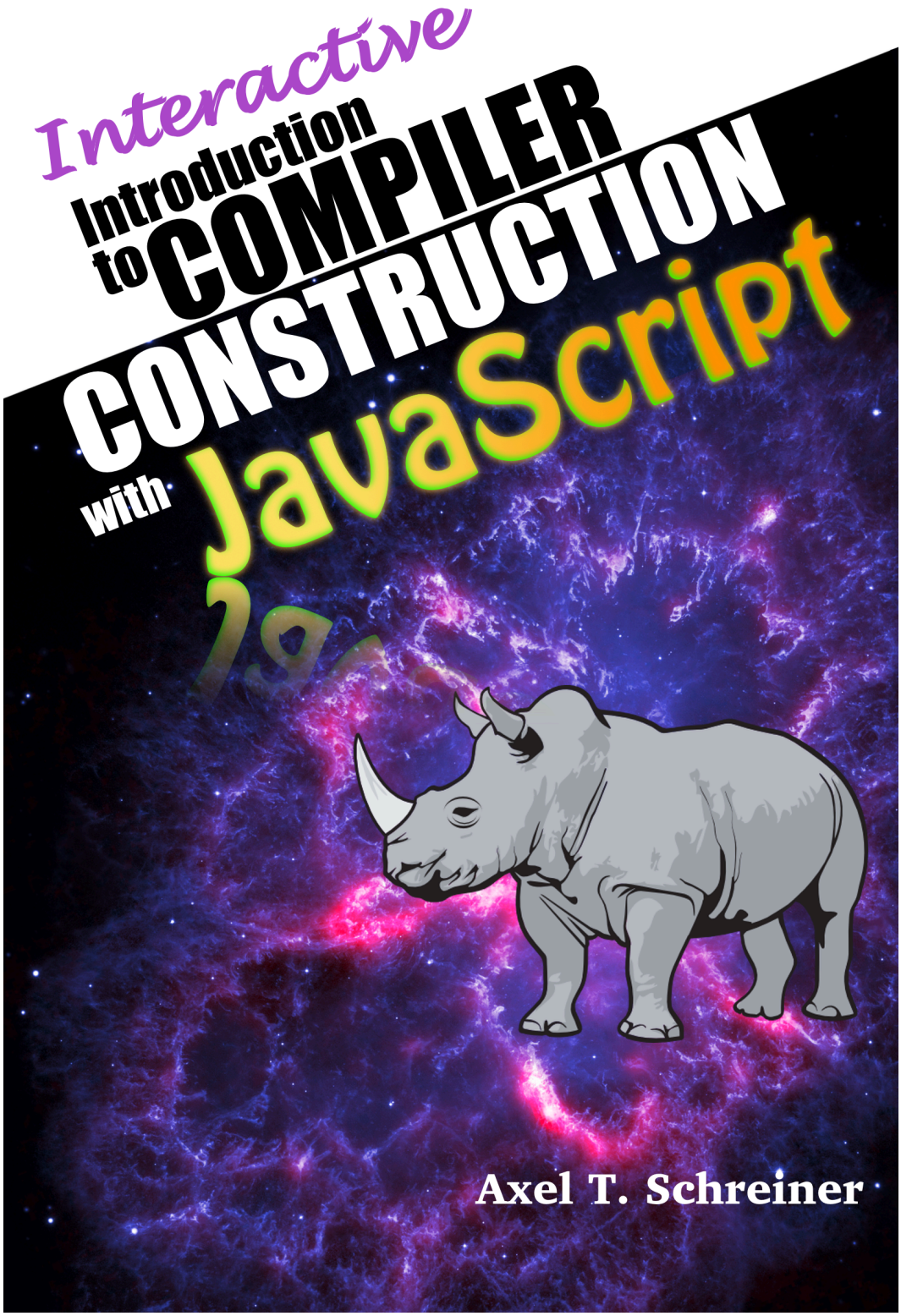

Interactive
Introduction to COMPILER CONSTRUCTION with JavaScript
Axel T. Schreiner

Overview

This book is a short introduction to compiler construction using JavaScript as the implementation language, recursive descent parsing‡ with the EBNF² parser generator and SLR(1)‡ parsing with BNF² as case studies for tools, and a little programming language with several extensions as working example. Here are the topics:

C: The One-Pass Compilers

The Method Browser
Rules and Actions
Symbol Table

D: The Compiler Kit

Tree Builders
Visitors

References

All references are links. Undecorated links are references within the book itself. Decorated links… reference the book's website:

- Examples[1] are linked to the practice page where they are automatically loaded to be executed, studied, changed, and executed again.
- Classes and methods developed in the examples[2] are linked to their documentation, to the documentation for classes and methods of the EBNF[2] and BNF[2] parser generators developed in the book, and to the documentation for scripting the practice page[2]. Documentation in turn links to the appropriate files and lines with the syntax-colored source code.
- Some links[3], mostly in chapter eight, reference the method browser where one can study how the implementation of a little programming language in this book evolves.

Other decorated links† reference the World Wide Web†:

- For JavaScript information, links point to the excellent MDN Web Docs†,
- a few links point to specific web pages, such as the original announcement of JavaScript†,
- and terminology is linked to definitions in Wikipedia‡ so that additional information is easily available — further references should be consulted from there.

Altogether, the book contains over 1100 unique links.

1. Compiler Terminology

What's in a Compiler?
What's in a Parser Generator?

A *compiler* translates a *source program* into an *executable*. It should also check that the source program is correct — at least in some sense.

The source program is written in the *source language,* the executable is written in the *target language,* the compiler itself is written in the *implementation language.* For this book, target and implementation language, both, are JavaScript because we will not discuss how to deal with low level languages such as assembler or machine language as target languages. However, beginning in chapter six, we will implement a small, typical programming language and target a simple stack machine which we will simulate in JavaScript.

Often, source and implementation language are the same, i.e., a compiler for a language is written in that language and can compile itself. Chapter nine illustrates how self-compilation for parser generators works.

Translation

The source program is presented as a text string. Translation is accomplished by cooperating tools which focus on specific aspects:

- A *scanner* represents the source program as a sequence of *terminals,* similar to words forming a sentence, and discards, e.g., white space and comments.
- A *parser* tries to construct a *syntax tree* from the terminals to represent the program structure.

Translation proceeds in several steps which might run sequentially or all at once:

- *Lexical analysis* is the process of running a scanner. Lexical analysis detects illegal characters which are neither terminals nor allowed to be ignored.
- *Syntax analysis* is the process of running a parser. The result is often called a *parse tree* or *syntax tree*. Syntax analysis detects syntax errors, e.g., unbalanced parentheses, unbalanced or missing keywords, bad punctuation, etc.
- *Semantic analysis* performs *type checking* and adds information to the parse tree, e.g., the data type produced by certain leaves or branches during execution. Semantic analysis should detect semantic errors, e.g., undefined values, illegal assignments, unsuitable function arguments, and type conflicts in general.
- Finally, the executable is created from the parse tree and the information collected by semantic analysis, if any.

Tools

Lexical and syntax analysis are fairly formal processes which are well supported by programming

tools such as EBNF[2], BNF[2], and their `Grammar`[2] classes:

- A *scanner generator* takes a description of terminals and constructs a scanner. The description is usually based on search patterns, also known as *regular expressions*‡. Chapter three explains how to create a scanner from regular expressions‡.
- A *parser generator* takes a description of a grammar and constructs a parser. Chapter two explains how to describe grammars and chapter four explains one technique to create a parser from a grammar; chapter ten discusses a more powerful technique. Chapter nine explains how to implement parser generators.
- Parser generators either produce trees to which semantic analysis can add annotations, or they provide a programming interface for the benefit of semantic analysis. Chapter five explains how to represent a syntax tree, and chapter six introduces a typical programming interface where semantic analysis or the creation of an executable can be attached.

Grammar and Trees

Syntax analysis depends on grammars and syntax trees which can be defined quite formally:

- A *context-free grammar*‡ consists of a finite set of terminals, a finite set of *non-terminals* — different from terminals and among them one called the *start symbol*, and a finite set of *rules*.
- Each *rule* is an ordered pair of a non-terminal and a finite, ordered sequence consisting of terminals and non-terminals. There may be different pairs with the same non-terminal and different sequences.

A *context-free grammar*‡ defines a syntax tree as follows:

- The root node of the syntax tree is labeled with the start symbol of the grammar.
- All branch nodes are labeled with non-terminals, all leaf nodes are labeled with terminals.
- A syntax tree is an *ordered tree*, i.e., subtrees appear in a branch node in a fixed order.
- Every ordered pair consisting of the non-terminal for a branch node and the ordered sequence of labels taken from the roots of the subtrees for the branch node must be in the rules of the grammar.

Language and Sentences

The rules of a grammar describe the branch nodes of every syntax tree, i.e., a grammar is the finite description of a language:

- A *sentence* is the (finite) ordered sequence of terminals which are the leaves of a syntax tree. There might be more than one syntax tree for a sentence.
- A *language* is the (usually) infinite set of all sentences which use the same grammar for their syntax trees. There might be more than one grammar for the same language.

Note that ordered pairs (rules) with empty sequences would result in leaf nodes which are labeled with non-terminals. They cannot be part of a sentence as defined above. Empty sequences look like a glitch in these definitions but chapter ten will illustrate that they are necessary. The previous definitions still work if a sentence is defined to consist of the leaves of a syntax tree with the exception of those leaves which are labeled with non-terminals.

2. Writing Grammars

What's in a Rule?
A Language for Rules

Formally, a rule is an ordered pair of two sequences of terminals and non-terminals, i.e., from the two sets of symbols in a grammar. For rules of a context-free grammars‡ the first sequence of the ordered pair is restricted to be a single non-terminal.

A grammar contains rules and describes sentences, i.e., sequences of input terminals. As an example we consider sentences which are sums of numbers such as

```
1 + 2
```

A rule to describe this sentence is, e.g.,

```
sum: '1' '+' '2';
```

So far, this notation for rules is inspired by the Backus-Naur form‡:

- `sum` is the non-terminal at left; we require that non-terminals are alphanumeric names, i.e., consist of letters, digits, and perhaps underscores, and start with a letter.
- The sequence of terminals on the right consists of `1`, `+`, and `2`; we enclose each in single quotes to indicate that they have to be spelled literally as shown. If we need a quote or backslash as part of a terminal we will precede it by a backslash.
- Finally, we require that the two parts of the rule are separated by a colon and that the entire rule is terminated by a semicolon.
- White space is necessary in a rule to separate non-terminal names, but it is ignored outside of terminals.

It should be noted that a list of such rules implicitly defines everything there is to know about a grammar: terminals and their representations in the input, non-terminals, rules, and the start symbol, by convention the non-terminal at left in the first rule — here it is `sum`.

Example 2/01[1] illustrates that this grammar can be used to recognize a sentence:

- Press **new grammar** to represent and check the grammar, and then
- press **parse** to perform syntax analysis, i.e., to try to recognize the sentence, i.e., to check whether a parse tree can be found.
- Now add or remove some spaces in the `program` area and press **parse** again to see that white space does not matter for recognition.
- Replace `1` and `2` in the `program` area and press **parse** again to see that other numbers cause errors as long as the rule in the `grammar` area requires `1` and `2`.
- Finally, replace `1` and `2` in the `grammar` area *and* in the `program` area and press **new grammar** and **parse** to allow different numbers.

Literals and Tokens

The example shows that spelling terminals literally in grammar rules may be useful but it is too restrictive. If we want to describe sums of numbers in general the specific value of a number is not important.

Therefore, we use two kinds of terminals:

- *Literals* represent themselves and are single-quoted strings using \ only to escape single quotes and \ itself.
- *Tokens* represent classes of similar inputs such as numbers, and are specified in rules as alphanumeric names which must be different from non-terminal names. For convenience we will use upper- and lower-case initials to immediately distinguish the two kinds of names.

Tokens have to be defined separately from the grammar itself. Chapter three explains in detail that regular expressions‡ are a very good way to describe tokens. For now it is sufficient that

```
/[0-9]+/
```

describes a `Number` of any length which consists of one or more of the digits from `0` to `9`.

Based on this definition the rule

```
sum: Number '+' Number;
```

describes all sums of two integer numbers such as

```
12 + 345
```

Example 2/02[1] contains the new rule in the `grammar` area and the definition of `Number` as a property with the token name and a (non-anchored) regular expression as the value in the `tokens` area:

- Press **new grammar** to represent and check the grammar.
- As it is set up, the example will produce an error when you press **parse** to create a parser and perform syntax analysis — the `program` area area contains a bit more than a sentence.
- Change either the `grammar` area or the `program` area to avoid the error.

Alternatives

A more useful sum of numbers has to contain at least one number, but it should be allowed to contain any number of numbers.

If the right hand side of a rule only contains terminals we would need very many rules. Here are the first three, all with the same non-terminal on the left:

```
sum: Number;
sum: Number '+' Number;
sum: Number '+' Number '+' Number;
```

There have been many extensions to the BNF notation‡ which inspired the grammar notation

used in most of this book and developed in this chapter. For alternatives, we collect all pairs with the same non-terminal at left and write instead:

```
1  sum: Number | Number '+' Number | Number '+' Number '+' Number;
```

We require that a *rule name*, i.e., the non-terminal at left, is unique, and we use `|` to concatenate separate alternative sequences for the right-hand side.

Example 2/03[1] illustrates that the grammar above has a problem:

- Press **new grammar** to represent and check the grammar and see the error message.
- You can toggle **no check** to suppress ambiguity checking and thus the error message when you press **new grammar**, but
- when you press **parse** to try and recognize the content of the `program` area you will find that only one number, i.e., the first alternative of the rule, can be recognized.
- Check out that once **no check** is set the order of the alternatives in the rule determines how many numbers can be in a sum.

In the formal definition of a grammar there are no restrictions on rules. However, it would be quite inefficient to have to try all alternatives to find one that actually confirms that an input is a sentence.

Therefore, we require that we must be able to select the appropriate alternative just by looking at the very next input — this is known as *one symbol lookahead* and it means that we will never back up in the input.

Example 2/04[1] shows a change to the grammar to circumvent the problem:

- Press **new grammar** to represent and check the grammar, and
- press **parse** to recognize the prefix and the sum with three numbers in the `program` area.
- Change the prefix and work with other sums in the `program` area.

The example uses the following grammar:

```
1  sum: 'a' Number
2       | 'b' Number '+' Number
3       | 'c' Number '+' Number '+' Number;
```

A prefix from `a` to `c` determines how many numbers the sum should contain. While this works it is not exactly user-friendly...

Chapter ten will show a less intuitive way to get from grammars to recognition which makes the grammar in example 2/03[1] perform as intended — still without backing up:

- The very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to process the grammar — this time there is no error message.
- Enter sums with one, two, or three numbers in the `program` area and press **parse** to see that all are recognized without errors.
- You can toggle **parser** to produce a trace (to be explained in chapter ten) which indicates when input is stacked (`shift`) and when a rule is satisfied (`reduce`, always followed by `goto`).

Recursion

Real sums contain any number of numbers, and they shouldn't require an obscure prefix.

Consider the following approach:

```
1  a: Number | Number '+' a;
```

This may look confusing but it is a finite description of an infinite number of sums with a number and as many additional plus signs with numbers as we choose:

- The first alternative states that `a` is a `Number`.
- Alternatively, `a` is a `Number`, a plus sign, and another `a` — which can be a single `Number`, or more...

Recursion is a finite way to describe infinite things. However, the particular rule above violates the principle that one symbol lookahead should be sufficient to select the successful alternative, i.e., two alternatives should not start with the same terminal — this was already illustrated in example 2/03[1].

The following rule seems to look better:

```
1  b: Number | b '+' Number;
```

This is another finite description of an infinite number of sums with as many numbers as we choose:

- The first alternative states that `b` is a `Number`.
- The second alternative allows us to append as many plus signs with numbers as we want.

However, example 2/05[1] which contains both rules, `a` and `b`, shows that there might still be problems:

- Press **new grammar** to represent and check the grammar and see an error message: `b` is left-recursive.
- Remove the rule for `b` from the grammar and remove `b` as an alternative in `sum`. Then press **new grammar** again to see another error message: `a` has ambiguous alternatives.

A rule, considered just as an ordered pair, is left-recursive‡ if the non-terminal at left appears first in the sequence at right, either directly, or even when tracing through other ordered pairs.

Example 2/06[1] contains a small change to `b` to illustrate that left recursion can involve several rules:

- Press **new grammar** to try to represent the grammar and see the error messages: this time `sum` is left-recursive.

```
1  sum: b;
2  b: Number | sum '+' Number;
```

Intuitively, this grammar does look doubtful — even without ambiguities. "One symbol lookahead" would select the first alternative of `b` and that is only a single number.

If the alternatives for `b` are interchanged and the rules are tried in order it looks like a `sum` is a `b` is a `sum` is a `b` ... will we ever start to look at the first number in the input?

It all depends on the approach used to interpret the grammar. The technique explained in chapter ten can deal with left recursion. Try example 2/06[1] again:

- The very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to process the grammar — this time there is no error message.
- Enter various sums in the `program` area and press **parse** to see that all are recognized.

Optional Brackets

We want to describe sums and avoid ambiguities and left recursion. Consider:

```
1 sum: Number more;
2 more: '+' sum |;
```

A `sum` consists of a number and `more`, and `more` adds a plus sign and another number or eventually nothing because `more` has an empty alternative which stops the right recursion on `sum`.

At the end of chapter one it was noted that empty alternatives are awkward and they can easily be overlooked in a rule. Therefore, we extend our grammar notation:

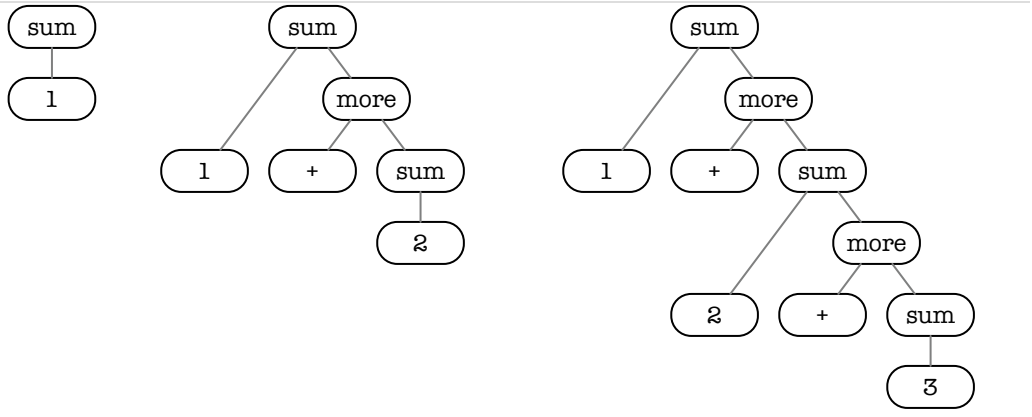
```
1 sum: Number [ more ];
2 more: '+' sum;
```

We use brackets to indicate that part of an alternative is optional and we forbid empty alternatives — including those which only consist of optional parts...

Work with this grammar in example 2/07[1] to see that we can finally have any number of numbers in a sum:

- Press **new grammar** to process the grammar,
- enter various sums in the `program` area, and
- press **parse** to see that all are recognized.

The grammar uses right recursion to get long sums with syntax trees such as the following:



If these trees were interpreted to perform arithmetic they would compute the sums from right to

left, i.e., right recursion in the grammar results in right-associative trees. This is suggested by the brackets in the output from example 2/07[1]:

```
[ '12' [ [ '+' [ '345' [ [ '+' [ '6789' null ] ] ] ] ] ] ]
```

Deeper nesting occurs at right.

Rules can contain alternatives. We can use this to allow addition and subtraction in the rule for `more`:

```
sum: Number [ more ];
more: '+' sum | '-' sum;
```

Brackets can contain alternatives, too. Here is another way to describe the same language, i.e., the same set of sentences, i.e., sums with any number of additions and subtractions:

```
sum: Number [ add | subtract ];
add: '+' sum;
subtract: '-' sum;
```

In chapter six this last version will turn out to be the most useful.

For now, you can experiment with the first variant in example 2/08[1] and the second variant in example 2/09[1]. Either one accepts arbitrary sums and differences and even produces the same set of brackets, numbers, and operators:

```
[ '12' [ [ '-' [ '345' [ [ '+' [ '6789' null ] ] ] ] ] ] ]
```

Chapter five will explain what all the brackets in the output mean. For now we note that the different grammars produce the same output, but something else is more significant. Consider

```
   12  (    -    345  (    +    6789          )    )
```

which is the same output with some clutter removed.

Mathematical conventions dictate that the innermost parentheses have to be elaborated first, i.e., this is likely to be interpreted as

```
add 6789 to 345
then subtract the result from 12
```

i.e.,

```
12 - (345 + 6789)
```

and that's probably not what the program

```
12 - 345 + 6789
```

intended...

Therefore, it is important to note that

- there can be different grammars describing the same sentences and language, and
- if a grammar is intended to suggest meaning, such as mathematical operator precedence and associativity, rules have to be crafted with that in mind —
- right recursion produces right associativity and for now left recursion is not allowed for the interpretation of rules from left to right even if it would produce the commonly used left associativity.

Some Braces

We started with the grammar notation known as Backus-Naur Form‡ which was created to describe the programming language Algol‡. Recursion was used to express any kind of iteration so that a finite grammar can describe sentences of arbitrary length.

Niklaus Wirth noticed that many programmers are more comfortable with explicit constructs for iteration and proposed the first of many extensions to BNF‡ which all offer such constructs.

Assuming that a sum has at least two numbers, here is another way to define a grammar:

```
1 sum: Number { '+' Number | '-' Number };
```

Unlike other extensions to BNF‡, this book uses braces to enclose alternatives which must occur *one or more* times, i.e., a sum conforming to this grammar contains at least two numbers.

Check out example 2/10[1] and compare the output

```
1 [ '12' [ [ '-' '345' ] [ '+' '6789' ] ] ]
```

to the output of example 2/08[1] and example 2/09[1] shown before:

```
1 [ '12' [ [ '-' [ '345' [ [ '+' [ '6789' null ] ] ] ] ] ] ]
```

Iteration makes for simpler looking output — this will be discussed further in chapter five and six.

Many Iterations

What about a sum that consists of a single number or more?

As part of a rule, both, brackets and braces, can contain one or more alternatives. Alternatives are on the right-hand side of rules, where braces and brackets made their appearance. Combining the ideas, this means that braces and brackets can be nested.

Clearly, it is superfluous to have brackets only contain another set of brackets or braces only contain another set of braces. More importantly, we avoid empty alternatives, i.e., an alternative cannot just consist of something optional in brackets, i.e., braces cannot just contain sets of brackets.

But — something in braces appears one or more times; therefore, when brackets directly contain

braces, together, they indicate something that can appear zero or more times:

```
sum: Number [{ '+' Number | '-' Number }];
```

Check out example 2/11[1] and, in particular, confirm that single numbers are recognized and compare the output for `1 + 2`

```
[ '1' [ [ [ '+' '2' ] ] ] ]
```

to the output for the same sum in example 2/10[1]:

```
[ '1' [ [ '+' '2' ] ] ]
```

We will return to this in chapter five.

Quick Summary

- *Terminals* are *literals* and *tokens*.
- *Literals* represent themselves as input and are single-quoted strings using `\` only to escape single quotes and `\` itself.
- *Tokens* represent sets of inputs, e.g., numbers, and are alphanumeric names starting with a letter — in our examples in upper case.
- Tokens are defined with search patterns, e.g., `/[0-9]+/` for numbers.
- Formally, *rules* are ordered pairs, each consisting of a non-terminal and a sequence of terminals and non-terminals.
- *Non-terminals* are alphanumeric names starting with a letter — in our examples in lower case. The sets of non-terminal names and token names must be disjoint.
- Backus-Naur Form‡ uses `:` to separate the non-terminal from the sequence in the rule, `|` to combine all alternative sequences for the same non-terminal, and `;` to terminate a rule.
- There has to be exactly one rule for each non-terminal and we call the non-terminal name the *rule name*.
- The rule name of the first rule in the list of rules of a grammar is the name of the start symbol.
- Extended BNF‡ adds notations for optional and repeated sets of alternatives. We use `[` and `]` (brackets) to enclose a set of *optional* alternatives and `{` and `}` (braces) to enclose a set of alternatives that must appear *some* times, i.e., one(!) or more times.
- We forbid empty alternatives, e.g., an empty sequence or a sequence containing only optional items. In particular, braces cannot directly contain only brackets.
- We allow brackets to directly contain braces because this denotes a set of alternatives which can appear *many* times, i.e., zero or more times.
- By convention, white space in a grammar and usually in input will be ignored.

Recursion Revisited

Example 2/12[1] contains a fairly comprehensive grammar describing a comma-separated list of

expressions, with the typical arithmetic operators and parentheses:

```
list: sum [{ ',' sum }];
sum: product [{ '+' product | '-' product }];
product: signed [{ '*' signed | '/' signed }];
signed: [ '-' | '+' ] term;
term: Number | '(' sum ')';
```

This grammar shows the necessary use of recursion for balanced notations such as parentheses. This is neither left- nor right-recursion.

The rule for `term` contains both parentheses to keep them balanced and by recursion nests a `sum` and thus balanced parentheses to any depth.

The rule for `sum` uses iteration to arrange for any number of `product` and thus `term` in sequence, and they can contain balanced parentheses, but the grammar does not permit unbalanced parentheses in a sentence.

Example 2/13[1] contains a grammar describing typical list constructs:

```
terminated: { [ separated ] ';' };
separated:  element [{ ',' element }];
element:    Number | enclosed;
enclosed:   '(' [ separated ] ')';
```

- A sentence is a list with one or more entries, each `terminated` with a semicolon.
- An entry itself can be empty (because `seperated` is optional), or it is a list of one or more elements which are `separated` by single commas.
- An `element` is either a `Number`, or a list `enclosed` by a balanced number of parentheses.
- The `enclosed` list consists of zero or more elements which are `separated` by single commas.
- `terminated` or `separated` are blueprints for statements in a programming language. `enclosed` is a blueprint for function parameters or arguments.

It is instructive to investigate various sentences, e.g.,

```
( (1,2), 3, (()) ); ;
```

even if the output looks much messier than the sentence itself.

Tokens Revisited

As a final example in this chapter, example 2/14[1] contains the grammar from example 2/12[1], with one more rule added to define `Number` right in the grammar from single digit literals:

```
list: ...
Number: { '0' | '1' | '2' | '3' | '4'
        | '5' | '6' | '7' | '8' | '9' };
```

This suggests that we might not need tokens, i.e., names for classes of input such as numbers or names.

However, example 2/14[1] demonstrates that the result is not quite the same. As we shall see in the next chapter, our recognition process usually ignores white space in the input. With the rule above, both of the following numbers

```
12,
1 2
```

are acceptable for a sentence and the result, even for a single number, can be unexpected:

```
[ [ [ [ null [ [ [ [ '1' ] [ '2' ] ] ] ] ] null ] null ] null ]
```

Plus, character recognition with a grammar, while mostly automated, is much less efficient than character recognition with regular expressions‡, and we will use these in recognizing literals anyhow.

3. Scanning Input

Terminals, Tokens, and Literals
Taking Input Apart

Chapter one explained that *lexical analysis*, the first phase of the translation process in a compiler, breaks the source program — usually a string — into parts called *terminals* in order to simplify the job of the next phase, *syntax analysis*.

Chapter two explained that a grammar describes *sentences*, i.e., *sequences of terminals* which conform to the rules of the grammar.

Therefore, a grammar will at least describe what terminals it needs for the rules, even if it does not really need to describe what the terminals look like in an actual source program.

Our grammar notation distinguishes two kinds of terminals, literals and tokens:

- A *literal* is a single-quoted string in a rule and the content of the string represents the literal in the source program. Backslashes are used to escape single quotes and backslashes in the string content in the rule.
- A *token* is just a name in a rule, different from all non-terminal names. Tokens are defined as key-value pairs (properties) of a JavaScript object where the key is the token name and the value is a regular expression‡ (search pattern) which describes how the token is represented in the source program.

If the grammar in example 2/10[1] is represented and checked by pressing **new grammar** the output is a short description of the grammar:

```
1    0 sum: Number { '+' Number | '-' Number };
2
3  literals: '+', '-'
4  tokens: Number
```

The description contains a numbered list of rules, a list of single-quoted literals, and a list of token names used in the rules. The token patterns are in the text of the `tokens` area:

```
1  { Number: /[0-9]+/ }
```

Together with the convention to ignore white space this is enough information to break the source program up into terminals:

- Press **new grammar** to represent and check the grammar and
- press **scan** to perform lexical analysis.

The output shows how the string in the `program` area

```
1  12 - 345 + 6789
```

is cut into pieces:

```
> g.scanner().pattern /(\s+)|([0-9]+)|(\-)|(\+)/gm
> g.scanner().scan(program)
(1) "12" Number
(1) '-'
(1) "345" Number
(1) '+'
(1) "6789" Number
```

The first line in the output displays the search pattern which is used for lexical analysis. It will be discussed below. The second line contains the JavaScript expression to create a scanner and perform lexical analysis.

Each of the remaining lines in the output describes one `Tuple`[2] object, consisting of a line number, a piece of the input, and a classification as a terminal:

- For a literal the piece of input is enclosed in single quotes.
- For a token the piece of input is enclosed in double quotes and the token name is shown, too.
- Illegal input is enclosed in double quotes and there is no token name.

The output is independent of any white space which may or may not be in the `program` area. If, for example, some newline characters are inserted between the numbers and operators, only the line numbers in the `Tuple`[2] objects will change.

The **scan** button uses the `scan()`[2] method of a `Scanner`[2] object which is constructed from the grammar[2]. This chapter explains how an object of the `Scanner`[2] class chops up the input.

Search Patterns

Computer users always search for specific pieces of text — buried in files or among collections of file names. Even the earliest versions of operating systems such as DOS‡, Unix®‡, or Windows‡, provided more than "literal" searches — they supported search patterns. For example

```
rm abc d?f g*i [0-9*a-z]\*
```

This Unix® command might not be terribly realistic, but it demonstrates the principle. It would remove

- the file literally named `abc`,
- all files with three-character names starting with `d` and ending with `f`,
- all files with names starting with `g`, ending with `i`, and containing zero or more characters in-between,
- and, finally, all files with a name consisting of a single digit, asterisk, or lower-case letter, followed by a literal asterisk.

Search patterns are usually called *regular expressions* because they employ a very concise notation for regular grammars‡ which are context-free grammars‡ where the right-hand side of rules can only use a terminal or a terminal followed by a single non-terminal.

The patterns above demonstrate that this allows for iteration, but it does *not* allow, e.g., for a construction that can balance parentheses.

Regular expressions can have very efficient implementations based on the theory of finite state automata‡.

One has to be cautious, however. While most modern programming languages support regular expressions for searching and text validation, neither the syntax nor the semantics of regular expressions is consistent across different languages, e.g.,

```
1 (a.*b)|[cde]+
```

should find `a` followed by any number of characters and then `b`, or find a sequence of one or more of the characters `c`, `d`, or `e`. However, depending on the platform, the expression might well find the first match, or the longest match, or even all matches...

The EBNF module/parser generator[2] is implemented in JavaScript; therefore these tutorials will use and discuss regular expressions as they are supported in JavaScript†.

Different flavors of regular expressions can be tested on the Regular Expressions 101 website†.

Regular Expressions 101

This section is a crash course on patterns as they are supported in JavaScript†, specifically, to deal with the question how typical terminals in programming languages can be expressed through patterns. Many of the more esoteric aspects of regular expressions are not covered here.

A pattern is used to recognize part of an input string. Patterns are a shorthand notation for a type of grammars; therefore, we should expect that there is a concept of literals and tokens for patterns:

- Letters, digits, and (very few) special characters are literals and recognize themselves. This can be used to recognize words in a programming language:

```
1 if then else while do
```

- A period is a token in the language of regular expressions which recognizes any character with the exception of a line separator:

```
1 .
```

- Brackets, `[` and `]`, enclose a sequence of characters, which is termed a *character class*. This token recognizes any single one of its constituent characters:

```
1 [0123456789]
```

- If the sequence inside brackets starts with `^` the token will recognize any single character *not* in the sequence — most of the time this is more than one bargains for:

```
1 [^a-z]
```

- `-` between two characters *inside* the sequence denotes the inclusive range.

All of this together can be used to recognize single digits, single lower-case letters, or all special

characters in the ASCII character set†

```
1 [0-9] [a-z] [!-/:-@[-`{-~]
```

- Most non-alphanumeric characters have a special meaning in a pattern. This can be defeated by preceding a character with the \ (backslash) character. In particular, \\ is the literal to recognize one backslash.

Simply preceding every character in a pattern with backslash is unwise because some letters acquire special meaning when preceded by backslash. For example, this literal recognizes any single white space character, even outside the ASCII character set‡:

```
1 \s
```

Chapter two explained that grammars use rules built from literals and tokens to describe more complicated sentences.

Instead of rules, regular expressions use certain operators to build complicated patterns from simpler pieces. From a semantic perspective the constructs have much in common with the extensions to BNF‡ discussed previously.

- To begin with, a sequence of pattern pieces recognizes a sequence of input parts each of which matches the corresponding pattern piece.

For example, this sequence recognizes command language variable names $0 through $9:

```
1 \$[0-9]
```

- One should always escape $ because $ at the end of a pattern *anchors* recognition, i.e., it demands that recognition extend all the way to the end of input.
- ^ at the beginning of a pattern also anchors recognition, i.e., recognition must start at the beginning of input.

For example, this pattern recognizes a string only if it consists exactly of two command variable names separated by exactly one white space character:

```
1 ^\$[0-9]\s\$[0-9]$
```

- Iteration is expressed by ?, +, or * directly following a pattern piece to indicate that the piece is optional, recognized one or more times, or both, in the input.

This can be used, e.g., to recognize optionally signed integer numbers (line 1 below), alphanumeric variable names starting with a letter (line 2), or such a name and number optionally surrounded and definitely separated by white space as the only input (line 3):

```
1 [-+]?[0-9]+
2 [a-zA-Z][a-zA-Z0-9]*
3 ^\s*[a-zA-Z][a-zA-Z0-9]*\s+[-+]?[0-9]+\s*$
```

- Patterns support alternatives: pattern pieces can be joined with the operator |.
- Just as in grammars, sequences take precedence over alternatives.
- Iterations take precedence over sequences.

Here is a pattern for a number which would be interpreted as decimal, hexadecimal, or octal, depending on the prefix:

```
1  [1-9][0-9]*|0[xX][0-9a-fA-F]+|0[0-7]*
```

The order of alternatives can make a difference because JavaScript is happy with the first alternative that matches. Use the Regular Expressions 101 website† to see that the previous pattern will not recognize `0xA` if the last alternative is specified first, i.e.,

```
1  0[0-7]*|[1-9][0-9]*|0[xX][0-9a-fA-F]+
```

- Parentheses `(?:` and `)` can be used to group pattern pieces and thus change precedence and nest alternatives into sequences and both into iterations.

Here is a pattern for a non-empty, single-quoted string which uses backslash to escape a single quote, backslash, or newline (represented as `\n`):

```
1  '(?:\\['\\\n]|[^'\\\n])+'
```

Admittedly, JavaScript patterns† also allow simple parentheses; however, simple parentheses introduce *capture groups* which serve a very special purpose that is critical for the scan()[2] method developed in the next section.

We have already seen the patterns needed to describe most of the tokens for a typical programming language. Among them, strings are the most complicated but the last example showed how escape conventions and newlines can be handled.

Slightly more complicated is the removal of comments from input. Patterns cannot balance nests of leading and trailing sequences; therefore, patterns cannot deal with nested comments. Other than that:

- Line comments are simple — they range from a leading sequence to the end of a line or of the input.

```
1  \/\/[^\n]*(?:\n|$)
```

- JavaScript uses `//` for the lead of a comment and also uses `/` to enclose regular expressions; therefore, the pattern above for JavaScript comments escapes the leading '//'.
- Comments enclosed by unique single characters use the same design. E.g., Pascal‡ encloses comments in braces and allows multiple lines in a comment:

```
1  {[^}]*}
```

- Finally, comments enclosed in overlapping sequences, such as `/*` and `*/` in JavaScript, require careful specification:

```
1  \/\*(?:[^*]|\*+[^/*])*\*+\/
```

Here, zero or more character groups follow the opening `/*` and one or more asterisks precede the closing slash. Each character group consists of any single character but an asterisk, or of one or more asterisks followed by a single character which is neither a slash nor an asterisk. Use the

Regular Expressions 101 website† to see that, e.g., `/***/` is recognized but `/*/` is not.

As far as possible, example 3/01[1] contains all the pattern examples in this section.

- Press **new grammar** to represent and check the grammar, and then
- press **scan** to see what tuples are found in the `program` area.
- Some definitions in the `tokens` area are *not* used in the grammar. Why not?
- Note that if you click on the label of a text area the area will expand. Shift-click will restore the default layout, and alt/option-click tries to open a separate, non-editable window with syntax highlighting.

`scan()`

Given the literals and descriptions for tokens, how does the `scan()`[2] method perform lexical analysis, i.e., take an input string and return the list of `Tuple`[2] objects which describe the terminals in the string?

JavaScript represents regular expressions as objects of the class `RegExp`†. An object can be constructed by enclosing a pattern in single slashes, or it can be constructed from a string — but then, backslash and quotes would have to be escaped according to string rules before they play their role in a pattern.

The `exec()`† method for a `RegExp` object accepts a string. If the regular expression matches the string somewhere the method returns an array, otherwise it returns `null`. The first element in the array is that part of the string which was matched. The remaining elements in the array, if any, contain the components of the matched part which the capture groups in the regular expression, if any, matched.

This gets tricky if capture groups are nested, but here is a simple example which should suggest how `exec()`† can be set up to do the job of lexical analysis. Consider a string with the alphabet

```
1  abcdefghijklmnopqrstuvwxyz
```

and consider a pattern with three capture groups

```
1  ([a-f]+)(.*)([u-z]+)
```

- Matches are greedy, i.e., the first group matches the beginning of the alphabet, and it will grab *all* of the letters `abcdef`,
- the third group only matches the end of the alphabet, i.e., in this case `z`,
- because the second group is greedy, too, and matches all the letters in-between.

Therefore, the JavaScript function call

```
1  /([a-f]+)(.*)([u-z]+)/.exec('abcdefghijklmnopqrstuvwxyz')
```

returns an array with four elements of length 26, 6, 19, and 1.

The pattern would also match the string `JavaScript` because the first capture group is happy with the first `a`, the second group can match nothing, and the third group matches `v`. The resulting array again contains four strings, containing `JavaScript`, `a`, nothing, and `v` — check it

out on the Regular Expressions 101 website†.

`exec()`† can be configured by appending certain flags† to the pattern, such as

- `g` (global) arranges that `exec()`† can be called multiple times to find all matches of the pattern in the string.
- `m` (multiline) allows the anchors `^` and `$` to match line feeds in the string.

A capture group extracts a piece of input and classifies it in the resulting list by the group's position in the pattern — extraction and classification is what lexical analysis has to do.

Example 3/02[1] creates the following pattern with four alternatives, each a capture group, namely white space, "names" (which include literals like `if`), "numbers", and the operator `<=`:

```
1  > g.scanner().pattern = /(\s+)|([a-z]+)|([0-9]+)|(\<\=)/gm
```

Depending on the string matched against the pattern, exactly one group will capture something — unless the entire match fails.

As for lexical analysis:

- If nothing is matched or if the match does not start at the beginning of the input string there are illegal characters in the input. They can be represented by a specific `Tuple`[2] object which does not classify the input.
- A match of the first group has to be trimmed off the input — the scanner skips, e.g., white space.
- The next set of groups consists of the tokens of the grammar which might have to be screened further. In this example, these are sequences of lower-case letters ("names") and sequences of digits ("numbers").
- Note that something like `iffy` should be recognized as a name and not as the literal `if` followed by the name `fy`.
- The last set of groups consists of the literals of the grammar which are not matched by the tokens, in this case `<=`.

The lexical analysis pattern can be composed as alternatives of capture groups with

- one pattern for white space and comments,
- one pattern for each token,
- and finally one pattern for each literal in order of decreasing length so that, e.g., `<=` is recognized before `<`.
- Patterns for literals which are matched by a token pattern can be omitted.

This assumes that white space and token patterns all match different input, and that the result of a token match such as "names" is screened, e.g., by looking the matched input up in a map of literal values which take precedence over the token.

In general, overlapping patterns can cause problems. It is not possible to check that the patterns are "different" enough when the `Scanner`[2] is constructed but example 3/01[1] and example 3/02[1] show how a grammar and some input can be used to test lexical analysis thoroughly:

- The grammar should contain all tokens and literals as simple alternatives. Press **new grammar** to represent the grammar.
- The `program` area should contain representations of all tokens and literals to be tested — the input does not have to be a sentence. Press **scan** to see what tuples are found.

Example 3/03[1] illustrates that the token patterns may have to be ordered. Consider integer constants, represented as digit strings, and floating point constants, required to contain a decimal point:

```
Int: /0|[1-9][0-9]*/,
Real: /(?:0|[1-9][0-9]*)\.[0-9]*/
```

The input string in the `program` area is

```
1 2.3
```

If the `scanner()`[2] places the `Int` pattern before the `Real` pattern lexical analysis will find three `Int` tokens and one illegal character:

```
> g.scanner().pattern =
  /(\s+)|(0|[1-9][0-9]*)|((?:0|[1-9][0-9]*)\.[0-9]*)/gm
> g.scanner().scan(program)
(1) "1" Int
(1) "2" Int
(1) "." 
(1) "3" Int
```

As a crutch to fix this problem, the token patterns are sorted by name before they are combined for the lexical analysis pattern. In example 3/04[1] the token name is changed from `Real` to `Float`.

- Press **new grammar** to prepare the grammar, and
- press **scan** to test the lexical analysis pattern.

Now the order of the token patterns is interchanged, `Float` before `Int`. The floating point pattern will find the floating point number and the output consists of two tuples:

```
> g.scanner().pattern =
  /(\s+)|((?:0|[1-9][0-9]*)\.[0-9]*)|(0|[1-9][0-9]*)/gm
> g.scanner().scan(program)
(1) "1" Int
(1) "2.3" Float
```

Note that `0.` and `0.1` are valid but `.1` is not. This could be changed with an alternative in the `Float` pattern.

Screening

Many programming languages use reserved words‡, i.e., they forbid that literals such as `if`, `then`, and `else` are used as variable names — these literals should always indicate a control structure.

From the perspective of a grammar for a programming language, variable names are a class of inputs, i.e., they are a token in the grammar, and the scanner must not report a reserved word like `if` as the particular token for variable names.

Screening prevents that.

A grammar is represented as a `Grammar`[2] object. The `scanner()`[2] method is called with an

optional pattern argument which describes the parts of input which should be ignored, by default white space.

`scanner()`[2] returns a `Scanner`[2] object which contains the lexical analysis `pattern` described above and a list, `terminals`, of the unique `Lit`[2] and `Token`[2] objects corresponding by position to the capture groups of the pattern.

The `scan()`[2] method of this `Scanner`[2] object takes an input string, applies the lexical analysis pattern, and classifies the matching part of the input as a literal or token, represented as a `Lit`[2] or `Token`[2] object, by using the index of the capture group and the `terminals` list.

If the `scan()`[2] method proposes a part of the input as a token while it could be a literal it is used as an index into a map of literals with which the token overlaps — this should be an efficient operation in JavaScript. If the string is found, the result is the corresponding `Lit`[2] object in place of the proposed `Token`[2] object. Thus, literals masquerading as tokens are detected and lexical analysis can report them properly.

Screening slows input processing down and should be used only if really necessary; e.g., a token for numbers will not require screening if all reserved words in the language consist of letters and all operators consist of special characters.

To streamline the process all literal values which can appear in the input are checked against all token patterns when the `Scanner`[2] object is constructed. If a literal value is classified as a token the corresponding `Lit`[2] object is added to a map `.screen` for the `Token`[2] object. If this map exists, it is consulted during `scan()`[2] as described above. Additionally, the literal value is dropped from the lexical analysis pattern because it would not match anything on its own.

Example 3/05[1] shows that things can go really wrong if a token pattern *partially* overlaps a literal. Patterns for "numbers" and alphabetic "names" would both overlap the literal `'Formula1'`.

Fortunately, this case is rare in the world of programming languages but the `scanner()`[2] method has an optional parameter so that the pattern pieces can be sorted explicitly. The parameter can be used in a JavaScript program; example 3/05[1] would be corrected as follows:

```
import * as EBNF from 'modules/ebnf.js';
const EBNF = await import("./ebnf.js");
const grammar = " examples: Name | Number | 'Formula1'; ",
  tokens = { Name: /[A-Za-z]+/, Number: /[0-9]+/ },
  program = " Formula 1 \n Formula1 ";

const g = new EBNF.Grammar(grammar, tokens); g.check();
console.log(g.toString());
console.log(g.scanner().pattern);
console.log(g.scanner().scan(program).join('\n')); // the mistake

const terminals = [ g.lit("'Formula1'"), g.token('Name'), g.token('Number') ];
const s = g.scanner(/\s+/, terminals); // the correction
console.log(s.pattern);
console.log(s.scan(program).join('\n')); // the intended output
```

In Node.js† this produces the intended output:

```
> console.log(s.pattern);
/(\s+)|(Formula1)|([A-Za-z]+)|([0-9]+)/gm
> console.log(s.scan(program).join('\n')); // the intended output
(1) "Formula" Name
(1) "1" Number
(2) 'Formula1'
```

Alternatively, an alphanumeric `Name`

```
Name: /[A-Za-z][A-Za-z0-9]*/
```

would overlap `Formula1` completely and screening would resolve `Formula1` correctly as a literal.

Quick Summary

- *Lexical analysis* breaks input into parts represented by the literals and tokens used in a grammar. It skips the irrelevant parts of the input — usually white space and comments. Sequences of one or more unrecognized input characters are also reported.
- Regular expressions‡ (search patterns) are the tool of choice to describe literals and tokens and they are the basis for automatic implementation of lexical analysis.
- `scanner()`[2] creates a `Scanner`[2] object with a `scan()`[2] method which accepts an input string and returns a list of `Tuple`[2] objects describing the input parts.
- `scan()`[2] uses a `RegExp`† composed of the patterns for all tokens and some literals in the grammar, plus a pattern for input parts to be skipped.
- The order of patterns is important: (usually) tokens, sorted by name, before literals, literals sorted by length in reverse order.
- Overlapping token patterns may cause issues, in particular, if a token pattern *partially* overlaps a literal. As a last resort, the pattern order can be set explicitly when the `Scanner`[2] object is created.
- Some tokens, such as the class of identifiers in a programming language, might hide *reserved words*, i.e., literals such as `if` `then` `else` which have a special meaning as a control structure.
- Such tokens are marked when the `Scanner`[2] object is created and matched input is checked against a token-specific map of hidden literals so that the input part can be reported correctly.
- The Regular Expressions 101 website† is great for testing.
- Example 3/01[1] contains many patterns for terminals used in programming languages. A few of these are too general to be used in the example...

4. Recognizing Sentences

What's in a Sentence?
Check Before You Call!

Chapter two explained how to write grammars and chapter three explained how to extract terminals from input. This chapter discusses how to decide if an input string, i.e., a sequence of terminals delivered by lexical analysis, is a sentence of a language described by a grammar. Chapter one called this *syntax analysis* and it should result in a *syntax tree* for the sentence.

A grammar contains all the information required to create a `Parser`[2] object with a method `parse()`[2] which implements syntax analysis — as long as the grammar is up to the job. This chapter also shows how to check a grammar to see if it can be used to create a `Parser`[2].

Both, this implementation of syntax analysis and the corresponding grammar check, are very intuitive and the technique is called LL(1)‡ — left-to-right parsing with leftmost derivation. Chapter 10 discusses SLR(1)‡ — left-to-right with rightmost derivation — an approach which is not quite as intuitive but which can use a larger class of grammars.

Trees

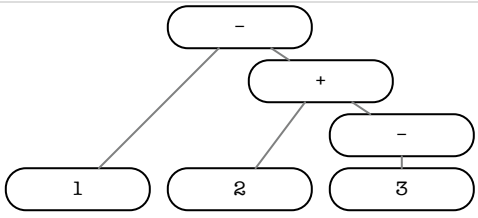
Here is a grammar for a sum of at least two, optionally signed numbers:

```
1  sum: term [{ more }];
2  more: '+' term | '-' term;
3  term: [ '+' | '-' ] Number | '(' sum ')';
```

The following input is in fact a sentence:

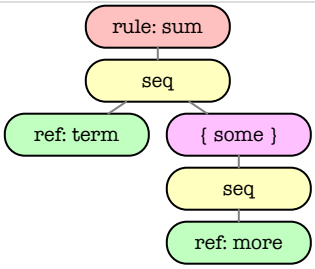
```
1  1 - (2 + - 3)
```

Arithmetic expressions such as this sum are often represented as expression trees where branch nodes are labeled with operators and leaf nodes are the numbers and variables in the expression.



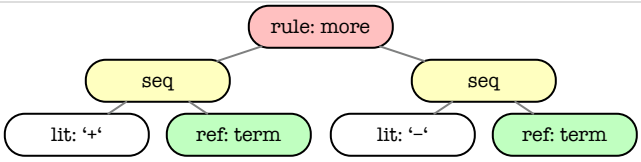
The levels of such an expression tree imply precedence, i.e., evaluation order; parentheses will not appear in an expression tree.

Similarly, we can visualize the rules of a grammar as trees. Here is `sum`:

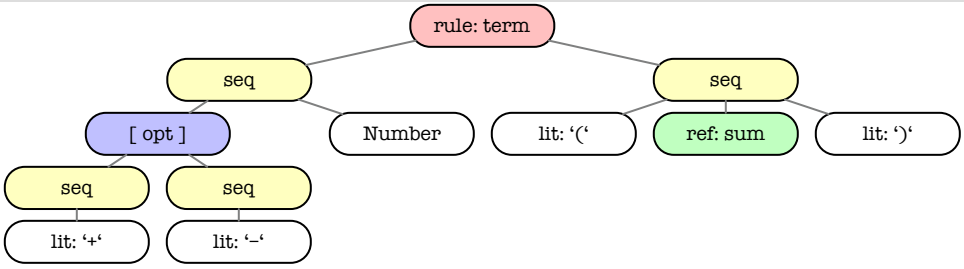


Recall that the right-hand side of a rule such as `sum` consists of one or more alternatives. Each of these is a sequence. In this case there is only one and it consists of a reference `term` to another rule and something in braces to be repeated one or more times.

Braces, in turn, contain alternatives, and each of those is again a sequence. In this case there is one alternative, namely a sequence consisting of a reference `more` to another rule. Here is the tree for the rule `more`:



`more` has two alternatives. Each must again be a sequence. One contains the literal `'+'`, the other contains `'-'`, and each sequence then references the third rule, `term`:



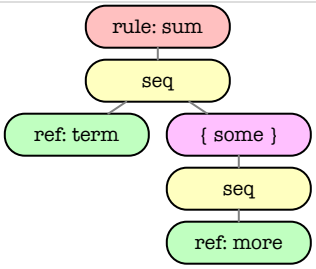
`term` also has two alternative sequences. One contains brackets with signs and a `Number` token, the other contains literals with parentheses enclosing a reference to a `sum`.

Brackets indicate something optional and contain alternatives, just as braces do. In this case there are two sequences and each only contains a literal with a sign.

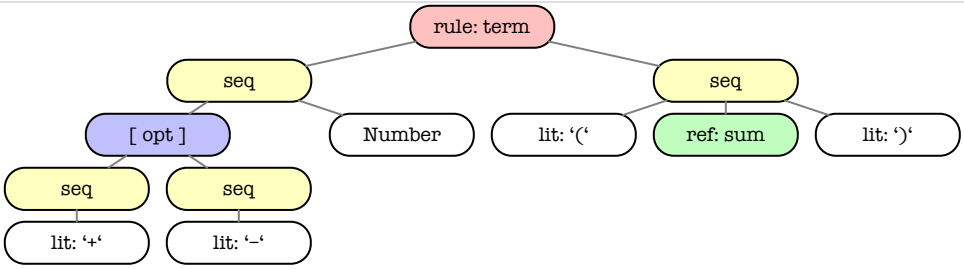
Recognition

For the purpose of recognition, rules can be viewed as functions. The diagrams suggest that the rule trees consist of objects which belong to a small set of classes each of which can have methods.

At the top level, the `parse()`[2] method of the `Parser`[2] object generated by the `parser()`[2] method from the grammar delegates the recognition problem to the start rule `sum` by calling a `parse()`[2] method on the `Rule`[2] object which is the root of the tree:



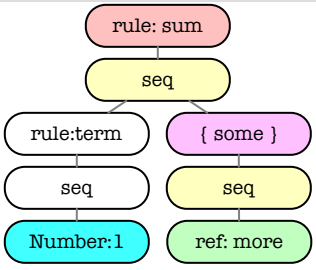
Within the `sum` rule, there is only one alternative and that sequence delegates to its first element, which references and delegates to the rule `term`:



`term` has two alternatives — time to look at the first terminal provided by lexical analysis. The proposed sentence is

```
1 - (2 + - 3)
```

and it starts with `1`. This `Number` only fits the first alternative of `term`, and only after the optional part of the sequence is skipped.



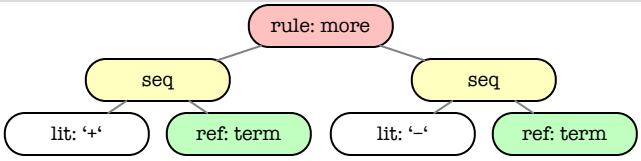
We are on our way to building the syntax tree which will prove that the input is a sentence. The process can be viewed in example 4/01[1]:

- Prepare the grammar by pressing **new grammar** and press **scan** to see what tuples lexical analysis creates.

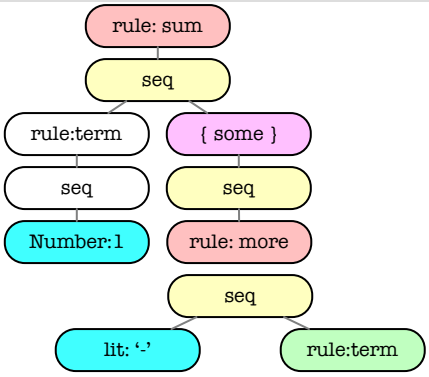
- Next toggle **lookahead** and press **parse** to watch how syntax analysis, i.e., the `parse()`[2] method of the `Parser`[2] object and the `parse()`[2] methods in the rule tree, consume the tuples — one up front, and a next one whenever syntax analysis reaches the current terminal:

```
> g.config.lookahead = true
> g.parser().parse(program)
parser lookahead: (1) "1" Number
Number lookahead: (1) '-'
'-' lookahead: (1) '('
'(' lookahead: (1) "2" Number
Number lookahead: (1) '+'
'+' lookahead: (1) '-'
'-' lookahead: (1) "3" Number
Number lookahead: (1) ')'
')' lookahead: end of input
```

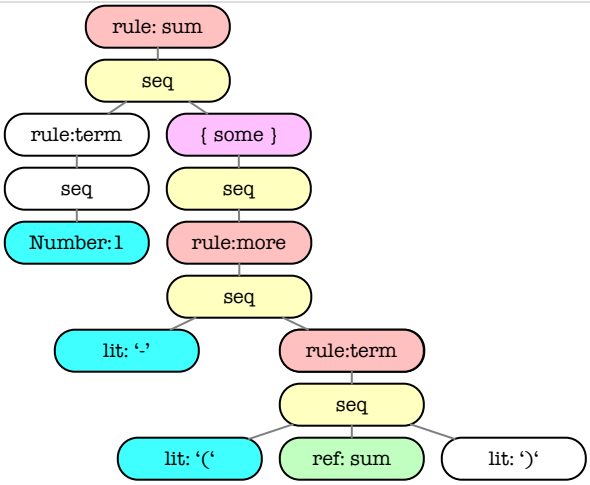
Back in the `sum`, after `term` has found `Number 1`, the sequence advances to the next element which requires finding `more` one or more times:



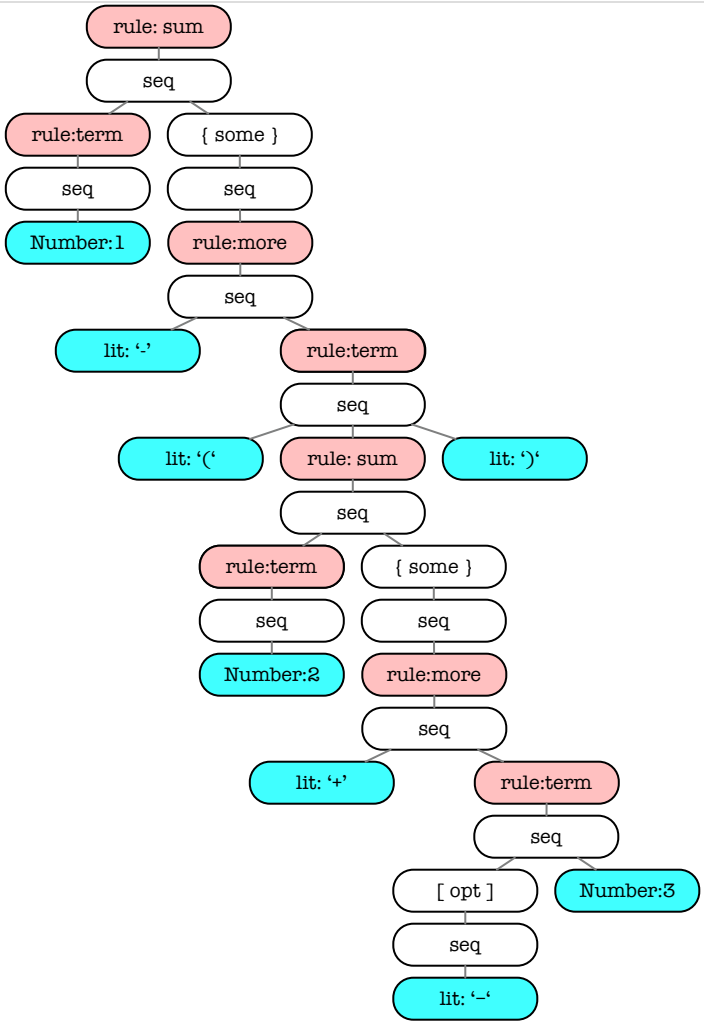
While there are two alternatives, the next input, literal `'-'`, only fits the second alternative and we get a little further in building the parse tree:



Syntax analysis continues, the `parse()` methods keep calling. The next input is literal `'('` which rule `term` can match with the second alternative:



Next, the sequence in `term` calls `sum` recursively before it will eventually match the trailing literal `')'`. Here is the result, where the rule calls and the terminals have been colored for emphasis:



The leaf nodes are all terminals, i.e., literals or tokens, and they match the proposed sentence

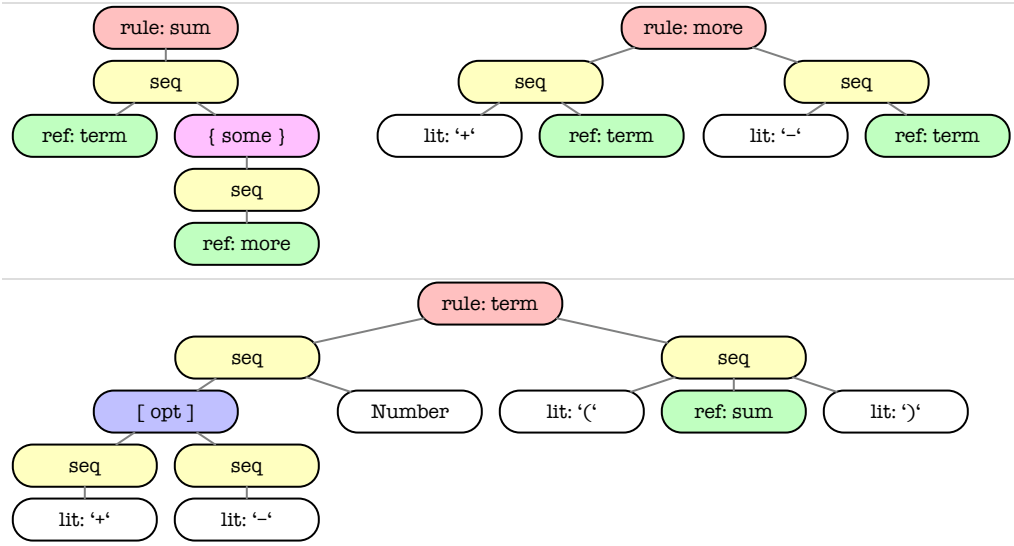
```
1 - (2 + - 3)
```

in order, i.e., we have found a parse tree and confirmed that the input is a sentence!

Strictly speaking, the diagram only shows that the input is a sequence of terminals (turquoise nodes) which are the leaves of an ordered tree. The definition of a syntax tree in chapter one requires that each branch node is labeled with a rule name and that the rule name and the sequence of subtrees (branch nodes and leaves) correspond to an ordered pair among the rules of the grammar. Chapter two extended the grammar notation and the extensions account for the white nodes in the diagram — sequences and the iteration constructs with brackets `[ opt ]` and braces `{ some }`. Still, the red and white branch nodes interpreted together as extended notation must appear in the grammar.

`Parser` and `parse()`

How do the `Parser`[2]'s `parse()`[2] method and the other `parse()`[2] methods work? The grammar diagrams in the preceding section showed that the job can be elegantly distributed among the nodes in the grammar trees for the rules `sum`, `more`, and `term`:



The colors suggest that the nodes belong to the following classes:

object	class	purpose
rule: sum	`Rule`[2]	represents a rule with a name and sequences as alternative descendants.
seq	`Seq`[2]	represents a sequence with terminals, rule references, braces, and brackets as descendants.
ref: term	`NT`[2]	represents a reference to a rule.
{ some }	`Some`[2]	represents braces with sequences as alternative descendants which are used one or more times.
lit: '+'	`Lit`[2]	represents a literal.
Number	`Token`[2]	represents a token.
[opt]	`Opt`[2]	represents brackets with sequences as alternative descendants which are optional.

For convenience there are two more classes:

- `Lit`[2] and `Token`[2] extend the class `T`[2] which represents terminals in general.
- `Rule`[2], `Some`[2], and `Opt`[2] extend the class `Alt`[2] which deals with alternative sequences in general.

We require that the branch nodes `Rule`[2], `Seq`[2], `Some`[2], and `Opt`[2], always have one or more descendants, and that `Seq`[2] cannot only have `Opt`[2] as descendants. This means that every `parse()` method will eventually find *something*.

Syntax analysis is implemented as `parse()` methods in six of the nine classes, the remaining three, `Opt`[2], `Lit`[2], and `Token`[2], simply inherit. The methods share a `Parser`[2] object which owns the input tuples; the `next()`[2] method is called to move on to the next tuple and it can trace lookahead. The actual implementation of syntax analysis is fairly simple:

- The `Grammar`[2]'s `parser()`[2] method creates the `Parser`[2] object and it's `parse()`[2] method is the function which will be called with the input string to create the tuples, check that the start rule can handle the first tuple, and call the `parse()`[2] method for the start rule.
- `parse()` for a `Rule`[2] delegates to the superclass — chapter five explains what else the method can do to create a convenient representation of the sentence.
- The superclass `Alt`[2] delegates to the particular descendant which can actually deal with the current tuple. Such a descendant exists — otherwise the method calls would not have reached this point.
- `parse()` for a `Some`[2], i.e., for braces, delegates to the superclass `Alt`[2] until there is no longer a descendant which can deal with the current tuple — the next section explains how this is determined.
- `parse()` for a `Seq`[2] delegates to one descendant after another — after first checking that the descendant can handle the current tuple and throwing an error if not — with one exception: a descendant `Opt`[2] is skipped if it cannot deal with the current tuple.
- `parse()` for a non-terminal[2] delegates to the referenced rule.
- Finally, `parse()` for literals and tokens[2] calls the `next()`[2] method to move on to the next tuple.

Yes, all of this is just a lot of delegating the work — with the requirement to "check before you call." `Alt`[2] and `Seq`[2] perform some traffic control and literals and tokens[2] nudge the scanner when they are reached.

The process can be observed in detail in example 4/01[1]:

- represent and check the grammar by pressing **new grammar**.
- Toggle **lookahead** and **parser** and press **parse** to watch how syntax analysis proceeds:

```
> g.config.lookahead = true
> g.config.parse = true
> g.parser().parse(program)
parser lookahead: (1) "1" Number
sum| Rule parse {
sum| super(sum: term [ { more } ];) parse {
sum| Seq(term [ { more } ]) parse {
sum| NT(term) parse {
  term| Rule parse {
  term| super(term: [ '+' | '-' ] Number | '(' sum ')';) parse {
  term| Seq([ '+' | '-' ] Number) parse {
  term| Token(Number) parse {
Number lookahead: (1) '-'
  term| Token(Number) parse }: '1'
  term| Seq([ '+' | '-' ] Number) parse }: [ null '1' ]
  term| super(term: [ '+' | '-' ] Number | '(' sum ')';) parse }: [ null '1' ]
  term| Rule parse }: [ null '1' ]
sum| NT(term) parse }: [ null '1' ]
```

Each line of output is prefixed with the rule name which has been called, nested rule calls are indented. Each line contains the class name of the node in the rule tree — or `super` for a call to the superclass, e.g., `Rule` to `Alt` (line 6) — and the rule fragment corresponding to the node. If a line ends with `{` it traces a call, otherwise it ends with `}:` and the return value — which will be explained in chapter five.

The above shows in complete detail how the function created by the `parser()`[2] method calls on `sum` (line 5) which calls on `term` (line 9) to recognize the `Number 1` (line 12). In particular, lines 12 to 14 above show that `Token`[2] advances to the next tuple.

expect

"Check before you call" seems like excellent advice, but just exactly what needs to be checked?

The `parse()` methods described in the previous section could be implemented in an error-tolerant fashion so that every alternative could be tried before syntax analysis throws the towel when it really cannot determine that some input is in fact a sentence. This works if there is no left recursion, but it is horribly inefficient, and it would usually require to move backwards in the input in order to try some other alternative.

Instead, each rule tree object contains a `Set`[2], called `expect`, of terminals, one of which the object wants to see as next tuple — the *lookahead* — when the object's `parse()` method is called. This can be seen in the output above and demonstrated in example 4/01[1] as suggested above.

Initially, the function created by the `parser()`[2] method sets the first lookahead up, and once the shared `parse()`[2] method for `Lit`[2] and `Token`[2] is reached it calls the `next()`[2] method to move on to the next lookahead.

"Check before you call" only allows to call `parse()` on an object if the current lookahead is in the object's `expect` set.

Obviously, terminals expect themselves, i.e., `expect` for literals and tokens are singleton sets which can be defined during construction. What about the other objects in the rule tree?

Naively

- a sequence should get `expect` from its first descendant,
- `expect` sets from alternative sequences should be merged,
- rules, braces and brackets, i.e., `Rule`[2], `Some`[2] and `Opt`[2], let the superclass `Alt`[2] of alternative sequences worry about `expect`,

and we are done?

Almost. Rule references, i.e., `NT`[2], have to take `expect` from their rules, and there things can get recursive... Therefore, computing `expect` takes two algorithms:

- `shallow` is applied to all rules, one after another, and proceeds along a sequence only far enough to determine `expect` for the sequence. It will detect (and fail for) left recursion.
- `deep` is applied to the start rule, down to all descendants, and along each sequence to the end, to determine `expect` for all nodes in the entire rule tree. `deep` cannot fail after `shallow` succeeded, and it detects rules that cannot be reached from the start rule.

All objects representing rules have `expect` sets. Therefore, all classes representing rules define `expect` and two methods `shallow()`[2] and `deep()`[2].

Representing `expect` is easy: all literal names are strings, single-quoted, and different, all token

names are strings, not quoted, and different. `expect` is a JavaScript map from a name to `true` indicating presence in the set. For convenience, this is encapsulated in a dedicated class `Set`[2] because the built-in class `Set`[†] currently does not provide the operations needed here.

The rule tree classes implement `shallow()` as follows:

class	shallow()
`Lit`[2] `Token`[2]	set `expect` to a singleton set with the terminal's name.
`Alt`[2]	delegates to all descendant sequences and merges their `expect` sets.
`NT`[2]	delegates to `Rule`[2] and detects undefined rules.
`Rule`[2]	marks, avoids, and complains about recursion, delegates to superclass `Alt`[2].
`Seq`[2]	proceeds from left to right merging descendant's `expect` until the descendant is not `Opt`[2]; note that we require that there will be such a descendant.

The remaining classes `Opt`[2] and `Some`[2] inherit from `Alt`[2].

The rule tree classes implement `deep()` as follows:

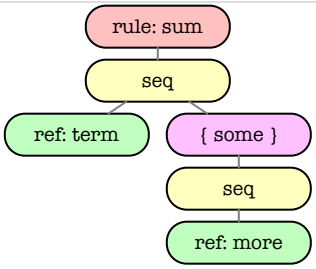
class	deep()
`Alt`[2]	needs to delegate and merge again to get to the end of all sequences.
`NT`[2]	delegates to `Rule`[2] again and might detect more undefined rules.
`Rule`[2]	marks a rule as reached from the start rule, delegates to superclass `Alt`[2] again to finish all sequences; recursion cannot happen.
`Seq`[2]	proceeds completely from *right to left* merging descendant's `expect` while the descendant is `Opt`[2]; the last `expect` is set for the sequence.

For `Lit`[2] and `Token`[2] there is nothing to do. `Opt`[2] and `Some`[2] again inherit from `Alt`[2].

The effects of `shallow()` and `deep()` can be seen in example 4/01[1]:

- Toggle **shallow** or **deep** and press **new grammar** to watch how each algorithm proceeds. Note that the algorithms are only applied once — during grammar checking.

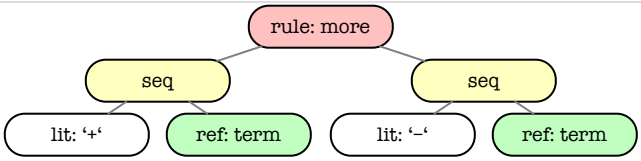
`shallow()` processes each rule, starting with `sum`:



The (shortened) output shows that `term` has to be analyzed before the result for `sum` is known:

```
> g = new EBNF.Grammar(grammar, tokens, { shallow: true })
> g.check()
sum| Rule shallow {
sum| super(sum: term [ { more } ];) shallow {
sum| Seq(term [ { more } ]) shallow {
sum| NT(term) shallow {
  term| Rule shallow {
    ...
  term| Rule shallow }: '+', '-', Number, '('
sum| NT(term) shallow }: '+', '-', Number, '('
sum| Seq(term [ { more } ]) shallow }: '+', '-', Number, '('
sum| super(sum: term [ { more } ];) shallow }: '+', '-', Number, '('
sum| Rule shallow }: '+', '-', Number, '('
```

As before, each line of output is prefixed with the rule name which has been called, nested rule calls are indented. Each line contains the class name of the node in the rule tree — or `super` for a call to the superclass, e.g., `Rule` to `Alt` (line 4) — and the rule fragment corresponding to the node. If a line ends with `{` it traces a call, otherwise it ends with `}:` and the elements of the resulting set.



The output shows that two alternative sequences have to be merged for `more`(line 11 below):

```
more| Rule shallow {
more| super(more: '+' term | '-' term;) shallow {
more| Seq('+' term) shallow {
more| Lit('+') shallow {
more| Lit('+') shallow }: '+'
more| Seq('+' term) shallow }: '+'
more| Seq('-' term) shallow {
more| Lit('-') shallow {
more| Lit('-') shallow }: '-'
more| Seq('-' term) shallow }: '-'
more| super(more: '+' term | '-' term;) shallow }: '+', '-'
more| Rule shallow }: '+', '-'
```

The output for `sum` also shows that `deep()` is required to compute `expect` for the sequence in

`sum` hidden under the `Some` node at right. Here is the (shortened) output from `deep`:

```
1  > g = new EBNF.Grammar(grammar, tokens, { deep: true })
2  > g.check()
3  sum| Rule deep {
4  sum| super(sum: term [ { more } ];) deep {
5  sum| Seq(term [ { more } ]) deep {
6  sum| Opt([ { more } ]) deep {
7  sum| Seq({ more }) deep {
8  sum| Some({ more }) deep {
9  sum| Seq(more) deep {
10 sum| NT(more) deep {
11   more| Rule deep {
12       ...
13   more| NT(term) deep {
14     term| Rule deep {
15       ...
16     term| Rule deep }: '+', '-', Number, '('
17   more| NT(term) deep }: '+', '-', Number, '('
18       ...
19   more| Rule deep }: '+', '-'
20 sum| NT(more) deep }: '+', '-'
21 sum| Seq(more) deep }: '+', '-'
22 sum| Some({ more }) deep }: '+', '-'
23 sum| Seq({ more }) deep }: '+', '-'
24 sum| Opt([ { more } ]) deep }: '+', '-'
25 sum| NT(term) deep {
26   term| Rule deep {
27   term| Rule deep }: '+', '-', Number, '('
28 sum| NT(term) deep }: '+', '-', Number, '('
29 sum| Seq(term [ { more } ]) deep }: '+', '-', Number, '('
30 sum| super(sum: term [ { more } ];) deep }: '+', '-', Number, '('
31 sum| Rule deep }: '+', '-', Number, '('
```

The computation happens in lines 7 to 22 above. In a Seq[2], `deep()` proceeds from right to left, i.e., in this case `more` is entered in line 10 before `sum` references `term`.

However, `more` depends on `term`, i.e., `term` will be processed in lines 14 to 16 before `more` is done in line 19. Later, when `sum` needs `term` in line 25, `deep()` does not have to enter the rule a second time.

Of course, `deep()` does not change `expect` for any of the rules themselves — that has been computed by `shallow()`.

Ambiguity

Clearly, `expect` is essential for the "check before you call" policy of calling a `parse()` method only if the callee expects the next input symbol.

Unfortunately, this is not the whole story as example 4/02[1] demonstrates. Here, the grammar has a new start rule `sums` which should allow input to consist of more than one sum of signed

numbers:

```
sums: { sum };
sum: term [{ more }];
more: '+' term | '-' term;
term: [ '+' | '-' ] Number | '(' sum ')';
```

The following input contains three sums:

```
1 - (2 + - 3)
4 + 5
- 6 - 7
```

It looks as everything works just fine:

- Toggle **no check** to avoid the ambiguity check and represent the grammar by pressing **new grammar**.
- Then toggle **lookahead** and **parser** and press **parse** to watch how syntax analysis proceeds.

There is voluminous output and there are no error messages. Chapter five will explain why the output contains so many brackets.

However, upon closer inspection (and considerable pruning) the output reveals a surprise:

```
1  > g.config.lookahead = true
2  > g.config.parse = true
3  > g.parser().parse(program)
4  parser lookahead: (1) "1" Number
5  sums| Rule parse {
6          ...
7    sum| Rule parse {
8          ...
9  ')' lookahead: (2) "4" Number
10         ...
11   sum| Rule parse }: [ [ null '1' ] [ [ [ [ '-' [ '('
12          [ [ null '2' ] [ [ [ [ '+' [ [ '-' ] '3' ] ] ] ] ] ]
13          ')' ] ] ] ] ] ]
14         ...
15   sum| Rule parse {
16         ...
17     more| Rule parse {
18         ...
19       term| Token(Number) parse {
20 Number lookahead: (3) '-'
21       term| Token(Number) parse }: '5'
22         ...
23     more| Rule parse }: [ '+' [ null '5' ] ]
```

`1 - (2 + - 3)`, the first `sum`, is recognized as before (line 11), and `4 + 5`, the second `sum`, is found as well. Unfortunately, `more` is happy to see `'-'` as a lookahead and continues to gobble up `- 6 -`

7 as part of a longer second sum:

```
      ...
sum| Rule parse }: [ [ null '4' ] [ [ [ [ '+' [ null '5' ] ] ]
      [ [ '-' [ null '6' ] ] ] [ [ '-' [ null '7' ] ] ] ] ] ]
```

Surprise: expect is consulted before descendants of Some[2] or Opt[2] are asked to parse(), but the process is greedy, i.e., whatever follows in a sequence containing Some[2] or Opt[2] only gets a chance if those two let go.

A grammar is called *ambiguous* if different parse trees can be constructed for the same sentence.

In this example, the rule sum could succeed three times

```
1 - (2 + - 3)
4 + 5
- 6 - 7
```

as the three lines of input might suggest. However, the three lines are just one long string — white space is skipped by lexical analysis — and the output above shows that sum can recognize 4 + 5 - 6 - 7 as a longer second sum. Overall, there are two different parse trees, with two or three top-level rule:sum nodes.

This grammar is ambiguous — a typical issue when two arithmetic expressions with signed numbers can follow each other. It is easy to repair, check out example 4/03[1].

new grammar will complain about ambiguities — unless **no check** suppresses the follow() and check() algorithms which normally are part of preparing the grammar.

We have ambiguity issues

- if two or more alternative sequences have overlapping expect sets, or
- if braces, i.e., a Some[2] object, and their successor in a sequence have overlapping expect sets, or
- if brackets, i.e., an Opt[2] object, and their successor in a sequence have overlapping expect sets.

Another ambiguity issue would arise if a sequence had a way to avoid getting to any terminal — in this case the expect of the sequence could overlap with whatever can follow the sequence. This is why our grammar definitions do not permit a sequence to only consist of brackets.

In all these cases "check before you call" has a choice within the overlaps, and blind trust in the closest expect set is not advisable.

follow()

How do we check for these ambiguities?

Overlapping expect sets among alternative sequences are easy to find, but the other cases involve

two successive items in a sequence. Unfortunately, already in example 4/02[1] of multiple sums

```
sums: { sum };
sum: term [{ more }];
more: '+' term | '-' term;
term: [ '+' | '-' ] Number | '(' sum ')';
```

it is not obvious which `expect` sets overlap.

Rather than hunting for successive items, the `follow()` algorithm computes an individual `follow` set for each object in the grammar rules. The set contains all terminals which can directly follow in the input after whatever the object itself recognizes during syntax analysis. Given the `follow` set, we can reason about ambiguities on a per-object basis.

Just like `shallow()` and `deep()`, the `follow()` algorithm is implemented as a method in the various classes which are used to represent grammar rules. The method is called for an object with the `Set`[2] of all terminals that can follow the object as a parameter.

The first step is that nothing can follow the start rule — at least at the top level. Therefore we call `follow()` for the start rule with an empty set to get things started.

class	follow(set)
`Alt`[2]	sends the incoming set to all descendant sequences because the set can follow each descendant.
`Some`[2]	adds its own `expect` set to the incoming set and sends the result to the descendant sequences because each descendant can be repeated and thus follow any other.
`Seq`[2]	processes all descendants from right to left: it sends the current set to a descendant and sends its `expect` to the next one — moving from the end to the beginning of the sequence. If the current descendant is an `Opt`[2], its `expect` is merged with the current set and sent to the predecessor because `Opt`[2]'s `expect` and `Opt`[2]'s `follow`, i.e., the current set, both, can follow the predecessor.
`NT`[2]	adds the parameter to the referenced rule, but only if that changes something.

`Rule`[2] and `Opt`[2] inherit from their superclass `Alt`[2].

There is nothing to do for `Lit`[2] and `Token`[2] because they don't require checking.

`NT`[2] is tricky: obviously it has to send the parameter to the referenced `Rule`[2] because all the terminals in the incoming set can follow the rule. But at this point, e.g., the start rule might find out that there are more terminals which can follow and the entire process has to start over. Therefore, `NT`[2] will only send the parameter to the referenced rule if the parameter really is bigger than the current `follow` of the referenced rule. This prevents infinite recursion because all sets can at most contain the finite number of terminals which the grammar has.

It should be noted that there is a more efficient way to compute `follow`: If the immediate `follow` relationship between any two items in all rules is noted in a matrix, Warshall's algorithm‡ is an efficient way to compute the "infinite" product of that matrix with itself and the resulting matrix describes the so-called *transitive closure* of the `follow` relation, which is just what we need. Brackets and braces, however, complicate the specification of the first matrix.

check()

The `expect` sets enable syntax analysis to "check before you call" and avoid trial and error.

The `follow` sets are used once during grammar preparation to ensure that "check before you call" always gets a unique answer, i.e., that syntax analysis cannot get greedy and must produce a unique parse tree.

A context-free grammar is called *LL(1)*‡ if syntax analysis can be performed by processing input left to right, looking for the left-most derivation, i.e., starting with the start rule of the grammar, and with one terminal lookahead. The approach is also called *top-down* because it starts with the root of the syntax tree, i.e., the start rule of the grammar.

The following `check()` determines if a grammar is LL(1)‡. It is based on the `expect` and `follow` sets and is again distributed over the classes for the objects representing the grammar rules.

class	check()
`Seq`[2]	each descendant must be checked.
`Alt`[2]	the alternatives' `expect` sets must not overlap.
`Opt`[2]	`expect` and `follow` must not overlap.
`Some`[2]	`expect` and `follow` must not overlap.
`NT`[2]	does nothing(!) because `check()` is applied to every rule, not just recursively to the start rule.

In `Alt`[2], `check()` is very simple to implement: an `Alt`[2] node has its own `expect` set which is the union of all alternatives. Therefore, the sum of the number of elements in the alternatives' `expect` sets and the number of elements in the `Alt`[2] node's own `expect` set must be the same.

Ambiguity Revisited

An ambiguous grammar might still be useful — as long as it recognizes only what is intended. Example 4/04[1] illustrates an `if` statement:

```
1  stmt: Text | if;
2  if: 'if' Text 'then' stmt [ 'else' stmt ] 'fi';
```

This grammar is not ambiguous and it recognizes for example the following program:

```
1  if a then
2    if b then c
3    else d fi
4  fi
```

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis.

The output can be reformatted

```
[ [ 'if' 'a' 'then' [
    [ 'if' 'b' 'then' [ 'c' ] [ 'else' [ 'd' ] ] 'fi' ]
  ] null 'fi' ] ]
```

and shows that `else d` is recognized as part of the inner `if` statement.

It turns out that this is due to the fact that in the grammar above `'if'` and `'fi'` are balanced.

Example 4/05[1] illustrates a more typical `if` statement without `fi`:

```
if a then
  if b then c
  else d
```

This grammar is reported to be ambiguous:

- Press **new grammar** to represent and check the grammar.
- Toggle **no check** to suppress ambiguity checking,
- press **new grammar** to represent and check the grammar again, and
- press **parse** to perform syntax analysis.

Again, the output can be reformatted for clarity

```
[ [ 'if' 'a' 'then' [
    [ 'if' 'b' 'then' [ 'c' ] [ 'else' [ 'd' ] ] ]
  ] null ] ]
```

Obviously, the output no longer contains the literal `'fi'` but it is otherwise unchanged!

Recognition is greedy, i.e., if it can accept an `else` because there is an `if` to be continued it will not check if there might be an outer `if`. Fortunately, this is how programming languages like to interpret `else` clauses, and in this particular case an ambiguous grammar does the right thing.

Quick Summary

- Syntax analysis receives terminals from lexical analysis and determines if the input is a sentence conforming to a grammar.
- Grammar rules can be viewed as functions performing recognition, and syntax analysis starts "top-down" by calling the start rule of the grammar.
- A grammar rule can be represented as a tree of nodes, i.e., objects of a few classes such as `Rule`[2].
- Algorithms, such as syntax analysis, are distributed as methods over these classes, such as `parse()`[2].
- Top-down, deterministic syntax analysis is based on the principle of "check before you call": `parse()` can only be called on a node if the *lookahead*, i.e., the next input symbol, matches what the node expects.

- "check before you call" is possible if a grammar is *LL(1)*‡ (and therefore not *ambiguous*). This needs to be checked once when the rules are represented as trees.
- The algorithms `shallow()` and `deep()` compute for each tree node the set `expect` of terminals which can be in the lookahead.
- The algorithm `follow()` computes for each tree node the set of terminals that can follow the input that the node recognizes.
- The algorithm `check()` compares these sets for each tree node and determines if a grammar is LL(1)‡ and can be used for this style of syntax analysis.
- Ambiguity is not always a bad thing; however, recognition based on an ambiguous grammar would have to be very carefully tested before it can be accepted for a project.
- Wherever classes and methods are mentioned in this book they are linked to the documentation and from there to the (syntax-colored) source lines.

Programming Note: `Set`²

Implementing specific set operations for LL(1) checking and syntax analysis in JavaScript (rather than using the built-in `Set`† class) turns out to be very easy.

A `Set`² is represented as an object with terminal names as properties which have an arbitrary value, e.g., `true`, just to denote their presence in the set. Note that token and literal names cannot overlap because literal names are single-quoted and tokens are not.

- `match()`² uses the operator `in`† to check if a terminal belongs to a set.
- `Object.assign`† imports the elements of a second set².
- `Object.keys`† returns the terminal names as an array, e.g., to count how many belong to the set.
- `every`† can be used to check if one set contains another²:

```
1  Object.keys(other.set).every(key => key in this.set)
```

- `reduce`† can be used to create a set of elements belonging to two sets²:

```
1  Object.keys(this.set).reduce((result, key) => {
2    if (key in other.set) result.set[key] = true;
3    return result;
4  }, new Set());
```

One word of caution: these sets are objects and, therefore, passed by reference. A new set has to be constructed, e.g., when a `Some`² node adds its own `expect` and `follow` sets to send them to the descendants...

Programming Note: `trace()`²

A prudent approach to programming is to assume a priori that something will go wrong and to always instrument code so that algorithms can be observed. Unfortunately, this is likely to hide the actual algorithm behind obscure scaffolding which should not be there for production runs,

e.g.,

```
if (debug) console.debug('some label', 'about to compute');
  // compute something here
if (debug) console.debug('some label', 'something to see');
```

In the object-oriented approach to syntax analysis described here, algorithms such as `parse()`, `shallow()`, `deep()`, `follow()`, and `check()` are distributed as methods of the classes involved in representing grammar rules.

The job of each method is quite small but the methods call each other recursively across the classes and the return values get more and more complicated. Example 4/01[1] illustrates how instructive it is to see method calls and return values.

- Toggle some of **shallow**, **deep**, and **follow**, then press **new grammar** and watch how the `expect` and `follow` sets are put together.
- Next toggle **parser**, then press **parse** and watch how the `parse()`[2] methods call on each other.

Tracing function calls and return values amounts to function composition:

- the algorithm is carried out by a method,
- a *tracer proxy* announces the method call and arguments, calls the algorithm method, reports the end of the method call and the result values, and returns the method's results.

Methods are functions stored in a class' prototype, i.e., a method can be cached and replaced by a tracer proxy which informs about and internally calls the cached method, e.g.

```
const cache = class_.prototype.method;          // cache
class_.prototype.method = function (...arg) {   // proxy
  console.debug('about to call', class_.name, cache.name);
  const result = cache.call(this, ...arg);
  console.debug('result:', result);
  return result;
}
```

`trace()`[2] manages caching and reporting for the essential algorithms of syntax analysis.

5. Translating Sentences

What's in a Parse Tree? What's with the Brackets?

Chapter four explained that syntax analysis tries to build a parse tree to show that a sequence of input terminals is a sentence conforming to a grammar.

Output from the examples suggests that, indeed, something gets built, and it contains strings related to the input terminals. However, the output seems to consist mostly of brackets, and it is very hard to see that the output models the parse tree as very deeply nested JavaScript lists.

This chapter explains how the lists come about and how syntax analysis can be augmented with `Action`[2] methods to translate input into something more convenient.

Lists

Trace output from example 5/01[1] demonstrates that the `parse()`[2] method of the `Parser`[2] object generated by the grammar's `parser()`[2] method collects a list containing the input strings from the `program` area which correspond to the terminals in the grammar rule.

Here is a sequence of eight examples, 5/01 through 5/08, which illustrate how rules, brackets, and braces collaborate to create nested lists. Each time

- press **new grammar** to represent and check the grammar;
- toggle **parser** and press **parse** to watch how syntax analysis builds the nested lists.

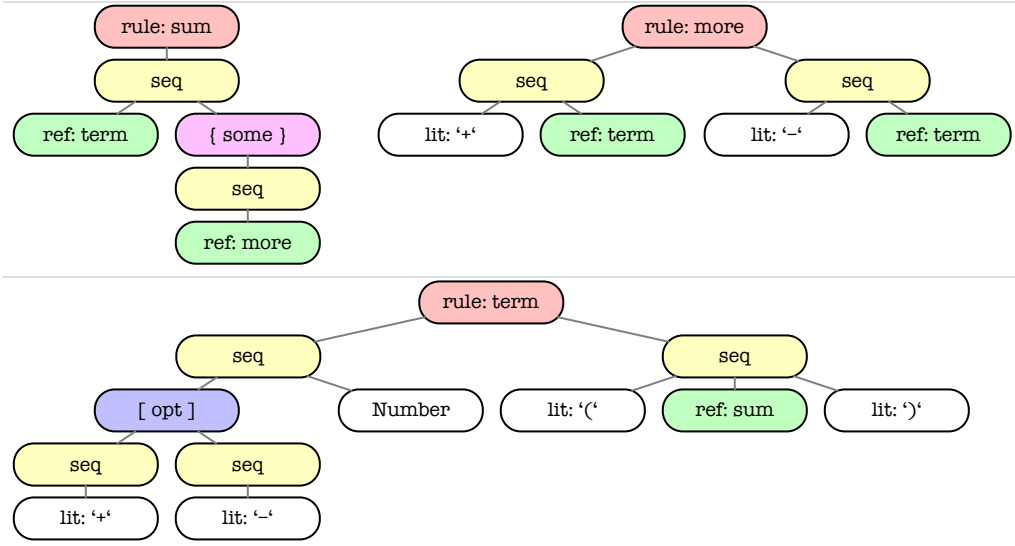
grammar	output	example
`sum: Number '+' Number;`	`[1 + 2]`	01: `sum` creates a list[1]
`sum: term '+' Number;` `term: Number;`	`[[1] + 2]`	02: `term` creates a list[1]
`sum: term '+' term;` `term: Number;`	`[[1] + [2]]`	03: `term` creates lists[1]
`sum: Number [ '+' Number ];`	`[1 null]` `[1 [+ 2]]`	04: brackets create `null` or a list[1]
`sum: term [ '+' term ];` `term: Number;`	`[[1] null]` `[[1] [+ [2]]]`	05: `term` creates lists[1]
`sum: Number { '+' Number };`	`[1 [[+ 2]]]` `[1 [[+ 2] [+ 3]]]`	06: braces create list of lists, one inner list per iteration[1]
`sum: term { '+' term };` `term: Number;`	`[[1] [[+ [2]]]]`	07: `term` creates lists[1]

`sum: Number { more };`
`more: '+' term;`
`term: Number;`
	`[1 [[[+ [2]]]]]`	08: `more` creates an extra list[1]

Terminals produce strings in the nested lists. In the actual output on the practice page… this is emphasized because strings are quoted there; in the **output** column above the quotes had to be omitted to save space.

In summary:

- The `parse()` algorithm calls the `parse()` methods for objects selected during a top-down traversal of the rule trees of the grammar. By default, most of these methods produce lists.



- The fundamental branch node in a rule tree is a `Seq`[2] object which always produces a list containing the values which the branch node's subtrees produce.
- Alternatives show up in a rule tree as branch nodes represented as `Rule`[2], `Opt`[2], and `Some`[2] objects.
- A `Rule`[2] selects one of it's `Seq`[2] descendants which produces a list for the rule's alternative matching the input.
- The list will contain exactly one entry for each item (terminal, rule reference, brackets, or braces) in the selected sequence.
- Brackets are represented as an `Opt`[2] object in the rule tree which again has `Seq`[2] descendants for the alternatives.
- If an alternative of `Opt`[2] (i.e., brackets) matches the input, it's `Seq`[2] will produce a list, otherwise the brackets themselves produce `null`.
- Braces are represented as a `Some`[2] object in the rule tree which again has `Seq`[2] descendants for the alternatives.
- `Some`[2] will match at least one alternative, but it might be able to match several times in a row — the same or a different alternative.
- Therefore, `Some`[2] (i.e., braces) will return a list of one or more inner lists, one for each iteration. Each inner list corresponds to one matched alternative and it is produced by the `Seq`[2] object representing the alternative in the rule tree.

The `Parser`'s `parse()`[2] method returns the nest of lists produced by the rule tree while syntax analysis recognizes a sentence. List nesting is completely under control of the grammar and the

list nesting principles described above are uniform and relatively simple; however, as the arithmetic expressions in example 2/12[1] or a simplified version just for sums in example 4/03[1] showed, while this result of syntax analysis is well-formed it is usually so overwhelming that it hardly invites further processing.

Action Methods

Fortunately, there is a way to interact with list creation whenever a rule completes recognition.

As discussed in chapter four, syntax analysis is performed by the `Parser`'s `parse()`[2] method which calls on the `parse()` methods of the various classes involved in representing a grammar as a rule tree.

Each grammar rule can be considered to be a function which has to recognize a part of the input which is described and structured according to the rule's right-hand side.

This function is the `parse()`[2] method associated with the `Rule`[2] object representing the rule in the grammar's rule tree. As described above, by default this `parse()`[2] method creates a list containing whatever the descendants' `parse()` methods return. The "check before you call" policy ensures that there is no backing up and re-creating — recognition cannot be undone.

The real power of considering a grammar rule as a function stems from the fact that each rule can be associated with an `Action`[2] method written in JavaScript

- which is called when the `parse()`[2] method is done collecting,
- which receives the list collected by `parse()`[2] as individual arguments, and
- which returns a value which `parse()`[2] returns in place of the collected list.

Thus far we have not specified any `Action`[2] methods — this is why we have seen deeply nested lists as results of syntax analysis.

The `Parser`'s `parse()`[2] method accepts an optional argument:

- a (singleton) object with methods names equal to some rule names, or
- a class which the parser instantiates to create such an object; the constructor is called with a reference to the parser to facilitate error reporting.

Example 5/09[1] contains different `Action`[2] methods to manipulate a sequence of non-whitespace characters using the following grammar:

```
chars: { char };
char: Char;
```

For example, the `Action`[2] methods

```
char (ch) { return ch; }
chars (some) {
  return some.reduceRight((rev, list) => rev += list[0], '');
}
```

reverse the order of the characters of the palindrome†

```
.ein.Esel.lese.nie!
```

- Press **new grammar** to represent and check the grammar, and
- press **parse** to see the output.
- Uncomment one action for `char` and/or one for `chars` in the `actions` area to see how the output changes.
- Toggle **parser** and/or **actions** and press **parse** to watch how input is found and converted by the `Action`[2] methods. In particular, if there is no `Action`[2] method for `char` the trace shows that the rule returns a list containing a single character which the `Action`[2] method for `chars` concatenates to a string...
- Erase either the methods or everything in the `actions` area and press **parse** to see that without action methods the lists are generated.

Example 5/10[1] with the same grammar shows how a property in the class can be used to communicate among the `Action`[2] methods:

```
class Actions {
  #queue = [];

  char (ch) { this.#queue.unshift(ch); }
  chars () { return this.#queue.join(''); }
}
```

Initially, `this.#queue` is an empty array.

Syntax analysis proceeds from the start rule of the grammar down to the terminal nodes which collect the input terminals from left to right. `unshift()`† stuffs arguments in front of a target array and returns the new length (which is not used in this case). Because of `unshift()` the action method for `char` stuffs the non-whitespace input characters into the array in reverse order.

The action method for the start rule `chars` returns the collected array elements as a string.

The following chapters illustrate with a few larger examples what `Action`[2] methods can do: we interpret arithmetic and compile a little language in chapter six, we perform some type checking and add functions and block structure in chapter seven, and we implement first-order functions‡ in chapter eight. Finally, we create a parser generator in chapter nine which implements lexical and syntax analysis for grammars — and which we have used all along.

Error Checking

Rules and `Action`[2] methods are very loosely coupled but by default the parser tries to check the number of values collected by `parse()`[2] against the number of parameters defined for the action method:

- The test is omitted if the action method has no parameters.
- Rest parameters† affect the count of expected parameters; in particular, the test is omitted if the entire parameter list is one rest parameter.
- The test can be suppressed by setting **no args**[2].

Idioms for Actions

Each rule presents structured data to it's `Action`[2] method. The `Action`[2] method for the rule will receive one argument for each top-level item in the rule. Consider the grammar in example 5/11[1]:

```
rule: 'literal' Number ref;
ref: Number;
```

There are two ways to write the parameter list for the `Action`[2] method, namely using rest parameter syntax† or using individual parameter names.

```
rule (...val) { /* ... */ }
```

With the rest parameter syntax† the array `val` will contain three values because there are three items in `rule`:

- `val[0]` is the string value `'literal'` itself,
- `parseInt(val[1], 10)`† is the decimal number computed from the string matched by the token `Number`, and
- `val[2]` contains the value returned by the `Action`[2] method, if any, for the referenced rule `ref`, or it contains the list of arguments which would have been presented to the referenced rule's `Action`[2] method.

A more convenient parameter list for an `Action`[2] method for `rule` uses different parameter names for the different argument values:

```
rule (literal, number, ref) { /* ... */ }
```

and in this case

- `literal` is the string value `literal` itself,
- `parseInt(number, 10)`† is the converted `Number`, and
- `ref` contains the value returned by the `Action`[2] method, if any, for the referenced rule `ref`, or it contains the list of arguments which would have been presented to the referenced rule's `Action`[2] method.

Note that if there is no `Action`[2] method for a rule, the rule returns a list containing the collected values. Without an `Action`[2] method the rule `ref` above collects a string matched by the token `Number` and returns it in an array.

- Therefore, if there is no `Action`[2] method for `ref`, the `Action`[2] method for `rule` can compute the decimal value as `parseInt(ref[0], 10)`.

Consider the following `Action`[2] methods for `rule` and `ref`:

```
rule (literal, number, ref) { /* ... */ }
ref (number) { return parseInt(number, 10); }
```

Now the `Action`[2] method for `rule` receives the decimal value in the parameter `ref` because the rule for `ref` has an `Action`[2] method which returns that value.

Brackets and braces in a rule amount to unnamed rules which cannot have `Action`[2] methods, i.e., they contain alternatives which in turn contain sequences. If such a sequence matches input it collects a list. The following sections explain how to drill down to access the values.

`[ optional ]` — maybe

The following rule accepts the string `literal`, optionally followed by either `maybe` or `not`, and either of those followed by a number, check example 5/12[1]:

```
rule: 'literal' [ 'maybe' Number | 'not' '!' Number ];
```

With explicit parameter names — one per top-level item in the rule —

```
rule (literal, opt) { /* ... */ }
```

here is how to get to the pieces:

- `literal` will contain the string value `literal` and `opt` will contain a list.
- `opt[0]` will contain one of the strings `maybe` or `not`, and
- `opt[1]` or `opt[2]`, respectively, will contain the string value collected for the `Number`.

However, `opt` receives the result produced by one of the alternatives in the brackets, and that need not match any input — in which case `opt` is `null`.

Therefore, an `Action`[2] method to display the decimal values in the output area could be

```
class Actions {
  // rule: 'literal' [ 'maybe' Number | 'not' '!' Number ];
  rule (literal, opt) {
    if (opt)
      switch (opt[0]) {
      case 'maybe': puts(opt[0], parseInt(opt[1], 10)); break;
      case 'not':   puts(opt[0], parseInt(opt[2], 10));
      }
  }
}
```

To `switch` or not to `switch` ...

Lexical analysis partitions the input into terminals. Syntax analysis collects the terminals in rules. It is inefficient if `Action`[2] methods have to use `switch` or `if` to deal with terminal strings again.

Example 5/13[1] accepts the same input as example 5/12[1] but two rules have been added to

simplify the corresponding Action[2] methods:

```
class Actions {
  // rule: 'literal' [ maybe | not ];
  rule (literal, opt) { if (opt) puts(opt[0]); }

  // maybe: 'maybe' Number;   // returns Number value
  maybe (m, number) { return parseInt(number, 10); }

  // not: 'not' '!' Number;   // returns Number value
  not (n, exclamation, number) { return parseInt(number, 10); }
}
```

Brackets and braces can contain alternatives. It is often easier to make each alternative a separate rule to simplify the Action[2] methods and avoid inspecting terminals explicitly.

{ some } — at least once

Example 5/14[1] contains a typical grammar for a list of statements, each terminated by a semicolon:

```
list: { item ';' };
item: dec | hex;
dec: 'dec' Decimal;
hex: 'hex' ref;
ref: Hex;
```

`list` accepts one or more numbers, each is preceded by `dec` or `hex` and followed by a semicolon. There are tokens for decimal and hexadecimal numbers; note that the token patterns overlap — hence the prefixes in the rules:

```
{ Decimal: /[0-9]+/, Hex: /[0-9a-fA-F]+/ }
```

- Press **new grammar** to represent and check the grammar,
- erase the `actions` area, and
- press **parse** to see the nested lists.
- Why does the input `hex0;` cause an error?

Without Action[2] methods the input

```
dec 10;
hexBad;
dec 20;
```

produces

```
1 [
2   [
3     [ [ [ 'dec' '10' ] ] ';' ]
4     [ [ [ 'hex' [ 'Bad' ] ] ] ';' ]
5     [ [ [ 'dec' '20' ] ] ';' ]
6   ]
7 ]
```

If there are no `Action`[2] methods the outermost list (lines 1 and 7) is produced by the sequence for `list`. The next list (lines 2 and 6) is produced by the braces (*some*) and contains one list for each iteration — the outer lists in lines 3 through 5. The inner lists in lines 3 and 5 are produced by the sequence for `dec` or `hex` and `item` adds one more list to these. `Bad` is in one more list, produced by the sequence for `ref` (line 4).

Trick question: Why are the blanks following `dec` in the program necessary, whereas `Bad` can immediately follow `hex`?

With explicit parameter names here is how to untangle a pair of braces using `Action`[2] methods:

```
1  class Actions14 {
2    /** `dec: 'dec' Decimal ';';` returns value */
3    dec (_, decimal) { return parseInt(decimal, 10); }
4
5    /** `hex: 'hex' ref ';';` returns value */
6    hex (_, ref) { return parseInt(ref[0], 16); }
7
8    /** `item: dec | hex;` returns `[ value ]`
9        `list: { item ';' };` displays value ... */
10   list (some) { puts(... some.map(list => list[0][0])); }
11 }
```

`dec()` and `hex()` decode the numbers. There is no need for an `Action`[2] method for `ref` if `hex()` extracts the input string from the list returned from the `ref` rule by default (line 6).

- Reload the example and press **new grammar** to represent and check the grammar.
- Toggle **actions**, and
- press **parse** to see how the numbers are extracted.

`list()` is now concerned with a list of lists which contain one number and a semicolon each. `map()`† is used to convert this nest into a simple list and the spread syntax† converts that into individual arguments for `puts()` to display (line 10).

`[{ many }]` — zero or more times

Example 5/15[1] is very similar, but for the fact that the statements are separated by commas.

```
1  list: item [{ ',' item }];
2  item: dec | hex;
3  dec: 'dec' Decimal;
4  hex: 'hex' ref;
5  ref: Hex;
```

The program

```
1  dec 10, hex Bad, dec 20
```

produces

```
1  > run = g.parser().parse(program, actions)
2  [ 10 2989 20 ]
```

With explicit parameter names here is how to untangle a comma-separated list of items, i.e., one `dec` or `hex` item followed by zero or more items, with a comma preceding each item but the first:

```
1  class Actions15 extends Five.Actions14 {
2    /** `item: dec | hex;` returns `[ value ]`
3        `list: { item ';' };` returns `[ value ... ]` */
4    list (item, many) {
5      return item.concat(many ? many[0].map(list => list[1][0]) : []);
6    }
7
8    // alternative solution
9    // list (item, many) {
10   //   return (many ? many[0] : []).reduce(
11   //     (result, list) => (result.push(list[1][0]), result), item);
12   // }
13 }
```

- Press **new grammar** to represent and check the grammar.
- Toggle **actions**, and
- press **parse** to see how the numbers are extracted.

`item` is defined by a separate rule (comment in line 2); without an `Action`[2] method for `item` the parameter `item` (line 4) will receive a list containing the first value.

This list is concatenated† either with a list of further values or with an empty list (which is discarded).

If the parameter `many` (line 4) is not `null`, it is a list containing the list of values for the alternatives created by the braces. `many[0]` is the inner list and `map()`† converts that into a list of values that can be concatenated with the first value.

The callback for `map()`† (line 5) each time receives a `list` produced by the sequence

```
1  ',' item
```

and `list[1][0]` drills down to the value in the second element of that list.

An alternative solution employs `reduce()`† (lines 9 to 12):

- If the parameter `many` is `null`, i.e., if there is only one `item`, `reduce()`† is applied to an empty list and returns the start value, i.e., the parameter `item`.
- Otherwise, each time the callback function receives a list, extracts a value, and appends that to the `item` parameter.

The result is the same; however, the solution with `concat()`† works on list lengths known a priori whereas the alternative solution dynamically extends the list passed as a parameter, i.e., one would opt for the solution with `concat()`†.

Example 5/16[1] has the same grammar, tokens, and program as example 5/15[1], but this time the `list` action returns the sum of the numbers, i.e., the list of items has to be processed into a single result:

```
1  class Actions16 extends Five.Actions14 {
2    // convoluted solution
3    // list (item, many) {
4    //   return item.concat(many ? many[0].map(list => list[1][0]) : []).
5    //     reduce((sum, value) => sum + value, 0);
6    // }
7
8    /** `item: dec | hex;` returns `[ value ]`
9        `list: [{ ',' item }];` returns `value +...` */
10   list (item, many) {
11     return (many ? many[0] : []).reduce(
12       (result, list) => result += list[1][0], item[0]);
13   }
14 }
```

A convoluted solution would use `map()`† as before to create a list of values (line 4) and follow up with `reduce()`† to produce the sum (line 5).

In this situation it is better to only apply `reduce()`†, either to the list of additional items or to an empty list (line 11), initialize with the first value and add the others on the fly (line 12). This avoids unneccesary dynamic list creation.

Extending Action Classes

The last three examples in this chapter share some grammar rules and the corresponding `Action`[2] methods: example 5/14[1] defines `item`, `dec`, `hex`, and `ref`, and examples 5/15[1] and 5/16[1] can just extend the action class defined for example 5/14[1] to inherit those methods and only overwrite the `Action`[2] method for `list` where the corresponding rule has been changed and/or a different algorithm should be applied.

The practice page supports this as follows:

- If class names end in digits, such as `Actions14` in example 5/14[1], the classes are exported from modules which are part of the environment provided by the practice page.
- The module names are global, e.g., `Five` for chapter five, so that, e.g., example 5/15[1] can inherit methods:

```
class Actions15 extends Five.Actions14 { ... }
```

It should be noted, however, that overwriting does not remove code. For example, `Actions15` contains it's own `list` method and `super.list()`, i.e., the method from `Actions14`, is still available, even if it is not used. Chapter eleven discusses how with careful planning mix-ins‡ can be used to avoid dead code.

The following chapters will rely on modules in order to focus on the `Action`[2] methods that are new or overwritten in a sequence of related examples. (Unfortunately, the practice page will not copy any other text — rules, tokens, or programs — from one example to another...)

Error Checking

A subclass inherits methods from the superclass and it can delegate to a method in the superclass using `super` in place of `this`.

Argument checking as described above applies implicitly if an action method is inherited, and it is explicitly applied if a call such as

```
super.method(arg1, .., argn)
```

is replaced by

```
this.parser.call(this, super.method, arg1, .., argn)
```

using the method `call()`[2] defined in the parser — assuming that the action object stores a reference to `parser` which is provided during construction.

Quick Summary

- The `Parser`'s `parse()`[2] method can be called with a class or a (singleton) object with `Action`[2] methods matched to the rule names in the grammar.
- Grammar rules plus their `Action`[2] methods, if any, are applied to the input. Calls proceed top-down, beginning with the start rule.
- Once a rule matches input an `Action`[2] method is selected which has the same name as the rule.
- The `Action`[2] method receives arguments corresponding to the alternative of the rule which matched the input, exactly one argument for each item (terminal, rule reference, brackets, or braces) in the alternative.
- Each set of brackets in the alternative produces either `null` or a list of values for the alternative in the brackets which matches the input.

- Each set of braces in the alternative produces a list which contains one inner list of values for each iteration.
- The iteration itself produces the (inner) list for the alternative in the braces which matches input during this iteration.
- The result value of the `Action`[2] method is received by the parent rule's `Action`[2] method or returned by the `Parser`'s `parse()`[2] method.
- If there is no `Action`[2] function the result value is a list of all argument values which the `Action`[2] method would have received.

6. Compiling Little Languages

Interpret Now, Compile for Later
Functions and a Stack Machine

Chapter two discussed how to write grammars and example 2/12[1] presented a typical grammar for a comma-separated list of arithmetic expressions:

```
1  list: sum [{ ',' sum }];
2  sum: product [{ '+' product | '-' product }];
3  product: signed [{ '*' signed | '/' signed }];
4  signed: [ '-' | '+' ] term;
5  term: Number | '(' sum ')';
```

Chapter three and chapter four explained that a grammar, token patterns, and the classes of the EBNF module[2] are all it takes to implement lexical and syntax analysis for sentences conforming to the grammar, in this case to recognize comma-separated lists of arithmetic expressions.

Chapter five introduced `Action`[2] methods (which we will just call *actions* from now on) to interact with syntax analysis and produce useful output rather than deeply nested lists.

This chapter looks at actions to evaluate arithmetic expressions or to translate them into different representations more suitable for interpretation, etc. Many examples reuse actions; all classes are available from the module `Six`[2] which is built into the practice page.

Arithmetic expressions are at the core of programming languages. This chapter also presents a grammar for a little language, actions to compile the language into JavaScript functions, and actions to compile the language for execution on a stack machine which will be simulated in JavaScript.

Immediate Evaluation

It makes sense to add a few more rules to the grammar above because they provide hooks for actions. For example, there is no point in letting syntax analysis distinguish addition and subtraction, only to have to check for `+` or `-` a second time when it comes to evaluation. Example 6/01[1] recognizes the same list of arithmetic expressions:

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis;
- compare to the original grammar in example 2/12[1] to see that the additional rules result in more deeply nested lists.

Here is the modified grammar:

```
1  list:     sum [{ ',' sum }];
2  sum:      product [{ add | subtract }];
3  add:      '+' product;
4  subtract: '-' product;
5  product:  signed [{ multiply | divide }];
6  multiply: '*' signed;
7  divide:   '/' signed;
8  signed:   [ '-' ] term;
9  term:     number | '(' sum ')';
10 number:   Number;
```

The goal is to immediately evaluate arithmetic expressions, i.e., each action should return the value which the corresponding rule has recognized. This results in obvious actions for the last three rules:

```
1  class Eval02 {
2    // list:     sum [{ ',' sum }];
3
4    /** `sum: product [{ add | subtract }];` */
5    sum (product, many) { puts(g.dump(product)); }
6
7    // add:      '+' product;
8    // subtract: '-' product;
9    // product:  signed [{ multiply | divide }];
10   // multiply: '*' signed;
11   // divide:   '/' signed;
12
13   /** `signed: [ '-' ] term;` */
14   signed (minus, term) { return minus ? - term : term; }
15
16   /** `term: number | '(' sum ')';` */
17   term (...val) { return val.length == 1 ? val[0] : val[1]; }
18
19   /** `number: Number;` */
20   number (number) { return parseInt(number, 10); }
21 }
```

Because of the placeholder for `sum` (line 5) very simple expressions can already be tested in example 6/02[1]:

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis and invoke the actions.
- Toggle **actions** and press **parse** again to see how the values are computed by the actions. Why is so much output `undefined`? Why are some numbers followed by commas in the output?

The next three actions are tricky but representative for any evaluation based on this kind of a

grammar:

```
1  class Eval03 extends Six.Eval02 {
2    // list:     sum [{ ',' sum }];
3    // sum:      product [{ add | subtract }];
4    // add:      '+' product;
5    // subtract: '-' product;
6
7    /** `product: signed [{ multiply | divide }];` */
8    product (signed, many) {
9      return (many ? many[0] : [ ]).
10       reduce((product, list) => list[0](product), signed);
11   }
12
13   /** `multiply: '*' signed;` */
14   multiply (_, right) { return left => left * right; }
15
16   /** `divide: '/' signed;` */
17   divide (_, right) { return left => left / right; }
18
19   // signed:   [ '-' ] term;
20   // term:     number | '(' sum ')';
21   // number:   Number;
22 }
```

Multiplication and division are left-associative operations, i.e., they are evaluated from left to right by `reduce()`† in the action for `product` (line 10 above). Therefore, the `multiply` and `divide` actions have to return functions which will take a left argument, combine it with the right argument which the `multiply` and `divide` rules have recognized and evaluated, and return the result of the computation (lines 14 and 17).

The `product` action gets a first `signed` value which the `signed` rule has already evaluated, and it gets `many` of these functions for the remainder of the chain of products (line 9).

The chain has to be processed as discussed in the *Idioms* in chapter five using `reduce()`†: If there are no functions, the first `signed` value is the result. Otherwise the callback function (line 10) is applied from left to right and each time applies a multiplication or division to the previous product, starting with `signed`.

Again, it might be a good idea to try this stage in example 6/03[1]:

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis and invoke the actions.

It seems to work at first, but `2 / (3*4)` produces the unexpected outputs `12` and `NaN`.

- Toggle **actions** and press **parse** again to see how the values are computed. Where do the *two* lines of output `12` and `NaN` come from? Which action has to be repaired?

Check out the complete immediate expression evaluation in example 6/04[1]. Addition and

subtraction use the same pattern as multiplication and division:

```
1  class Eval04 extends Six.Eval03 {
2    /** `list: sum [{ ',' sum }];` */
3    list (sum, many) {
4      puts(sum);
5      if (many) many[0].forEach(seq => puts(seq[1]));
6    }
7
8    /** `sum: product [{ add | subtract }];` */
9    sum (product, many) {
10     return this.product(product, many);
11   }
12
13   /** `add: '+' product;` */
14   add (_, right) { return left => left + right; }
15
16   /** `subtract: '-' product;` */
17   subtract (_, right) { return left => left - right; }
18
19   // product:  signed [{ multiply | divide }];
20   // multiply: '*' signed;
21   // divide:   '/' signed;
22   // signed:   [ '-' ] term;
23   // term:     number | '(' sum ')';
24   // number:   Number;
25 }
```

`sum` is used recursively inside parentheses, i.e., it needs to return a value and it cannot output the final result. Therefore, the action for the start rule has to display each top-level `sum` (lines 3 to 6 above).

Functional Evaluation

Returning functions for some arithmetic operations suggests that the *entire* evaluation could be delayed — every action returns a function, the result is stored as an "executable" and can be evaluated (executed) several times over. This is more useful if simple variables are added to the grammar in example 6/05[1].

There has to be a token for names:

```
1  Name: /[a-z]+/
```

There has to be a rule to recognize a name

```
1  name: Name;
```

and the rule for `term` has to recognize a `name` just like a `number`:

```
1  term: number | name | '(' sum ')';
```

The action for `name` (lines 27 to 30 below) has to return a value. For now, a local object `memory` maps names to values (line 29) and it is generated if needed (line 28).

```
1  class Functions05 extends Six.Eval04 {
2    #parser;                                  // for error messages
3    get parser () { return this.#parser; }
4
5    constructor (parser) { super(); this.#parser = parser; }
6
7    // list: sum [{ ',' sum }];
8
9    /** `sum: 'let' Name '=' sum | product [{ add | subtract }];` */
10   // arg          [1]      [3]   [0]     [1]
11   sum (... arg) {
12     if (arg.length < 4) return this.parser.call(this, super.sum, arg[0], arg[1]);
13     if (!this.memory) this.memory = { };
14     return this.memory[arg[1]] = arg[3];
15   }
16
17   // add: '+' product;
18   // subtract: '-' product;
19   // product:  signed [{ multiply | divide }];
20   // multiply: '*' signed;
21   // divide: '/' signed;
22   // signed: [ '-' ] term;
23   // term: number | name | '(' sum ')';
24   // number: Number;
25
26   /** `name: Name;` returns value or `0` */
27   name (name) {
28     if (!this.memory) this.memory = { };
29     return name in this.memory ? this.memory[name] : 0;
30   }
31 }
```

This is only useful if there is a way to assign a value to a name. Therefore, the rule for `sum` is modified as shown in the comment (line 9 above).

The action for `sum` defers to the superclass for simple sums (line 12) or it creates `memory` if needed (line 13) and performs the assignment (line 14).

Check this stage out in example 6/05[1]:

- Press **new grammar** to represent and check the grammar and

- press **parse** to evaluate the expressions in the `program` area.

```
1  let x = 3,
2  (x + 1) / (y - 2) * 3,
3  y + (x + 1) / ((let y = 1) - 2) * 3
```

The assignment in line 1 returns `3`. So far, `y` is undefined, i.e., zero; therefore, line 2 produces `-6`. In line 3, `y` is set to `1` but only once the denominator is evaluated, i.e., the output is `-12` and not `-11`.

At this point, expressions are evaluated as they are recognized and the top-level `list` action returns `undefined` to the `Parser`'s `parse()`[2] method which returns it to be shown in the output area.

If every action returns a function rather than a value we can execute the top-level function more than once. This is interesting if there is input during execution; therefore, example 6/06[1] adds `input` as a "reserved" variable name, optionally with a default value:

```
1  term: input | number | name | '(' sum ')';
2  input: 'input' [ Number ];
```

Here are the first few modified actions which return functions:

```
1  class Functions06 extends Six.Functions05 {
2    // sum: product [{ add | subtract }];
3    // add: '+' product;
4    // subtract: '-' product;
5    // product: signed [{ multiply | divide }];
6    // multiply: '*' signed;
7    // divide: '/' signed;
8    // signed: [ '-' ] term;
9    // term: input | number | name | '(' sum ')';
10
11   /** `input: 'input' [ Number ];` returns fct */
12   input (_, number) {
13     const dflt = String(number !== null ? number[0] : 0);
14     return () => parseInt(prompt('input', dflt), 10);
15   }
16
17   /** `number: Number;` returns fct */
18   number (number) {
19     const result = parseInt(number, 10);
20     return () => result;
21   }
22
23   /** `name: Name;` returns fct */
24   name (name) {
25     return memory => name in memory ? memory[name] : 0;
26   }
27 }
```

The `number` action returns a function returning a constant value. The action optimizes and converts the string only once (line 19 above). The returned function takes advantage of a closure† (line 20).

The `name` action returns a function which checks memory *at run time*, i.e., it expects to receive `memory` when the executable is run (line 25).

Finally, the `input` action returns a function which invokes `prompt()`† and converts the resulting string to a number if possible (line 14); `parseInt()`† returns `NaN` if it cannot convert.

The `term` action can be inherited because now it juggles functions: the action of every descendant of the rule must deliver a function and the `term` action picks it out of the list produced by recognition.

If `sum` and `product` recognize a single descendant they will just pass on whatever the descendant's action returns, i.e., if `sum` is the start rule the actions can be inherited and one simple expression such as `input` will be compiled into a JavaScript function.

Again, it might be a good idea to try this stage in example 6/06[1]:

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis and invoke the actions, i.e., to compile simple programs such as `10` or `(input)`.
- Press **run** to execute the compiled program.
- Note that a program consisting only of a variable name such as `x` is compiled into a function which expects `memory` as an argument, i.e., execution with **run** will fail:

```
> run = g.parser().parse(program, actions)
memory => name in memory ? memory[name] : 0
> run()
memory is not an Object. (evaluating 'name in memory')
```

Example 6/07[1] converts the remaining actions so that they return functions. The action for `signed` is patterned after those for `term` and `name`:

```
class Functions07 extends Six.Functions06 {
  // ...
  signed (minus, term) {
    return minus ? memory => - term(memory) : term;
  }
```

If there really is a sign change, there has to be a new function, otherwise the old function will do.

All of these functions have to accept `memory` because `term` could be the function created by `name` and, therefore, require that argument.

Actions for `add`, `subtract`, `multiply`, and `divide` again are tricky, for example:

```
  multiply (_, right) {
    return left => memory => left(memory) * right(memory);
  }
```

`multiply` expects two functions, `left` and `right`, each with an argument `memory`, to return the left and right operand value for the multiplication.

The function `right` is handed to the `multiply` action as the result of recognizing the right operand, i.e., it is the result of the `signed` action and as such expects `memory` as argument.

The `multiply` action returns a function which expects a function `left` as an argument and returns the function which expects `memory` as an argument and produces the result of the multiplication, i.e., a function which fits the pattern of `signed` and `name`.

This complicates the `product` and `sum` actions a bit, for example:

```
1   product (signed, many) {
2     const c = (a, b) => b(a);  // function composition
3     return (many ? many[0] : []).
4       reduce((product, list) => c(product, list[0]), signed);
5   }
```

The action for `product` fits the idiom discussed in chapter five: there is one function `signed` and there can be `many` more. The functions have to be composed with `reduce()`† and the callback function (line 2) has to compose two functions:

- `a` is a function which expects `memory` as an argument, i.e., a function like the one returned by the action for `signed`.
- `b` is a function which expects something like `a` as an argument, i.e., a function like the one returned by the action for `multiply`.
- As discussed for `multiply` above, `b(a)` is in fact a function.

The result of `reduce()`† is the last function produced with the composition `c()`, i.e., it is a function which can be applied to `memory` to carry out the calculations required to evaluate the `product`. Note that `reduce()`† is executed at compile time, not at run time.

The actions for `divide`, `add`, and `subtract` follow the pattern of `multiply`. The action for `sum` either produces a function to implement the assignment or it delegates to the action for `product`:

```
1   sum (... arg) {
2     if (arg.length == 4)
3       return memory => memory[arg[1]] = arg[3](memory);
4     else
5       return this.product(arg[0], arg[1]);
6   }
```

Finally, the action for `list` has to return the executable, i.e., a function which does not accept an argument and displays the result of each sum in the comma-separated list:

```
1   list (sum, many) {
2     const list = [ sum ].
3       concat(many ? many[0].map(seq => seq[1]) : [ ]);
4     return () => {
5       const memory = { };
6       puts(... list.map(fct => fct(memory)));
7     }
8   }
9 }
```

The list of functions to be called can be produced during recognition (lines 2 and 3 above) based

on the idiom discussed in chapter five. The executable creates `memory` (line 5), uses `map()`† to produce an array of results, and uses spread syntax† to display it on one line, i.e., with a single call to `puts()` (line 6).

Try the functional expression compiler in example 6/07[1]:

- Press **new grammar** to represent and check the grammar and
- press **parse** to perform syntax analysis and invoke the actions, i.e., to compile a program such as

```
1  let x = input 3, (x + 1) / (y - 2) * 3,
2  y + ((let y = 2) + 1) / (x - 2) + y
```

- Press **run** and execute the program for different inputs.

Stack Evaluation

In this section arithmetic expressions are compiled into code for a stack machine‡.

Example 6/08[1] contains new actions for the single expression grammar from example 6/06[1]

which simply output the operation to be performed:

```
class Postfix08 {
  // sum: product [{ add | subtract }];

  /** `add: '+' right;` */
  add (_, r) { puts('add'); }

  /** `subtract: '-' right;` */
  subtract (_, r) { puts('subtract'); }

  // product: signed [{ multiply | divide }];

  /** `multiply: '*' right;` */
  multiply (_, r) { puts('multiply'); }

  /** `divide: '/' signed;` */
  divide (_, r) { puts('divide'); }

  /** `signed: [ '-' ] term;` */
  signed (minus, t) {  if (minus) puts('minus'); }

  // term: input | number | name | '(' sum ')';

  /** `input: 'input' [ Number ];` */
  input (i, n) { puts('input'); }

  /** `number: Number;` */
  number (number) { puts(number); }

  /** `name: Name;` */
  name (name) { puts(name); }
}
```

There are no actions for `sum`, `product`, and `term`. The program

```
  (x + 1) / (input - -2) * 3
```

produces the output (one item per line)

```
x 1 add input 2 minus subtract divide 3 multiply
```

This is known as *Reverse Polish Notation*‡ — operators follow their operands and no parentheses are needed as long as it is clear how many operands belong to each operator. This notation can be viewed as machine language for a stack machine‡:

- operands are pushed onto the stack,
- operators are applied to the top entries on the stack,
- pop the entries off the stack, and
- push the result onto the stack.

The example suggests that syntax analysis with a grammar should make it near trivial to produce stack machine code to evaluate an arithmetic expression.

Example 6/09[1] shows how to generate "code" and simulate a stack machine in JavaScript. The grammar from example 6/07[1] can be used but the start rule has to be modified to avoid collecting all results of `sum` on the stack:

```
1  list: stmt [{ ';' stmt }];
2  stmt: sum;
3  sum: 'let' Name '=' sum | product [{ add | subtract }];
```

A program is a `list` of statements, a statement is a `sum` and can start with an assignment prefixed by `let` to avoid an ambiguity. In the `list` above, statements are semicolon-separated. Alternatively

```
1  list: { stmt ';' };
```

would make statements semicolon-terminated.

Here is a small program:

```
1  let x = let y = input 4; (x + 1) / (y - 2) * 3;
2  y + ((let y = 2) + 1) / (x - 2) + y
```

Note that assignment is right-associative because it involves right recursion.

It turns out to be a good idea to encapsulate the infrastructure for a stack machine as a class to separate it from the compiler:

```
1  class Machine09 {
2    code = [ ];                              // holds the instructions
3
4    /** Represents `code` as text */
5    toString () {
6      return this.code.map((f, n) => n + ': ' + f).join('\n');
7    }
8
9    /** Creates stack machine */
10   run (memorySize) {
11     return () => {
12       const memory = Array(memorySize).fill(0);    // create memory
13       this.code.forEach(code => code(memory));           // execute
14       return memory;
15     };
16   }
17 }
```

A program is compiled into an array `code[]` of JavaScript functions (line 2 above) which simulate the instructions of the stack machine. The method `toString()` can be used to display the instructions (line 5).

The method `run()` creates and returns the stack machine interpreter (line 10) which must be a

parameterless function.

Runtime values are stored in an array `memory[]` which contains the variables' values followed by the stack. Thankfully, JavaScript imposes no restrictions on array length. `memory[]` is created when the stack machine starts (line 12) and manipulated by the JavaScript functions which simulate the instructions of the stack machine (line 13). At this point, each instruction in `code[]` is executed sequentially just once.

The actions need access to the stack machine infrastructure and to a symbol table which maps variable names to memory addresses. The actions class has construction parameters so that future subclasses can extend the machine infrastructure and/or the symbol table:

```
1  class Arithmetic09 {
2    #parser;                                  // for error messages
3    get parser () { return this.#parser; }
4    #machine;                                  // handles execution
5    get machine () { return this.#machine; }
6    #symbols = new Map();    // symbol table, maps names to addresses
7    get symbols () { return this.#symbols; }
8
9    constructor (parser, machine = new Machine09 ()) {
10     this.#parser = parser;
11     this.#machine = machine;
12   }
13   // ...
```

Every variable has an address that is determined at *compile time*, i.e., during syntax analysis, and stored in the symbol table. The symbol table is managed by a method `_alloc()` which assigns new addresses to as yet unknown names:

```
1    /** Returns memory address for name */
2    _alloc (name) {
3      let addr = this.symbols.get(name);              // known name?
4      if (typeof addr == 'undefined')
5        this.symbols.set(name,                           // new name
6          addr = this.symbols.size);       // allocate, starting at 0
7      return addr;
8    }
```

With all of this in place, the actions for the operands are easy to write:

```
1   /** `input: 'input' [ Number ];` */
2   input (_, number) {
3     const dflt = String(number !== null ? number[0] : 0);
4     this.machine.code.push(
5       memory => memory.push(parseInt(prompt('input', dflt), 10))
6     );
7   }
8
9   /** `number: Number;` */
10  number (number) {
11    const result = parseInt(number, 10);
12    this.machine.code.push(memory => memory.push(result));
13  }
14
15  /** `name: Name;` */
16  name (name) {
17    const addr = this._alloc(name);
18    this.machine.code.push(memory => memory.push(memory[addr]));
19  }
```

The `input` action creates a machine instruction which uses `prompt()`† to obtain a string, converts it to a number, and pushes the value onto the stack (line 5 above).

The `number` action "compiles" the numerical value of the string collected by the token (line 11 above) and stores a machine instruction as generated code which will push this numerical value onto the stack (line 12).

The `name` action consults `_alloc()` for the address of the variable in `memory[]` (line 17) and creates a machine instruction to push the value of the variable onto the stack (line 18).

Moving on to the arithmetic operations:

```
1   /** `add: '+' right;` */
2   add (_, r) {
3     this.machine.code.push(
4       memory => memory.splice(-2, 2, memory.at(-2) + memory.at(-1))
5     );
6   }
```

The `add` action is a typical example. The machine instruction has to add the top two values on the stack and replace them by the single result value — a simple job for the `splice()`† method.

Just as in example 6/08[1] there are no actions for `term` and `product` because their results are already on the stack, but `sum` has to implement the embedded assignment, if any:

```
1   /** `sum: 'let' Name '=' sum | product [{ add | subtract }];` */
2   sum (...val) {
3     if (val.length < 4) return;
4     const addr = this._alloc(val[1]);
5     this.machine.code.push(memory => memory[addr] = memory.at(-1));
6   }
```

`_alloc()` finds or creates an address (line 4 above) and the machine instruction copies the value from the top of the stack to that address in `memory[]` (line 5). Note that the value remains on top of the stack.

The action for `stmt` generates a machine instruction to display the result of a top-level `sum` (or assignment) as it is popped off the stack (line 13 below).

The action for the start rule `list` is executed after all statements have been compiled. It displays the results of the compilation (lines 3 to 7) and calls `run()` to create the stack machine function (line 8):

```
1    /** `list: stmt [{ ';' stmt }];` */
2    list (s, many) {
3      puts(this.machine.toString());                    // show code
4      this.symbols.forEach(                           // show variables
5        (value, name) => puts(name, 'at', value));
6      const size = this.symbols.size;          // number of variables
7      puts('stack starts at', size);
8      return this.machine.run(size);                 // stack machine
9    }
10
11   /** `stmt: sum;` */
12   stmt (s) {                                 // print and clear stack
13     this.machine.code.push(memory => puts(memory.pop()));
14   }
15 }
```

The compiler is complete but it consist of two classes. The `actions` area has to evaluate to a single class or object with the `Action`[2] methods for the parser. It takes a little trick to accomplish this:

```
1  (() => {  // define and immediately use an anonymous function
2    class Machine09 { ... }
3    class Arithmetic09 { ... }
4    return Arithmetic09;
5  }) ()
```

The two class definitions are placed into an anonymous function where `return` delivers the actual value for the `actions` (line 4) and the anonymous function is called immediately after being defined (line 5).

It is instructive to see the actual code. For example `a - - b` compiles into

```
1  > run = g.parser().parse(program, actions)
2  0: memory => memory.push(memory[addr])
3  1: memory => memory.push(memory[addr])
4  2: memory => memory.splice(-1, 1, -memory.at(-1))
5  3: memory => memory.splice(-2, 2, memory.at(-2) - memory.at(-1))
6  4: memory => puts(memory.pop())
7  a at 0
8  b at 1
9  stack starts at 2
```

The output shows the machine instruction functions but not the variable addresses because the functions use closure† to capture the addresses computed by `_alloc()`. Try out example 6/09[1]:

- Press **new grammar** to represent and check the grammar,
- press **parse** to compile the program in the `program` area, and
- press **run** to execute the stack machine.

The output shown above is created by converting the values in `code[]` from JavaScript functions to strings, i.e., the output shows the actual JavaScript functions which simulate what hardware instructions would do to `memory[]`. As such, the output is precise but a bit hard to read.

Example 6/10[1] is a slightly modified version of example 6/09[1] which makes the generated machine instructions easier to read. The following output, again for `a - - b`, still consists of JavaScript functions converted to strings, but it looks like it is closer to the reverse Polish notation‡ introduced at the beginning of this section:

```
1  > run = g.parser().parse(program, actions)
2  0: memory => this.Load(0)(memory)
3  1: memory => this.Load(1)(memory)
4  2: memory => this.Minus(memory)
5  3: memory => this.Subtract(memory)
6  4: memory => this.Pop(memory)
7  a at 0
8  b at 1
9  stack starts at 2
```

The infrastructure for the stack machine is extended. In `Machine10` the machine instructions are defined using methods so that they are more meaningful when displayed as strings, for example:

```
1  class Machine10 extends Six.Machine09 {
2    // ...
3    /** `stack: ... a b -> ... a-b` */            Subtract (memory) {
4      memory.splice(-2, 2, memory.at(-2) - memory.at(-1));
5    }
6
7    /** `stack: ... -> ... memory[addr]` */             Load (addr) {
8      return memory => memory.push(memory[addr]);
9    }
10 }
```

`code[]` is local to the `Machine10` object and contains the functions shown in the output above where `this` refers to the `Machine10` object because the stack machine executable is defined in the `run()` method as an arrow function†, i.e., it inherits `this` and uses it when interpreting `code[]`.

As a result, `this.Subtract(memory)` acts just like a simple function with `memory` as argument (lines 3 to 5 above). `this.Load(0)(memory)` (line 2) is more costly because it generates the function to manipulate `memory` during execution (line 8) — the price to pay for the mnemonic display — but the address argument is computed at compile time.

Actions generate code as follows:

```
1  class Arithmetic10 extends Six.Arithmetic09 {
2    constructor (parser, machine = new Machine10()) {
3      super(parser, machine);
4    }
5    // ...
6
7    /** `subtract: '-' product;` */
8    subtract () { this.machine.gen('Subtract'); }
9
10   /** `name: Name;` */
11   name (name) {
12     this.machine.gen('Load', this._alloc(name));
13   }
14 }
```

`Arithmetic10` uses `Machine10` unless a subclass should decide otherwise (line 2 above). Code generation is implemented in `Machine10` as a method `gen()` (line 2 below) which takes the instruction name and an optional list of values, calls the method `ins()` to produce the machine instruction, and then adds the function to `code[]`:

```
1    /** returns `code.length` */
2    gen (name, ... args) {
3      return this.code.push(this.ins(name, ... args));
4    }
5
6    /** returns instruction function */
7    ins (name, ... args) {
8      return args.length ?
9        eval(`memory => this.${name}(${args.join(', ')})(memory)`) :
10       eval(`memory => this.${name}(memory)`);
11   }
```

The operations on `memory` are still the same JavaScript functions, but they are wrapped into outer functions with expressive names.

`ins()` uses `eval()`† (lines 9 and 10 above) to perform closure† so that, e.g., the address used in a `Load` instruction is visible when the function is converted into a string for display purposes. Using the `Function`† constructor would be safer; however, the resulting function then has no display name.

Unfortunately, extending to `Arithmetic10` and `Machine10` results in a lot of unused code:

- All machine instruction functions have been copied from `Arithmetic09` into `Machine10`.
- All actions have been overwritten in `Arithmetic10`, only `_alloc()` has been inherited.

This could have been avoided if the mnemonic display of the instructions had been introduced immediately.

Control Structures

In this section arithmetic expressions will be extended with typical control structures and implemented for the stack machine‡. Changes to the grammar can be seen on this page…, new stack machine and action method classes can be seen in the method browser[3]:

- press **show** to see the methods, sorted alphabetically,
- press **by class or mix-in** to sort by module, class or mix-in, and finally method name,
- select individual classes and/or methods to see less.

Thus far the stack machine used `forEach()`† to process the machine instructions in `code[]` just once, one after another. Real machines have a program counter which is advanced as an instruction is fetched from memory. Here is an idea for a modification to the stack machine:

```
1  run (size, startAddr) {                    // create stack machine
2    return () => {
3      memory.pc = startAddr;             // initialize program counter
4      ...
```

Once properties are added to `memory` to simulate machine registers such as a "program counter" there can be simulated branch instructions, for example:

```
1  class Machine11 extends Six.Machine10 {
2    // ...
3    Branch (a) {                           // stack: ... -> ... | pc: a
4      return memory => memory.pc = a;
5    }
6    Bzero (a) {                 // stack: ... bool -> ... | pc: !bool? a
7      return memory => { if (!memory.pop()) memory.pc = a; }
8    }
```

If `this.Branch(a)(memory)` is executed the stack machine takes the next instruction from address `a`.

If `this.Bzero(a)(memory)` is executed the stack is checked and popped, and if the top value was zero the stack machine takes the next instruction from address `a`.

It is helpful if the stack machine‡ can display instructions as they are executed, and if tracing can be controlled from within a program, e.g., tracing could be unconditional or depend on the current value of a variable. `Machine11` adds a method `trace()` which accepts `true` to create an unconditional tracing function (lines 3 to 5 below) or a numerical memory `address` to create a

function which traces if the current value in `memory[address]` is non-negative (lines 6 to 10):

```
1   /** Returns trace function, if any */
2   trace (address) {
3     if (address === true)                    // unconditional trace
4       return (memory, pc) =>              // traces instruction at pc
5         puts(memory.toString(), pc+':', this.code[pc].toString());
6     if (typeof address == 'number') // address of control variable?
7       return (memory, pc) => {            // traces instruction at pc
8         if (memory[address] >= 0) // variable at addr non-negative?
9           puts(memory.toString(), pc+':', this.code[pc].toString());
10      };
11  }
12
13  get Memory () { return this.#Memory ??= class extends Array {
14      toString () { return '[ ' + this.join(' ') + ' ]'; }
15    };
16  }
17  #Memory;
```

Either function displays the instruction stored in `code[pc]`. Each trace output includes a display of the current memory contents produced by a locally modified `toString()` method (lines 13 to 17 above). Eventually, `memory[]` will be more than just a linear list and `toString()` can be overwritten again.

The stack machine interpreter, i.e., the executable returned by `run()`, still creates `memory[]` and initializes the variables to zero but it now has to consider the program counter and tracing:

```
1   run (size, startAddr = 0, traceAddr) {
2     let t;                       // [closure] trace function, if any
3     const StackMachine = (memory, steps) => {
4       if (!memory) {                                  // initialize?
5         if (steps) t = this.trace(true);  // steps? permanent trace
6         else {                           // no steps: don't suspend
7           t = this.trace(traceAddr); steps = Infinity;
8         }
9         memory = new this.Memory(size).fill(0);    // create memory
10        memory.pc = startAddr;        // initialize program counter
11        t && puts(memory.toString());             // initial memory
12      }
13      while (steps -- && memory.pc < this.code.length) {  // steps?
14        const pc = memory.pc ++;         // advance program counter
15        this.code[pc](memory);            // execute at previous pc
16        t && t(memory, pc);           // trace executed instruction
17      }
18      memory.continue = memory.pc < this.code.length;     // again?
19      return memory;
20    };
21    return (memory, steps) => StackMachine(memory, steps);
22  }
```

`run()` is called with the `size` of the initialized part of memory, the start address for the program counter, and an address to control tracing if any (line 1 above).

The resulting `StackMachine` accepts two optional arguments, the `memory[]` array and `steps`, the number of instructions to execute (line 3), and eventually returns the modified `memory[]` (line 19).

- `StackMachine()` will create `memory[]` and run without limit.
- `StackMachine(null, 10)` will create `memory[]` and execute and trace 10 instructions.
- `StackMachine(memory, 10)` will then execute another 10 instructions.

Initialization happens if there is no `memory[]` (line 4):

- Permanent tracing is set up if `steps` is defined and not zero (line 5), otherwise tracing depends on the control address handed to `run()` and there is no limit on `steps` (line 7).
- `memory[]` is created and zero-filled (line 9), the program counter at `memory.pc` is initialized (line 10), and the initial `memory[]` is shown if there is tracing (line 11).

Instructions are executed as long as there are `steps` left and if the program counter does not point beyond `code[]` (line 13). The program counter is incremented (line 14), the instruction is executed (line 15) and traced if tracing was set up (line 16).

Once there are no more `steps`, a property is set to indicate if more steps could be executed (line 18) and `memory[]` is returned.

These modifications do not change the basic behavior of the stack machine, i.e., arithmetic expressions from example 6/09[1] — but for embedded assignment — can be part of a little language that has control structures.

Example 6/11[1] uses Euclid's algorithm‡ to compute the greatest common divisor of two numbers with a trace of one of the subtractions:

```
1  x = input 36; y = input 54;
2  while x <> y do
3    if x > y then x = x - y
4    else y = y - x
5    fi
6  od;
7  print x
```

- Press **new grammar** to represent and check the grammar,
- press **parse** to perform syntax analysis and generate code, and
- press **run** to execute the program, reply `36` and `54` to the requests for input, and see the output:

```
1  > run()
2  18
3  [ 18 18 ]
```

The program requires that the grammar from example 6/09[1] is extended to support control

structures:

```
prog:     stmts;
stmts:    stmt [{ ';' stmt }];
stmt:     assign | print | loop | select;
assign:   Name '=' sum;
print:    'print' sums;
sums:     sum [{ ',' sum }];
loop:     While cmp Do stmts 'od';
While:    'while';
Do:       'do';
select:   'if' cmp Then stmts [ Else stmts ] 'fi';
Then:     'then';
Else:     'else';
sum: ...
```

This grammar makes a clear distinction between statements (`stmt`), expressions (`sum`), and conditions (`cmp`). This avoids a possible ambiguity between an expression and an assignment: without `let` both can start with a variable name.

This grammar for control structures uses terminating literals to avoid ambiguity issues like the *dangling else*‡ problem presented in example 4/05[1]. Some control structure literals such as `'while'` have been turned into simple rules so that actions can be attached — this is discussed below.

Conditions could be about as complex as arithmetic expressions; however, for the little language comparisons will suffice, e.g.,

```
cmp: eq | ne;
eq: sum '=' sum;
ne: sum '<>' sum;
```

Unfortunately, these rules are ambiguous because both alternatives start with `sum`. Here is a better approach:

```
cmp:      sum rel;
rel:      eq | ne | gt | ge | lt | le;
eq:       '=' sum;
ne:       '<>' sum;
gt:       '>' sum;
ge:       '>=' sum;
lt:       '<' sum;
le:       '<=' sum;
```

While this is patterned after `sum` and `product`, this grammar does not allow cascading comparisons (which, e.g., JavaScript does).

Here is the action for one of the comparisons:

```
1  class Control11 extends Six.Arithmetic10 {
2    constructor (parser, machine = new Machine11()) {
3      super(parser, machine);
4    }
5
6    /** `eq: '=' sum;` */   eq () { this.machine.gen('Eq'); }
```

The extended actions class requires additional machine instructions and, therefore, inserts `Machine11` during construction (line 2 above). Here is one of the comparison instructions in `Machine11`:

```
1  Eq (memory) {                       // stack: ... a b -> ... a == b
2    memory.splice(-2, 2, memory.at(-2) == memory.at(-1));
3  }
```

This instruction will replace the top two values on the stack with the result of the comparison, i.e., `true` or `false`. Technically, these are very special values in the little language. The syntax does not permit them to be assigned to variables. Instead, they will be consumed by branch instructions for the control structures.

Just like compiling sums, compiling comparisons also amounts to reverse Polish notation‡, i.e., the rules `rel` and `cmp` require no actions.

However, the `assign` and `print` statements consume sums and require actions:

```
1    /** `assign: Name '=' sum;` stores and pops stack */
2    assign (name, e, s) {
3      this.machine.gen('Store', this._alloc(name));
4      this.machine.gen('Pop');
5    }
6
7    /** `print: 'print' sums;` */
8    print (_, sums) { this.machine.gen('Print', sums); }
9
10   /** `sums: sum [{ ',' sum }];` returns number of values */
11   sums (sum, many) { return 1 + (many ? many[0].length : 0); }
```

The machine instructions `Store` and `Pop` were already defined in `Machine10`, i.e., in example 6/10[1]. The `assign` action generates them to copy the top value on the stack into the memory location for the variable (line 3 above) and to clear the stack at the end of the statement (line 4).

A `print` statement displays a list of `sum` values. It receives all arguments on the stack. The number of arguments is counted at compile time (line 11) and compiled into the `Print` instruction (line 8). At run time the `Print` instruction pops the arguments off the stack and hands them to `puts()` for display:

```
1  Print (n) {                              // stack: ... n*val -> ...
2    return memory => puts(... memory.splice(- n));
3  }
```

This little language supports control structures. The actions for `loop`, `While`, and `Do` have to implement the following branch structure:

label	code
`While:`	
	`cmp`
`Do:`	`branch if zero to 'od'`
	`stmts`
	`branch to 'While'`
`od:`	

This explains what the actions have to do:

```
1   /** `While: 'while';` returns address for branch to `while` */
2   While (w) { return this.machine.code.length; }
3
4   /** `Do: 'do';` returns address of slot for bzero to `od` */
5   Do (d) { return this.machine.code.push(null) - 1; }
6
7   /** `loop: While cmp Do stmts 'od';` */
8   loop (While, _, Do, s, o) {
9     const od = this.machine.gen('Branch', While);
10    this.machine.code[Do] = this.machine.ins('Bzero', od);
11  }
```

The `While` action just returns the next address in the generated code (line 2 above).

The `Do` action allocates a slot in the generated code and returns the address of the slot (line 5) because `push()`† always returns the new array length.

Finally, `loop` puts it all together: the action adds an unconditional branch from the end of the loop body back to `While` (line 9) and populates the slot allocated by `Do` with a conditional branch to `od` to terminate the loop (line 10).

Yes, the conditional branch at `Do` fails each time through the loop until it succeeds only once, at the end of the loop. An obvious optimization is to reverse the order of the loop body and the condition in the generated code and cut the number of branch executions in half; however, moving code when syntax analysis generates it directly is not really an option — absolute branches like the ones used here would break.

`select` is a little harder because there are two possibilities:

label	code with `else`	code without `else`
	`cmp`	`cmp`
`Then:`	`branch if zero to 'Else'`	`branch if zero to 'fi'`
	`stmts`	`stmts`
	`branch to 'fi'`	
`Else:`		
	`stmts`	
`fi:`		

The actions have to create the appropriate branch instructions:

```
1   /** `Then: 'then';` returns address for bzero to `else` `fi` */
2   Then (t) { return this.machine.code.push(null) - 1; }
3
4   /** `Else: 'else';` creates slot for branch to `fi`,
5       returns address of `else` */
6   Else (e) { return this.machine.code.push(null); }
7
8   /** `select: 'if' cmp Then stmts [ Else stmts ] 'fi';` */
9   select (i, c, Then, s, Else, f) {
10    const fi = this.machine.code.length;         // address of 'fi'
11    if (Else) {
12      Else = Else[0];               // address after branch to 'fi'
13      this.machine.code[Then] = this.machine.ins('Bzero', Else);
14      this.machine.code[Else - 1] = this.machine.ins('Branch', fi);
15    } else
16      this.machine.code[Then] = this.machine.ins('Bzero', fi);
17  }
```

The `Then` action allocates a slot in the generated code and returns the address of the slot (line 2 above).

The `Else` action, if called, also allocates a slot in the generated code (for a branch to bypass the `else` clause) but it returns the address following the slot (line 6).

Finally, the `select` action checks if there is an argument created by `else` (line 11) and creates and inserts the branch instructions as described in the pseudo code above.

Code generation is complete. The top-level action for `prog` shows the generated code and provides tracing if a variable `trace` is in the program:

```
1   /** `prog: stmts;` returns executable */
2   prog (_) {
3     const size = this.symbols.size,          // number of variables
4       traceAddr = this.symbols.get('trace'); // if a variable named
5     if (typeof traceAddr != 'undefined') {     // ...'trace' exists
6       puts(this.machine.toString());                  // show code
7       this.symbols.forEach(              // show variable addresses
8         (addr, name) => puts(`${name} at ${addr}`)
9       );
10      puts('stack starts at', size);
11    }
12    return this.machine.run(size, 0, traceAddr);   // stack machine
13  }
```

If the variable exists (lines 4 and 5 above) the generated code and variable addresses are shown (lines 6 to 10). The `run()` method of `Machine11` discussed earlier is called (line 12) to create the stack machine.

In example 6/11[1]

- Press **new grammar** to represent and check the grammar,

- press **parse** to perform syntax analysis and generate code, and
- press **1**, **10**, or **100** to step through the program:

```
> memory = run(null, 1)
0:[ 0 0 ]
[ 0 0 36 ] 0: memory => this.Input(memory)
> memory = run(memory, 1)
[ 36 0 36 ] 1: memory => this.Store(1)(memory)
  ...
```

- Add statements like `trace = -1` or `trace = 1` to display the generated code and to turn tracing off and on within the program:

```
trace = -1; x = input 36; y = input 54;
  ...
```

- Again, press **parse** to perform syntax analysis and generate code:

```
> run = g.parser().parse(program, actions)
0: memory => this.Push(1)(memory)
1: memory => this.Minus(memory)
2: memory => this.Store(1)(memory)
  ...
program counter at 0
trace at 1
x at 2
y at 3
stack starts at 4
(memory, steps) => StackMachine(memory, steps)
```

- Now press **run** to execute the program and see the trace:

```
> run()
0:[ 0 0 0 ]
[ 0 0 0 1 ] 0: memory => this.Push(1)(memory)
[ 0 0 0 -1 ] 1: memory => this.Minus(memory)
18
[ -1 18 18 ]
```

The stack is cleared at the end of each statement because the value of each `sum` is assigned or printed and then popped of the stack or two sums are compared and popped and the result of the comparison is tested and popped, i.e., the stack will grow and shrink during expression evaluation but it cannot grow out of bounds. However, the little language can still be used to crash the practice page — just program an infinite loop and don't step execution...

Functional Programming

Fans of functional programming — especially in JavaScript — should enjoy example 6/12[1], the

implementation of the little language in the style of the functional evaluation of arithmetic expressions as discussed previously. Functional programming can be very elegant, but — unlike the stack machine "instructions" — functions created by composition cannot be decomposed for display. At least, **actions** can be toggled to see what functions the actions create.

The grammar is almost the same as in example 6/11[1] — the extra rules for the keywords in `loop` and `select` are not needed:

```
1   loop:     'while' cmp 'do' stmts 'od';
2   select:   'if' cmp 'then' stmts [ 'else' stmts ] 'fi';
```

The actions for `sum` and all the other rules for arithmetic expressions remain unchanged from example 6/07[1] discussed previously.

Comparisons are implemented just like `add`, and they return functions which compute the right operand and still need a function to compute the left operand value (lines 9 to 11 below):

```
1  class Functions12 extends Six.Functions07 {
2
3    /** `cmp: sum rel;` returns fct */
4    cmp (sum, rel) { return memory => rel[0](sum)(memory); }
5
6    // rel:      eq | ne | gt | ge | lt | le;
7
8    /** `eq: '=' expr;` returns fct for composition */
9    eq (_, right) {
10     return left => memory => left(memory) == right(memory);
11   }
```

The grammar does not allow cascading comparisons. Therefore, the action for `cmp` is a single function composition and does not require `reduce()`† (line 4 above).

Compiling into functions is particularly elegant for control structures. Instead of returning addresses and inserting branch instructions later, the control structures of the little language are simply turned into JavaScript control structures inside functions. Here is the action for `loop`:

```
1   /** `loop: 'while' cmp 'do' stmts 'od';` returns fct */
2   loop (w, cmp, d, stmts, o) {
3     return memory => { while (cmp(memory)) stmts(memory); };
4   }
```

The action for `select` returns one of two functions depending on the presence of `else`:

```
1   /** `select: 'if' cmp 'then' stmts [ 'else' stmts ] 'fi';` returns fct */
2   select (i, cmp, t, stmts, opt, f) {
3     return opt ?
4       (memory => cmp(memory) ? stmts(memory) : opt[1](memory)) :
5       (memory => { if (cmp(memory)) stmts(memory); });
6   }
```

If `else` is present, the returned function uses conditional evaluation (line 4 above), otherwise it uses an `if` statement in JavaScript (line 5).

`loop` and `select` are the only places where the result of the `cmp` action is used. No matter which type JavaScript uses to represent the result of a comparison in the compiled comparisons, the use of this type is consistent with the intent of the compiled control structures — the joy of translating from one language to another...

Assignment now is a statement; therefore, the action for `sum` is overwritten (to omit assignment there) and a new action for `assign` creates the function to perform assignment:

```
1   /** `assign: Name '=' sum;` returns fct */
2   assign (name, e, sum) {
3     return memory => memory[name] = sum(memory);
4   }
```

The stack machine implementation had to pop the `sum` off the stack, but there is no stack in the functional implementation.

The action for `sums` creates a list of functions (line 8 below) and the action for `print` creates a function which elaborates the list to get the values and display them with `puts()` (line 3):

```
1   /** `print: 'print' sums;` returns function */
2   print (p, sums) {
3     return memory => puts(... sums.map(fct => fct(memory)));
4   }
5
6   /** `sums: sum [{ ',' sum }];` returns list of functions */
7   sums (sum, many) {
8     return [ sum ].concat(many ? many[0].map(seq => seq[1]) : []);
9   }
```

The action for `stmts` uses `reduce()`† and the comma operator† to create a function which will execute the statement functions, one after another:

```
1   /** `stmts: stmt [{ ';' stmt }];` returns fct */
2   stmts (stmt, many) {
3     return (many ? many[0] : []).
4       reduce((left, list) =>
5         memory => (left(memory), list[1][0](memory)), stmt[0]);
6   }
```

As before, all the functions expect a `memory` object as an argument which is used to map names to values. The start rule of the grammar has to create this object. `stmts` is referenced in `loop` and `select`; therefore, there is an extra top-level start rule to create `memory` in a parameter-less executable:

```
1   /** `prog: stmts;` returns executable */
2   prog (stmts) { return () => stmts({ }); }
```

Check out example 6/12[1]:

- Press **new grammar** to represent and check the grammar and
- press **parse** to compile the program.

- Toggle **actions** and press **parse** again to see some of the functions created by the actions.

Quick Summary

- Action methods[2] can be used in syntax analysis to interpret — rather than just recognize — a sentence. For example, an arithmetic expression can be translated into reverse Polish notation‡ or evaluated while it is recognized.
- Variable values (or their descriptions) can be stored as properties of a JavaScript object or in a `Map`†.
- Programming languages contain control structures and interpretation of a program requires an intermediate representation so that loops can be interpreted more than once and selections can be partially skipped.
- A stack machine‡B: The Stack Machine is both, very easy to simulate, and very easy to generate code for. It can use a fixed-size array as program storage and a variable-size array as data stack. A program counter can be a property of either array. Global variables can be persistent at the beginning of the data stack. Machine instructions are JavaScript functions which manipulate the data stack, fetch and store variable values in that array, and change the program counter for branching.
- Alternatively, composition of JavaScript functions can also be used as an executable intermediate representation. Unlike the stack machine, however, the internals of this representation cannot be displayed as text.

7. Language Features

Type Checking
Functions, Scopes, and Nesting

Chapter six explained how to use actions to interpret or compile arithmetic expressions. Simple variables, input, output, and control structures were added, resulting in a little language which was compiled into JavaScript functions, or into machine instructions for a stack machine simulated in JavaScript.

This chapter adds specific language features to the little language and shows how to implement them, primarily for the stack machine. All classes are available from the module `Seven`[2] which is built into the practice page.

Syntactic Type Checking

The result types of arithmetic and other operations depend on both, the operators, and the types of the operand values.

Dynamically typed‡ languages such as JavaScript tend to allow operators to combine fairly arbitrary types and let variables store values of different types at different times. The results can be surprising:

- Strings and numbers can be compared in JavaScript, but the string is first converted to a number, i.e., `'A' > '1'` is true (string comparison) and `'A' > 1` is false (number comparison).
- Strings and numbers can be added in JavaScript, but the number is converted to a string and strings are concatenated with the `+` operator, i.e., `1 + '2'` is `'12'` (string sum) rather than `3` (number sum).
- These effects are more surprising if variables are involved.

Strongly typed‡ languages such as Java determine the types of all values at compile time so that they can select the most efficient machine instructions for implementation. This requires either explicitly declaring or implicitly deducing the data types which variables may store, and flagging every operation which does not use compatible data types.

Example 7/01[1] shows that strong type checking can be accomplished with a restrictive grammar where type mismatches will be reported as syntax errors.

- Press **new grammar** to represent and check the grammar, and
- press **parse** to perform syntax analysis:

```
> run = g.parser().parse(program, actions)
error: at (4) '-': in sequence, Lit.parse(): expects 'fi'
```

Here is line 4 of the program:

```
      x = x - y
```

A deep dive into the grammar will reveal that a `Name` such as `x` is recognized as a `stringTerm` which cannot appear adjacent to the operator `-`, i.e., the error message is right on target — but hardly helpful.

Constants in a program tend to exhibit their data type by their looks:

- integers are represented as digit strings,
- floating point numbers require a decimal point, an exponent, or both,
- strings are enclosed in quotes,
- Boolean constants usually are symbols such as `true` and `false`, etc.

Fortran‡ considered a variable to be "integer" if the name started with one of the letters `I` through `N` and "floating point" otherwise, but this idea of *implicit typing* has long since been abandoned. Instead, variable names are usually declared and typed before use, i.e., usually there is only one token for variable names

```
{
  Number: /0|[1-9][0-9]*/,
  String: /'(?:[^'\\]|\\['\\])+'/,
  Name: /[a-z]+/
}
```

and the type of a variable is determined by actions which interpret declarations.

Numbers can be converted to strings; therefore, this little language requires all variables to be strings. Example 7/01[1] more or less duplicates the rules for `cmp` and `sum` from example 6/12[1] to separate string and number operations and fuse comparisons and terms:

```
1  cmp: sum rel | stringSum stringRel;
2
3  term: number | '(' sum ')' | 'number' stringTerm;
4
5  stringRel: stringEq | stringNe | stringGt | stringGe | stringLt | stringLe;
6  stringEq: '=' stringSum;
7    ...
8
9  stringSum: stringTerm [{ stringTerm }];
10 stringTerm: string | name | input | 'string' term;
11 string: String;
12 name: Name;
13 input: 'input' String String;
```

`cmp` requires that numbers are compared to numbers, and strings to strings (line 1 above) because all comparison rules have been duplicated for strings (lines 5 to 7) and only allow `stringSum` operands (lines 1 and 6).

`sum` still recognizes arithmetic with numbers, but `name` and `input` have been moved from a (numerical) `term` (line 3) to a `stringTerm` (line 10), i.e., `name` and `input` only allow strings. `input` uses two `String` tokens to specify a prompt text and a default input value (line 13).

A `stringTerm` can be converted to a number by preceding it with the literal `number` as a cast operation (line 3) and a (numerical) `term` similarly can be converted to a string (line 10).

A `stringSum` allows concatenation using one or more `stringTerm` (line 9); there is no explicit

operator to mark this operation.

Example 7/01[1] still implements Euclid's Algorithm‡:

```
1  x = input 'x' '36'; y = input 'y' '       54';
2  while x <> y do
3    if x > y then
4      x = x - y
5    else
6      y = y - x
7    fi
8  od;
9  print '\'gcd\': ' x
```

It is unchanged but for some extra strings for `input` and `print` (lines 1 and 9 above). There are essentially three errors in the program, the last one serious enough to crash the practice page:

- `number` Conversions are required before `y` can be subtracted from `x` and vice versa (lines 4 and 6).
- `string` Conversions are required before the results can be assigned back to `x` and `y` (lines 4 and 6).
- The default strings for `x` and `y` are not equal (line 1) but string comparison would consider any positive or negative number in `x` to be greater than `y` (line 3) because of the leading blanks in the default input (line 1) and 'x' would be set to ever more negative numbers — the program loops!

The last error demonstrates that strong typing will not necessarily catch serious flaws in a program. It should also be noted that this syntactic approach to type checking requires that there are few types and that operations are well separated.

The actions from Example 6/12[1] can be extended in example 7/01[1] to compile this little language into JavaScript functions:

```
1  class TCheck01 extends Six.Functions12 {
2    // ...
3    /** `stringEq: '=' stringSum;` returns fct for composition */
4    stringEq (_, right) { return this.parser.call(this, super.eq, _, right); }
5
6    /** `stringSum: stringTerm [{ stringTerm }];` returns fct */
7    stringSum (term, many) {
8      const c = (a, b) => memory => a(memory) + b(memory);
9      return (many ? many[0] : []).
10       reduce((sum, list) => c(sum, list[0]), term);
11   }
```

The string comparisons can delegate to the numerical comparisons (line 4 above) because the target language JavaScript is dynamically typed. `stringSum` implements string concatenation in

the callback function for `reduce()`† (line 4).

```
1   // term: number | '(' sum ')' | 'number' stringTerm;
2   //       [0]          [1]                [1]
3   term (...val) {
4     switch (val.length) {
5     case 1: return val[0];
6     case 3: return val[1];
7     case 2: return memory => parseInt(val[1](memory), 10);
8     }
9   }
10
11  /** `stringTerm: string | name | input | 'string' term;` */
12  //              [0]      [0]    [0]              [1]
13  stringTerm (...val) {
14    return val.length == 1 ? val[0] :
15      memory => String(val[1](memory));
16  }
```

`term` has to be overwritten to return a function if a program calls for an explicit conversion (line 7 above). Similarly, `stringTerm` has to implement a conversion to a string value if needed (line 15).

```
1   /** Removes quotes and backslash */
2   _unq (s) {
3     return  s.slice(1,-1).replace(/\\([\\'])/g, "$1");
4   }
5
6   /** `string: String;` returns fct */
7   string (s) { return () => this._unq(s); }
8
9   /** `input: 'input' String String;` [replace] returns fct */
10  input (i, prmpt, dflt) {
11    return () => prompt(this._unq(prmpt), this._unq(dflt));
12  }
```

Finally, the `string` and `input` actions have to convert a single-quoted string literal in the program into the corresponding string value. A helper method `_unq()` removes the outer quotes and interprets backslashes if any (line 3 above). Just like the `number` action, the `string` action has to return a function which will return a constant, literal string value (line 7). `input` is overwritten to use the prompt and default strings and to not perform implicit conversion of the incoming string value into a number (line 11).

Type Checking by Interpretation

In this section the functional implementation of the little language with control structures from chapter six will be extended with strings and floating point and integer numbers. Changes to the grammar can be seen on this page…, new action methods can be seen in the method browser[3].

The best conclusion from the previous section is that syntax analysis, i.e., sentence structure, and

semantic analysis, i.e., sentence meaning, require separate mechanisms.

If variables are declared before use, type checking amounts to symbolic interpretation of a program — with types taking the place of values. Chapter six showed how to implement interpretation, i.e., immediate evaluation at compile time — at least for arithmetic expressions without variables.

Example 7/02[1] shows how to interpret types while translating a program into JavaScript functions. Variables have to be declared before use and there are integer and floating point variables with mixed arithmetic, strings with input, printing, and concatenation, strict comparisons and assignments, implicit conversion only for numerical operations, and explicit type casting operations. Here is the typed version of Euclid's algorithm‡:

```
int x, y;
x = int (input 'x' '36');
y = int (input 'y' '54');
while x <> y do
  if x > y then
    x = x - y
  else
    y = y - x
  fi
od;
print '\'gcd\': ' + x
```

- Press **new grammar** to represent and check the grammar,
- press **parse** to perform syntax and semantic analysis and generate an executable, and
- press **run** to execute the compiled program.
- Toggle **actions** and press **parse** again to see what JavaScript functions are created.

In this kind of type checking, actions immediately interpret types similar to values while generating and returning JavaScript functions to the parent rules and actions. The new action class is mostly concerned with type interpretation. Almost all function generation is inherited

from example 6/12[1] and example 7/01[1]. `TCheck02` first defines some infrastructure:

```
1  class TCheck02 extends Seven.TCheck01 {
2    /** For error messages */
3    get parser () { return this.#parser; }
4    #parser;
5    /** Symbol table, maps names to types */
6    get symbols () { return this.#symbols; }
7    #symbols;
8    /** For symbolic computing with types */
9    get stack () { return this.#stack; }
10   #stack = [ ];
11
12   constructor (parser, symbols = new Map()) {
13     super();
14     this.#parser = parser;
15     this.#symbols = symbols;
16   }
17
18   /** Returns type of name, message if undefined */
19   _type (name) {
20     const type = this.symbols.get(name);
21     if (type) return type;
22     this.parser.error(name + ': undeclared');   // return undefined
23   }
```

Error messages require access to the parser (lines 3, 12, and 14 above). There have to be a map from variable names to their types (lines 6, 12, and 15) and a stack (lines 9 and 10) where actions such as `number`, `string`, etc., will push a type and actions such as `product`, `sum`, etc., will pop and combine operand types and push result types.

`_type()` expects to find and return the type declared for a variable name in the symbol table (lines 20 and 21). If not, the method displays an error message (line 22) and returns `undefined`.

If a `Name` is recognized, the `name` action determines the type, pushes it onto the compile time type stack (line 3 below), and delegates to the superclass(es) to generate an appropriate JavaScript function (line 4).

```
1    /** `name: Name;`  returns `fct:_type(name)` */
2    name (name) {
3      this.stack.push(this._type(name));
4      return this.parser.call(this, super.name, name);
5    }
```

This pattern is typical for `int` — which pushes `'int'` and delegates to `number` — and `string` and `input` — which push `'string'`. The `float` action has to generate a new function:

```
1    /** `float: Float;` returns `fct:float` */
2    float (float) {
3      this.stack.push('float'); return () => parseFloat(float);
4    }
```

The first heavy lifting happens when a `term` contains an explicit conversion (line 12 below):

```
1   // term: int | float | string | name | input
2   //       [0]
3   //     | 'int' term | 'float' term | 'string' term
4   //       [0]   [1]
5   //     | '(' sum ')';
6   //           [1]
7   term (...val) {
8     switch (val.length) {
9     case 1: return val[0];
10    case 3: return val[1];
11    }
12    const to = val[0], from = this.stack.pop();
13    this.stack.push(to);
14    return this._cast(val[1], from, to);
15  }
```

In this case the `term` action is called with two arguments, namely the target type `val[0]` which is stored as `to` and the function `val[1]` which will produce the value which may have to be converted. The type of this value `from` is popped off the type stack (line 12 above). The result type of the explicit conversion will be `to` and it is pushed onto the type stack (line 13). Conversions might be applied elsewhere; therefore, it is the job of a new method `_cast()` to compose a suitable JavaScript function:

```
1   /** Converts `fct:from` into `fct:to` if needed */
2   _cast (fct, from, to) {
3     switch (`${to} <- ${from}`) {
4     default:
5       this.parser.error('impossible cast from', from, 'to', to);
6
7     case 'int <- int': case 'float <- float':
8     case 'string <- string': case 'float <- int':
9       return fct;
10
11    case 'int <- float':
12      return memory => Math.round(fct(memory));
13
14    case 'int <- string':
15      return memory => parseInt(fct(memory), 10);
16
17    case 'float <- string':
18      return memory => parseFloat(fct(memory));
19
20    case 'string <- int': case 'string <- float':
21      return memory => String(fct(memory));
22    }
23  }
```

Nothing needs to be done in four of the nine possible cases (lines 7 to 9 above) because three

cases are identities, and integers and floating point values, both, are represented as numbers in the implementation language.

The five other cases are handled by composing four new functions which involve `Math.round()`†, `parseInt()`†, `parseFloat()`†, and `String()`†. The latter, again, works for both, integers and floating point values.

Arithmetic operations allow integers and floating point arguments, e.g.:

```
1   /** `subtract: '-' product;` returns `fct:int|float` */
2   subtract (_, right) {
3     const [ l, r ] = this.stack.splice(-2, 2);
4     if (l == 'string' || r == 'string')
5       this.parser.error("cannot apply '-' to string");
6     this.stack.push(l == 'int' && r == 'int' ? 'int' : 'float');
7     return this.parser.call(this, super.subtract, _, right);
8   }
```

The types of the left and right operand are on the stack (line 3 above). Subtraction and most other operations flag strings as errors (lines 4 and 5) and replace the operand types by the result type — `'int'` only if all operands are integers (line 6). Division always produces `'float'`. Code generation is always delegated to the superclass (line 7).

```
1   /** `add: '+' product;` returns `fct:string|int|float` */
2   add (_, right) {
3     const [ l, r ] = this.stack.splice(-2, 2);
4     this.stack.push(l == 'string' || r == 'string' ? 'string' :
5         l == 'int' && r == 'int' ? 'int' : 'float');
6     return this.parser.call(this, super.add, _, right);
7   }
```

The operator `+` specifies concatenation if a string is involved and addition otherwise — this determines the result type (lines 4 and 5 above). This makes no difference for the generated function (line 6) because JavaScript has the same semantics.

```
1   /** `cmp: sum rel;` returns fct */
2   cmp (sum, rel) {
3     const [ l, r ] = this.stack.splice(-2, 2);
4     if ((l == 'string' || r == 'string') && l != r)
5       this.parser.error('must compare strings to strings');
6     return this.parser.call(this, super.cmp, sum, rel);
7   }
```

`cmp` flags comparisons between strings and numbers (lines 4 and 5 above). The action discards the operand types because the parent rule is a statement and can only be `select` or `loop` which need no additional checking.

Assignment insists that variable and value type are equal (lines 3 to 6 below) and pops the value's

type, clearing the stack at the statement level (line 4):

```
1   /** `assign: Name '=' sum;` returns fct */
2   assign (name, _, sum) {
3     const type = this._type(name),
4       r = this.stack.pop();
5     if (type != r)
6         this.parser.error(`assigning ${r} to ${type} ${name}`);
7     return this.parser.call(this, super.assign, name, _, sum);
8   }
```

`print` accepts only string values and pops all their types:

```
1   /** `print: 'print' sums;` returns fct, string arguments only */
2   print (p, sums) {
3     if (! this.stack.splice(- sums.length, sums.length).
4             every(type => type == 'string'))
5       this.parser.error('can only print strings');
6     return this.parser.call(this, super.print, p, sums);
7   }
```

As an alternative, `printAny()`, is an action for the same rule which implicitly converts all arguments to strings using the `_cast()` method described above:

```
1    /** `printAny: 'print' sums;` returns fct */
2    printAny (p, sums) {     // implicitly casts non-string arguments
3      sums.reverse().map((sum, n) => {         // check each argument
4        const type = this.stack.pop();      // requires reverse order
5        if (type != 'string') {
6          sum = this._cast(sum, type, 'string');  // apply conversion
7          puts(`print argument ${sums.length - n} was ${type}`);
8        }
9        return sum;                              // returns fct:string
10     }).reverse();
11     return this.parser.call(this, super.print, p, sums);
12   }
```

`sums`, the list of functions computing the values to be printed, is reversed so that the order matches the type stack top-down. For each function a type is popped off the stack (line 4 above). If the type is not `'string'` already an implicit conversion is applied (line 6). Finally, the resulting list of functions is reversed again (line 10) just before delegation to `print()` in the superclass (line 11) to generate code for the actual print operation.

Essential for type checking in example 7/02[1] is that variables have to be declared before they are used. The rule for `prog` is revised to include declarations before statements (line 1 below). The

action has to be overwritten to send the function generated for `stmts` to the superclass (line 2):

```
1    /** `prog: [{ decl ';' }] stmts;` returns executable */
2    prog (many, stmts) { return this.parser.call(this, super.prog, stmts); }
3
4    /** `decl: type Name [{ ',' Name }];` */
5    decl (type, name, many) {
6      [ name ].concat(many ? many[0].map(list => list[1]) : []).
7        forEach(name => {
8          if (this.symbols.has(name))
9            this.parser.error(`${name}: duplicate`);
10         this.symbols.set(name, type[0]);
11       });
12   }
13 }
```

Each name in each declaration is checked to catch duplicate declarations (line 8 above) and it's type is recorded in the symbol table (line 10).

In example 7/02[1]

- Press **new grammar** to represent and check the grammar.
- Replace + in the last line in the `program` area with a comma so that there are two arguments to `print` and
- press **parse** to perform syntax and semantic analysis to determine that `print` only permits string arguments.
- Replace the name `print` in two(!) rules by `printAny` in the `grammar` area to use the alternative action,
- press **new grammar** to represent and check the changed grammar, and
- press **parse** to perform syntax and semantic analysis again to see that there now is an implicit conversion.

Semantic analysis is done. It should be noted that the little language is more restrictive than the implementation language but benefits from some shortcuts due to dynamic typing and powerful implicit conversions in JavaScript.

All error messages show the typical dilemma: an action method could `throw` an error and forcibly terminate syntax analysis, but then only one error would be detected. Usually, it is better to continue syntax analysis with incorrect code generation and mark the executable as defective. Error reporting through the parser counts errors and reports them at the end of recognition.

It is also quite apparent that implementing semantic analysis points out design choices that were made for the little language. They may be a matter of taste, but they can surprise or annoy users endlessly...

Example 7/03[1] uses the same compiler `Seven.TCheck02`[2] and contains statements with a number of semantic errors.

- Press **new grammar** to represent and check the grammar,
- press **parse** to perform syntax and semantic analysis.
- Repair the semantic errors, e.g., by deleting the offending statements, and recompile with **parse**.

- Finally, press **run** to execute the compiled program. The last two lines of output should be 3 and 4.

Functions

In this section the little language with control structures will be extended with functions which can be called recursively. Changes to the grammar can be seen on this page…, new stack machine and action methods can be seen in the method browser[3].

Just like control structures, functions depend either on language features of the implementation language, i.e., functions in JavaScript to which the source language functions can be mapped, or on branch instructions which capture return information. This section looks at an implementation of functions without parameters for the stack machine.

Example 7/04[1] contains two mutually recursive functions which together implement Euclid's algorithm‡ without an explicit loop:

```
1  var x, y;
2  function fa;
3  function fb;
4
5  function main begin
6    x = input 36; y = input 54;
7    fa; print x;
8    x = 90; y = 72;
9    return fb; print 99
10 end;
11
12 function fa begin
13   if x = y then return x fi;
14   if x > y then x = x - y else y = y - x fi;
15   fa = fb
16 end;
17
18 function fb begin
19   if x = y then fb = x else
20     if x > y then x = x - y else y = y - x fi;
21     fb = fa
22   fi
23 end;
```

The program starts in `main`, inputs decimal values for `x` and `y`, calls the algorithm, and prints the result (lines 6 and 7 above). It then sets `x` and `y` to other values, calls the algorithm again, and returns the result. `99` will not be printed (line 9).

- Press **new grammar** to represent and check the grammar,
- press **parse** to create the executable,
- press **run** to execute, and
- input two arbitrary positive integers for the first call to the algorithm.

The second output (line 3 below) is the greatest common divisor of 90 and 72. It is returned from

`main` (line 9 above) and printed by the caller. After execution memory contains the values `18` for the variables `x` and `y` (line 4).

```
1  > run()
2  0
3  18
4  [ 18 18 ]
```

A program consists of global variable definitions followed by function definitions; one of the functions must be called `main`. When the executable is run `main` is called implicitly and the result is printed.

Functions must be declared before they can be used, i.e., mutual recursion requires a declaration of `function fb` (line 3 in the program above) before it can be used in the definition of `function fa` (lines 12 to 16). If `function fb` is declared it must later be defined (lines 18 to 23).

`return` terminates function execution (lines 9 and 13) and can arrange for a result to be delivered to the point of call. The result value of a function is either specified explicitly for `return` or it is assigned to the function name, or it is zero. Technically, there is no need for a `return` statement because a result value can be assigned to the function name within the function body (lines 15, 19 and 21); however, `return` can be convenient for early termination (e.g., line 13 for `fa`).

Functions can be called in the context of expressions or as statements. If a function is called in a statement (line 7) the result value is discarded; otherwise the result value can be used in an expression. E.g., `return fb` in the definition of `main` (line 9) first makes a call to `fb` and then terminates the (implicit) call to `main` and returns the result of the call to `fb` as result of the call to `main`.

This step in the evolution of the little language requires a bit of semantic analysis:

- Variables are global and need not be declared.
- A variable cannot be called as if it were a function.
- Functions must be declared or defined before they can be used.
- A function cannot be declared after it has already been defined.
- Assignment to a function name is only allowed in the function body.

Grammar Modifications

A program consists of an optional list of global variables followed by one or more function declarations and definitions, each terminated by a semicolon. A function declaration has no body (line 5 below):

```
1  prog: [ vars ] funs;
2  vars: 'var' names ';';
3  names: Name [{ ',' Name }];
4  funs: { fun };
5  fun: head [ 'begin' stmts 'end' ] ';';
6  head: 'function' Name;
```

There are only two more extensions to the grammar:

```
1   stmt: assign | print | return | loop | select;
2   assign: Name [ '=' sum ];
3   return: 'return' [ sum ];
```

A procedure call has to be recognized as an assignment where the actual assignment operation is omitted (line 2 above). `return` is added as a new statement. All other changes are semantic in nature, i.e., they concern the fact that a `Name` can refer to a function or a variable.

Machine Instructions

The stack machine needs three new instructions:

```
1  class Machine04 extends Six.Machine11 {
2    /** `stack: ... -> ... old-pc | pc: addr` */
3    Call (addr) {
4      return memory => (memory.push(memory.pc), memory.pc = addr);
5    }
6    /** `stack: ... old-pc -> ,,, 0 old-pc` */
7    Entry (memory) {
8      memory.splice(-1, 0, 0);
9    }
10   /** `stack: ... old-pc -> ... | pc: old-pc` */
11   Return (memory) {
12     memory.pc = memory.pop();
13   }
14   /** `stack: ... x old-pc result -> ... result old-pc result` */
15   ReturnValue (memory) {
16     memory.splice(-3, 1, memory.at(-1));
17   }
18 }
```

Functions are control structures and require a machine instruction which remembers the address to return to. Recursive calls are possible as long as this address is deposited on the stack. `Call(addr)` pushes the stack machine's current program counter onto the stack and places `addr` into the program counter (line 4 above).

`Return` undoes the effect of `Call`: It expects the function result value on top of the stack and the return address just below. It removes the return address from the stack and assigns it to the program counter (line 8). This leaves the function result value on top of the stack.

`ReturnValue(memory)` expects the function result value on top of the stack, just above the return address and the slot for the function result. It copies the result value from the top of the stack into the result slot (line 12). Depending on context the value may then have to be popped off the stack.

Infrastructure

The symbol table now contains descriptions of variables and functions; therefore, a separate counter `size`[2] is needed to allocate variable addresses (lines 3 and 4 below):

```
1  class Functions04 extends Six.Control11 {
2    /** Manages next (global) variable address */
3    get size () { return this.#size; }
4    set size (size) { this.#size = size; }
5    #size = 0;
6
7    /** Describes current function */
8    get funct () { return this.#funct; }
9    set funct (sym) { this.#funct = sym; }
10   #funct;
```

`funct`[2] contains the symbol description of the current function (lines 8 and 9 above). Getters and setters are used because they can be overwritten in subclasses and they cannot be mistaken for action methods.

Symbol descriptions will change as the little language evolves. All of them contain a name and a reference to the actions class which uses them, i.e., they are best represented by inner classes which have `this.Symbol`[2] as the common base class:

```
1    get Symbol () { return this.#Symbol ??= class {
2        owner;                                  // surrounding class
3        name;                                // variable/function name
4
5        constructor (owner, name) {
6          this.owner = owner; this.name = name;
7        }
8      };
9    }
10   #Symbol;
```

`this.Var`[2] is the class to represent variables. This kind of symbol description must be created

with a new global address (line 5 below):

```
1   get Var () { return this.#Var ??= class extends this.Symbol {
2       addr;                                          // memory address
3
4       constructor (owner, name, addr) {
5         super(owner, name); this.addr = addr;
6       }
7       load () {                          // generate load instruction
8         this.owner.machine.gen('Load', this.addr);
9       }
10      storeOk () { return true; }       // always permit assignment
11      store () {                        // generate store instruction
12        this.owner.machine.gen('Store', this.addr);
13      }
14      toString () { return `${this.name} at ${this.addr}`; }
15    };
16  }
17  #Var;
```

The class has a few convenience methods to create instructions to copy the variable value to the stack (line 8 above) and back (line 12) and to format information about the symbol (line 14).

`this.Fun`[2] is the class to represent functions and has similar convenience methods for code

generation:

```
  get Fun () { return this.#Fun ??= class extends this.Symbol {
      start = false;                  // start address, not yet set
      calls = [];                    // forward references to entry
      returns = [];                   // forward references to exit

      entry () { // defines start address, arranges slot for result
        this.start = this.owner.machine.gen('Entry') - 1;
      }
      undo () {               // ends a declaration, undoes entry()
        this.owner.machine.code.length = this.start;
        this.start = false;
      }
      call () {    // create Call or save address for slot for Call
        if (typeof this.start == 'number')
          this.owner.machine.gen('Call', this.start);
        else
          this.calls.push(this.owner.machine.code.push(null) - 1);
      }
      return () {           // create slot for Branch, save address
        this.returns.push(this.owner.machine.code.push(null) - 1);
      }
      storeOk () {                     // ok to store result value?
        if (this == this.owner.funct) return true;
        this.owner.parser.error(`${this.name}: ` +
            `assigned to outside function`);
        return false;
      }
      store () {              // store top of stack as result value
        this.owner.machine.gen('ReturnValue');
      }
      end () {          // resolve calls and returns if any, exit()
        const call = this.owner.machine.ins('Call', this.start);
        this.calls.forEach(c => this.owner.machine.code[c] = call);
        this.calls.length = 0;

        const br = this.owner.machine.ins('Branch',
          this.owner.machine.code.length);
        this.returns.forEach(c => this.owner.machine.code[c] = br);
        this.returns.length = 0;
        this.exit();
      }
      exit () {      // generates code to return from function call
        this.owner.machine.gen('Return');
      }
      toString () {
        return `function ${this.name} start ${this.start}`;
      }
    };
  }
  #Fun;
```

A function result value is zero by default; therefore, the slot for the function result is created at the beginning of a function by pushing zero onto the stack (line 7 above).

Function declarations and definitions look more or less alike and both will cause a call to `entry()` which a declaration can undo by calling `undo()` (line 9).

Because of function declarations a function can be called before it is defined; therefore, a method `call()` might create an empty slot for a `Call` instruction (line 17) and a method `end()` is called when the function definition is complete to fill these slots (lines 32 to 34).

Similarly, `return()` directs all function returns to the end of the function code (line 20) and `end()` inserts the necessary forward branch instructions (lines 36 to 39).

`storeOk()` checks if a function result is assigned within the current function definition (line 23) and displays an error message if not.

`store()` codes the `ReturnValue` instruction which sets the function result (line 29).

`end()` delegates to `exit()` to code the `Return` instruction which ends function execution (line 43).

All of these methods are designed to be overwritten if function setup and cleanup have to be changed.

All three symbol classes are defined using getters so that they can be overwritten in action subclasses. The result values of the getters are memoized‡ using the `??=` operator† so that tests like `symbol instanceof this.Fun` produce the intended result.

`_find()`[2] and `_dcl()`[2] manage the symbol table, `_alloc()`[2] allocates addresses for new variables:

```
1   /** Returns symbol description for name, if any */
2   _find (name, report) {
3     const sym = this.symbols.get(name);
4     if (report && !sym) this.parser.error(`${name}: undefined`);
5     return sym;
6   }
7
8   /** (Re-)defines and returns `sym`, cannot be undefined */
9   _dcl (sym, report) {
10    if (report && this.symbols.get(sym.name))
11      this.parser.error(`${sym.name}: duplicate`);
12    this.symbols.set(sym.name, sym);
13    return sym;
14  }
15
16  /** Returns new `Var` at next global address. */
17  _alloc (name) { return new this.Var(this, name, this.size ++); }
```

`_find()`[2] returns a symbol description for a name and can optionally report that the symbol is undefined. `_dcl()`[2] enters a symbol description into the symbol table and can optionally report that this is a duplicate definition.

Actions

Given functions, `Name` can refer to either a function or a variable; therefore, actions have to be

created or revised for all rules involving `Name`.

```
1    // names: Name [{ ',' Name }];
2   names (name, some) {
3      const dcl = name => this._dcl(this._alloc(name), true);
4      dcl(name);
5      if (some) some[0].forEach(list => dcl(list[1]));
6      return 1 + (some ? some[0].length : 0);
7   }
```

`names` creates variable descriptions for a list of names and uses `_dcl()`[2] to enter them into the symbol table and report duplicates.

```
1   // name: Name;
2   name (name) {
3     const sym = this._find(name, true);
4     if (sym instanceof this.Fun) sym.call();
5     else if (sym) sym.load();
6   }
```

In an expression, a `Name` can trigger a function call (line 4 above) which will eventually leave the result on top of the stack, or it can cause the current value of a variable to be pushed onto the stack (line 5). Code generation for either case is handled by the symbol description as discussed above. If a name is undefined `_find()`[2] reports an error and no code is generated (line 5).

```
1    // assign: Name [ '=' sum ];
2    assign (name, sum) {
3      const sym = this._find(name, true);
4      if (sym) {
5        if (sym instanceof this.Var)
6          if (sum && sym.storeOk()) sym.store();   // variable = sum
7          else this.parser.error(`${name}: cannot call a variable`);
8        else if (!sum) sym.call();                  // function call
9        else if (sym.storeOk()) sym.store();       // function = sum
10       this.machine.gen('Pop');                     // clear stack
11     }
12   }
```

In an assignment statement `Name` can refer to a function or a variable. Code is only generated if `_find()`[2] provides a description and does not report an error (line 4 above). If it is a variable (line 5) and if there is something to assign `sym.store()` is called to generate the `Store` instruction (line 6). Otherwise there is an error because a variable cannot be called (line 7). If it is a function name and nothing to assign `sym.call()` takes care of the `Call` instruction (line 8). If there is something to assign `sym.store()` can be called if the name refers to the current function, i.e., if the assignment is in the function body (line 9); otherwise `sym.storeOK()` reports the assignment to the function name as an error. Finally, the stack has to be popped to discard the assigned value

or the function result (line 10).

```
1   // return: 'return' [ sum ];
2   return (r, sum) {
3     if (sum && this.funct.storeOk())
4       (this.funct.store(), this.machine.gen('Pop'));
5     this.funct.return();
6   }
```

If a `return` statement includes a value, the current function's `store()` method will assign it as a function result and then the value has to be popped off the stack (line 4 above). Function termination is arranged by the function description's `return()` method (line 5).

Functions can be declared so that they can be called before they are defined:

```
1  funs: { fun };
2  fun: head [ 'begin' stmts 'end' ] ';';
3  head: 'function' Name;
```

`head` is recognized before `fun` and the function body if any, i.e., the action for `head` happens first:

```
1   // head: 'function' Name;
2   head (f, name) {
3     let sym = this._find(name);
4     if (! (sym instanceof this.Fun)) {
5       if (sym instanceof this.Var)
6         this.parser.error(`${name}: used as variable and function`);
7       sym = this._dcl(new this.Fun(this, name));
8     }
9     if (typeof sym.start == 'number') {
10      this.parser.error(`${name}: duplicate`);
11      sym = this._dcl(new this.Fun(this, name));          // patch
12    }
13    sym.entry();               // generate code for function entry
14    return this.funct = sym;                        // in function
15  }
```

First there is some semantic analysis:

- A variable name cannot be a function name (line 5 above); to allow further processing it is redeclared as a function (line 7).
- An undefined name is declared as a function (line 7).
- A function cannot be *defined* more than once (line 9) — `sym.start` will contain an address once there is code for the function.

`head` may be the beginning of a function definition; therefore `sym.entry()` is called to generate the code to start the function (line 13). Finally, the current function is posted in `this.funct` and returned (line 14).

Once code generation for the function body is complete — or immediately if there is no body —

the `fun` action performs some cleanup:

```
1   // head [ 'begin' stmts 'end' ] ';';
2   fun (head, opt, _) {
3     if (opt) head.end();            // function definition: wrap up
4     else head.undo();   // function declaration: discard entry code
5     this.funct = null;                          // not in function
6   }
```

`head` provides access to the function description. If there is a function body the `end()` method described previously inserts `Call` and `Branch` instructions and generates the code to return from the function call (line 3 above). Otherwise this was only a function declaration and `undo()` removes the effect of `entry()` in the `head` action (line 4). Finally, `this.funct` is set to `null` to signal that code generation has left the body of the function (line 5).

All actions for functions and their consequences are in place. Expressions can be retained from the previous (untyped) version of the little language, i.e., from example 6/11[1]. The `prog` action performs some final cleanup and generates the stack machine:

```
1   // prog: [ vars ] funs;
2   prog (v, f) {
3     const main = this._check_defs(this.symbols),  // flag undefined
4       startAddr = this.machine.code.length,      // at startup code
5       trace = this._find('trace'),  // does variable 'trace' exist?
6       traceAddr = trace instanceof this.Var ? trace.addr : false;
7     if (main) this._startup(main);         // generate call to main
8     else this.parser.error('main: undefined');
9     if (traceAddr !== false) {                // if 'trace' exists
10      puts(this.machine.toString());                  // show code,
11      this.symbols.forEach(s => puts(s.toString()));    // symbols,
12      puts('stack starts at', this.size);   // variable memory size
13      if (main) puts('execution starts at', startAddr);
14    }
15    return this.machine.run(this.size, startAddr, traceAddr);
16  }
```

Just as before tracing depends on the existence of a symbol named `trace` (line 5 above) but `trace` has to be a variable (line 6). If so, the generated code, the symbols, and the initial memory size are shown (lines 9 to 14). The stack machine generator creates the executable as before (line 15); however, functions definitions have to be checked (line 2) and the initial call to `main` has to be generated (line 7) which will determine the start address of the executable.

`_check_defs()`[2] checks all symbols to make sure that all functions have a defined start address

and it returns the description for `main` if it is a defined function:

```
  _check_defs (map) {
    let main = undefined;
    map.forEach(sym => {
      if (sym instanceof this.Fun)
        if (typeof sym.start != 'number')
          this.parser.error(`${sym.name}: undefined function`);
        else if (sym.name == 'main')
          main = sym;
    });
    return main;
  }
```

If `main` was defined, `startup()`[2] generates instructions to call the function and print the result and returns the start address of the instructions.

```
  _startup (main) {
    this.machine.gen('Call', main.start);     // call main function
    this.machine.gen('Print', 1);                  // print and pop
  }
}
```

Tracing was introduced in chapter six. Check out how a function is called and how it returns a result in example 7/04[1]:

- Press **new grammar** to represent and check the grammar,
- press **parse** to create the executable, and
- repeatedly press **1** to execute one step at a time.
- Alternatively, add statements `trace = -1`, `trace = 1`, or `trace = input` to turn tracing off and on, or
- add `trace` to the list of global variables (which are initialized to zero) to trace everything.

Example 7/05[1] demonstrates 13 semantic errors:

```
1  var a, a;
2  function undefined;
3  function a;
4  function g begin g = 1 end;
5  function f begin f = 2; g = 3; v = 4 end;
6  function v;
7  function f begin
8    return;
9    v = 5; v; undef; f = 6
10 end;
11 function g;
```

- duplicate global variable names (in line 1 above),
- a name `a` which is defined as a variable and forward declared as a function (line 3),

- a function names `g` and `v` which are assigned to outside the function definitions (lines 5 and 9),
- an undefined variable name 'v' (line 5),
- a duplicate function definition (line 7),
- a function `undef` which is used but not declared (or defined) (line 9),
- forward declarations for functions `a`, `v`, and `undefined` which are never defined (lines 2, 3, and 6),
- a forward declaration for `g` *after* the function has been defined (line 11),
- and no definition for a `main` function.

Local Variables

In this section the little language with functions will be extended with local variables. Changes to the grammar can be seen on this page…, new stack machine and action methods can be seen in the method browser[3].

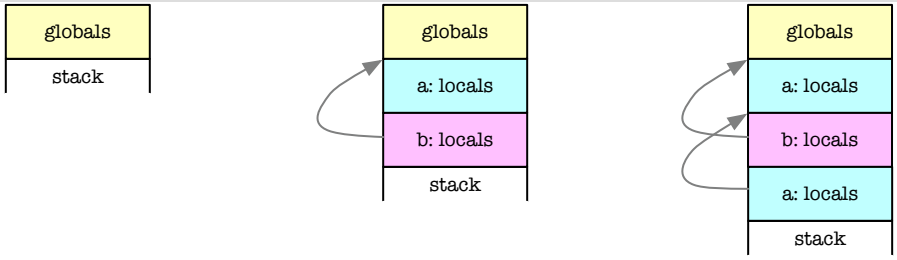
Frames

Real functions have parameters and local variables. This complicates the code when entering and exiting a function:

- At entry, argument values are connected to parameter names and memory is allocated for local variables.
- At exit, cleanup is required so that `Return` only leaves a result behind — this is why a `return` inside a function body is coded as a branch to the exit code.

Recursion complicates access to parameters and local variables even more because the values exist once for each level of recursion and the names refer to the latest activation of each function.

Rather than just stacking a return address and a result value slot as in example 7/04[1], now frames, also called activation records‡, have to be stacked which additionally contain argument values and local variables:



Memory use evolves:

- Initially there are global variables.
- Once function `a` is called and itself calls function `b` there are two *locals* frames.
- If there is a recursive call from `b` to `a` there are three *locals* records.

The *locals* are chained on a *dynamic link*‡ so that the exit code and `Return` can clean them up in

reverse order.

Addressing also gets more complicated:

- Global variables have *absolute addresses*, determined at compile time.
- Local variables have *relative addresses*, i.e., offsets within their frame, which have to be added to the *base address* where the frame begins at run time. The offset is determined at compile time but — as the graphic indicates — the base address depends on the call history.

In summary, if *parms* is the number of parameters of a function, a frame contains the following:

offset	**content**
`+0`...	argument values for the parameters, if any
+*parms*	the return address for the function
+*parms* `+1`	a slot for the dynamic link, i.e., the address of the frame of the caller
+*parms* `+2`	a slot for the function result
+*parms* `+3`...	values for the local variables

The `Memory`[2] class from example 7/04[1] is extended with a *frame pointer* register, represented as a property `.fp` (line 4 below), and `toString()` is overwritten to display the stack as a sequence of frames:

```
1  class Machine06 extends Seven.Machine04 {
2    get Memory () {
3      return this.#Memory ??= class extends super.Memory {
4        fp = 0;                            // global frame starts at 0
5        frames = [ 0 ];   // toString(): list of number of parameters
6
7        toString () {
8          let fp = this.fp,                   // begin of (top) frame
9            to = this.length;                   // end of (top) frame
10         return this.frames.map((parms, n) => {
11           try {
12             return `${fp}:[ ${this.slice(fp, to).
13               map(slot => this.mapSlot(slot)).join(' ')} ]`;
14           } catch (e) { throw e;                // shouldn't happen
15           } finally {
16             to = fp;             // end and begin of previous frame
17             if (n == this.frames.length-1) fp = 0;       // globals
18             else fp = this[fp + parms + 1];       // previous frame
19           }
20         }).reverse().join(' ');
21       }
22
23       mapSlot (slot) {                   // hook to interpret slots
24         return typeof slot == 'undefined' ? 'undefined' : slot;
25       }
26     };
27   }
28   #Memory;
```

The frames can be displayed by following the dynamic link which at this point depends on the number of parameters in each frame. At this point this is stored in a list represented as a property `.frames[]` (line 5 above).

The top frame on the stack extends from `.fp` to the current end of the memory array (lines 8 and 9). In a loop over the frames (line 10) a frame is accessed using `slice()`† (line 12) and each slot is represented as a string using a function `mapSlot()` which might be overwritten (lines 13 and 24).

Machine Instructions

Argument values are pushed onto the stack before a function is called. The `Entry` instruction at the beginning of each function has to initialize the frame as described above, given that `parms` is the number of parameters of the function and `size` is the memory requirement of the new frame:

```
1   /** `stack: ... arguments old-pc
2       -> ... arguments old-pc old-fp result locals` */
3   Entry (parms, size) {
4     return memory => {
5       const locals = size - parms - 3,           // local variables
6         fp = memory.length - parms - 1;         // new frame's base
7       memory.push(memory.fp, 0);              // push old-fp, result
8       if (locals)                    // push local variables, if any
9         memory.push(... Array(locals).fill(0));
10      memory.fp = fp;                            // new dynamic link
11      memory.frames.unshift(parms);  // push frames stack for trace
12    };
13  }
```

The frame contains a slot for the function result, initialized to `0` (line 7 above). This slot can be treated just like a local variable but for the fact that it is only assigned to.

The `Exit` instruction at the end of each function reverts to the previous frame pointer (line 6 below) and removes all but the return address and result value from the stack as the `Return` instruction expects it (lines 7 and 8):

```
1   /** `stack: ... arguments old-pc old-fp result locals
2       -> ... result old-pc` */
3   Exit (parms) {
4     return memory => {
5       const fp = memory.fp;                          // current frame
6       memory.fp = memory[fp + parms + 1];   // restore dynamic link
7       memory.splice(fp, Infinity,  // pop frame, push result old-pc
8         memory[fp + parms + 2], memory[fp + parms]);
9       memory.frames.shift();             // pop frames stack (trace)
10    };
11  }
```

`return` statements within the body of the function are coded as branches to the `Exit` instruction.

Parameters and local variables have to be addressed relative to the frame pointer using two new

instructions:

```
  /** `stack: ... -> ... frame[addr]` */
  LoadFP (addr) {
    return memory => memory.push(memory[memory.fp + addr]);
  }

  /** `stack: ... val -> ... val | frame[addr]: val` */
  StoreFP (addr) {
    return memory => memory[memory.fp + addr] = memory.at(-1);
  }
}
```

Grammar Modifications

To enable semantic analysis in the actions of syntax analysis, functions have to be declared before use, and a design decision is necessary to determine the scope of declarations in a way that is compatible with the activation record stack:

- A function should not have access to the local variables and parameters of a different function because its frame might not exist.
- Functions may have access to global variables.
- Local names may shadow‡ global names.

For example, "declare before use" for variables could be done prior to functions and at the beginning of function bodies, and missing declarations could be considered errors:

```
prog: GLOBAL { fun };
fun: head [ 'begin' [ LOCAL ] stmts 'end' ];
head: 'function' Name '(' [ parms ] ')';
parms: Name [{ ',' Name }];
```

Alternatively, in the absence of `GLOBAL` and `LOCAL` above, unknown names could be resolved in the closest scope, i.e., enclosing function.

In this case there could be `GLOBAL` import declarations so that functions could share some global variables:

```
fun: head [ 'begin' [ GLOBAL ] stmts 'end' ];
```

As a third alternative, unknown names could be resolved in the global scope, i.e., visible to any function unless shadowed‡ by parameter names.

In this case there could be `LOCAL` definitions as above so that functions can have local variables in addition to parameters.

The design choice among these alternatives depends on perceived convenience vs. the wish to prevent errors due to scope violations.

The little language in example 7/06[1] requires strict "declare before use" and uses global and local

declarations:

```
1  prog: [ vars ] funs;
2  vars: 'var' names ';';
3  names: Name [{ ',' Name }];
4  funs: { fun };
5  fun: head parms [ block ] ';';
6  head: 'function' Name;
7  parms: '(' [ names ] ')';
8  block: 'begin' [ vars ] stmts 'end';
```

`names` before all functions define global variables (lines 1 to 3 above), `names` in a `block` define local variables (line 8), and `names` in `parms` declare parameter names local to the function (line 7).

`begin` and `end` delimit a `block` (line 8) to avoid an ambiguity between a semicolon separating statements and a semicolon terminating a function definition.

On the other hand, the semicolons terminating function and variable declarations are not required to disambiguate — they are punctuation in the spirit of `end`, `fi`, and `od`.

A forward declaration omits the function body (line 5). The number of parameters in a forward declaration has to match the function definition, the parameter names need not.

A `Name` can be assigned to or called with function arguments in a statement, or it can be referenced and called with function arguments in an expression. The common aspects can be emphasized by changes to the grammar:

```
1  assign: symbol action;
2  action: store | call;
3  store: '=' sum;
4  call: args;
5  args: '(' [ sums ] ')';
6
7  name: symbol [ args ];
8  symbol: Name;
```

In addition to distinguishing the use of variable and function names, semantic analysis now has to check that the numbers of parameters and argument values agree and that a forward declaration of a function specifies as many parameters — irrespective of the actual names — as the function definition.

Example 7/07[1] contains all semantic errors:

```
1  var f, dup, dup;
2  function undefined ();
3  function a ();
4  function a (dup) begin var dup; dup = 1 end;
5  function a (x) begin var y; y = 1 end;
6  function b () begin b = 2 end;
7  function f () begin f = 3; g = 4 end;
8  function main () begin
9    a();
10   a = 5;
11   undef();
12   dup()
13 end;
```

- a global variable `dup` defined more than once (line 1),
- a function `a` declared forward with no parameters but defined with one (lines 3 and 4),
- a local variable `dup` with the same name as a parameter (line 4),
- a second definition for the function `a` (line 5),
- a function `f` with the same name as a global variable (line 7),
- an undefined variable `g` (line 7),
- a call to `a` where the number of arguments and parameters differ (line 9),
- an assignment to the function name `a` outside the function definition (line 10),
- an undefined function `undef` (line 11),
- a function call to the variable `dup` (line 12), and
- a forward declared function `undefined` (line 2) which is not defined.

Infrastructure

Local names can shadow‡ global names, i.e., attributes such as type or address can differ. Therefore, a function needs a local symbol table (line 5 below):

```
1  class Parameters06 extends Seven.Functions04 {
2    get Fun () { return this.#Fun ??= class extends super.Fun {
3        parms;                                  // number of parameters
4        addr;              // offset of function result slot in frame
5        #locals = new Map();      // maps local names to descriptions
6        get locals () { return this.#locals; }
7        set locals (locals) { this.#locals = locals; }
8        #size = 0;                          // next address, frame size
9        get size () { return this.#size; }
10       set size (size) { this.#size = size; }
```

`Fun`[2], the class of function descriptions in the symbol table, is extended (line 2 above). It now contains the number of parameters (line 3), the relative address of the function result in the frame

(line 4), the symbol table for parameters and local variables (lines 5 to 7), and the size of the frame (lines 8 to 10). Getters and setters† are used so that further restrictions on the visibility of names can be added.

Code generation for functions has to be modified and extended:

```
1       entry () {  // defines start address, arranges slot for Entry
2         this.start = this.owner.machine.code.push(null) - 1;
3       }
4       setParms () {         // frame: parms, old-pc, old-fp, result
5         if (typeof this.parms != 'undefined'
6             && this.parms != this.size)
7           this.owner.parser.error(`${this.name} parameters: ` +
8             `previously ${this.parms}, now ${this.size}`);
9         this.parms = this.size;         // set number of parameters
10        this.size += 2;         // leave room for old pc and old fp
11        this.addr = this.size ++;          // leave slot for result
12      }
13      undo () {
14        this.locals = new Map();             // undefine parameters
15        this.size = 0;           // reset next address, frame size
16        super.undo();
17      }
18      store () {                                  // use `StoreFP`
19        this.owner.machine.gen('StoreFP', this.addr);
20      }
21      exit () {           // fills Entry, generates Exit and Return
22        this.owner.machine.code[this.start] =
23          this.owner.machine.ins('Entry', this.parms, this.size);
24        this.owner.machine.gen('Exit', this.parms);
25        this.owner.machine.gen('Return');
26      }
27      toString () {
28        const names = [];
29        this.locals.forEach(sym => names.push(sym.name));
30        return `function ${this.name} start ${this.start} ` +
31          `parms ${this.parms} size ${this.size} ` +
32          `[ ${names.join(' ')} ]`;
33      }
34    };
35  }
36  #Fun;
```

`entry()` leaves room for an `Entry` instruction (line 2 above). This instruction can only be created by `exit()` once the number of parameters and the frame size are known (lines 22 and 23).

`setParms()` will be called once the names in the parameter list, if any, have been declared. The method ensures that declarations match (line 6), records the number of parameters, leaves room for the old frame pointer and function result in the frame, and records the address of the function result (lines 9 to 11).

`setParms()` is called for both, a forward declaration and a definition of a function. `undo()` is called for a forward declaration and is extended to remove the symbol table in order to discard

the parameter names in the declaration and reset the frame size so that nothing has been allocated in the frame yet (lines 14 and 15).

The `ReturnValue` instruction is no longer used. `store()` uses a `StoreFP` instruction to copy the function result from the stack into the slot at `.addr` reserved by `setParms()` (line 19).

As before, `exit()` generates a `Return` instruction, but it is also responsible for inserting the `Entry` and `Exit` instructions which require information about parameters and frame size (lines 22 to 25). Needless to say, on real hardware there would be boilerplate instruction sequences for function entry and exit which `exit()` would still generate.

Variables can be local in a frame or global, and different instructions are needed to access their values. Therefore, `Var`[2], the class of variable descriptions in the symbol table, has to be extended, too:

```
1   get Var () { return this.#Var ??= class extends super.Var {
2       depth;                                // 0: global, else local
3
4       constructor (owner, name, addr, depth) {
5         super(owner, name, addr); this.depth = depth;
6       }
7       load () {                          // generate load instruction
8         if (this.depth)                                    // local
9           this.owner.machine.gen('LoadFP', this.addr);
10        else                                              // global
11          this.owner.machine.gen('Load', this.addr);
12      }
13      store () {                        // generate store instruction
14        if (this.depth)                                    // local
15          this.owner.machine.gen('StoreFP', this.addr);
16        else                                              // global
17          this.owner.machine.gen('Store', this.addr);
18      }
19      toString () {
20        return `${this.name} at ${this.depth ? '+' : ''}${this.addr}`;
21      }
22    };
23  }
24  #Var;
```

A new property `.depth` differentiates between local and global variables (line 2 above). It has to be set when the variable is created (lines 4 and 5). `load()` and `store()` consult the property to

create the appropriate instruction for local and global access.

```
1   /** Replace: returns new `Var` at next local/global address. */
2   _alloc (name) {
3     if (this.funct)                     // create local variable
4       return new this.Var(this, name, this.funct.size ++, 1);
5     else                                // create global variable
6       return new this.Var(this, name, this.size ++, 0);
7   }
8
9   /** Extend: checks local then global map, returns `sym` */
10  _find (name, report) {
11    let sym;
12    if (this.funct && (sym = this.funct.locals.get(name)))
13      return sym;                                          // local
14    return super._find(name, report);                     // global
15  }
16
17  /** Replace: sets innermost map, returns `sym` */
18  _dcl (sym, report) {
19    const map = this.funct ? this.funct.locals : this.symbols;
20
21    if (report && map.get(sym.name))
22      this.parser.error(`${sym.name}: duplicate`);
23    map.set(sym.name, sym);
24    return sym;
25  }
```

`funct`[2] contains the description of a function exactly while the function is compiled.

`_alloc()`[2] uses `funct`[2] to create a local variable in the frame described by `funct`[2] (line 4 above) or a global variable (line 6). In each case a new address is created from the corresponding `size` counter.

`_find()`[2] uses `funct`[2] to check for local names before global names (line 12). This implements shadowing‡.

Similarly, `_dcl()`[2] uses `funct`[2] to declare a name in a local or global symbol table (line 19) and report duplicates, if any.

```
1   _startup (main) {
2     for (let p = 0; p < main.parms; ++ p)
3       this.machine.gen('Push', 0);
4     super._startup(main);
5   }
```

Finally, `_startup()`[2] has to be extended to allocate room for the parameters of the `main()` function (lines 2 and 3 above) before the function can be called to start execution.

Statements

While the grammar now unifies the use of a `Name` in an assignment statement and in an expression, the rule changes described earlier create a problem:

```
1  assign: symbol action;
2  action: store | call;
3  store: '=' sum;
4  call: args;
5  args: '(' [ sums ] ')';
6
7  name: symbol [ args ];
8  symbol: Name;
```

An `assign` statement operates on a `symbol`, i.e., a `Name` (lines 1 and 8 above). However, there are two possible operations (line 2): either a value is assigned to a variable (line 3) or a function is called with an argument list (line 4). Similarly, in the context of an expression, either a variable value is pushed onto the stack or — again — a function is called with an argument list (line 7).

Recognition starts top-down with the start rule of the grammar, but the rules call their actions from leaf nodes such as `symbol` back to the start rule. It is predictable from the grammar in which exact order recognition will complete:

```
1  symbol: Name;
2    store: '=' sum;
3    args: '(' [ sums ] ')';
4      call: args;
5        action: store | call;
6          assign: symbol action;
7          name: symbol [ args ];
```

Given the order in which the actions are called, it easy to design what each action should do:

rule	**action**
`symbol: Name;`	find description
`store: '=' sum;`	if variable generate `Store`
`args: '(' [ sums ] ')';`	generate code to stack each value finally call function
`call: args;`	*nothing to do*
`action: store | call;`	*nothing to do*
`assign: symbol action;`	pop stack
`name: symbol [ args ];`	if variable generate `Load` else *nothing to do*

However, there is a lack of information flow while `assign` is recognized, e.g., the action for `symbol` finds a description but the action for `store` will not receive it as a parameter. Similarly, the action for `args` seems to have no access to the description of the function to be called.

Example 5/10[1] demonstrated that action methods should use the actions class as context to communicate with each other, i.e., `symbol` needs to put the description somewhere into `this` so

that `store` and `args` — one of which is called later — can access it.

`this.context`[2] has to be a stack because the `store` and `args` rules both involve `sum` which in turn could involve another `name` and `symbol`... Therefore, `this.context`[2] is implemented with `get` and `set`[†]:

```
1   get context () {
2     if (this.#contexts.length) return this.#contexts.at(-1);
3     throw 'no context';                               //can't happen
4   }
5   set context (context) {            // push a value, pop with null
6     if (context) this.#contexts.push(context);
7     else this.#contexts.pop();
8   }
9   #contexts = [];
```

The action for `symbol`[2] creates a context entry (line 5 below),

```
1   // symbol: Name;
2   symbol (name) {
3     let sym = this._find(name, true);
4     if (!sym) sym = this._dcl(this._alloc(name));          // patch
5     this.context = { symbol: sym };      // push symbol description
6     return sym;
7   }
```

The actions for `store`[2] and `args`[2] access the context entry:

```
1   // store: '=' sum;
2   store (_, sum) {
3     if (this.context.symbol.storeOk())
4       this.context.symbol.store();
5   }
6
7   // args: '(' [ sums ] ')';
8   args (lp, sums, rp) {
9     const sym = this.context.symbol,            // to apply args to
10      nargs = sums ? sums[0] : 0;                 // # of arguments
11    if (!(sym instanceof this.Fun))
12      this.parser.error(`${sym.name}: not a function`);
13    else if (nargs != sym.parms)
14      this.parser.error(`${sym.name} arguments: ` +
15        `expected ${sym.parms}, specified ${nargs}`);
16    else
17      sym.call();                                   // call function
18  }
```

Calling `storeOk()`[2] ensures that assignment is allowed and `store()`[2] generates the appropriate code (lines 3 and 4 above).

`args()`[2] ensures that a function is called (line 11) and that the numbers of parameters and argument values match (line 13) before `call()`[2] is used to generate code (line 17).

Finally, the actions for `assign`[2] and `name`[2] delete the context entry (lines 3 and 13 below):

```
1   // assign: symbol action;
2   assign (symbol, action) {
3     this.machine.gen('Pop'); this.context = null;    // pop context
4   }
5
6   // name: symbol [ args ];
7   name (sym, args) {
8     if (!args)
9       if (sym instanceof this.Fun)
10        this.parser.error(`${sym.name}: no argument list`);
11      else
12        sym.load();                          // variable reference
13    this.context = null;                           // pop context
14  }
```

`name()`[2] marks a function reference without arguments as an error (line 9 above). The grammar guarantees that the context stack remains balanced at all times.

Just One More

Next up on the way to the start rule of the grammar are function definitions. Semantic analysis and code generation for functions consist of five steps:

1. declare a function with a local symbol table,
2. declare parameters,
3. define local variables,
4. generate code for the function body, and
5. wrap up.

An action happens after all input required by the corresponding rule has been recognized. If something has to happen earlier, the rule is split into several rules to get the proper timing for the actions:

```
1  fun: 'function' Name '(' [ names ] ')'
2         [ 'begin' [ 'var' names ';' ] stmts 'end' ] ';';
```

This rule would have a single action — plus more for `names` and `stmts` — and it could not handle the timing of the five steps as described above. Instead:

```
1  head: 'function' Name;
2  names: Name [{ ',' Name }];
3  parms: '(' [ names ] ')';
4  vars: 'var' names ';';
5  block: 'begin' [ vars ] stmts 'end';
6  fun: head parms [ block ] ';';
```

This amounts to the same rule but the steps can now be assigned to different actions, here in the

order in which they will happen:

rule	action	return
head[2]	[inherited] create function description, set funct[2]	symbol
names[2]	[inherited] declare parameters in funct.locals[2]	count
parms[2]	call setParms()[2] to layout frame	
names[2]	[inherited] declare variables in funct.locals[2]	count
stmts	[inherited] generate code	
block	*nothing to do*	
fun	[inherited] call funct.end() or funct.undo()[2], pop funct[2]	

It turns out that there is only one new action: parms()[2] calls funct.setParms()[2] to record the number of parameters and layout the function's frame.

Tracing in example 7/06[1] confirms the order:

- Enter only `function main(a) begin var b; print 1 end;` into the `program` area.
- Press **new grammar** to represent and check the grammar,
- toggle **actions**, and
- press **parse**.

Here is the redacted output:

```
head('function', 'main') returns
  { function main start 0 parms undefined size 0 [  ] }
names('a', null) returns 1
parms('(', [ 1 ], ')') returns undefined
names('b', null) returns 1
...
fun(...) returns undefined
prog(...) returns (memory, steps) => StackMachine(memory, steps)
```

Everything is in place, we have reached the top-level rules of the grammar which remain unchanged from the previous version of the little language:

```
prog: [ vars ] funs;
vars: 'var' names ';';
names: Name [{ ',' Name }];
funs: { fun };
```

The actions remain unchanged, too. At the top level, funct[2] is `null`; therefore _dcl()[2] defines the variable names in the global symbol table.

Example 7/08[1] contains mutually recursive functions just as in example 7/04[1], but this time one version of Euclid's algorithm takes advantage of function parameters.

Tracing

It is often useful to see how the frames are laid out, and what they contain when the program executes, as well as the machine instructions before and during execution.

In example 7/06[1]:

- Add `var trace;` before the definition of `euclid` in the `program` area to define the global variable.
- As usual, press **new grammar** to represent and check the grammar, and
- press **parse** to compile the executable and show the extra information:

```
> run = g.parser().parse(program, actions)
0: memory => this.Entry(2, 5)(memory)
1: memory => this.LoadFP(0)(memory)
2: memory => this.LoadFP(1)(memory)
3: memory => this.Gt(memory)
  ...
30: memory => this.Entry(0, 3)(memory)
31: memory => this.Input(36)(memory)
32: memory => this.Input(54)(memory)
  ...
42: memory => this.Call(30)(memory)
43: memory => this.Print(1)(memory)
trace at 0
function euclid start 0 parms 2 size 5 [ x y ]
function main start 30 parms 0 size 3 [  ]
stack starts at 1
execution starts at 42
(memory, steps) => StackMachine(memory, steps)
```

- press **run** to execute. Global variables such as `trace` are initialized to `0`, i.e., execution will be traced:

```
> run()
0:[ 0 ]
0:[ 0 43 ] 42: memory => this.Call(30)(memory)
0:[ 0 ] 1:[ 43 0 0 ] 30: memory => this.Entry(0, 3)(memory)
0:[ 0 ] 1:[ 43 0 0 36 ] 31: memory => this.Input(36)(memory)
0:[ 0 ] 1:[ 43 0 0 36 54 ] 32: memory => this.Input(54)(memory)
0:[ 0 ] 1:[ 43 0 0 36 54 34 ] 33: memory => this.Call(0)(memory)
0:[ 0 ] 1:[ 43 0 0 ] 4:[ 36 54 34 1 0 ] 0: memory => this.Entry(2, 5)(memory)
  ...
0:[ 0 ] 1:[ 43 0 0 38 18 ] 28: memory => this.Exit(2)(memory)
0:[ 0 ] 1:[ 43 0 0 18 ] 29: memory => this.Return(memory)
0:[ 0 ] 1:[ 43 0 18 18 ] 38: memory => this.StoreFP(2)(memory)
0:[ 0 ] 1:[ 43 0 18 ] 39: memory => this.Pop(memory)
0:[ 0 43 18 ] 40: memory => this.Exit(0)(memory)
0:[ 0 18 ] 41: memory => this.Return(memory)
18
0:[ 0 ] 43: memory => this.Print(1)(memory)
0:[ 0 ]
```

The output shows the machine instructions as they are executed, preceded by the frames. First, there is the global frame (line 2 above), `main()` is called (line 3), there are two frames, and the input values `36` and `54` are pushed onto the stack (lines 5 and 6). Then the call to `euclid()`

happens (line 7) and there is a third frame which starts at location `4` in `memory` (line 8). Eventually, `euclid()` exits (line 10), the result value `18` is left on the stack (line 11), is assigned as result of `main()` and popped (lines 12 and 13), `main()` exits and leaves `18` on top of the stack (lines 14 and 15), ready to be printed.

Example 7/08[1] contains several functions and, therefore, has a much more interesting trace...

Block Scopes

In this section the little language with local variables will be extended with block scopes for the variables. Changes to the grammar can be seen on this page…, new stack machine and action methods can be seen in the method browser[3].

At compile time a local scope is pushed on top of a global scope to implement shadowing‡. So far, for the little language with local variables, shadowing‡ means that parameters and local variables owned by a function hide global variables and function names. The scope where a name is visible is either the enclosing function or the entire program.

Languages like JavaScript also have block scopes, i.e., a name is visible only within the block which contains the definition. In example 7/09[1] some variables are declared before use and can be shadowed‡ within blocks:

```
1  var trace;
2  function main (x) begin
3        var a;    print x; x = 1; print x;
4  begin var b;    print x; x = 2; print x end;
5  begin var c, x; print x; x = 3; print x; main = 99 end;
6  begin var d, x; print x; x = 4; print x end;
7                  print x
8  end;
```

The output is produced as follows:

line	code	scope	output
1	`var trace;`	global	
2	`function main (x)`	`main()` parameters	
	`begin`	`main()` block	
3	`var a; print x;`		`0`
	`x = 1; print x;`		`1`
4	`begin var b; print x;`	first inner block	`1`
	`x = 2; print x end;`		`2`
5	`begin var c, x; print x;`	second inner block	`0`
	`x = 3; print x; main = 99 end;`		`3`
6	`begin var d, x; print x;`	third inner block	`3`
	`x = 4; print x end;`		`4`
7	`print x`	`main()` block	`2`
8	`end;`	global	`99`

The first few lines of output are no surprise. `trace` is a global variable, initialized to `0` (program

line 1 above). `x` is declared as parameter of `main()` and the startup code sets the corresponding argument to `0`. `begin` starts the scope for local variables of `main()` (line 2). `x` is printed, set to `1`, and printed again (line 3).

The first inner block does not declare another `x` (line 4); therefore, the parameter `x` is not shadowed‡. It is printed, set to `2`, and printed again.

The second, parallel, inner block declares a new `x` which shadows‡ the parameter `x` (line 5). The local variable region of the frame is initialized to `0`; therefore the inner `x` prints as `0`, is set to `3` and printed again.

The third, again parallel, inner block declares another `x` which shadows‡ the parameter `x` (line 6). In a parallel block it shares memory (by location, not by name); therefore this `x` prints as `3`, is set to `4` and printed again.

Back in the scope for local variables of `main()` the parameter `x` is still `2` and is printed as such (line 7).

Deeply nested in the second inner block the result of `main()` was set to 99 (line 5) and that is printed by the startup code.

Example 7/09[1] demonstrates shadowing‡: there are exactly two memory slots for various versions of `x`. The inner two declarations for `x` are in parallel blocks; therefore, the two `x` share one memory slot.

- As usual, press **new grammar** to represent and check the grammar, and
- press **parse** to compile the executable and show the extra information provided because there is a global variable `trace`:

```
1  > run = g.parser().parse(program, actions)
2  block [ b at +5 ]
3  block [ c at +5, x at +6 ]
4  block [ d at +5, x at +6 ]
5  block [ a at +4 ]
6  block [ x at +0 ]
7  0: memory => this.Entry(1, 7)(memory)
8    ...
9  trace at 0
10 function main start 0 parms 1 frame size 7
11 stack starts at 1
12 execution starts at 36
13 (memory, steps) => StackMachine(memory, steps)
```

The global variable `trace` is at memory location `0` (line 9 above). Each block layout is displayed when recognition of the block is complete. The first inner block contains variable `b` at offset `5` in the frame for `main()` (line 2). Each of the next two inner blocks contains a variable `x` at offset `6` in the frame for `main()` (lines 3 and 4). The `main()` block contains variable `a` at offset `4` in the frame for `main()` (line 5). Finally, the parameter `x` is at offset `0` in the frame for `main()` (line 6).

All the blocks belong to the frame for `main()`. The highest offset in the frame for `main()` is `6` (lines 3 and 4); therefore, the frame size is `7` (line 10).

Note that the variables `b`, `c`, and `d` are declared first in the parallel inner blocks; therefore they share offset `5` in the frame for `main()` (lines 2 to 4).

- What changes if the order of `d` and `x` is interchanged in program line 6?

Example 7/10[1] has more names, more functions, several frames, and more scopes:

```
var a, b, c;
function f (x);
function d (d) begin print 100, a, d, x end;
function e (a) begin print 200, a, b end;
function f (a) begin
  a = 1; b = 2; c = 3; d(4); e(5);
  begin var a, b;
    a = 10; b = 20;
    print 300, a, b, c; d(40); e(50)
  end;
  print 400, a, b, c
end;
function main () begin c = 5; f(6) end;
$ /^%% grammar/,/^%% actions/ | 07/09.eg
Seven.Blocks09
```

- Prepare the grammar and compile.
- There is one error. Repair it by globally defining the missing variable and a variable `trace`.
- Recompile and explain how the listing

```
> run = g.parser().parse(program, actions)
block [  ]
block [ d at +0 ]
block [  ]
block [ a at +0 ]
block [ a at +4, b at +5 ]
block [  ]
block [ a at +0 ]
block [  ]
block [  ]
  ...
a at 0
b at 1
c at 2
x at 3
trace at 4
function f start 15 parms 1 frame size 6
function d start 0 parms 1 frame size 4
function e start 8 parms 1 frame size 4
function main start 55 parms 0 frame size 3
stack starts at 5
```

corresponds to the following frame layouts:

frame	+0	+1	+2	+3	+4	+5
global	a	b	c	x	trace	
d()	d	old pc	old fp	result		
e()	a	old pc	old fp	result		
f()	a	old pc	old fp	result		
					a	b
main	old pc	old fp	result			

- Remove `trace`, compile and run the program, and explain the output:

```
> run()
100 0 4 0
200 5 2
300 10 20 3
100 0 40 0
200 50 2
400 1 2 3
0
0:[ 0 2 3 0 ]
```

Example 7/11 [1] contains a recursive version of Euclid's algorithm‡ with a global `trace` and a rather risky use of nested blocks.

```
var trace;
function euclid (x,y) begin
  if x > y then
    var gt; gt = euclid(x-y, y)
  else
    var lt;
    if y > x then lt = euclid(x, y-x)
    else return x
    fi
  fi;
  begin var common; print common; euclid = common end
end;

function main (x, y) begin
  x = input 54; y = input 36;
  return euclid(x, y)
end;
$ /^%% grammar/,/^%% actions/ | 07/09.eg
Seven.Blocks09
```

- Compile the program and confirm that the frames are laid out as follows:

frame	+0	+1	+2	+3	+4	+5
global	trace					
euclid()	x	y	old pc	old fp	result	
						gt
						lt
						common
main()	x	y	old pc	old fp	result	

- Run the program and confirm that it stacks a maximum of five frames, right when 18 is returned as the greatest common divisor of 36 and 54:

```
0:[ 0 ]
  1:[ 54 36 54 0 0 ]
    6:[ 54 36 46 1 0 0 ]
      12:[ 18 36 10 6 0 0 ]
        18:[ 18 18 22 12 18 0 18 ] 26: memory => this.StoreFP(4)(memory)
```

- Where is the risky use of nested blocks -- why does this program produce correct results?

Grammar Modifications

There are a few changes to the grammar:

```
1  prog: [ vars ] funs;
2  vars: 'var' names ';';
3  names: Name [{ ',' Name }];
4  funs: { fun };
5  fun: head parms [ block ] ';';
6  head: 'function' Name;
7  parms: '(' [ names ] ')';
8  block: begin [ vars ] stmts 'end';
9  begin: 'begin';
10
11 stmt: assign | print | return | block | loop | select;
12 loop: While cmp Do [ vars ] stmts 'od';
13 select: 'if' cmp then [ else ] 'fi';
14 then: Then [ [ vars ] stmts ];
15 else: Else [ vars ] stmts;
```

As before, global variables can be defined with a `vars` phrase prior to functions (lines 1 and 2 above). They can be accessed wherever they are not shadowed‡.

As before, parameters can be declared with a `names` phrase enclosed in parentheses following a function name in a function declaration or definition (lines 5 and 7). They can be accessed throughout the function body as long as they are not shadowed‡. Parameters themselves shadow‡ global variables.

As before, a `block` is the body of a function (line 5); however, it can also be a statement, i.e., blocks can now be nested (line 11).

A `block` delimits the scope of the variables defined by its `vars` phrase (line 8) and entered into a symbol table for the scope by the `names()`[2] action (line 3). They will shadow‡ all, global variables, parameters of the encompassing function, and variables in encompassing blocks — this is the semantics of JavaScript's `let`†, rather than `var`†.

Similarly, variables can also be defined local to a `loop` body (line 12) or the `then` or `else` branch of a `select` statement (lines 14 and 15).

Obviously, actions will have to create and discard the symbol tables which represent the scopes and it is tempting to assign these responsibilities to the `vars` and `parms` rules to create as few scopes as possible. However, a symbol table must exist exactly while the closest scope within a program is recognized which contains the `names` phrase, i.e., the actions for the `fun`, `block`, `loop`, `then`, and `else` rules are involved in managing these symbol tables.

These actions happen when recognition is complete, i.e., they will discard the symbol tables, if any, but each of the corresponding rules needs a hook where an action happens early enough to create a symbol table before the `names()`[2] action enters names. The obvious hooks are the literals `'var'` and `'('` at the beginning of the `vars` and `parms` rules; however, `vars` is also used to define global variables (line 1) and the global symbol table `symbols` is allocated as a property of the actions class. Therefore, the actions for `head`[2], `begin`[2], `Do`[2], `Then`[2], and `Else`[2] will have to arrange for new scopes.

Scope Management

Previously, a function description[2] contained a `Map`† `.locals` for parameter and local variable names and a property `.size` which determined the frame size and which `_alloc()`[2] uses to assign new addresses. `_dcl()`[2] stores new entries in `funct.locals`[2] and `_find()`[2] searched `funct.locals`[2] before the global symbol table.

To implement block scopes, `.locals` has to refer to the top-most `Map`† of a stack of symbol tables, one per nested scope, `.size` has to refer to the top-most address of a stack of addresses for new variables in a scope, and `_find()`[2] has to search backwards through the stack of symbol tables before consulting the global symbol table. Together, these stacks describe one frame, i.e., they should be owned by the function description for the frame.

It helps to create a class Block[2] for scope descriptions which contain a `Map`† `.locals` for names defined in the scope and a property `.size` to use for new addresses:

```
1   get Block () { return this.#Block ??= class {
2       locals = new Map();    // maps names in block to descriptions
3       size;                       // next variable address in block
4
5       constructor (size) { this.size = size; }
6       toString () // ...
7     };
8   }
9   #Block;
```

`.size` must be initialized when a new scope is created.

Previously, `.locals` and `.size` were implemented with getters and setters† and they can be

overwritten in an extended function description[2]:

```
1  class Blocks09 extends Seven.Parameters06 {
2    get Fun () { return this.#Fun ??= class extends super.Fun {
3        frameSize = 0;        // because this.size is local to blocks
4        blocks;                // block stack, [0] is innermost block
5        get locals () { return this.blocks[0].locals; }
6        set locals (locals) {
7          try { return this.blocks[0].locals = locals; }
8          catch (e) { console.trace(e); throw e; } }
9        get size () { return this.blocks[0].size; }
10       set size (size) { return this.blocks[0].size = size; }
11
12       constructor (owner, name) {        // creates outermost block
13         super(owner, name);
14         this.blocks = [ new this.owner.Block(0) ];
15       }
```

`.blocks[]` is a list which represents the nested scopes (line 4 above). It is created and initialized for the scope of parameters when a function description is constructed (line 14). Parameter addresses start in the frame at `0`.

To simplify searching, the list will contain the scopes in order from innermost to outermost. Therefore, `.locals` and `.size` can simply be delegated to `block[0]` (lines 5 to 10).

`.frameSize` is a new property because `.size` will not necessarily be both, the next new address and the total size of the frame (line 3).

```
1        push () {           // add block, start in encompassing block
2          this.blocks.unshift(
3            new this.owner.Block(this.blocks[0].size)
4          );
5        }
6
7        pop () {         // remove block, maintain maximum frame size
8          this.frameSize =
9            Math.max(this.frameSize, this.blocks[0].size);
10         if (this.owner.symbols.get('trace')              // trace?
11               instanceof this.owner.Var)
12           puts(this.blocks[0].toString());
13         this.blocks.shift();
14       }
15
16       end () {                    // [extend] pop outermost block
17         this.pop(); super.end();
18       }
19       // ...
20     };
21   }
22   #Fun;
```

There is no need to reset `.frameSize` if parameters are allocated during a forward declaration for

a function because the function definition will later allocate the same amount of memory.

`push()` creates a new innermost scope (line 2 above) which will use new addresses at `.size` of the current scope (line 3).

`pop()` maintains `.frameSize` as the maximum `.size` reached by all nested scopes (lines 8 and 9), displays the definitions in the scope (line 12) if there is a global variable `trace` (lines 10 and 11), and discards the scope (line 13). At this point `.size` reverts to the next outer scope, i.e., the part of the frame used by the discarded scope will be reused.

Finally, `end()` is extended to discard the scope created by the constructor (line 17).

With the block stack in place, `_find()`[2] can be extended to search innermost to outermost through the symbol tables (lines 5 to 8 below) before consulting the global symbol table (line 9):

```
1   _find (name, report) {
2     let sym;
3     try {
4       if (this.funct)                 // loop inner to outer block
5         this.funct.blocks.forEach(block => {
6           sym = block.locals.get(name);
7           if (typeof sym != 'undefined') throw sym;
8         });
9       return sym = this.symbols.get(name);              // global
10    } catch (sym) {                           // found in a block
11      if (sym instanceof Error) throw sym;    // shouldn't happen
12      return sym;
13    } finally {
14      if (report && !sym)
15          this.parser.error(`${name}: undefined`);
16    }
17  }
```

Actions

Scopes manage absolute global and relative local memory addresses at compile time, frames manage memory at run time, i.e., instruction generation does not change for block scopes.

There are new actions for `begin` and `block` which call `funct.push()` and `funct.pop()` at compile time, respectively, to create and discard a scope for a `block` as a statement or function body.

Similarly, the actions for `Do`, `Then`, and `Else` are extended to call `funct.push()` and the actions for `loop`, `then`, and `else` are extended to call `funct.pop()` to create and discard their respective

scopes, for example:

```
  // Do: 'do';
  Do () { this.funct.push(); return super.Do(); }

  //  loop: While cmp Do [ vars ] stmts 'od';
  loop (While, c, Do, v, s, o) {
    this.funct.pop(); super.loop(While, c, Do);
  }

  // ...
}
```

Nested Functions

In this section the little language with block scopes will be extended to permit nested function definitions. Changes to the grammar can be seen on this page…, new stack machine and action methods can be seen in the method browser[3].

Nested function definitions provide another level of scopes and shadowing‡, e.g., to hide details and instrumentation of an algorithm from the actual use of the algorithm. Consider example 7/12[1]:

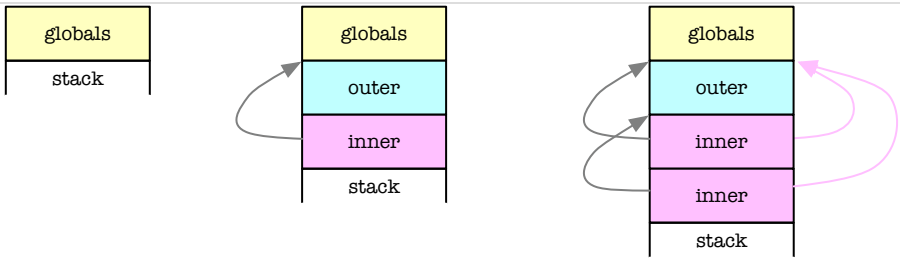
```
1  function factorial (n) begin
2    var f, calls;
3    function factorial (n) begin
4      calls = calls + 1;
5      if n <= 1 then return 1 fi;
6      factorial = n * factorial(n - 1)
7    end;
8    if n >= 0 then
9      f = factorial(n);
10     print calls;
11     return f
12   fi
13 end;
14
15 function main (n) begin
16   while n <= 5 do
17     print n, factorial(n);
18     n = n + 1
19   od
20 end;
```

The outer `factorial()` (line 1 above) accepts any value as an argument but only performs the requested computation for non-negative numbers (line 8). The number of calls to the inner `factorial()` (line 3) is counted (line 4) and reported (line 10), but the counter variable `calls` is hidden inside the outer `factorial()` (line 2).

The scope of `calls` demonstrates that nesting functions requires a modification to the

management of frames for local variables as discussed earlier:

- `calls` is not global, it is in the frame of the outer `factorial()`.
- `calls` is accessed from within each activation of the inner `factorial()`.
- Because of recursion, there can be many such inner activations stacked up, i.e., each inner activation needs a reference to the common outer frame.



Memory use evolves:

- Initially there is the *globals* frame.
- Once the outer `factorial()` is called and itself calls the inner `factorial()` there are two more frames.
- If there are recursive calls from the inner `factorial()` to itself there are more and more frames stacked up.

All frames are chained on the *dynamic link*‡ (at left) so that the code at the function exits can clean them up in reverse order.

Global variables have *absolute addresses*, determined at compile time.

Local variables have *relative addresses*, i.e., offsets within their frame, which have to be added to the *base address* where the frame begins at run time. The offset is determined at compile time but — as the graphic indicates — the base address depends on the call history.

If function definitions are nested, the inner function — i.e., frame — needs to know the base address of the outer frame.

This is known as the *static link*‡ (at right). It points from an inner frame to the topmost frame for the outer function which the inner function is nested into — the "static" nesting at compile time which determines which outer names are visible.

The base address for a local variable can be found by following the static link for as many levels outward as the variable name is in a scope more outward at compile time, i.e., the deeper functions are nested the more address tracing is required to access more global variables at run time.

The elements of the static link are "somewhere" in each frame, i.e., above the function parameters if any. Therefore, just like there is a frame pointer register there has to be a register pointing to the beginning of the static link, and it, too, has to be saved when a function is called and restored on return.

An alternative implementation, the so called *display*‡, requires a single level of address tracing — at the cost of one memory slot per function nesting level in each frame.

The display is the stack of currently visible base addresses for a frame, i.e., it is a copy of all elements of the static link of the frame. A *display pointer register* is used to locate the bottom of the display. At the bottom of the display is the address of the topmost frame for the outermost

function. The top of the display is the address of the display's own frame.

In summary, if *parms* is the number of parameters of a function and *depth* is the nesting level of the function, starting with `1` for global functions, the frame contains

register	offset	content
`.fp` →	`+0` ...	argument values for the parameters, if any
	+*parms*	the return address for the function, i.e., the previous value of `.pc`
	+*parms* `+1`	a slot for the dynamic link, i.e., the address of the frame of the caller, i.e., the previous value of `.fp`
	+*parms* `+2`	a slot for the static link, i.e., the address of the display of the caller, i.e., the previous value of `.dp`
`.dp` →	+*parms* `+3`	a slot for the function result
	+*parms* `+3 +1`	the end of the static link, i.e., the address of the outermost encompassing function's topmost frame
	+*parms* `+3 +` ...	parts of the static link, i.e., the addresses of other encompassing functions' topmost frames
	+*parms* `+3` +*depth*	begin of the static link, i.e., the address of the function call's frame, i.e., the address which is also in `.fp`
	+*parms* `+3` +*depth*`+` ...	values for the local variables and stack

Global variables have absolute addresses, determined at compile time.

Local variables in the current frame can be addressed relative to the base address of the current frame in `.fp` of the stack machine.

Local variables in outer frames are addressed by using the nesting level of their function relative to the bottom of the display to retrieve the base address of the appropriate frame. Nesting levels for functions start at `1`. They can be directly used as indices into the display because `.dp` is set to point to the function result slot at offset *parms* `+3` which directly precedes the display.

The display itself is copied to the new frame when a function is called:

- if the new nesting level is less than the current one, the new display is popped until it has a length equal to the new nesting level minus 1 — because only something that is visible at compile time can be called,
- if the nesting level stays the same, only the top entry is popped — because something at the same nesting level can access the same things which the caller can, and it cannot access the caller itself,
- otherwise, the nesting level can only increase by `1`.
- In all cases the current base address from `.fp` — which points to the current frame — is pushed onto the new display.

This section explains how nested functions are implemented in example 7/12[1] on top of the block scopes implemented in example 7/09[1].

Grammar Modifications

The grammar has to be modified so that function definitions can be nested:

```
1  block: begin body 'end';
2  begin: 'begin';
3  body: [ vars ] [ funs ] stmts;
4
5  loop: While cmp Do body 'od';
6  While: 'while';
7  Do: 'do';
8  select: 'if' cmp then [ else ] 'fi';
9  then: Then [ body ];
10 else: Else body;
11 Then: 'then';
12 Else: 'else';
```

A `body` contains variable definitions, if any, followed by function declarations and definitions, if any, followed by one or more statements (line 3 above). The names defined within the `body` are only visible within the body, i.e., within the functions and statements, unless shadowed‡.

A `body` is part of a `block` (line 1), i.e., a function definition or a `block` statement, and it is also part of a `loop` (line 5) and the `then` and `else` parts of a `select` statement (lines 9 and 10).

Display Implementation

Argument values are pushed onto the stack before a function is called. The `Entry`[2] instruction at the beginning of each function has to initialize the frame as described above, given that `parms` is the number of parameters of the function, `depth` is the nesting depth, starting with `1` for global functions, and `size` is the memory requirement of the new frame:

```
1  class Machine13 extends Seven.Machine06 {
2    Entry (parms, depth, size) {
3      return memory => {
4        const locals = size - parms - 4 - depth,   // local variables
5          fp = memory.length - parms - 1,         // new frame's base
6          dp = memory.length + 2;             // new display's bottom
7                                       // push old-fp, old-dp, result
8        memory.push(memory.fp, memory.dp, 0);
9        if (depth > 1) memory.push(           // push part of display
10         ... memory.slice(memory.dp + 1, memory.dp + depth) );
11       memory.push(fp);                      // push new frame's base
12       if (locals)                    // push local variables if any
13         memory.push(... Array(locals).fill(0));
14       memory.fp = fp;                           // new dynamic link
15       memory.dp = dp;                      // new display's bottom
16     };
17   }
```

The frame contains a slot for the function result, initialized to `0` (line 8 above). The initial part of the display is copied from the previous frame (lines 9 and 10) and the address of the new frame is added on top (line 11).

The `Exit`[2] instruction at the end of each function no longer needs to know the number of parameters:

```
1   Exit (memory) {
2
3     const fp = memory.fp,        // current frame, i.e., @arguments
4       dp = memory.dp;             // current display, i.e., @result
5     memory.fp = memory[dp - 2];                  // restore old-fp
6     memory.dp = memory[dp - 1];                  // restore old-dp
7                                  // pop frame, push old-pc result
8     memory.splice(fp, Infinity, memory[dp], memory[dp - 3]);
9   }
```

It reverts to the previous frame and display pointers (lines 5 and 6 above) and removes all but the return address and result value from the stack as the `Return` instruction expects it (lines 8).

Parameters and local variables have to be addressed relative to the display pointer using two new instructions:

```
1   /** `stack: ... -> ... frame[depth][addr]` */
2   LoadDP (addr, depth) {
3     return memory =>
4       memory.push(memory[memory[memory.dp + depth] + addr]);
5   }
```

Both, `LoadDP`[2] and `StoreDP`[2], need to know the static nesting `depth` at which a variable is defined at compile time and the variable's offset in the frame. `depth` is used as an index into the display to obtain the appropriate frame's address which is combined with the offset (lines 4 and 10 above).

The display pointer simplifies displaying memory for a trace, i.e., `toString()` is overwritten in

the Memory[2] class:

```
1   get Memory () {
2     return this.#Memory ??= class extends super.Memory {
3       dp = 0;                       // display (static link) pointer
4       // frames[] is no longer used
5
6       toString () {
7         let fp = this.fp,                  // begin of (top) frame
8           to = this.length,                  // end of (top) frame
9           dp = this.dp,                             // static link
10          output = [];
11        do {
12          if (!dp) fp = 0;                          // global frame
13          output.unshift(`${fp}:[ ${this.slice(fp, to).
14            map(slot => this.mapSlot(slot)).join(' ')} ]`);
15          to = fp;               // end and begin of previous frame
16          fp = this[dp - 2];              // previous frame pointer
17          dp = this[dp - 1];                // previous static link
18        } while (to);
19        return output.join(' ');
20      }
21    };
22  }
23  #Memory;
24 }
```

The current frame extends from the frame pointer to the end of memory (lines 7 and 8 above). The previous frame and display pointers can be obtained relative to the display pointer (lines 16 and 17). Output is collated in reverse order using unshift()† (line 13).

Infrastructure

Nesting functions is a natural extension of block structure. However, chapter eight will show that the feature can cause a very significant complication for runtime memory management; therefore, it should be an optional part of the inheritance chain of the action classes for the little language versions. Nesting functions is implemented as a function which can be applied to extend an action class such as Blocks09[2] which supports block structure:

```
1 const Nest13 = superclass => class extends superclass {
2   constructor (parser, machine = new Machine13()) {
3     super(parser, machine);
4   }
5   // ...
6 };
```

The result of the function call Nest13(Blocks09) is a class. The function Nest13()[2] overwrites and adds methods and properties of its argument class. In particular, by default it uses Machine13[2] to generate the stack machine (line 2 above).

`funct`[2] references the current function description. If functions are nested, this description has to be the top entry of a stack so that recognition of a nested function can interrupt recognition of the encompassing functions:

```
1  get funct () { return this.functs[0]; }
2  set funct (sym) {
3    if (sym) this.functs.unshift(sym);            // push function
4    else this.functs.shift();                      // pop function
5  }
6
7  get functs () { return this.#functs; }  // current function stack
8  #functs = [ null ];
```

`funct`[2] is overwritten as a getter† where `functs[0]` describes the current function (line 1 above). The list is initialized with `null` to indicate that there is no current function (line 8). The stack is pushed with `unshift()`† if a function description is assigned to `funct`[2] (line 3) and popped with `shift()`† if `null` is assigned (line 4).

Variable access uses new instructions; therefore, the class of variable descriptions[2] has to be extended:

```
1  get Var () { return this.#Var ??= class extends super.Var {
2      load () {                        // [replace] load by depth
3        if (!this.depth)                                  // global
4          this.owner.machine.gen('Load', this.addr);
5        else if (this.depth+1 != this.owner.functs.length)// nested
6          this.owner.machine.gen('LoadDP', this.addr, this.depth);
7        else                                               // local
8          this.owner.machine.gen('LoadFP', this.addr);
9      }
10     store () {                      // [replace] store by depth
11       if (!this.depth)                                  // global
12         this.owner.machine.gen('Store', this.addr);
13       else if (this.depth+1 != this.owner.functs.length)// nested
14         this.owner.machine.gen('StoreDP', this.addr, this.depth);
15       else                                               // local
16         this.owner.machine.gen('StoreFP', this.addr);
17     }
18     toString () {
19       // ...
20     }
21   };
22 }
23 #Var;
```

`load()` and `store()` use direct addressing (lines 4 and 12 above), addressing relative to the display (lines 6 and 14), or relative to the frame pointer (line 8 and 16), depending at what

nesting depth the variable is defined by `_alloc()`[2]:

```
1   _alloc (name) {
2     if (this.funct)                        // create local variable
3       return new this.Var(this, name, this.funct.size ++,
4         this.funct.depth);
5     else                                  // create global variable
6       return new this.Var(this, name, this.size ++, 0);
7   }
```

For global variables, `depth` is set to `0` (line 6 above) Otherwise it is taken from `funct`[2] (line 4). The class of function descriptions[2] has to be extended to account for the changes in frame layout:

```
1   get Fun () { return this.#Fun ??= class extends super.Fun {
2       depth;        // length of static link, 1 for global function
3
4       constructor (owner, name) {   // sets depth from owner.functs
5         super(owner, name);
6         this.depth = owner.functs.length;        // functs[.. null]
7       }
```

`depth` records the nesting depth, i.e., the current length of `functs[]`[2] (lines 2 and 6 above), `setParms()` is extended because the display pointer and the display will also be stored in the frame (lines 3 and 4 below):

```
1       setParms () { // frame: parms, old-pc, old-fp, old-dp, result
2         super.setParms();
3         this.addr = this.size ++;         // insert slot for old-dp
4         this.size += this.depth;         // leave slots for display
5       }
```

`storeOk()` has to check all of `functs[]`[2] to see if a function result is assigned within the function (line 2 below):

```
1       storeOk () {           // [replace] consider outer functions
2         if (this.owner.functs.some(f => f == this)) return true;
3         this.owner.parser.error(`${this.name}: ` +
4           `assigned to outside function`);
5         return false;
6       }
7
8       store () {                       // [replace] consider depth
9         if (this == this.owner.funct)                      // local
10          this.owner.machine.gen('StoreFP', this.addr);
11        else                                      // outer function
12          this.owner.machine.gen('StoreDP', this.addr, this.depth);
13      }
```

`LoadFP`[2] can still be used to store a function result in the current function (line 9 above) but `LoadDP`[2] is necessary to store the result of an encompassing function (line 12).

`entry()` and `exit()` have to be overwritten because the `Entry`[2] and `Exit`[2] instructions expect different arguments. More importantly, function definitions can appear at the beginning of a block, i.e., within a `begin`, `loop`, or `select` statement. Code which is generated during recognition of a function definition cannot easily be moved aside; therefore, nested function definitions have to be bypassed with branch instructions.

One branch before the `Entry`[2] of the first in a sequence of function definitions is sufficient if it lands after the `Exit`[2] and `Return`[2] of the last function in sequence; i.e., `entry()` and `exit()` together have to code the branch and they can share information in the `Block`[2] where the sequence of functions is defined:

```
1       entry () {                  // [extend] make room for bypass
2         if (this.depth > 1) {                    // nested function
3                                 // remember where this is defined
4           this.scope = this.owner.funct.blocks[0];
5           if (typeof this.scope.bypass == 'undefined')
6             this.scope.bypass =      // make room for bypass branch
7               this.owner.machine.code.push(null) - 1;
8         }
9         super.entry();
10      }
11
12      exit () {               // [replace] uses depth, fixes bypass
13        this.owner.machine.code[this.start] =
14          this.owner.machine.ins('Entry', this.parms,
15            this.depth, this.frameSize);
16        this.owner.machine.gen('Exit');      // needs no parms info
17        const end = this.owner.machine.gen('Return');
18        if (this.scope)                    // need to repair bypass
19          this.owner.machine.code[this.scope.bypass] =
20            this.owner.machine.ins('Branch', end);
21      }
```

A bypass branch is only required for nested functions (line 2 above). `entry()` is called when `funct`[2] still references the encompassing function; therefore, a reference to the `Block`[2] which contains the description of the new function can be stored as property `.scope` of the new function description (line 4). The same `Block`[2] contains the descriptions of all functions defined in a sequence in the same scope; therefore, a new property `.bypass` in this `Block`[2] can be shared by all functions in the sequence:

- If `.bypass` is undefined, `entry()` for the first function in the sequence allocates a slot for a branch and stores the address in `.bypass` (lines 5 to 7).
- `exit()` for each function checks for the shared `Block`[2] and inserts an appropriate branch instruction in the `.bypass` slot (lines 18 to 20).

The result is a single branch instruction preceding all function definitions in the same block and landing past the exit of the last of these function definitions.

One last extension: function definitions have to be checked at the end of every block (line 2

below), not only once globally:

```
      pop () {            // [extend] check for undefined functions
        this.owner._check_defs(this.locals);
        super.pop();
      }
    };
  }
```

Actions

The actions for `block`, `loop`, and `else` now receive fewer parameters than before, e.g.,

```
loop: While cmp Do [ vars ] stmts 'od';
  vs.
loop: While cmp Do body 'od';
```

The parser checks the argument count for actions; therefore, these actions have to be overwritten, for example:

```
  loop (While, cmp, Do, body, od) {
    super.loop(While, cmp, Do, undefined, undefined, od);
  }
```

Other than that, only one more action has to be overwritten:

```
 // head: 'function' Name;
 head (_, name) {
   let sym = this._find(name);
   try {
     if (sym instanceof this.Fun) {
       if (sym.depth >= this.functs.length) {  // same nesting level
         if (typeof sym.start != 'number') throw true;    // forward
         this.parser.error(`${name}: duplicate`);
       }                     // else define at deeper nesting level
     } else if (sym instanceof this.Var &&     // same nesting level
                 sym.depth >= this.functs.length - 1)
       this.parser.error(`${name}: used as variable and function`);
     sym = this._dcl(new this.Fun(this, name));       // (re-)define
   } catch (e) { throw e;                        // shouldn't happen
   } finally {
     sym.entry();                // generate code for function entry
     return this.funct = sym;                        // in function
   }
 }
```

`head()`[2] defines a new `Name` as a function (line 13 above), generates code at function entry (line 16) and pushes the description as the new value of `funct`[2] (line 17). If a prior definition for the

`Name` can be found (line 3) it has to be investigated:

- The description of a function declaration at the same nesting level (line 6) without a start address (line 7) can be used to define the function,
- otherwise it is a duplicate name (line 8).
- A more global function description can be shadowed‡ by a new local definition.
- A variable definition at the same level is an error (lines 10 to 12).

If an error is found a local definition is created to continue recognition (line 13).

`_find()`[2] searches for a name beginning with the innermost `Block`[2] outward through all nested scopes (line 6 below) and functions (line 4) and ending with the global symbol table (line 11):

```
1   _find (name, report) {
2     let sym;
3     try {
4       this.functs.forEach(funct => {   // loop inner to outer funct
5         if (funct)                     // loop inner to outer block
6           funct.blocks.forEach(block => {
7             sym = block.locals.get(name);
8             if (typeof sym != 'undefined') throw sym;
9           });
10      });
11      return sym = this.symbols.get(name);               // global
12    } catch (sym) {                            // found in a block
13      if (sym instanceof Error) throw sym;     // shouldn't happen
14      return sym;
15    } finally {
16      if (report && !sym)
17          this.parser.error(`${name}: undefined`);
18    }
19  }
```

`forEach()`† can be used because scopes and functions are stored innermost first. `throw`† is used to abort the search as soon as a match is found (line 8) and a `catch` clause converts the `throw` into a `return` (line 14). It is only prudent to rethrow any errors caught in this construction (line 13). Finally, an error is reported if the name cannot be found and a report was requested (lines 16 and 17).

Examples

Example 7/13[1] shows a function which uses an inner helper function to compute the factorial for non-negative numbers. There is significant shadowing‡:

- Add a global `trace` variable to see which names belong to which scope.
- Trace execution to see which variable is accessed using the display and when.

Example 7/14[1] is yet another variant of Euclid's algorithm‡ with a helper function which returns zero for invalid arguments. Again, there is shadowing‡.

Example 7/15[1] is the variant of Euclid's algorithm‡ with mutually recursive functions studied in example 7/08[1], modified to take advantage of nested functions.

Example 7/16[1] is a variant of example 7/10[1] where some functions are nested.

- Add a `trace` variable to see which names belong to which scope.
- Compare to the previous output (below) and explain the difference:

```
100 1 4 0
200 5 2
300 10 20 3
100 1 40 0
200 50 2
400 1 2 3
```

- Delete the first assignment to `c` and explain the difference in output.

Example 7/17[1] is a puzzle with nested functions.

- Explain the output — and then add a variable definition to obtain the expected output:

expected	actual
`3 -1`	`-13 -11`
`4 -2`	`4 -2`

Example 7/18[1] demonstrates that inner functions can set the result of outer functions which they are nested into (line 4 below) — function names do act as assign-only variables:

```
1  function main () begin
2    function a (x) begin
3      function b (x) begin
4        function c (x) begin c = x + 1; b = x + 2; a = x + 3 end;
5        print c(100 + x)
6      end;
7      print b(10 + x)
8    end;
9    print a(5)
10 end;
```

Finally, example 7/19[1] is another collection of semantic errors:

```
1  function main ();
2  function a ();
3  function a (dup, dup) begin var dup; dup = 1 end;
4  function a (x) begin var y; y = 1 end;
5  function f () begin var g;
6    function undef ();
7    function main (x) begin return y end;
8    function g () begin print 2 end;
9    f = 3; g = 4
10 end;
```

- a duplicate parameter `dup` (line 3 above)
- a mismatch in parameter counts (lines 2 and 3)
- a duplicate function definition `a()` (line 4)
- an undefined variable `y` (line 7)

- a name `g` used as a variable and nested function (lines 5 and 8)
- an assignment to `g` outside the function `g()` because `g` was redefined as a function (line 9)
- an undefined nested function (line 10)
- there is a function definition for `main()` (line 7) but it is not global

Note that a local variable can shadow‡ a parameter (line 3).

Quick Summary

- A programming language can be strongly typed — to be checked at compile time — or dynamically typed like JavaScript — where all values are typed and operators convert as necessary.
- Grammar rules can be type-aware; however, this approach to type checking fails unless there are very few mixed type operations in a language.
- Type checking can be done during syntax analysis and code generation for a stack machine, mostly by checking argument types for machine instructions and returning types from the actions generating the instructions.
- Functions require branch instructions which capture an address to return to. Recursive functions require stacking return addresses.
- Argument values and return values for functions can be passed on the stack machine's stack.
- The values of local variables for a recursive function have to be on the stack. Together with argument values, return value, return address, and other administrative information they are the *activation record* or *frame*‡ of a function call.
- Local variables and parameters are addressed relative to a *frame pointer* or *activation record pointer* which is used to locate the frame and which also has to be stacked during a call to another function.
- The visibility of names and *shadowing*‡ of global names by local names can be implemented by stacking symbol tables at compile time. Variables within the same frame in non-intersecting scopes can share memory but they are not always initialized.
- Nesting functions at compile time requires a *static link*‡ at run time which connects the visible nested scopes for variable access.
- Addressing variables in encompassing scopes requires only a single level of indirection through the *display*‡ which is a complete copy of the static link in each frame.
- If a sequence of actions within a parent rule need to share information they can use a stacked *context* which the first action creates and the action of the parent rule deletes.

8. Functions as Values

First-Order Functions
Stack versus Closure

Chapter seven explained how to perform type checking in the actions of syntax analysis and how to compile global and nested functions into machine instructions for a stack machine simulated in JavaScript.

This chapter investigates the use of functions as parameter and variable values and as function results. If function definitions can be nested the current stack machine only supports functions as parameters. Other uses of functions as values require a significant change to the stack machine's memory management. Appendix B summarizes the evolution of the stack machine and appendix C outlines the changes to the infrastructure for code generation. All classes are available from the module `Eight`[2] which is built into the practice page.

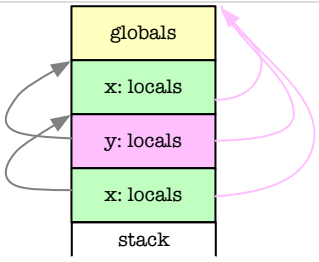
First-Order Functions

The term *first-order functions*‡ refers to the fact that they can be passed as function arguments, stored in variables, called, and returned as function results — in other words they can be used just like numbers or any other value which is manipulated by a program.

Consider the following scenario for global functions with block scopes as implemented previously:

```
1  var global;
2  function x (p) begin var v; ... end x;
3  function y (q) begin var w; ... end y;
4    ...
```

There is a global frame and — depending on call history — there can be frames for multiple nested calls to `x` and `y`:



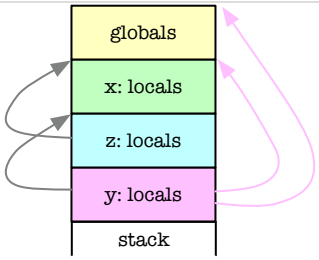
The dynamic link (frame pointers) is shown at left, the static link (visibility) at right. In addition to the global variables each activation of a global function can only access its own parameters and local variables, i.e., the stack of frames can handle any visibility issues no matter where a global function is called from.

Global functions are first-order functions. The start address of a global function represents the function as a value — no additional information is required.

As an alternative, consider the following scenario for nested functions as implemented previously:

```
1  var global;
2  function x (p) begin var v;
3    function y (q) begin var w; ... end y;
4    ... z(y); ...
5  end x;
6  function z (y) begin ... y(...); ... end z;
7  ... x(...); ...
```

Assume there is a call sequence through x to z where x passes function y as a parameter and where z then activates y.



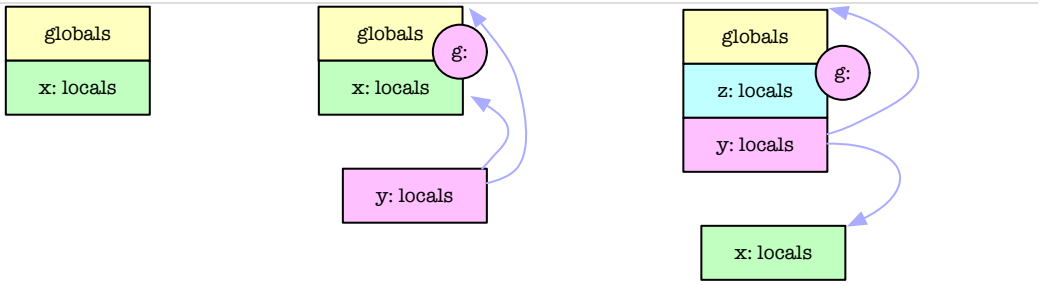
The diagram shows at right the display for the activation of y. It has access to the local information in x and to the global variables — because these are visible at compile time — and x is still active on the stack because it awaits a return from z which in turn awaits a return from y.

As long as nested functions are only passed as parameters there are no problems if the frames are stacked. However, passing a function as a value requires both, the address of the function and the display on which the function value can be called, so that a proper display can be constructed to activate the function.

Finally, consider the same setup for nested functions as before

```
1  var g;
2  function x (p) begin var v;
3    function y (q) begin var w; ... end y;
4    ... return y
5  end x;
6  function z (y) begin ... g(...); ... end z;
7  ... g = x(...); ... z(...); ...
```

but this time assume that a call to x returns the function y to the global variable g and during a later call to the function z the function y is activated by calling on g.



The left picture shows the frame for a call to `x`. The center picture shows the display which any activation of `y` will have — the display must include a frame for `x`. This information would be assigned to a more global variable `g`.

The right picture shows the situation for a call to `z` and from there by way of the variable `g` to `y`. There must be frames for `z` and `y`, but there also has to be the frame for `x` which was assigned to `g` together with the address of `y` — and this time the frame for `x` is no longer on the stack!

This effect is known as *closure*† and it poses a problem if a function defined at a deeper level such as `y` is passed outward, e.g., to `g`.

The picture at right shows that nested first-order functions cannot be supported with the strict stack discipline which has been used for frames so far. If functions can be passed to lower depths, higher depth frames still have to exist until they can no longer be referenced — they need to be garbage-collected‡.

This chapter will implement little languages for each of these three scenarios.

Function Typing

Functions and numbers are very different kinds of values, e.g., functions can be called, numbers can be added. When functions are called the arguments have to match expectations, e.g., a parameter can only be called if the corresponding argument is a function.

If functions are values the little language must support an infinite number of types:

- functions with zero to many parameters and perhaps a result
- each of which can be a number or a function
- which in turn can have parameters and might have a result, etc.

Fortunately, this infinite set of types can be described with a small grammar such as the following:

```
1  prog: [ typedcls ] [ vars ] funs;
2  typedcls: { 'type' typedcl [{ ',' typedcl }] ';' };
3  typedcl: Name '(' [ types ] ')' [ ':' typename ];
4  types: typename [{ ',' typename }];
5  typename: Name | 'number';
```

Type declarations are global and first in a program (line 1 above). The literal `type` introduces one or more type declarations, separated by commas and terminated with a semicolon (line 2). Each declaration describes a function type and starts with the type name (line 3). Parentheses enclose the list of zero or more parameter type names (line 4), and a colon precedes the result type name

if any. The literal `number` is used to indicate a number as a parameter or result type (line 5).

Type names need not be different from other global names because they will be stored in a separate type table. In a program, context determines if a name refers to a type.

```
1  type Euclid (number, number): number, Sum (number, number): number;
```

The example above declares each, `Euclid` and `Sum`, as a function type with two numbers as parameters and a number as result.

The little typing (sub-)language defined by the grammar above is restrictive:

- The number and type of parameters for a function are fixed, i.e., the exact sequence of parameters of a function is declared.
- Types are unique based on their names, i.e., there is type identity, not type equivalence. The `Euclid` and `Sum` function types above, both, require two numbers as arguments and return a number, but they are considered different types.
- As a benefit, type names can be used before they are declared, i.e., there is no need for forward type declarations, and recursion is allowed, e.g., a function can be typed to return itself.
- `number` is predefined and could be replaced by a richer set of types such as string, integer, etc. `main` is predefined as a function without parameters and a `number` result. All other type names must eventually be declared.

Once function types can be declared, variables and functions in the little language can be strongly typed, e.g.,

```
1  vars: 'var' varname [{ ',' varname }] ';';
2  varname: Name [ ':' type ];
3  type: Name | 'number';
4  funs: { fun };
5  fun: head parms [ block ] ';';
6  head: 'function' Name;
7  parms: '(' [ names ] ')' [ ':' Name ];
```

Variables are numbers by default, but in a definition a variable name can be explicitly typed by appending a type name (line 2 above).

Similarly, a function can be typed by appending a type name to the parameter list (line 7). If there is no explicit type name, the function name is the type name and the type must be declared — unless it is `main`.

Parameters are implicitly typed because a function type declaration includes types for the parameters and the result.

For example:

```
1  type Euclid (number, number): number;
2  var e: Euclid;
3  function euclid (x, y): Euclid begin ...  return 18; ... end;
```

Given these definitions, the assignment `e = euclid;` would be acceptable.

Global First-Order Functions

In this section the little language with global functions and block scopes will be extended to allow functions as variable values, argument values, and function results. Changes to the grammar can be seen on this page…, new stack machine and action methods can be seen in the method browser[3].

Example 8/01[1] prints a list of values for two global functions, `square()` and `cube()`:

```
type Calc (number): number;
function square (x): Calc begin square = x * x end;
function cube (x): Calc begin cube = x * x * x end;
```

Two more global functions implement a loop to print a list of function values: `up()` is used if the function argument increases along the list, `down()` is used otherwise. They differ only in the loop condition. Here is `up()`:

```
type Printer (Calc);
var from, to, step;
function up (calc): Printer begin
  var f;
  f = from;
  while f <= to do
    print f, calc(f);
    f = f + step
  od
end;
```

Unfortunately, in this little language there is no closure†, i.e., the loop range has to be captured in global variables.

One more global function, `loop()`, initializes the loop range and returns the appropriate `Printer` function:

```
type loop (number, number, number): Printer;
function loop (f, t, s) begin
  loop = up; from = f; to = t; step = s;
  if step < 0 then loop = down
    else if step = 0 then to = f; step = 1 fi
  fi
end;
```

Note that `loop()` shares nothing with the function which will be tabulated. Only the main program puts it all together by creating a `Printer` with `loop()` and immediately calling the

resulting function with the function to be evaluated:

```
1  function main () begin
2    loop (1, 5, 1) (square);
3    loop (10, 7, -1) (cube);
4    loop (6, 7, 0) (square)
5  end;
```

Check out example 8/01[1] which demonstrates functions used as parameters and function results:

- Note that the type declarations have to be at the beginning of the program. Compare the loop conditions in `up()` and `down()`.
- Press **new grammar** to represent and check the grammar,
- press **parse** to compile the program, and
- press **run** to see the result.
- Can you assign the loops to variables and use each twice?

Grammar Modifications

The compilers developed in example 7/09[1] and example 7/13[1] support nested variable definitions but the latter supports nested function definitions. Therefore, a compiler for global first-order functions is best developed as a function to extend the compiler from example 7/09[1].

Most changes to the grammar have already been discussed above:

- the typing (sub-) language is included up front,
- function definitions are global and can include a type name, and
- variable names can be defined in any scope and can include a type.

The main program in example 8/01[1] showed that it is convenient if a function result can be called immediately:

```
1    loop (1, 5, 1) (square);
```

For the grammar this means that a function name can be followed by several sets of arguments:

```
1  assign: symbol action;
2  action: store | call;
3  store: '=' sum;
4  call: { args };
5
6  name: symbol [{ args }];
7  symbol: Name;
```

At the statement level (line 1 above) a function name must be followed by at least one set of arguments (line 4) because a function name alone cannot be a statement. In an expression (line 6), a function name without arguments refers to the function as a (constant) value, a function name with one or more sets of arguments denotes the return value of — potentially cascaded — function calls.

Types

Type declarations are global in the little language and they are stored in a separate type table[2]:

```
1  const Global01 = superclass => class extends superclass {
2    get typeSymbols () { return this.#typeSymbols; }
3    #typeSymbols = new Map();
```

A type description[2] contains the type name, a list of parameter types, and a result type if any:

```
1     get Type () { return this.#Type ??= class extends super.Symbol {
2         parms = [];   // list of parameter types, `null` for 'number'
3         returns;                                // result type if any
4         get isFun () { return this.parms !== null; }
5
6         constructor (owner, name, parms, returns) {
7           super(owner, name);
8           this.parms = parms; this.returns = returns;
9         }
10        toString () { /* ... */ }
11      };
12    }
13    #Type;
```

Two types are predefined and created when the singleton object with action methods is constructed:

```
1     get numberType () { return this.#numberType; }
2     #numberType;
3
4     get mainType () { return this.#mainType; }
5     #mainType;
6
7     constructor (parser, machine) {
8       super(parser, machine ?? new (Machine01(Seven.Machine06))());
9       this.typeSymbols.set('number',
10        this.#numberType = new this.Type(this, 'number', null, null));
11      this.typeSymbols.set('main',
12        this.#mainType =
13          new this.Type(this, 'main', [ ], this.numberType));
14    }
```

`numberType`[2] is a scalar type, i.e., it has `null` in place of a parameter list (line 10 above). `mainType`[2] is a function type with an empty parameter list and `number` as a return type (lines 12 and 13).

The actions for the typing (sub-)language fill and check the type table:

```
1   // typename: Name | 'number';
2   typename (name) { return name; }
3
4   // types: typename [{ ',' typename }];
5   types (typename, many) {
6     return [ typename ].
7       concat(many ? many[0].map(list => list[1]) : []);
8   }
9
10  // typedcl: Name '(' [ types ] ')' [ ':' typename ];
11  typedcl (name, lp, types, rp, returns) {
12    if (this.typeSymbols.get(name))
13      this.parser.error(`${name}: duplicate type`);
14    else
15      this.typeSymbols.set(name, new this.Type(this, name,
16        types ? types[0] : [], returns ? returns[1] : null));
17  }
```

`typename()`[2] returns a name (line 2 above) and `types()`[2] returns a list of one or more names (lines 6 and 7). `typedcl()`[2] checks if a name has already been declared (line 12) and if not builds a new entry in the type table (lines 15 and 16).

At this point all the types, including `number`, are represented as strings. Once all declarations have been recognized the type descriptions have to be modified so that they reference each other and they have to be checked for completeness:

```
1   // typedcls: { 'type' typedcl [{ ',' typedcl }] ';' };
2   typedcls (some) {
3     this.typeSymbols.forEach(sym => {  // check and translate types
4       if (sym.isFun) {                      // avoid non-functions
5         const check = name => { // return type description for name
6           const type = this.typeSymbols.get(name);
7           if (type) return type;
8           this.parser.error(`${name}: not a type`);
9           return this.numberType;                          // patch
10        };
11        sym.parms = sym.parms.map(check);     // convert to symbols
12        if (typeof sym.returns == 'string')
13          sym.returns = check(sym.returns);
14      }
15    });
16  }
```

`typedcls()`[2] looks at every function type in the type table (lines 3 and 4 above) and replaces the parameter and return type names by references to their descriptions (lines 11 to 13). Undefined type names are reported and replaced by references to `number` to let recognition continue (lines 6 to 9).

Variables

Variables now have a type which is set by an action when a variable is defined:

```
1   // type: Name | 'number';
2   type (name) {
3     const type = this.typeSymbols.get(name);
4     if (type) return type;
5     this.parser.error(`${name}: not a type`);
6     return this.numberType;
7   }
8
9   // vars: 'var' varname [{ ',' varname }] ';';
10  // varname: Name [ ':' type ];
11  varname (...arg) {
12    let [ name, type ] = arg;
13    type = type ? type[1] : this.numberType;
14    this._dcl(this._alloc(name), true).type = type;
15  }
```

`type()`[2] checks that a type name has been declared (lines 2 to 7 above). If not, it is reported as an error and `number` is substituted to let recognition continue (line 6).

`varname()`[2] creates the variable description, enters it into the symbol table, and sets a `.type` property, by default `numberType` (lines 12 and 13).

`Var`[2], the class of variable descriptions, has to be extended:

```
1   get Var () { return this.#Var ??= class extends super.Var {
2       type;                                       // variable's type
3
4       storeOk (type) {                      // [replace] check type
5         if (this.type == type) return true;
6         this.owner.parser.error(`${this.name}: ` +
7             `expects ${this.type}, not ${type}`);
8         return false;
9       }
10
11      call () { this.load(); this.owner.machine.gen('CallValue'); }
12
13      toString () { /* ... */ }
14    };
15  }
16  #Var;
```

Assignment is only allowed if the value to be stored and the variable have identical types (lines 4 to 9 above).

`call()` implements code generation if a variable is called as a function (line 11): `load()` pushes the variable's current value onto the stack and a new instruction `CallValue`[2] branches to that value and replaces it on the stack by the return address, similar to `Call`[2].

Functions

Functions now have a type which is set in the function declaration because it has to be known even for a forward declaration:

```
1   // fun: head parms [ block ] ';';
2   // head: 'function' Name;
3   // parms: '(' [ names ] ')' [ ':' Name ];
4   parms (lp, names, rp, name) {   // funtion's name is default type
5     this.funct.setParms(name ? name[1] : this.funct.name);
6   }
```

The function type name is recognized together with the parameter list. It is either an explicit type name or the name of the function (line 5 above). The `parms()`[2] action calls `setParms()` to store the type as part of the description of the current function[2]. `Fun`[2], the class of function descriptions, has to be extended:

```
1   get Fun () { return this.#Fun ??= class extends super.Fun {
2       type;                                       // function's type
3       loads = [];                       // forward references to push
4
5       setParms (name) {           // [replace] sets parameter types
6         this.parms = this.locals.size;   // may be wrong, see below
7         this.size += 2;         // leave room for old pc and old fp
8         this.addr = this.size ++;          // leave slot for result
9         try {
10          const type = this.owner.typeSymbols.get(name);
11          if (!type) throw `${name}: not a type`;
12          if (!type.isFun) throw `${name}: not a function type`;
13          if (this.type && this.type != type)
14            throw `${name} ${this.name}: ` +
15              `previously declared as ${this.type.name}`;
16          if (type.parms.length != this.locals.size)
17            throw `${name} ${this.name} arguments: expects ` +
18              `${type.parms.length}, receives ${this.locals.size}`;
19          this.type = type;
20          let n = 0;              // Map.forEach does not provide n
21          this.locals.forEach(parm => parm.type = type.parms[n ++]);
22        } catch (e) {
23          if (e instanceof Error) throw e;      // shouldn't happen
24          this.owner.parser.error(e);           // report an error
25        }
26      }
27      // ...
28    };
29  }
30  #Fun;
```

`setParms()` is extended to set the function's type (line 19 above) — which must match a previous declaration, if any (line 13) — and assign types to the parameter names. When

`setParms()` is called the symbol table `.locals` contains the parameter descriptions. `Map`'s `forEach()`† visits entries in insertion order, corresponding to the types in the function type's parameter list, if any. The types are copied to the parameter descriptions (lines 20 and 21 above). Errors are reported and `number` is substituted as necessary to continue recognition.

The function type has to be checked when a return value is assigned:

```
1       storeOk (type) {                      // [extend] checks type
2         try {
3           if (this.type.returns) {        // return value expected?
4             if (!type)                          // no return value?
5               throw `must return ${this.type.returns}`;
6             else if (this.type.returns != type)     // wrong type?
7               throw `expects ${this.type.returns}, not ${type}`;
8           } else if (type)            // return value not expected?
9             throw  `doesn't return a value`;
10          return super.storeOk();               // inside function?
11        } catch (e) {
12          if (e instanceof Error) throw e;      // shouldn't happen
13          this.owner.parser.error(`${this.name}: ${e}`);
14          return false;
15        }
16      }
```

`storeOk()` now has to be called with the type of the value to be returned which has to match the expected `returns` type (lines 3 to 9 above) and, as before, assignment is only allowed within the current function (line 10).

In an expression a function name can be specified without arguments to denote the function as a value which might be assigned or passed as an argument, i.e., just like `Var`[2], the class of variable descriptions, `Fun`[2], the class of function descriptions, has to support a `load()` operation which generates code to put a function value onto the stack. At this point, for global functions, this means that the function's start address has to be pushed onto the stack (line 3 below) — even if it is not yet known:

```
1       load () {                           // generates 'Push start'
2         if (typeof this.start == 'number')
3           this.owner.machine.gen('Push', this.start);
4         else
5           this.loads.push(this.owner.machine.code.push(null) - 1);
6       }
7
8       end () {                           // [extend] resolves loads
9         const push = this.owner.machine.ins('Push', this.start);
10        this.loads.forEach(p => this.owner.machine.code[p] = push);
11        this.loads.length = 0;
12        super.end();
13      }
```

Just as for the `call()` and `return()` methods[2], if the address is not yet known, `load()` reserves a code memory slot and stores the address in a list (line 5 above) so that the correct instruction can

be inserted by `end()` when the function definition is completed (line 10).

Names

A name can be recognized as the source of a value or as the target of an assignment:

```
1   // symbol: Name;
2   // name: symbol [{ args }];
3   name (sym, args) {
4     const context = this.context; this.context = null;
5     if (args) return context.type;
6     sym.load();
7     return sym.type;
8   }
9
10  // assign: symbol action;
11  // action: store | call;
12  // store: '=' sum;
13  store (_, sum) {
14    if (this.context.symbol.storeOk(sum))
15      this.context.symbol.store();
16  }
17
18  // call: { args };
```

Chapter seven discussed that the `symbol()`[2] action sets up a `context` with a reference to the `Name`, intended to be available during recognition of `args` or `store` and discarded by the actions for either `name` or `assign`.

Chapter seven also indicated that for type checking during recognition the actions have to indicate what type results from the generated code.

Therefore, if there are no arguments applied to a `Name`, the `name()`[2] action asks the `Name`'s description to generate code to push the appropriate value onto the stack (line 6 above) and it returns the type declared for the `Name` (line 7). If there are arguments, the `args()`[2] action is responsible for leaving a result type in the `context` which `name()`[2] will return (line 5). Note that both, a variable or a function description, have a `.type` property and support a `load()` operation.

A result type will eventually be returned by the action for `sum` and the `store()`[2] action passes it to `storeOk()` to check if an assignment is possible (line 14) which `store()` will generate code for (line 15). Again, a variable or a function description, both, support the `storeOk()` and `store()` operations.

The main program of example 8/01[1] contains cascaded function calls, e.g.,

```
1  loop (1, 5, 1) (square);
```

In this case there are two argument lists which have to be recognized by two successive `args()`[2]

actions — and there could be many moreL

```
1   // args: '(' [ sums ] ')';
2   args (lp, sums, rp) {
3     const args = sums === null ? [ ] : sums[0];    // list of types
4     const type = 'type' in this.context ?   // chained call if true
5       this.context.type : this.context.symbol.type;
6     try {
7       if (!type) throw 'too many argument lists';
8       if (!type.isFun) throw 'not a function';
9       if (type.parms.length != args.length)
10        throw `arguments: ${type.parms.length} expected, ` +
11          `${args.length} specified`;
12      const errors = [];
13      type.parms.forEach(
14        (parm, n) => { if (parm != args[n]) errors.push(
15          `argument ${n+1} is ${args[n].toString()}, ` +
16          `not ${parm.toString()}`
17        ); });
18      if (errors.length) throw errors.join('; ');
19      if ('type' in this.context) {                 // chained call
20        this._lift(args);   // move function address past arguments
21        this.machine.gen('CallValue');     // call address on stack
22      } else this.context.symbol.call();  // call function/variable
23    } catch (e) {
24      if (e instanceof Error) throw e;         // should not happen
25      this.parser.error(`call to ${this.context.symbol.name}: ${e}`);
26    }
27    this.context.type = type ? type.returns : null;  // result type
28  }
```

`args()`[2] is called after `sums()`[2] has generated code to push argument values onto the stack and it is responsible for type checking (lines 3 to 18 above), generating the actual function call (lines 19 to 22), and determining the result type (line 27).

As discussed above, the `sum()`[2] action will return the type of the value which the generated code produces. Therefore, `sums()`[2] will return a list of types and `args()`[2] matches this list to the parameters which the function expects (lines 13 to 17).

The first of a sequence of argument lists is applied to a name and `context.symbol` contains the description of the name — which can be a variable or a function, but which must have a function type. For subsequent argument lists, `context.type` must contain the type which the preceding argument list produced. Therefore, `args()`[2] stores the result type as `context.type` for next time (line 27) and fetches its function type from `context.type`, if any, or from the name described in the context (lines 4 and 5). Given that `type`, parameter checking is straight-forward (lines 7 to 18).

The first of a sequence of argument lists is applied to a name, i.e., code for the actual function call is generated by the `call()` operation of the name's description (line 22) — both, variables and functions now support this operation.

For each subsequent argument list the code generated for the preceding argument list has (hopefully) resulted in a function value on top of the stack. Unfortunately, code for the

subsequent argument list will push the argument values on top of that function value. Therefore, the function value has to be lifted to the top of the stack (line 20) where the `CallValue`[2] instruction expects it (line 21).

```
1   _lift (args) {
2     if (args.length) this.machine.gen('Rotate', args.length);
3   }
```

A new `Rotate`[2] instruction may be needed to move the function value past the argument values to the top of the stack (line 2 above). This part of `args()`[2] is encapsulated as a separate method so that it can be replaced later.

Type Checking

The grammar guarantees that the type table is built and checked before any code is generated; therefore, the action methods can ensure that function and `number` values are only used as intended. Similar to the type checking implementation in example 7/02[1], actions involved in expressions again have to report and check at compile time what types of values will be produced at runtime:

```
1   cmp: sum rel;
2   rel: eq | ne | gt | ge | lt | le;
3   eq: '=' sum;
4     ...
5   sum: product [{ add | subtract }];
6   add: '+' product;
7     ...
8   product: signed [{ multiply | divide }];
9   multiply: '*' signed;
10    ...
11  signed: [ '-' ] term;
12  term: input | number | name | '(' sum ')';
13  input: 'input' [ Number ];
14  number: Number;
15  name: symbol [{ args }];
```

For the grammar excerpt above, recognition succeeds and the actions are called in order from bottom to top. `input()`[2], `number()`[2], and `name()`[2] report to `term()`[2] which, together with `signed()`[2], eventually reports to `product()`[2] and from there to `sum()`[2] which reports to the comparisons for the control structures and to the `print` and `return` statements.

Analysis is simpler than in example 7/02[1] because the arithmetic and comparison operations cannot be applied to functions and there are no cast operations which could modify function types. For the most part, the action methods have to return `number` or function types and flag illegal operations.

`input()`[2] and `number()`[2] return `numberType`[2]. As discussed above, `name()`[2] can return a function

type description. `term()`[2] returns the type received from it's descendants (line 2 below):

```
1   // term: input | number | name | '(' sum ')';
2   term (...val) { return val.length > 1 ? val[1] : val[0]; }
3
4   // signed: [ '-' ] term;
5   signed (minus, term) {
6     if (minus && term != this.numberType)
7       this.parser.error(`cannot apply '-' to ${term.toString()}`);
8     else this.parser.call(this, super.signed, minus, term);
9     return term;
10  }
```

`signed()`[2] accepts any type but complains if a minus sign is applied to a function (line 6 above); code generation for `number` values is always delegated to the superclass (line 8).

```
1   // multiply: '*' signed;
2   multiply (_, signed) {
3     if (signed != this.numberType)
4       this.parser.error(`cannot apply '*' to ${signed.toString()}`);
5     else this.parser.call(this, super.multiply);
6   }
7
8   // product: signed [{ multiply | divide }];
9   product (signed, many) {
10    if (many && signed != this.numberType)
11      this.parser.error(`cannot apply '*' or '/' ` +
12        `to ${signed.toString()}`);
13    return signed;
14  }
```

`multiply()`[2] and the other arithmetic operators complain if they have a function as a right operand (line 3 above); they don't have to return anything.

`product()`[2] and `sum()`[2] return a `number` or function type (line 13) but complain if a function is the left operand in an arithmetic operation (line 10).

Finally, comparisons complain if they are applied to functions:

```
1   // cmp: sum rel;
2   cmp (sum, _) {
3     if (sum != this.numberType)
4       this.parser.error(`cannot compare ${sum.toString()}`);
5   }
6
7   // rel: eq | ne | gt | ge | lt | le;
8   // eq: '=' sum;
9   eq (_, sum) {
10    if (sum != this.numberType)
11      this.parser.error(`cannot apply '=' to ${sum.toString()}`);
12    else this.parser.call(this, super.eq);
13  }
```

There is a small amount of semantic analysis for statements as well. As discussed above, `sums()`[2] is extended to return a list with the type of each `sum` (lines 3 and 4 below):

```
1   // sums: sum [{ ',' sum }];
2   sums (sum, many) {
3     return [ sum ].
4       concat(many ? many[0].map(list => list[1]) : []);
5   }
6
7   // print: 'print' sums;
8   print (p, sums) {
9     if (!sums.every(sum => sum == this.numberType))
10      this.parser.error('can only print numbers');
11    this.parser.call(this, super.print, p, sums.length);
12  }
13
14  // return: 'return' [ sum ];
15  return (_, sum) {
16    if (this.funct.storeOk(sum ? sum[0] : null))
17      if (sum)
18        (this.funct.store(), this.machine.gen('Pop'));
19    this.funct.return();
20  }
21 };
```

The `print` action[2] restricts printing to numbers (line 9 above) and then delegates to the superclass (line 11).

The `return` action[2], just like the `store` action[2], calls `storeOk()`[2] to see if a result value has the expected type and is even expected (line 16), and it calls `store()`[2] to generate code to store the value, if any, in the result slot (line 18).

Bottom line: the more types the more ways to make mistakes in a program, but also more chances to catch mistakes by type checking. And — little things to be thankful for — the control structures are not affected by function types and their actions remain unchanged.

Examples

This is still a compiler for the little language with block scopes and previous examples can be changed to use the new features.

Example 8/02[1] and example 8/03[1] implement Euclid's algorithm‡ from example 7/11[1] and example 7/08[1] but they refer to functions with variables. Example 8/02[1] shows that `main()` need not have the default type and, e.g., can have parameters.

Example 8/04[1] and example 8/05[1] demonstrate block scopes and the effects of shadowing‡ from example 7/09[1] and example 7/10[1].

Finally, example 8/06[1] contains a collection of errors which semantic analysis detects:

```
1  type F (), G(number), H(): number, aa(number, number): bb, bb(number): cc, cc():
2
3  var f, dup, dup;
4
5  function undefined (): Undefined;
6  function a (): F;
7  function a (dup): G begin var dup; dup = 1 end;
8  function a (x): G begin var y; y = 1 end;
9  function b (): H begin b = 2 end;
10 function f (): H begin f = 3; g = 4 end;
11
12 function cc () begin return b end;
13 function bb (x) begin return cc end;
14 function aa (x, y) begin return bb end;
15
16 function main () begin
17   a();
18   a = 5;
19   b = 5;
20   undef();
21   dup();
22   dup = aa(1,2)(3)()();
23   dup = aa(1,2)(3)();
24   aa();
25   aa(4,5)();
26   aa(5,6)(7)(8)
27 end;
```

- a duplicate variable name `dup` (line 3 above),
- an undeclared type `Undefined` (line 5),
- a forward declaration for `a` with a different type than the definition (lines 6 vs. 7),
- a duplicate definition for `a` (line 8),
- a global variable `f` which is redefined as a function in the same scope (lines 3 and 10),
- an undefined name `g` which has no type (line 10),
- a mismatch in the number of arguments when calling `a` (line 17),
- an assignment to a function name `a` which has no result (line 18),
- an assignment to a function name `b` outside the body (line 19),

- an undefined name `undef` without a type (line 20),
- a call to `dup` which is not a function (line 21),
- an assignment of a function to a variable `dup` which expects a `number` (line 23),
- mismatches in the number of arguments in cascades of function calls (line 24 to 26),
- and finally an undefined function `undefined` (line 5).

Function Composition

Example 8/07[1] makes an attempt at function composition, i.e., combining two functions to produce a new function:

```
1  type Binary (number, number): number,
2    Ternary (number, number, number): number,
3    Compose (Binary, Binary): Ternary;
4
5  var a: Binary, b: Binary;
6
7  function add (x, y): Binary begin return x + y end;
8  function sub (x, y): Binary begin return x - y end;
9
10 function sum (x, y, z): Ternary;
11
12 function compose (aa, bb): Compose begin
13   a = aa; b = bb; return sum
14 end;
15
16 function sum (x, y, z): Ternary begin
17   return b(a(x, y), z)
18 end;
19
20 function main () begin
21   print compose(add, sub)(1, 2, 3), compose(sub, add)(1, 2, 3);
```

`add()` and `sub()` are `Binary` functions — they accept two numbers and return their sum or difference, respectively (lines 7 and 8 above).

`compose()` takes two `Binary` functions and creates a `Ternary` function which accepts three numbers and returns a number. The goal is that the main program (line 21) should print `0` and `2` because it should be equivalent to the following operations

```
1  1 2 add 3 sub print
2  1 2 sub 3 add print
```

in postfix notation‡, i.e., the result of the first `Binary` function should be the argument of the second `Binary` function handed to `compose()`.

`compose()` (lines 12 to 14) has to return a function. Function definitions cannot be nested in this little language, i.e., the function `sum()`, the result of `compose()`, has to be defined as a global function (lines 16 to 18).

`compose()` stores the argument functions in two variables, `a` and `b`, which unfortunately also

have to be defined globally (line 5) because they are shared between `compose()` and `sum()`. When called, `sum()` will apply `a` to it's own first two argument values and `b` to the result of `a()` and the third argument value — a fairly convoluted way to compute and print `0` and `2`...

Unfortunately, the following block is still part of the example and it exhibits a serious flaw:

```
1   begin var as: Ternary, sa: Ternary;
2     as = compose(add, sub); sa = compose(sub, add);
3     print as(1, 2, 3), sa(1, 2, 3)
4   end
5 end;
```

The functions assigned to `as` and `sa` are composed (line 2 above) just as before in the `print` statement, but this time the output is `2` and `2`!

The problem is that `sum`, the result function of `compose()`, applies whatever is stored in the variables `a` and `b` at the time the function is *executed*, not at the time it is *created*. The variables are global, i.e., they are shared between all uses of `compose()`; therefore, both function calls in the second `print` statement (line 3) will produce the same result.

This problem will be fixed once first-order functions are nested and closure† is available, see example 8/21 below.

Functions as Argument Values

In this section the little language with nested functions and the little language with global first-order functions will be merged to allow nested functions as argument values — but not yet as variable values or function results.

Example 8/01[1] suffers from a similar flaw as example 8/07[1] just discussed above. The main program

```
1 function main () begin
2   loop (1, 5, 1) (square);
3   loop (10, 7, -1) (cube);
4   loop (6, 7, 0) (square)
5 end;
```

should work the same if it is changed to

```
1 var up: Printer, down: Printer, single: Printer;
2 up = loop(1, 5, 1); down = loop(10, 7, -1); single = loop(6, 7, 0);
3 up(square); down(cube); single(square)
```

- Load example 8/01[1].
- Press **new grammar** to represent and check the grammar.
- Edit the main program using the text above.
- Press **parse** to compile the program.
- Do *not* press **run**!
- Instead, press **100** a few times to see that the program goes into an infinite loop tabulating `cube()`.

All functions returned by `loop()` share the same range because `from`, `to`, and `step` have to be stored in global variables. Therefore, `up(square)` will print one line within the first 200 steps but then `down(cube)` will use a positive step size in a loop condition that relies on a negative step size...

Function nesting comes to the rescue because it allows to hide the range. Example 8/08[1] retains the type declarations for `Calc` and `Printer` and the function definitions for `square()` and `cube()`

```
type Calc (number): number;
type Printer (Calc);

function square (x): Calc begin square = x * x end;
function cube (x): Calc begin cube = x * x * x end;

function main () begin
  loop(1, 5, 1, square);
  loop(10, 7, -1, cube);
  loop(6, 7, 0, square)
end;
```

but `up()` and `down()` are nested into `loop()`:

```
type loop (number, number, number, Calc);

function loop (from, to, step, calc) begin

  function up (calc): Printer begin
    while from <= to do
      print from, calc(from);
      from = from + step
    od
  end;

  function down (calc): Printer begin
    while from >= to do
      print from, calc(from);
      from = from + step
    od
  end;

  if step < 0 then down(calc)
  else
    if step = 0 then to = from; step = 1 fi;
    up(calc)
  fi
end;
```

The functions to be tabulated and the loop construction are still separate. The range is still shared between the two `Printer` functions `up()` and `down()` but only one of them will actually be executed and the range, i.e., the parameters of `loop()`, cannot be reused.

Example 8/08[1] requires function nesting (e.g., `up()` and `down()` in `loop()`) and the ability to pass a function as an argument value — even over several levels (e.g., `calc()` into `loop()` and then into `up()` or `down()`). This is the second scenario discussed above and the little language can still be implemented for the stack machine using stacked frames.

- Press **new grammar** to represent and check the grammar.
- Press **parse** to compile the program.
- Press **10** once or twice and check out how near the start of the program a new instruction `PushDP`[2] is involved in creating an argument with the function value `square` and in calling `loop()`.

Grammar Modifications

Changes to the grammar can be seen here for the nested function compiler… and here for the global first-order function compiler….

Obviously, the typing (sub-)language has to be included

```
1  prog: [ typedcls ] [ vars ] funs;
2  typedcls: { 'type' typedcl [{ ',' typedcl }] ';' };
3  typedcl: Name '(' [ types ] ')' [ ':' 'number' ];
4  types: typename [{ ',' typename }];
5  typename: Name | 'number';
```

but functions can only return numbers, not functions (line 3 above).

```
1  block: begin body 'end';
2  begin: 'begin';
3  body: [ vars ] [ funs ] stmts;
4
5  varname: Name;
6
7  fun: head parms [ block ] ';';
8  head: 'function' Name;
9  parms: '(' [ names ] ')' [ ':' Name ];
10
11 loop: While cmp Do body 'od';
12 select: 'if' cmp then [ else ] 'fi';
13 then: Then [ body ];
14 else: Else body;
```

Functions can be declared in every scope (line 3 above) — even at the statement level (lines 11 to 14) — and variables *cannot* have function types (line 5). As before, a function definition can include a type name (line 9) or default to it's own name as a type name.

Finally, the rules for function calls change. A single set of arguments is all that can be applied to a function or variable name because there are no function values as results:

```
1  call: args;
2  name: symbol [ args ];
```

As noted above the grammar forbids that a variable name can be declared with a function value type. However, the typing (sub-)language has to allow that a parameter can have a function value type so that function values can be passed as arguments. To stick with the second scenario (no variables with function values) it has to be enforced that such parameters are read-only — the grammar itself cannot ensure that.

What's in a Function Value?

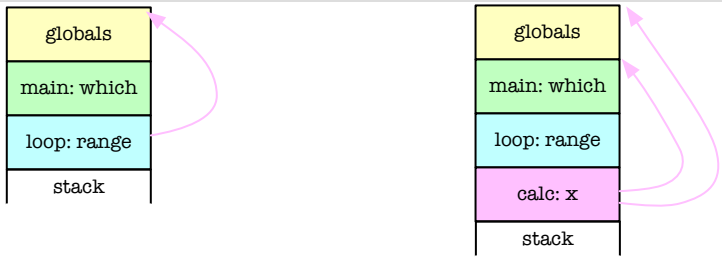
Previously, a function was represented by it's address in code storage because variables could only be global or local — no display was required. With function nesting the main program from example 8/08[1] is replaced in example 8/09[1] as follows:

```
1   function main () begin
2     var which;
3
4     function calc (x): Calc begin
5       if which > 0 then return square(x) fi; return cube(x)
6     end;
7
8     which = input 0; loop(1, 5, 1, calc); loop(10, 6, -1, calc)
9   end;
```

Depending on input (line 8 above) the output will be a table of squares or cubes:

- `loop()` is handed a function `calc()` to be calculated (line 8).
- `calc()` is nested into `main()` (lines 4 to 6). Both, `main()` and `loop()`, are global.
- `calc()` depends on a local variable `which`, defined in `main()` (line 2) and therefore invisible to `loop()`.

As the left diagram below shows, `which` is not reachable from `loop()`'s display:



When `loop()` calls `calc()` the display shown at right in the right diagram above has to be constructed. `calc()` is nested into `main()`, i.e., it requires a display which contains a frame for `main()` and thus can reach `which`.

Therefore, when `main()` sends `calc()` as an argument to `loop()` it has to send the starting code address of `calc()` and `main()`'s own display — at least up to the depth of `calc()` — because that covers what is visible to `calc()`. This is the so-called *closure*† which `calc()` requires during execution.

Function Value Management

If functions can be nested a function value consists of information to construct the display for the function value plus the starting code address for the function. In the little language with nested functions each frame contains the display which provides access to all visible frames and the address of the current display is stored in `memory.dp`, i.e., a *display register*. This value — obtained at a point where the name of a function is visible — provides the information.

In example 8/09[1]

- press **new grammar** to represent and check the grammar,
- press **parse** to compile the program, and
- press **10** twice and check out how `PushDP`[2] instructions at addresses 112 and 114 are involved in creating the function value for `calc` (start address 87) and in calling `loop()` (start address 18):

```
0:[ 130 0 0 0 0 0 1 ] 109: memory => this.Push(1)(memory)
0:[ 130 0 0 0 0 0 1 5 ] 110: memory => this.Push(5)(memory)
0:[ 130 0 0 0 0 0 1 5 1 ] 111: memory => this.Push(1)(memory)
> memory = run(memory, 10)
0:[ 130 0 0 0 0 0 1 5 1 3 ] 112: memory => this.PushDP(memory)
0:[ 130 0 0 0 0 0 1 5 1 3 87 ] 113: memory => this.Push(87)(memory)
0:[ 130 0 0 0 0 0 1 5 1 3 87 3 ] 114: memory => this.PushDP(memory)
0:[ 130 0 0 0 0 0 1 5 1 3 87 3 116 ] 115: memory => this.Call(18)(memory)
```

`PushDP`[2] is a new instruction implemented in the `Machine08`[2] mix-in for the stack machine:

```
const Machine08 = superclass => class extends superclass {
  // stack: ... -> ... dp
  PushDP (memory) {
    memory.push(memory.dp);
  }
```

The instruction pushes the current display pointer onto the stack where it will form part of a function value.

This instruction is first used by the `_startup()`[2] method which generates the code for the initial call to a program's `main()` function and which has to be replaced:

```
const Pass08 = superclass => class extends superclass {
  constructor (parser, machine) {
    super(parser, machine ?? new (Machine08(Machine01(Seven.Machine13)))());
  }

  _startup (main) {
    for (let p = 0; p < main.parms; ++ p)  // push arguments if any
      this.machine.gen('Push', 0);
    this.machine.gen('PushDP');             // push display pointer
    this.machine.gen('Call', main.start);     // call main function
    this.machine.gen('Print', 1);                  // print and pop
  }
```

The Pass08[2] mix-in for the action methods by default includes the Machine08[2] mix-in (line 3 above). _startup()[2] generates PushDP[2] right before Call[2] transfers control to main() (lines 9 and 10).

A change to the structure of a function value and, in particular, to the size of a function value — two memory slots rather than one — requires changes to the Entry[2] and Exit[2] instructions shown below as well as changes to the classes Var[2] and Fun[2] which represent parameters, variables, and functions in the symbol table. Surprisingly, there are no changes to the action methods themselves. All changes can be seen in the method browser[3].

Both, parameters and variables, are represented as Var[2] objects. If a parameter has a function value a second memory slot is allocated for the parameter, see below. The load() method is responsible for generating code to push a parameter (or variable) value onto the stack. For function values this requires two instructions:

```
1   get Var () { return this.#Var ??= class extends super.Var {
2       load () {       // [replace] load two slots for function type
3         const load = addr => {
4           if (!this.depth)                                  // global
5             this.owner.machine.gen('Load', addr);
6           else if (this.depth+1 != this.owner.functs.length)
7                                                             // nested
8             this.owner.machine.gen('LoadDP', addr, this.depth);
9           else this.owner.machine.gen('LoadFP', addr);     // local
10        };
11        load(this.addr);              // top:value or below:display
12        if (this.type.isFun) load(this.addr + 1);  // + top:address
13      }
```

Effectively, the super.load() method is turned into a local function (lines 2 to 9 above) which is called once for the parameter's memory slot at this.addr (line 10) and once again for the next address (line 11) if the parameter has a function value.

The grammar cannot prevent assignment to a parameter which has a function value. Instead, storeOk() reports this as an error and does not allow an assignment:

```
1       storeOk (type) {    // [extend] read-only function parameters
2         if (this.type?.isFun) {
3           this.owner.parser.error(`${this.name}: read only parameter`);
4           return false;
5         }
6         return super.storeOk(type);
7       }
8     };
9   }
10  #Var;
```

Functions are represented as Fun[2] objects. The call() and load() methods are responsible for

generating code to call a function or push the function value onto the stack, respectively:

```
1   get Fun () { return this.#Fun ??= class extends super.Fun {
2       call () {                       // [extend] generate 'PushDP'
3         this.owner.machine.gen('PushDP'); super.call();
4       }
5
6       load () {                       // [extend] generate 'PushDP'
7         this.owner.machine.gen('PushDP'); super.load();
8       }
```

Each method generates PushDP[2] to push the display pointer onto the stack and then delegates to it's superclass method to either generate a Call[2] or Push[2] instruction directly or defer actual code generation until the start address is known.

`setParms()` assigns the function type to a function description and implicitly the types to the parameters. This is where the extra memory slots are allocated to the parameters, i.e., `setParms()` is responsible for the layout of the frame:

```
1       setParms (name) {           // [replace] sets parameter types
2         try {
3           const type = this.owner.typeSymbols.get(name);
4           if (!type) throw `${name}: not a type`;
5           if (!type.isFun) throw `${name}: not a function type`;
6           if (this.type && this.type != type)
7             throw `${name} ${this.name}: ` +
8               `previously declared as ${this.type.name}`;
9           if (type.parms.length != this.locals.size)
10            throw `${name} ${this.name} arguments: expects ` +
11              `${type.parms.length}, receives ${this.locals.size}`;
12          this.type = type;
13          this.size = 0;          // parameter addresses start at 0
14          let n = 0;              // Map.forEach does not provide n
15          this.locals.forEach(parm => {
16            parm.addr = this.size ++;      // set parameter address
17            parm.type = type.parms[n ++];     // set parameter type
18            if (parm.type.isFun) ++ this.size; // function argument
19          });
20          this.parms = this.size;                // argument slots
21          this.size += 3;        // room for old pc, old fp, old dp
22          this.addr = this.size;               // address of result
23          this.size += 1 + this.depth;  // room for result, display
24        } catch (e) {
25          if (e instanceof Error) throw e;      // shouldn't happen
26          this.owner.parser.error(e);           // report an error
27        }
28      }
```

After some error checking the type is assigned (line 12 above) and `this.size` is reset to `0` because the parameters are at the beginning of the frame (line 13). As a Set†, `this.locals` contains the parameter descriptions in insertion order, i.e., in the order of the types in

`type.parms`. Each parameter receives an address and a type and if necessary an additional memory slot (lines 16 to 18). Finally, the total number of slots for the argument values is recorded in `this.parms` (line 20), `this.addr` is set to the address of the function result within the frame (line 22), and `this.size` is adjusted to leave room for the return address, old frame and display pointers, and the display (lines 21 to 23), so that it points to the start address for local variables, if any.

When the `Entry`[2] instruction at the beginning of a function is reached the stack contains the argument values, the incoming display pointer, and the return address. From that, `Entry`[2] determines the new frame pointer, extracts the incoming display, saves the current frame and display pointers, and allocates a result slot (lines 5 to 8 below):

```
1    // stack: ... arguments dp old-pc
2    // -> ... arguments old-pc old-fp old-dp result display locals
3    Entry (args, depth, vars) {
4      return memory => {
5        const fp = memory.length - args - 2,        // next memory.fp
6          dp = memory.splice(-1, 1, memory.pop(),    // retain old-pc
7                 memory.fp, memory.dp, 0  // push fp, dp, result slot
8            )[0];                         // extract incoming display
9        memory.fp = fp;                             // new frame's base
10       memory.dp = memory.length - 1;           // new display's base
11                           // copy incoming display up to depth-1
12       memory.push(... memory.slice(dp + 1, dp + depth),
13         memory.fp,                                // append new frame
14         ... Array(vars).fill(0));     // initialize local variables
15     };
16   }
```

The new frame and display pointers are set (lines 9 and 10 above) and the new display is constructed by copying part of the incoming display (line 12) and inserting the new frame pointer (line 13). Finally, the local variables are allocated (line 14), if any.

The `Exit`[2] instruction at the end of a function reverses most of this:

```
1    // stack: ... arguments old-pc old-fp old-dp result display locals
2    // -> ... result old-pc
3    Exit (args) {
4      return memory => {
5        const fp = memory.fp;                         // current frame
6        memory.splice(fp, args,              // remove argument values
7          memory[fp + args + 3]);                     // insert result
8                             // restore old fp dp, free rest of frame
9        [ memory.fp, memory.dp ] = memory.splice(fp + 2, Infinity);
10     };
11   }
12 };
```

The argument values are discarded (line 6 above) and replaced by the result value (line 7), the old frame and display pointers are restored, and the rest of the frame is discarded (line 9). The stack now contains the result value and the return address and is ready for a `Return`[2] instruction.

The `Entry`[2] and `Exit`[2] instructions require other parameters then before. These instructions are generated by the method `exit()` which, therefore, has to be replaced in `Fun`[2]:

```
      exit () {                   // [replace] new 'Entry', 'Exit'
        this.owner.machine.code[this.start] =
          this.owner.machine.ins('Entry', this.parms,  // arguments
            this.depth,                 // display, variable slots
            this.frameSize - (this.parms + 4 + this.depth));
        this.owner.machine.gen('Exit', this.parms);
        const end = this.owner.machine.gen('Return');
        if (this.scope)                    // need to repair bypass
          this.owner.machine.code[this.scope.bypass] =
            this.owner.machine.ins('Branch', end);
      }
    };
  }
  #Fun;
};
```

Examples

Example 8/10[1] is a nasty nested way to input three numbers, `i`, by default 3 (line 22 below), `j`, and `n`, by default 4 and 5 (lines 17 and 18), and return `2 * i * n + j`, by default 34:

```
type a (): number,
     b (number): number,
     c (b, number): number,
     d (b),
     e (b): number;

function main () begin
  var i;
  function a () begin
    var n, j; function c (b, n);
    function e (f) begin e = c(f, 2 * n) end;
    function c (f, n) begin
      function d (f) begin c = f(n) end;
      d(f)
    end;
    function b (n) begin b = n * i + j end;
    j = input 4;
    n = input 5;
    a = e(b)
  end;
  i = input 3;
  main = a()
end;
```

In example 8/10[1]

- press **new grammar** to represent and check the grammar,
- press **parse** to compile the program, and
- press **run** to see the result.
- Add a global variable `trace`, press **parse** again, and compare the frame layouts with the following table:

offset → **↓ type**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**depth**
main (): number	pc	fp	dp	0	main()	i					1
a (): number	pc	fp	dp	0	main()	a()	n	j			2
b (number): number	n	pc	fp	dp	0	main()	a()	b()			3
c (b, number): number	f		n	pc	fp	dp	0	main()	a()	c()	3
d (b)	f		pc	fp	dp	0	main()	a()	c()	d()	4
e (b): number	f		pc	fp	dp	0	main()	a()	e()		3

Each table row is a frame where the display contains references such as `main()` to the corresponding frames.

Example 8/11[1] is yet another take on Euclid's algorithm‡:

```
1  type euclid (number, number): number;
2  type Runner (Run): number, Run (number): number;
3
4  function euclid(x, y) begin
5    function run (run): Runner begin return run(y) end;
6
7    if x > 0 then
8      if y > 0 then
9
10       function euclid (y): Run begin
11         euclid = x;
12         if x > y then x = x - y; euclid = euclid(y) fi;
13         if y > x then euclid = euclid(y-x) fi
14       end;
15
16       return run(euclid)
17     fi
18   fi
19 end;
20
21 function main () begin
22   print euclid(-36, -54); main = euclid(input 36, input 54)
23 end;
```

The actual algorithm is in the helper function `euclid()` (line 10 above) which shares `x` with the global function `euclid()` (line 4). The helper is executed by the `Runner` (line 5) if the parameters for the global function are positive (lines 7 and 8) and receives `y` from the `Runner` (line 5) and from recursive calls (lines 12 and 13). Admittedly contrived, but there is a nested function as an argument value (line 16)...

As an aside, if a parameter is non-positive the global function does not reach the `return` statement (line 16) and the result is zero because that is set up by `Entry`[2].

Example 8/12[1] builds a deeper display:

```
1  type F (number), G (number, H), H (number): number;
2
3  function x (a, f): G begin
4    function y (b): F begin
5      function z (c): F begin
6        print 111, a, b, c, f(222) end;
7      z(b+1) end;
8    y(a+1) end;
9
10 function a (a): F begin
11   function b (b): F begin
12     function c (c): F begin
13       function d (d): F begin
14         function e (e): H begin
15           function f (f): H begin
16             print a, b, c, d, e, f; return f+1 end;
17           x(e+1, f) end;
18         e(d+1) end;
19       d(c+1) end;
20     c(b+1) end;
21   b(a+1) end;
22
23 function main () begin a(1) end;
```

The main program calls `a()` (line 23) which results in a chain of calls (line 21 back to line 18) until `e()` passes the function `f()` to `x()` (line 17). `x()` builds another chain of calls (lines 8 and 7) until `z()` calls the parameter function (line 6) with the argument `222`. The parameters are incremented along the chains and the expected output is

```
1  1 2 3 4 5 222
2  111 6 7 8 223
3  0
```

The first line of output is printed by `f()` (line 16) before the second line is printed by `z()` (line 6); the last line is printed by the code generated by `_startup()`[2].

- `f()` is passed once as a parameter and called once. Press **10** or **100** a few times to see the frames evolve.

Example 8/13[1] is a typed version of example 7/17[1] with three function values as arguments which

are defined at different depths.

```
1  type f (number, number),
2       add (number): number,
3       set (number),
4       sub (): number,
5       out (),
6       act (add, sub, out);
7
8  var g;
9
10 function f (x, y) begin var a;
11   function add (a);
12   function set (z) begin var s;
13     function sub() begin sub = x - y - z end;
14     function out () begin
15       print a, s;
16       if s <> -2 then set(1) fi
17     end;
18     function act (add, sub, out) begin
19       a = add(z); s = sub(); out()
20     end;
21     act(add, sub, out)
22   end;
23   function add (p) begin add = x + y + p end;
24   set(g)
25 end;
26
27 function main () begin g = 10; f(1,2) end;
```

- `act()` does the actual work (line 19 above) and receives three function values (line 21) which were defined at different depths. Use **100** to observe function values on the stack and stored locally.

Finally, the following program fragment can be used to demonstrate that assignments to `number` parameters are allowed (line 7 below) and assignments to parameters with function values (line 4) or mismatched types (line 5) are reported as errors:

```
1  type a (number, f), f ();
2  function a (n, f) begin
3    function a (nn, ff) begin
4      f = ff; f = 1;
5      n = ff
6    end;
7    n = 10
8  end
```

Nested first-order Functions

In this section the restrictions on typing in the little language with functions as argument values

will be removed to allow nested functions as variable and argument values and function results.

Nesting first-order functions is the third scenario discussed above and it can be implemented for the stack machine as long as the frames are garbage-collected‡. Nested first-order functions are used everywhere in JavaScript, i.e., garbage collection is readily available. In this section frames are implemented as arrays which JavaScript will manage dynamically as needed. One could say that nested first-order functions finally push the envelope of the "stack" machine...

Grammar Modifications

In the previous section the little language was restricted so that functions could only be passed as argument values. This restriction is now removed. Changes to the grammar can be seen here for the compiler from the previous section….

The typing (sub-)language has to be changed to again allow function types as result types:

```
1 typedcl: Name '(' [ types ] ')' [ ':' typename ];
2 typename: Name | 'number';
```

Variables can be typed, in particular, with function types:

```
1 varname: Name [ ':' type ];
2 type: Name | 'number';
```

Finally, the rules for function calls change. More than one set of arguments can be applied to a function or variable name because function values as results are available:

```
1 call: { args };
2 name: symbol [{ args }];
```

Memory Management

It even turns out that frame management is simplified by using JavaScript arrays. Global variables and the value stack remain in `memory` which now has the following layout:

`memory`	**use**
`.pc` *register*	next address in `code` to execute
`.fp` *register*	`null` or `Array` of current frame
`[ 0` ...	values of global variables
... `]`	stack

All machine instructions are functions manipulating `memory`, i.e., the stack will remain at the end of `memory`, independent of what happens with the frames. However, argument values handed to parameters during a function call become part of the function's frame, i.e., they will have to be moved. The layout of a frame is as follows:

`frame[]`	**use**
`0`	return address in `code` for function call
`1`	`null` or `Array` of previous frame
`1+1` ...	arrays of visible frames
`1+`*depth*	`Array` of this frame (at *depth*)
`2+`*depth*	result value of function call
`3+`*depth*	extra slot, exactly if result value is function value
...	argument values
... *frame size*`-1`	local variable values

All administrative information can be reached at fixed (relative) addresses within each frame, i.e., the display pointer register `memory.dp` is no longer used. A frame references itself because it contains it's own address at the end of the display.

`Machine14`[2] is a new mix-in which replaces `Machine08`[2] to support first-order function values. It contains new instructions to deal with frames:

```
1  const Machine14 = superclass => class extends superclass {
2    // stack: ... -> ... fp
3    PushFP (memory) {
4      memory.push(memory.fp);
5    }
6
7    // stack: ... -> ... frame[depth][addr]
8    LoadGC (addr, depth) {
9      return memory => memory.push(memory.fp[1 + depth][addr]);
10   }
11
12   // stack: ... val -> ... val | frame[depth][addr]: val
13   StoreGC (addr, depth) {
14     return memory =>
15       (memory.dirty = memory.fp[1 + depth])[addr] = memory.at(-1);
16   }
```

`PushFP`[2] pushes the current frame pointer, i.e., `null` or an `Array` value, onto the stack (line 4 above). This instruction replaces `PushDP`[2] when a function value is created.

`LoadGC`[2] replaces both, `LoadFP`[2] and `LoadDP`[2], to push a value from a frame onto the stack using the display within the current frame (line 9).

`StoreGC`[2] replaces both, `StoreFP`[2] and `StoreDP`[2], to copy a value from the stack into a frame using the display within the current frame (lines 14 to 15).

`memory.dirty` acts as a register which — for the benefit of tracing execution — is set whenever a frame is modified by `StoreGC`[2]. It contains the `Array` which was last modified.

The `Entry`[2] and `Exit`[2] instructions are responsible for the setup and tear-down of a frame. They

have to be modified once the layout of a frame is changed:

```
1   // stack: ... arguments fp old-pc
2   // -> ... | frame: old-pc old-fp display result arguments locals
3   Entry (args, depth, result, vars) {
4     return memory => {
5       const frame = [ memory.pop(), memory.fp ];  // old-pc, old-fp
6       frame.id = memory.newId;                    // label new frame
7       if (depth > 1)     // push (part of) incoming display, if any
8         frame.push(... memory.pop().slice(1 + 1, 1 + depth));
9       else memory.pop();                                 // pop frame
10      frame.push(frame);                    // push new frame's base
11      frame.push(... Array(result).fill(0));   // push result value
12      if (args)                            // move arguments to frame
13        frame.push(... memory.splice(- args, Infinity));
14      if (vars)                            // create local variables
15        frame.push(... Array(vars).fill(0));
16      memory.dirty = memory.fp = frame;                  // new fp
17    };
18  }
```

`Entry`[2] creates a new array for the frame (line 5 above). A property `.id` with a unique sequence number taken from `memory.id` is attached (line 6) so that the frame arrays can be identified in a trace.

Unless the called function is global, i.e., at depth `1` (line 7), most of the new display is copied from the display in the incoming frame (line 8) which is now at the top of the stack because the return address has been popped off earlier (line 5). This concludes access to the incoming frame which was part of the function value referencing this `Entry`[2] instruction. The new frame array is assigned to the top of the new display (line 10).

The slot(s) for the function result are allocated following the display (line 11). If there are argument values (line 12) they are popped off the stack and moved to the new frame (line 13). If there are local variables (line 14) they are allocated next (line 15). Unlike `memory` the size of the new frame array is now fixed.

Finally, the array is set as the new frame pointer and recorded in `memory.dirty` in case execution is traced (line 16).

```
1   // stack: ... | frame: old-pc old-fp display result ...
2   // -> ... result old-pc | fp: old-fp | frame unchanged
3   Exit (depth, result) {
4     return memory => {
5       memory.push(                                  // push result
6         ... memory.fp.slice(2 + depth, 2 + depth + result),
7         memory.fp[0]);                              // push old pc
8       memory.fp = memory.fp[1];                // set previous frame
9     };
10  }
```

The use of arrays as frames significantly simplifies `Exit`[2]. The result value and the return address are pushed onto the value stack (lines 5 to 7 above) where the immediately following `Return`[2]

instruction expects it. The frame pointer is restored from the old value in the frame (line 8).

Done — the frame array is silently abandoned. Unless a function value with the frame was created in the course of activation, JavaScript will reclaim the space eventually; otherwise, the array can be referenced as long as such a function value is among the values accessible to the program.

Execution Trace

The `memory` array still holds the global variables and the stack but the `Memory`[2] class needs some modifications to support tracing execution:

```
1   get Memory () {
2     return this.#Memory ??= class extends super.Memory {
3       get newId () { ++ this.#id; return this.id; }
4       get id () {          // returns a letter or a sequence number
5         return this.#id <= 26 ? String.fromCharCode(96 + this.#id) :
6           this.#id <= 52 ? String.fromCharCode(64 + this.#id - 26) :
7           String(this.#id - 52);
8       }
9       #id = 0;                                  // current uniqe id
```

For tracing, each frame array is labeled with an `.id` property which has a unique value maintained by `memory.newId` (line 3 above). The value is an upper-case letter (line 5), a lower-case letter (line 6), or a number starting from `1` (line 7).

```
1       dirty = null;                        // frame to be displayed
2
3       toString () {      // [replace] global memory and dirty frame
4         const dump = slot =>
5           slot === null ? 'null' :
6           slot instanceof Array ?
7             'id' in slot ? `${slot.id}:[]` : '[?]' :
8           slot;
9         let result = 'mem:[ ' + this.map(dump).join(' ') + ' ] ' +
10                     `fp: ${dump(this.fp)}`;
11        if (this.dirty) {
12          result += ` ${this.dirty.id}:[ ` +
13                     this.dirty.map(dump).join(' ') + ' ]';
14          this.dirty = null;
15        }
16        return result;
17      }
18    };
19  }
20  #Memory;
21 };
```

Memory slots may have to be interpreted symbolically: they can contain `null` (line 5 above), an array reference which should have an `.id` property (lines 6 and 7), or a plain value (line 8).

A line of trace output contains `memory.toString()` which at least contains a symbolic dump of `memory` (line 9) and the current value of the frame pointer (line 10). The most recently changed frame is referenced in `memory.dirty` and if there is one it's symbolic dump is added to the trace (lines 12 and 13) and `memory.dirty` is cleared (line 14).

If there is no trace, the very last line in the `output` area after a **run** contains the last dirty frame.

As an example consider the main program in example 8/14[1]

```
1  begin var printer: Printer;
2    printer = loop(1, 5, 1);
3    printer(square); printer(cube)
4  end
```

which computes tables of squares and cubes, similar to earlier examples.

- Press **new grammar** to represent and check the grammar,
- press **parse** to compile the program, and
- press **10** to see the the first few trace lines:

```
1  mem:[  ] fp: null
2  mem:[ null ] fp: null 125: memory => this.PushFP(memory)
3  mem:[ null 127 ] fp: null 126: memory => this.Call(0)(memory)
4  mem:[  ] fp: a:[] a:[ 127 null a:[] 0 0 0 ] 0: memory => this.Entry(0, 1, 1, 2)(m
5  mem:[  ] fp: a:[] 1: memory => this.Branch(100)(memory)
6  mem:[ 1 ] fp: a:[] 100: memory => this.Push(1)(memory)
7  mem:[ 1 5 ] fp: a:[] 101: memory => this.Push(5)(memory)
8  mem:[ 1 5 1 ] fp: a:[] 102: memory => this.Push(1)(memory)
9  mem:[ 1 5 1 a:[] ] fp: a:[] 103: memory => this.PushFP(memory)
10 mem:[ 1 5 1 a:[] 105 ] fp: a:[] 104: memory => this.Call(20)(memory)
11 mem:[  ] fp: b:[] b:[ 105 a:[] a:[] b:[] 0 0 1 5 1 ] 20: memory => this.Entry(3,
```

`memory` is initially empty and the frame pointer is `null` (line 1 above). The first two instructions call `main()` where `Entry`[2] builds the first frame represented as `a:[]` (line 4). This frame contains the return address `127`, the previous frame pointer `null`, a display of length 1 which just represents it's own frame `a:[]`, a slot for a `number` result, and two slots for the local variable `printer`.

The code now pushes the arguments `1`, `5`, and `1` onto the stack and calls `loop()` where `Entry`[2] builds the second frame represented as `b:[]` (line 11). This frame contains a longer display which ends in `b:[]`, followed by two slots for the function value result of `loop()`, and followed by the arguments moved off the stack. `memory` itself is empty and the frame pointer references `b:[]`.

Other Modifications

Based on the little language with global first-order functions, i.e., on the `Global01`[2] mix-in, this little language with nested first-order functions results in two major changes: function values require two memory slots and frames are arrays. The stack machine has a different frame layout and new instructions as discussed above. These require changes to the classes `Var`[2] and `Fun`[2] for variable and function representations and changes to some action methods.

Most of the changes are very similar to the changes made for the little language with functions

as argument values. This section describes each change and includes links, marked with [3], to the method browser where the new code can be directly compared to the corresponding code for the little language with functions as argument values.

Variables are represented as `Var`[2] objects. The `load()`[3] method generates code to push a variable value onto the stack. It now has to generate `LoadGC`[2] instructions for parameters and local variables and it takes two instructions for a function value. The `store()`[3] method generates code to copy the top value on the stack to a variable. It now has to generate `StoreGC`[2] instructions for parameters and local variables and it takes two instructions for a function value.

Functions are represented as `Fun`[2] objects. The `call()`[3] method generates code to call the function. The `load()`[3] method generates code to push a reference to the function onto the stack. Both now include the current frame pointer which requires a `PushFP`[2] instruction. The `store()`[3] method generates code to copy a value from the top of the stack to the slot(s) for the function's result value in the frame. This now requires one or two `StoreGC`[2] instructions. The `setParms()`[3] method has to assign types to the function parameters and define their addresses within the frame. This is now based on the new frame layout. The `exit()`[3] method has to generate the new `Entry`[2] and `Exit`[2] instructions and accommodate function values, i.e., two memory slots, for function results.

The `_startup()`[3] method generates code to call `main()` which now requires a `PushFP`[2] instruction.

The `varname()`[3] action method is responsible for defining a variable which includes creating a `Var`[2] object and allocating a global or local memory slot. This now requires an additional memory slot for function values.

The grammar rules

```
1 assign: symbol action;
2 action: store | call;
```

ensure that both action methods, `store()`[3] and `call()`[3], will be followed by a call to the `assign()`[3] action method which generates a single `Pop`[2] instruction to remove the result from the top of the stack. The `store()`[3] and `call()`[3] action methods now have to consider that function values require two slots on the stack and generate an additional `Pop`[2] instruction. `storeOk()`[3] reverts back, allows assignment for equal types, and no longer prevents assignment to parameters with function values.

The `_lift()`[3] method generates code to move a function value past it's arguments to the top of the stack when function calls are cascaded. It now has to move two slots for the value and it has to consider that function values among the arguments require two slots in order to compute where the value is on the stack.

Finally, the `return()`[3] action method is responsible to generate code to remove the return value of a function from the stack after it has called `store()`[3] to set the value, if any, in the frame. It now has to consider that function values require two slots on the stack.

Closure Examples

Example 8/14[1] revisits the `loop` and `Printer` functions implemented in example 8/01[1] and improved in example 8/08[1]. Thanks to function nesting and therefore closure[†], creation and

repeatable use of a loop range can be separated and the range shielded from modification:

```
type loop (number, number, number): Printer,
     Printer (Calc),
     Calc (number): number;

function main () begin
  function square (x): Calc begin square = x * x end;
    ...
  function loop (from, to, step) begin
    function up (calc): Printer begin
      var f;
      f = from;
      while f <= to do print f, calc(f); f = f + step od
    end;
    loop = up;

    if step < 0 then
      function down (calc): Printer begin
        ...
      end;
      loop = down;
      ...
  begin var printer: Printer;
    printer = loop(1, 5, 1); printer(square); printer(cube)
  end
end;
```

- Prepare the grammar and compile the program as usual.
- Press **10** once and observe that at code address `20` (begin of `loop()`) frame `b:[]` contains the loop range `1 5 1`.
- Press **10** once more and observe that at code address `48` frame `b:[]` and address 22 are the function value of `up()` stored as result in `loop()`'s frame `b:[]`.
- Press **10** twice more and follow how this value ends up as value of `printer` at the end of `main()`'s frame `a:[]` at code address `107`.

Closure† is also employed in yet another implementation of Euclid's algorithm‡ in example 8/

15[1]:

```
1  type Euclid (number, number): Run, Run (): number;
2
3  function euclid (x, y): Euclid begin
4
5    function fail (): Run begin return 0 end;
6
7    function euclid (): Run begin
8      if x = y then return x fi;
9      if x > y then x = x - y else y = y - x fi;
10     euclid()
11   end;
12
13   if x > 0 then if y > 0 then return euclid fi fi;
14   return fail
15 end;
16
17 function main () begin var run: Run;
18   run = euclid(36, 54);
19   print run(), run(), euclid(0, 1)()
20 end;
```

For positive arguments `euclid()` returns a function (line 13 above) which needs no arguments and recursively performs the calculation when called (lines 7 to 11). If an argument is non-positive the returned function (line 14) does nothing (line 5).

- Find all three uses of first-order functions in this example.
- The default output is `0 18 0`, rather than `18 18 0`. Why?
- `run()` is a function without arguments, computed in line 18 and called twice in line 19. How can it produce two different results?
- It takes one word to repair the program...

Example 8/16[1] is intended as a testbed for closure:

```
1  type a (), b (), c (), x ();
2  var f: c;
3  function x () begin print 3; f() end;
4  function a () begin
5    var a;
6    function b () begin
7      var b;
8      function c () begin print a, b end;
9      b = 1; f = c
10   end;
11   a = 2; b()
12 end;
13 function main () begin
14   a(); x()
15 end;
```

Unchanged, it outputs a line containing `3` (line 3 above) and a line containing `2` and `1` (line 8).

- Press **run**. The last line of output

```
1  mem:[ c:[] 13 ] fp: null e:[ 6 d:[] b:[] c:[] e:[] 0 ]
```

shows the last modified frame `e:[]` which has to belong to a call to `c()` because the display has three entries.

- Press **100** to see this frame `e:[]` when `c()` is called from code address `5` and entered into at code address `13`. Why is this the last modified frame?
- As one test, move `c()` out of `b()` to a lower depth, modify the `print` statement to account for arguments which are no longer in scope, step execution, and check the frames again.

Example 8/17[1] is a nesting puzzle:

```
1  type F (), Fr (): number, Fv (number), Fvr (number): number, Ffr (Fr): number;
2  var add: F, sub: Fvr, mul: Fvr, div: Fvr;
3  function a (a): Fv begin
4    function b (): Fr begin var x, y;
5      function c (x): Fvr begin
6        function a ():  F   begin print 100, x + y end;
7        function s (s): Fvr begin return x - s end;
8        function m (m): Fvr begin return x * m end;
9        function d (d): Fvr begin return x / d end;
10       add = a; sub = s; mul = m; div = d;
11       y = input 36; c = x
12     end;
13     x = input 54; print 200 + a, c(x); b = y
14   end;
15   function d (d): Ffr begin a = 2 * a; return d() end;
16   print 300 + a, d(b)
17 end;
18 function main () begin
19   a(1); add(); print 400, sub(2), mul(3), div(4)
20 end;
```

The default output is

```
1  202 54
2  301 36
3  100 90
4  400 52 162 13.5
5  0
6  mem:[ e:[] 6 e:[] 14 e:[] 23 e:[] 32 ] fp: null
7    i:[ 142 a:[] b:[] d:[] e:[] i:[] 13.5 4 ]
```

The post-mortem dump of `memory` (lines 6 and 7 above) shows that the function values for the four innermost functions `a`, `s`, `m`, and `d` in the four global variables `add`, `sub`, `mul`, and `div` share the same frame `e:[]` which belongs to the call to the function `c()` because the function values were assigned in `c()` (line 10 in the program).

Frames `a:[]` and `b:[]` belong to the initial calls to `main()` and from there to `a()` (line 19 to line

3). The last modified frame `i:[]` is at depth 4 and contains the result value `13.5`, i.e., it belongs to the call to the variable `div` (line 19) which contains the function value of `d()` (line 10); `4` in that frame is the argument value (from line 19); the modification is the setting of the result value at code address `36`.

Currying

Example 8/18[1] demonstrates a pattern for Currying‡, i.e., transforming a function with multiple arguments into a sequence of single-argument functions. Javascript can do this in a very elegant fashion:

```
1  const f = (a, b, c) => a + b * c;
2  const g = a => b => c => a + b * c;
3  f(1, 10, 100) == g(1)(10)(100)
```

In the little language the intermediate steps have to be named and typed in the spirit of this piece of JavaScript code:

```
1  const h = a => {
2    const g = b => {
3      const i = c => a + b * c;
4      return i;
5    }
6    return g;
7  };
8  f(1, 10, 100) == h(1)(10)(100)
```

Note that the last line in both JavaScript fragments is `true`.

To curry functions with up to four arguments, example 8/18[1] changes the `tokens` definition to allow for alphanumeric names

```
1  { Number: /0|[1-9][0-9]*/, Name: /[a-zA-Z][a-zA-Z0-9]*/ }
```

and declares types to define the Curry operations:

```
1  type f  (number): number,
2       f2 (number, number): number,
3       f3 (number, number, number): number,
4       f4 (number, number, number, number): number,
5       curry  (f2): ff,   ff (number): f,
6       curry3 (f3): fff,  fff (number): ff,
7       curry4 (f4): ffff, ffff (number): fff;
```

Types `f` through `f4` describe functions with one or more `number` parameters which produce a `number` result. Types `ff` through `ffff` describe functions with a single `number` parameter which can be cascaded. The `curry` types describe functions which perform the Curry transformations‡ for functions with two to four `number` arguments.

`main()` illustrates how functions with these types can be used:

```
function main () begin
  function f2 (a, b) begin return a + b end;
  function f3 (a, b, c) begin return a + b * c end;
  function f4 (a, b, c, d) begin return (a + b) * (c - d) end;
  print 1 + 2,             f2(1, 2),       curry (f2) (1)(2);
  print 1 + 2 * 3,         f3(1, 2, 3),    curry3(f3) (1)(2)(3);
  print (1 + 2) * (3 - 4), f4(1, 2, 3, 4), curry4(f4) (1)(2)(3)(4)
end;
```

The output is

```
3 3 3
7 7 7
-3 -3 -3
0
```

i.e., the explicit expressions, the functions with two to four parameters, and the cascaded curried functions, all, produce the same results — as expected.

Example 8/18[1] demonstrates a design pattern:

```
function curry (body) begin
  function ff (a) begin
    function f (b) begin f = body(a, b) end;
    ff = f
  end;
  curry = ff
end;
```

`body` is the function to be curried. There are nested function definitions, each accepts one `number` parameter and together they accept as many as `body`. Each deeper nested function is the result of the next encompassing function. Altogether, the code makes it clear that currying‡ heavily depends on closure†.

- Confirm that the definitions of `curry3()` and `curry4()` follow the design pattern.
- Press **new grammar** and **parse** as usual.
- Press **run** and confirm that a total of 18 frames are generated.
- Press **100** three times and confirm that frame `r:[]` — which is created at address `89` for `f()` in `curry4()` and is defined at depth `5` — in fact has a display with 5 entries.

Composition Revisited

Chapter 6 started with interpreting and compiling arithmetic expressions. Example 8/19[1]

demonstrates a pattern for an arithmetic expression compiled into nested function calls:

```
1  type f  (number): number,
2       f2 (number, number): number;
3
4  function add (x, y): f2 begin return x + y end;
```

Type `f` describes the resulting function which allows setting one "variable" and returns a number (line 1 above). Type `f2` describes binary operators which accept two numbers and return a number (line 2). `add()` defines the addition operator (line 4).

The main program shows how to implement the expression `(x + 1) / (x - 2) * 3`:

```
1  function main () begin
2    function f (x) begin
3      f = mul(
4            div(
5              add(x, 1),
6              sub(x, 2)),
7            3)
8    end;
9    print f(0), f(1), f(3)
10 end;
```

The resulting function `f()` (lines 2 to 8 above) essentially is reverse Polish notation‡ but in reverse order, i.e., when reading from left to right and top to bottom the leaves (numbers and variables) appear in order, the operators precede their operands — and are, therefore, in reverse order at each precedence level. Nested calls arrange for operator precedence.

Example 8/20[1] copies currying‡ from example 8/18[1] and the operators from example 8/19[1] to demonstrate how the arithmetic expression can be implemented with curried functions:

```
1  function f (x) begin
2    var a: ff, s: ff, m: ff, d: ff;
3    a = curry(add); s = curry(sub); m = curry(mul); d = curry(div);
4    f = m(
5          d(
6            a(x)(1))
7            (s(x)(2)))
8          (3)
9  end;
```

The order of operands and operations is the same as before; however, using the curried functions results in cascaded calls.

The first compiler for arithmetic expressions developed in example 6/07[1] uses functional programming. It includes the following action methods for `product` and `multiply` which reduce a list of one-argument functions produced by `signed` into a single function using function

composition‡:

```
1    // product: signed [{ multiply | divide }];
2  product (signed, many) {
3    const c = (a, b) => b(a);  // function composition
4    return (many ? many[0] : []).
5      reduce((product, list) => c(product, list[0]), signed);
6  }
7
8    // multiply: '*' signed;
9  multiply (_, right) {
10   return left => memory => left(memory) * right(memory);
11 }
```

`product()`[2] and `multiply()`[2] receive functions from the `signed()`[2] action method which manipulate `memory`. `memory` could be a map from variable names to variable values.

`product()`[2] is expected to return such a function by combining a function produced by `signed()`[2] with a list of zero or more results produced by the `multiply()`[2] and `divide()`[2] action methods.

To support list reduction, each `multiply()`[2] action method returns a curried function (line 10 above) which `product()`[2] composes along the list (lines 3 and 5).

Example 8/21[1] implements this pattern in the little language with nested first-order functions. It starts with the following types:

```
1  type value (number): number, leaf (number): value;
```

`value` is the result of representing any arithmetic expression. It describes functions which should accept variable values and return the value of an expression.

`leaf` describes functions `num()` and `name()` which are used to represent constants and variables:

```
1  function num (n): leaf begin
2    function value (ignore) begin value = n end;
3    num = value
4  end;
5
6  function name (index): leaf begin
7    function value (memory) begin
8      var n; n = 0;
9      while memory > 10 do memory = memory - 10; n = n + 1 od;
10     if index > 0 then value = memory else value = n fi
11   end;
12   name = value
13 end;
```

`num()` creates a `value` which always returns the same number, originally specified as argument to `num()`.

`name()` creates a `value` which will always return the same digit from it's argument; the argument to `name()` determines which digit it is. This is a rudimentary implementation of a memory

containing single-digit integers for the names `0` and `1`.

```
type operator (value): operation, operation (value): value;
```

`operator` is used to represent a binary operator such as `*` together with it's right-hand argument `value`. It returns an `operation`, i.e., a (curried) function which needs a left-hand argument `value` and returns the `value` of applying `*` to the two arguments:

```
function multiply (rvalue): operator begin
  function operation (lvalue) begin
    function value (memory) begin
      value = lvalue(memory) * rvalue(memory)
    end;
    operation = value
  end;
  multiply = operation
end;
```

Function composition, finally, takes a left-hand `value` and an `operation` and returns the combined `value`:

```
type compose (value, operation): value;

function compose (lvalue, roperation) begin
  compose = roperation(lvalue)
end;
```

Given this infrastructure, here is how to construct a function `f()` which can evaluate the expression `(x + 1) / (y - 2) * 3`:

```
function main () begin
  var x: value, y: value, f: value, g: value;
  x = name(0);
  y = name(1);
  f = multiply(num(3))(divide(sub(num(2))(y))(add(num(1))(x)));
  g = compose(
        compose(
          compose(x, add(num(1))),
          divide(
            compose(y, sub(num(2))))),
        multiply(num(3)));


  print x(0),   y(0), f(0),  g(0);
  print x(21), y(21), f(21), g(21);
  print x(43), y(43), f(43), g(43)
```

The first implementation, `f()` (line 5 above), again is reverse Polish notation‡ in reverse order, i.e., the leaves are in reverse order, the operators precede their operands, and composition is accomplished by cascading function calls. The second implementation, `g()` (lines 6 to 11), uses

`compose()` and, therefore, does without reverse order and cascaded calls.

The `print` statement evaluates the function for three pairs of values for `x` and `y`. The complete source in example 8/21[1] is set up for tracing.

- Press **new grammar** to represent and check the grammar and press **parse** to see how many functions are created for the example.
- Press **run** and check the last modified frame in the post-mortem dump to see that over 80 frames were created:

```
1  mem:[ -1 ] fp: null 29:[ 371 a:[] A:[] C:[] 29:[] 15 43 ]
```

- Change the value of `trace` or press **100** a few times to find the last modification to the first frame `a:[]` which belongs to the last and only call to `main()`:

```
1  a:[ 376 null a:[] 0 b:[] 20 c:[] 20 n:[] 142 C:[] 142 ]
```

The four local variables contain function values of `name()` at address `20` and the `value()` function at address `142` nested into `multiply()` at address `138`, each with two different frames containing different argument values.

Quick Summary

- First-order values can be assigned to variables, sent as arguments to functions, and returned as results. Numbers and the addresses of global functions are first-order values.
- If function definitions can be nested, a non-global function can access frames on the static link.
- If frames are strictly stacked, the address of a non-global function, together with the base address of the display at the point of call, can be used as an argument value, but it cannot be supported as a full first-order value.
- If frames are garbage collected‡ the address of a non-global function, together with a reference to the frame at the point of call (or just the display) is a first-order value.
- The display reference implements closure†, i.e., a nested function has access to the variables visible at compile time with the values at the point where the value of the nested function is captured.
- First-order functions can be used to implement functional programming with manipulations such as composition‡ and currying‡.
- Type checking is necessary if functions are first-order values.
- Type identity checking can be implemented based on a small grammar for defining unique type names and using these names to strongly type variables and functions.
- Type equivalence checking would require comparing ordered trees.

9. Compiling Grammars

What's in a Parser Generator?
`Grammar`[2] (De-)Constructed

Among other things, chapter one defined context-free grammars‡ and chapter two introduced the version of extended BNF‡ to specify grammars which we have been used throughout.

Given some grammar rules written in this notation, chapter three explained how to mechanically create a scanner function from a grammar to break up an input string into the literals and tokens used as terminal alphabet in the grammar. Chapter four introduced classes for objects that can represent grammar rules as trees in such a way that a grammar rule can be viewed as a function to recognize a piece of input and, in particular, the start rule will recognize sentences of the language described by the grammar.

Finally, chapter five explained how the rules, i.e., recognition functions, can be augmented by action methods to translate a sentence from the input terminals into some convenient representation, or to manipulate it in other ways.

All of this works as soon as the grammar rules are represented with objects from `Rule`[2] and the other magic classes. This chapter shows how this is done.

The Grammars' Grammar

Throughout, grammars have been written in a specific version of extended BNF notation‡. Formally speaking, the notation amounts to a language where the sentences are grammars, and as such there has to be a grammar describing this language. Indeed, this *grammars' grammar* is `Grammar.ebnf`[2], a static ingredient of the `Grammar`[2] class, and it is another sentence in the language of grammars:

```
1  grammar: { rule };
2  rule:    Token ':' alt ';';
3  alt:     seq [{ '|' seq }];
4  seq:     { lit | ref | opt | some };
5  lit:     Lit;
6  ref:     Token;
7  opt:     '[' alt ']';
8  some:    '{' alt '}';
```

To begin with, the above is just another grammar which can be studied in example 9/01[1].

- As usual, press **new grammar** to represent and check the grammar and see that there are no issues — the grammars' grammar is LL(1)‡.

The output shows the usual grammar description created by `Grammar.toString()`[2] with the literals

- `':'` and `';'` to separate a rule name from the right-hand side and to terminate a rule (line 2 above),

- the literal `'|'` to separate alternatives (line 3),
- brackets, `'['` and `']'`, to enclose an optional list of alternatives (line 7), and
- braces, `'{'` and `'}'`, to enclose a list of alternatives which can be repeated (line 8).

Such a self-referential description tends to be confusing, but the output also shows that this grammar has two tokens, aptly named `Token` and `Lit`:

- `Token` appears in line 2 at the beginning of the grammar rule for a grammar rule (ouch!) and, therefore, corresponds to anything in the input which can be a rule name.
- `Lit` is the other token (ouch!) and appears in line 5 and by reference in line 4. From the looks of the grammars' grammar, line 4 describes what can be in a sequence; therefore, `Lit` must correspond to anything in the input which can be a literal.

Unfortunately, lines 4 and 6 together point out a problem: `Token` stands for *any* name used in a grammar, i.e., for a rule name or a token name. One could have taken the position that any name *not* defined as a rule name must eventually be defined as a token name and provided with a pattern. Instead, `Grammar`[2] construction requires the token definitions up front and complains if they are used as rule names.

The token definitions for the grammars' grammar must be specified in the `tokens` area and are as expected:

```
1  Lit /'(?:[^'\\]|\\['\\])+'/
2  Token /[A-Za-z][A-Za-z0-9_]*/
```

i.e., when represented in the input, `Lit` are surrounded by single quotes, with single quotes and backslashes escaped by backslashes, and `Token` are alphanumeric. The `tokens` area references `Grammar.terminals`[2], another static ingredient of the `Grammar`[2] class.

This suggests two experiments. In example 9/01[1]:

- Press **new grammar** to represent and check the grammars' grammar for use.
- Copy the `grammar` area to the `program` area and
- press **parse** to see that the grammars' grammar recognizes itself, i.e., the grammars' grammar is a sentence in the language described by the grammars' grammar.

Is it the only one?

```
1  grammar is do rule end end
2  rule    is Token \: alt \; end
3  alt     is seq maybe do \| seq end end end
4  seq     is do lit or ref or opt or some end end
5  lit     is Lit end
6  ref     is Token end
7  opt     is \[ alt \] end
8  some    is \{ alt \} end
```

The grammars' grammar would not recognize the above as a sentence — literals are just backslash-escaped single characters, rules are not marked up with colons and semicolons, etc.

However, you can check in example 9/02[1] that it just takes minor tweaks to literals and token definitions for the grammars' grammar to recognize the above as a sentence:

- Press **new grammar** to represent and check the tweaked grammars' grammar for use, and
- press **parse** to see that the above is a sentence for the tweaked grammars' grammar.

Representing the Grammars' Grammar

The examples suggest that there are different ways to spell grammar rules — there might even be extensions to the notation used so far. Example 9/03[1] contains:

```
seq:    { lit | ref | opt | some | parens };
parens: '(' alt ')';
```

This addition would allow parentheses containing one or more alternatives — just like brackets and braces — to be part of a sequence, i.e., the parentheses would arrange for precedence and we could embed alternatives for something that appears exactly once right into a sequence. If this were the grammars' grammar, a grammar for a sum could be

```
sum: Number [{ ( '+' | '-' ) Number }];
```

Extending the grammars' grammar is the subject of an upcoming section.

First, however, there is still the question: how does the **new grammar** button on the practice page represent a grammar — specified only as a string in the `grammar` area — with objects from `Rule`[2] and all the other magic classes so that functions to perform lexical and syntax analysis based on that grammar can be called by the **parse** button?

The documentation of the `Grammar` constructor[2] makes it look very easy:

```
const g = new Grammar(grammar, tokens);
```

The construction only requires a string `grammar` straight from the `grammar` area and an object `tokens` which maps token names to regular expressions describing their representation.

An empty string can even be mapped to a regular expression which describes everything which is to be ignored in the `grammar` area and the `program` area, such as white space and comments.

The practice page cannot hand the *string* from the `tokens` area to the `Grammar` constructor[2] directly, but it can apply eval† to the string and hand the resulting JavaScript object to the constructor.

So, the `Grammar` constructor[2] must recognize an arbitrary `grammar` string as a sentence — that is a job for syntax analysis based on the grammars' grammar represented in the magic objects.

If we had this representation, we could apply it to get it — a clear case of bootstrapping‡.

A `Grammar`[2] object has factory methods such as `rule()`[2] which construct `Rule`[2] and all the other magic objects and handle all the bookkeeping, too. A rule of the grammars' grammar such as

```
rule: Token ':' alt ';';
```

is represented manually as follows:

```
Grammar.grammar = new Grammar(Grammar.terminals);
Grammar.grammar.rule(Grammar.grammar.nt('rule'),
  Grammar.grammar.seq([
    Grammar.grammar.token('Token'),
    Grammar.grammar.lit("':'"),
    Grammar.grammar.nt('alt'),
    Grammar.grammar.lit("';'")
  ], null)
);
```

Thankfully, this has to be done manually only once, for the grammars' grammar, and the resulting JavaScript code, `Grammar.grammar`[2], is another static ingredient of the class.

Any other grammar — as long as it is a sentence for the grammars' grammar — is recognized using the `parse()`[2] method of a `Parser`[2] created by the `Grammar.parser()`[2] method which only depends on the fact that the rules of the grammars' grammar have been represented using `Rule`[2] and all the other magic classes, and that the grammar tree has been checked as described in chapter four.

```
class Grammar {
  constructor (grammar, tokens, configuration) {
    ...
    Grammar.grammar.parser().parse(grammar, ... );
```

The `Grammar.parser()`[2] factory method creates a `Parser`[2] which calls the factory method `Grammar.scanner()`[2] to create a `Scanner`[2]. The `Parser.parse()`[2] method accepts a string, uses the `Scanner`[2] to break it up into a list of `Tuple`[2] objects each of which contains an input position, part of the input string, and a `Lit`[2] or `Token`[2] object representing this piece of the input. Finally, the `Parser.parse()`[2] method starts syntax analysis by calling the `Rule.parse()`[2] method of the start rule if the terminal in the first `Tuple`[2] is in the `expect` set of the rule.

Representing Any Grammar

So far, `Grammar.grammar`[2] can recognize any grammar that is a sentence conforming to the grammars' grammar.

However, mere recognition of some other grammar does not mean that there is lexical and syntax analysis for the other grammar. To get that, the rules of the other grammar have to be represented with `Rule`[2] and all the other magic classes.

This could again be done manually, but that's an error-prone process. If the `Parser.parse()`[2] function can *recognize* grammars, it can call action methods so that grammars can be *represented* — using the very same classes that are involved in recognition in the first place. That is *bootstrapping*!

The `Actions`[2] class contains the action methods which are used during construction of a `Grammar`[2] object for some grammar string describing a sentence conforming to the grammars'

grammar:

```
class Grammar {
  constructor (grammar, tokens, configuration) {
    ...
    Grammar.grammar.parser().parse(grammar, new Actions(this));
```

The `Parser.parse()`[2] function takes the sentence string and the action methods corresponding to the rules of the grammars' grammar which build and check the trees for the rules in the sentence. All action methods have access to a new, empty `Grammar`[2] object `this.g`, set by the constructor, and can use factory methods such as `Grammar.rule()`[2] to build the trees just as it was done manually once only for the grammar's grammar. No more manual labor!

The action methods of the class `Actions`[2] are surprisingly simple. Each action method receives the items collected by the rule's `parse()` method[2] and returns a new object. For example:

```
// lit: Lit;
lit (literal) { return this.g.lit(literal, true); }
```

If a literal is recognized, the name is collected by the `Lit`[2] object in the rule tree. The action method uses the factory method of `this.g` to create a `Lit`[2] object for the new grammar.

```
// ref: Token;
ref (name) {
  if (name in this.g.tokensByName) return this.g.token(name, undefined, true);
  return this.g.nt(name);
}
```

Similarly, a `Token`[2] object collects a name and creates either a `Token`[2] or an `NT`[2] object for the new grammar.

```
// seq: { lit | ref | opt | some };
seq (some) {
  if (some.flat().every(node => node instanceof Opt))
    throw this.g.error(some.flat().join(', ') +
      ': all sequence elements are optional');
  return this.g.seq(some.flat(), null);
}
```

Braces collect a list of lists. Here, the content of each of the inner lists is produced by a rule, i.e., it will be new objects such as the two produced above. `flat()`† moves these objects to the outer list. Spread syntax†, i.e., `...`, passes the objects collected by the braces as individual arguments to `Grammar.seq()`[2], the factory method to represent a sequence.

```
// alt: seq [{ '|' seq }];
alt (seq, many) {
  return many ?
    this.g.alt(seq, ... many.flat(1).map(elt => elt[1])) :
    this.g.alt(seq);
}
```

Alternatives are easy if there is only one — it was represented as a `Seq`[2] object by the previous action and it needs to be wrapped into an `Alt`[2] object.

Otherwise the brackets collect a list which contains the list of lists produced by the braces. `many.flat()` lifts the innermost lists up where `map()` can pick the second element of each of the innermost lists. Each `elt[0]` would have been the string representing the `'|'`, each `elt[1]` is a `Seq`[2] object. Spread syntax† and the factory method `Grammar.alt()`[2] take care of representing the list of alternative sequences.

`Alt`[2] objects never show up in a grammar tree — `Alt`[2] is the hidden superclass to represent rules, brackets, and braces. The next three actions use spread syntax and factory methods to revise the representation:

```
// rule: Token ':' alt ';';
rule (name, _, alt, s) { return this.g.rule(this.g.nt(name), ...alt.seqs); }

// opt: '[' alt ']';
opt (lb, alt, rb) { return this.g.opt(...alt.seqs); }

// some: '{' alt '}';
some (lb, alt, rb) { return this.g.some(...alt.seqs); }
```

A grammar contains some rules. By now they have been collected and the factory methods have attached everything to the new grammar. Finally, the start rule action just returns the grammar, not yet checked:

```
// grammar: { rule };
grammar (r) { return this.g; }
```

It is up to the caller of the `Parser.parse()`[2] function to check the grammar, at least far enough to create the `expect` sets.

In retrospect, a few comments:

- The spread syntax† `...` seems unnecessary — the factory methods could have been designed to accept arrays instead. However, the grammars' grammar had to be represented manually. A look at the source of `Grammar.grammar`[2] shows how requiring individual descendants as arguments facilitates nested calls to the factory methods.
- Creating and immediately destroying the `Alt`[2] objects could have been avoided. Instead, the `alt` action could have returned a list of alternative sequences. However, some action is necessary to smooth out the different nesting depths of one and more alternatives.
- `Alt`[2] objects seem to be destined to represent alternative sequences nested into a sequence when the grammars' grammar is extended with parentheses for grouping. Could there be an obstacle?

Quick Summary

- Grammar rules can be represented as nested constructor or factory method calls to create `Rule`[2] and other objects.

- The classes implement algorithms for grammar checking such as `shallow()`[2], `deep()`[2], and `check()`[2], and recognition, i.e., `parse()`[2].
- As a result. once the grammars' grammar is (manually) represented, any grammar can be recognized.
- Following recognition, action methods for the rules of the grammars' grammar take care of representing the rules of any grammar string which is a sentence.
- In summary, the grammars' grammar, once represented, can represent other grammars, and this in turn implements lexical and syntax analysis for other grammars. Their rules, too, can be augmented with actions to implement translations.

Bootstrap Example

Example 9/04[1] demonstrates that the grammars' grammar can recognize and represent itself.

```
1  class Actions extends EBNF.Actions {  // modify top-level action
2    static count = 0;                   // count created grammars
3
4    constructor () {                    // create next grammar
5      super(new EBNF.Grammar(EBNF.Grammar.terminals));
6      this.g.count = ++ Actions.count;
7    }
8
9    grammar (some) {
10     this.g.check();                   // check next grammar
11     puts(this.g.toString());          // display next grammar
12     puts('count', this.g.count);      // display it's count
13
14     // executable uses next grammar to parse it's own rules
15     return () => this.g.parser().parse(
16       this.g.count % 2 ? program : grammar, Actions);
17   }
18 }
```

The `Actions`[2] class is extended and contains a `static` variable `count` to identify how often an object is created (lines 2 and 6 above). Whenever an object is created it uses a new, empty `Grammar`[2] object (line 5).

The top-level action is replaced to check and display the grammar, together with it's `count` (lines 10 to 12). Rather than returning the grammar it returns a function which will use the new `Grammar`[2] object to parse the text in the `grammar` area or the `program` area and represent it using a new `Actions`[2] object (lines 15 and 16).

- Press **new grammar** to represent and check the grammars' grammar.
- Press **parse** to let the grammars' grammar represent and check itself and store the parser from the new representation as the result in `run`.
- Press **run** to apply this parser to the `grammar` area or the `program` area which both contain the grammar's grammar.
- Press **run** a few more times: the `count` value in the `output` area increases because the grammar keeps representing itself.

Extending the Grammars' Grammar

The grammars' grammar can represent itself. Therefore, it is possible to extend the grammars' grammar, i.e., to modify or extend the notation in which we have been writing grammars. Example 9/05[1] tries to add parentheses to the notation as discussed earlier, so that alternatives can be embedded into sequences.

The extended grammar's grammar is in the `grammar` area. The modifications are

```
seq:     { lit | ref | opt | some | parens };
parens:  '(' alt ')';
```

- Press **new grammar** to represent and check it: it is a sentence!

The new grammar for `sum`

```
sum: Number [{ ( '+' | '-' ) Number }];
```

is in the `program` area.

The `actions` area again extends the `Actions`[2] class to provide a definition for `Number` and replace the top-level action for `grammar` to check and display the new grammar and return an executable which will call the `Parser.parse()`[2] method based on the new grammar with a sum `1 + 2 - 3`. It also contains an action for `parens`

```
// parens: '(' alt ')';
parens (lp, alt, rp) {              // returns the alternatives
  return alt;
}
```

which simply returns the `Alt`[2] object to be included in the sequence.

- Press **parse** to represent and check the grammar for `sum`.

Unfortunately...

An `Alt`[2] object cannot be a descendant of a `Seq`[2] object — sequences were supposed to only contain terminals, rule references, brackets and braces. Not allowing parentheses in sequences is a syntax issue, checking that there are no other descendants is a semantic issue. The semantic check is part of the `Seq`[2] constructor.

A rule reference is allowed in a sequence, and a rule contains alternatives. Example 9/06[1], the next attempt, tries to add a hidden rule:

```
// parens: '(' alt ')';
parens (lp, alt, rp) {   // returns reference to auxiliary rule
  const uniq = this.g.nt();              // unique non-terminal
  this.g.rule(uniq, ... alt.seqs);                // uniq: alt;
  return uniq;
}
```

The only change from the previous example is a different action for `parens`. This action creates a

rule with a unique name and returns a reference to it which will be added to the parent sequence.

- Press **new grammar** to represent and check the extended grammars' grammar and press **parse** to let the extended grammars' grammar represent and check the new grammar for `sum` found in the `program` area.

Unfortunately...

The rule for `sum` got lost: the `Parser.parse()`[2] method looks for a `grammar`, from there for a `rule`, `alt`, `seq`, and eventually `parens` — this is the stack of open calls on the `parse()` methods by the time we reach the new action which creates the hidden rule. This action happens to create the first `Rule`[2] object ever which becomes the start rule, rather than `sum`.

Third time is a charm: the action for `parens` in example 9/07[1] is more careful and moves the new rule to the end of the list of rules (lines 2, 8, and 14 below):

```
1  class Actions extends EBNF.Actions {
2    #hidden = [];
3
4    // parens: '(' alt ')';
5    parens (lp, alt, rp) {   // returns reference to auxiliary rule
6      const uniq = this.g.nt();              // unique non-terminal
7      this.g.rule(uniq, ... alt.seqs);               // uniq: alt;
8      this.#hidden.push(this.g.rules.pop()); // can't be start rule
9      return uniq;
10   }
11
12   // grammar: { rule };
13   grammar (some) {
14     this.g.rules.push(... this.#hidden);   // append hidden rules
15       ...
```

- Again, press **new grammar** to represent and check the extended grammars' grammar,
- press **parse** to let the extended grammars' grammar represent and check the new grammar for `sum` found in the `program` area, and
- finally, press **run** to apply the new grammar for `sum` to the expression `1 + 2 - 3`.

Bottom line: the grammars' grammar can translate itself and this allows for changes to the notation as long as `Rule`[2] and the other magic classes support the semantics or new classes are added to support entirely new constructs.

According to the definition of bootstrapping‡ this can go on for many iterations — but it will eventually reach the limits of the practice page...

10. Recognition Revisited

Try (Almost) Everything?
Conflicts and Errors

Chapter two defined *context-free grammars*‡, the BNF notation‡ for grammar rules, and the version of Extended BNF‡ used throughout this book — rules are alternative sequences which contain literals, tokens, references to other rules, and additionally brackets for optional alternatives and braces for alternatives which can appear one or more times.

All the examples so far employed the *recursive descent*‡ algorithm for recognition implemented by the EBNF module[2] and described in chapter four:

- Consider the grammar rules to be recognition functions.
- Call the start rule.
- The lookahead (next input symbol) uniquely selects an alternative within the rule.
- If the next item in the rule is a literal or token it is collected and recognition moves on.
- If the next item is a reference another rule is called.
- Brackets are entered if the lookahead fits.
- Braces are entered and iterated as long as the lookahead fits.
- Eventually the start rule has to be completed.

As an aside, example 2/05[1] illustrated that this algorithm cannot handle left recursion in grammar rules.

This chapter discusses a stack-based parser to replace the *recursive descent*‡ algorithm. The parser is constructed from BNF‡ rules and supports left recursion. It is based on an idea by Don Knuth‡.

It turns out that EBNF‡ rules can be translated to BNF‡ in such a way that the stack-based parser executes the same actions as before, i.e., the objective of this chapter is to construct a more powerful parser from grammars which are not suitable for *recursive descent*‡.

All numbered classes in the examples are available from the module `Ten`[2] which is built into the practice page.

All examples in the previous chapters can use either the recursive descent parser or the stack-based parser — no changes are required!

The Idea

The construction is based on BNF notation, i.e., rules are ordered pairs with a non-terminal on the left and a symbol sequence — terminals and non-terminals — on the right. Alternatives are expressed by rules which have the same non-terminal at left.

The stack-based parser moves from left to right along the input and records each move on the stack until the top values on the stack fit a rule of the grammar. At this point there are at least as many values on the stack as there are symbols in the rule's sequence.

At this point the values corresponding to the rule are popped off the stack, optionally processed by an action, and a value corresponding to the rule is placed on the stack instead.

The process continues until the input is exhausted and the stack just contains a value corresponding to the start rule of the grammar. Success!

This looks like the obvious algorithm — but the problem is to recognize that *the top values on the stack fit a rule of the grammar*.

Example 10/01[1] shows how this might be accomplished:

- The very first button should show bnf. If not, click it until it does.
- Toggle **states** and
- press **new grammar** to represent and check the grammar.
- Press **parse** to see that the text in the `program` area is recognized.
- Remove `b` or `c` from the `program` area and each time press **parse** again.

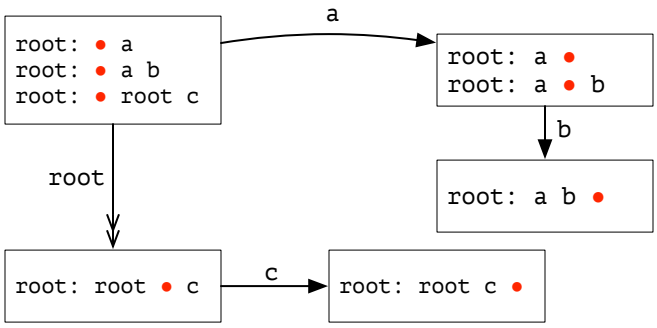
The grammar describes sequences starting with a single `a`, optionally followed by a single `b`, and ending in any number of `c`:

```
1  root: 'a';
2  root: 'a' 'b';
3  root: root 'c';
```

The grammar is left-recursive and the first two rules for `root` start with the same terminal symbol, i.e., the grammar is definitely not LL(1) and it is not suitable for *recursive descent*‡.

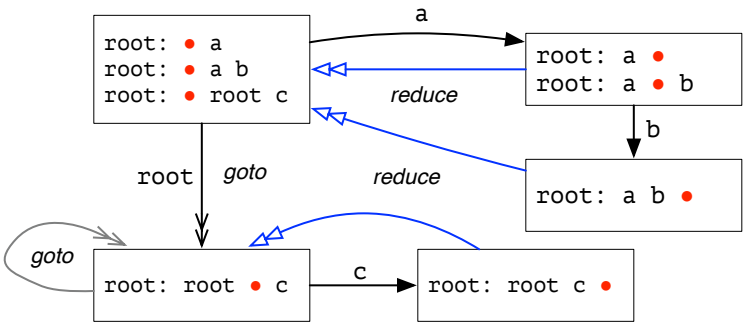
However, the grammar is not ambiguous because a sentence starts either with `a` or with `a` and `b`, and that uniquely determines how each syntax tree is built.

How should recognition proceed if it is based on this grammar? The stack-based parser is controlled by a transition diagram which is constructed by simulating execution: a position marker • is placed at the beginning of the rules and is advanced as input arrives:



The result is a directed graph where the nodes are recognition states which contain marked rules and the edges indicate transitions based on input terminals (single arrowhead) or recognized non-terminals (double arrowhead).

A non-terminal is recognized exactly when a rule can be satisfied, i.e., the graph needs edges for completed rules — rules with the marker at the end of the rule. Edges must lead from states with completed rules to states with rules where the marker is just before the non-terminal at left in the completed rule:



In the diagram above these edges are blue, labeled *reduce*, and are marked with double arrowheads.

Finally, there are transitions based on non-terminals (black with double arrowheads). A transition based on a terminal is called a *shift* and a transition based on a non-terminal is called a *goto*. A *reduce* transition is always immediately followed by a *goto* transition.

To get unique start and end states for the transition diagram a *rule zero* is added to the grammar

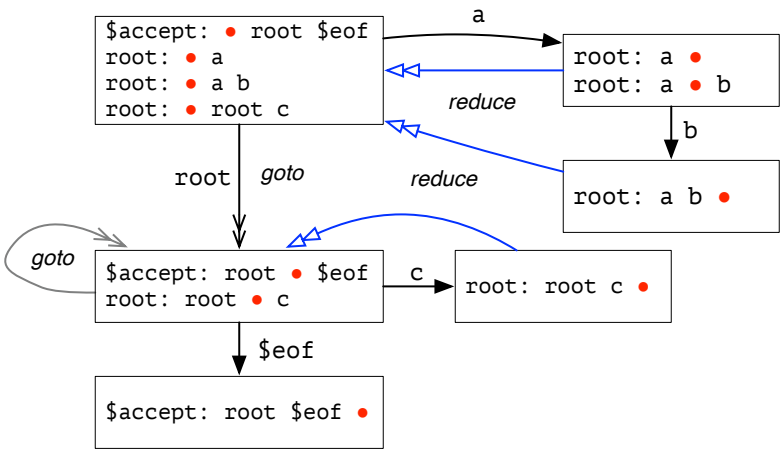
```
1  $accept: root $eof
```

where `root` is the start non-terminal of the grammar.

This provides a unique start state 0 for the transition diagram, namely the state which contains rule 0 marked at the beginning.

Whenever the position marker is right before a non-terminal, *closure*‡ is applied (recursively), i.e., all rules for the non-terminal are marked at the beginning and are added to the state.

Rule zero also provides a unique state where the transitions end successfully, namely the state which contains rule 0 marked at the end, following `$eof`, the end of input. Here is the complete transition diagram:



States and transitions for example 10/01[1] can be viewed:

- The very first button should show bnf. If not, click it until it does.

- Toggle **states** so that the effect of closure‡ is displayed and
- press **new grammar** to represent and check the grammar.
- Check the `output` area. It contains the states and the marked rules defining each state. Each state is followed by the transition messages.

One more thought and the recognition technique is complete: The transition diagram is finite because the number and length of rules, both, are finite. However, a grammar can, for example, describe a set of nested parentheses of arbitrary depth, i.e., recognition needs a mechanism for unlimited counting which a finite diagram cannot provide.

Therefore, the states are kept on a *state stack* with the current state on top. Each *shift* and *goto* transition, i.e., recognizing an input terminal or a completed non-terminal, pushes one new state on top of the stack. A *reduce* transition pops states off the stack, namely exactly as many states as it took to complete the rule which is reduced, i.e., exactly as many states as there are symbols in the right-hand side of the rule. This can be counted for any specific path through the diagram, too.

The transition diagram really tries "everything." It starts with the parser positioned at the beginning of rule 0. It transitions and pushes the state stack for every possible input. It pops the stack for completed rules and transitions beyond them until, hopefully, the end of rule 0 is reached. The states are computed in a way that "everything" is tried in parallel, thus avoiding the complications of trial and error.

There has to be a catch — not every grammar will result in this kind of a transition diagram. The catch will show up once the state table is constructed but example 4/04[1] which was discussed previously provides a quick preview:

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar — no complaints because the `if` statement is defined with a trailing `fi`.
- Remove the `'fi'` in the `grammar` area and press **new grammar** again to see that the `check()` algorithm is not happy — the grammar is now ambiguous.
- Switch to BNF: press the very first button until it shows stack-based, i.e., the EBNF grammar will be translated to BNF and the stack-based parser is used.
- Toggle **states** to see the effect of closure‡ and
- press **new grammar** to translate the grammar to BNF and create the state table.
- Check the `output` area: The last line complains about a *shift/reduce conflict*.
- If you scroll to state 7 you can see that it contains both, a complete rule which could be reduced and the opportunity to shift `else`.

The Architecture

The EBNF[2] and BNF[2] modules use similar class architectures to represent grammars: A BNF[2] `Grammar`[2] object wraps a grammar for parsing and BNF[2] classes `Rule`[2], `Lit`[2], `Token`[2], and `NT`[2] are used to represent the rules.

A BNF[2] `Grammar`[2] object can be constructed from a string with rules expressed in BNF notation using the kind of bootstrap technique described in chapter nine which is demonstrated in example 10/02 for BNF[1]:

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the BNF grammars' grammar.

- Press **parse** to let the BNF grammars' grammar represent itself and store the parser from the new representation as the result in `run`.
- Press **run** to apply this parser to the `program` area which also contains the BNF grammar's grammar.
- Press **run** a few more times: The `count` in the `output` area increases because the grammar keeps representing itself.

The EBNF[2] and BNF[2] modules share some common code included from the Base[2] module. In particular, the ability to create a scanner from the literals and token patterns in a grammar is implemented in the Base[2] module.

The design objective was to decouple the modules as far as possible and avoid duplicating code, but for the fact that a BNF[2] `Grammar`[2] can be constructed from an EBNF[2] `Grammar`[2] using the BNF[2] factory method `Grammar.fromEBNF()`[2].

There are some subtle but significant differences which reduce the number of classes used to represent a BNF‡ grammar:

- a BNF[2] `Rule`[2] is an ordered pair which contains a BNF[2] non-terminal `NT`[2] at left and a flat list of terminals and non-terminals at right,
- the same non-terminal can be at left in several rules, and
- a non-terminal has a list of rules where it is at left.

The state table is computed by `check()`[2] and owned by the BNF[2] `Grammar`[2] object. A BNF[2] `Parser`[2] is a separate object which owns the state stack and is consumed during the parsing operation, i.e., a BNF[2] `Parser`[2] cannot be reused once all input has been processed.

As before, a `Scanner`[2] is a separate object, created from the terminals of a BNF[2] `Grammar`[2] with the technique explained in chapter three. It translates a string into a list of `Tuple`[2] objects each of which contains a piece of the input string, the line number in the input, and a BNF[2] `Lit`[2] or `Token`[2] object representing the piece of the input.

- In Example 10/01[1] the very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar.
- Press **scan** to see the pattern used in the scanner and the `Tuple`[2] objects for the text in the `program` area.

The BNF[2] parsing method `Parser.parse()`[2] can either be called with a string, or it can be called repeatedly, either with a list of `Tuple`[2] objects or with a function providing such lists. Because the parsing state is stacked, successive input lists can be pushed to the parsing method, i.e., parsing could even operate interactively. `null` acts as end of all input and matches `$eof` in rule zero.

An illegal input character is represented as a `Tuple`[2] containing `null` rather than a `Lit`[2] or `Token`[2] object. This kind of `Tuple`[2] will cause an error during recognition.

- In Example 10/01[1] the very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar.
- Toggle either **parser** to see all transitions or **actions** to see only `reduce` and
- toggle **build lists** to collect the input into lists.

- Finally, press **parse** to observe recognition:

```
> g.config.build = true
> g.config.trace = /./
> g.parser().parse(program)
STATE TUPLE    MESSAGE                 RETURNS
  0  (1) 'a'   shift   1               null
  1  (1) 'b'   shift   3               null
  3  (1) 'c'   reduce  root: 'a' 'b';  [ 'a' 'b' ]
  0  (1) 'c'   goto    2               null
  2  (1) 'c'   shift   4               null
  4  (1) 'c'   reduce  root: root 'c'; [ [ 'a' 'b' ] 'c' ]
  0  (1) 'c'   goto    2               null
  2  (1) 'c'   shift   4               null
  4  eof $eof  reduce  root: root 'c'; [ [ [ 'a' 'b' ] 'c' ] 'c' ]
  0  eof $eof  goto    2               null
  2  eof $eof  accept                  [ [ [ 'a' 'b' ] 'c' ] 'c' ]
[ [ [ 'a' 'b' ] 'c' ] 'c' ]
```

Recognition implicitly contains an *Observer* design pattern‡. Input terminals are processed and trigger some of the state changes and transitions described earlier and the transitions result in messages: `shift`, `reduce`, `goto`, `accept`, and perhaps `error`. The parser's `observe()`[2] method reacts to these messages. It sends the message to the `trace()`[2] method if the configuration option `Grammar.config.trace`[2] is set, i.e., **parser** is set on the practice page, and to the `build()`[2] method if `Grammar.config.build`[2], i.e., **build lists**, is set, or if `Action`[2] methods were supplied in the first call of the `parse()`[2] method. For `reduce` messages `build()`[2] assembles recognized input into lists and sends them to `Action`[2] methods, if any.

The classic implementations of this parsing technique, yacc‡ and bison‡, spend considerable effort on optimizing the state table. `BNF`[2] was designed to be more tutorial in nature; therefore, the state table is a list of `State`[2] objects. Each holds `marks`, a list of one or more `Mark`[2] objects, i.e., the marked rules, and a `messages` collection which maps the symbols for the outgoing edges of the state to the possible messages which are sent to `observe()`[2] and which define the operations on the stack. For display purposes these messages are encoded as `Message`[2] objects.

As an aside, this chapter uses the term *actions* for semantic actions implemented as `Action`[2] methods as introduced in previous chapters. In contradistinction, the term *messages* is used for the operations on the state stack because they are also sent as messages to `observe()`[2]. The `reduce` *messages* trigger the semantic *actions*, if any.

Constructing the State Table

The state table is constructed by the `check()`[2] method which is called once after the rules of a `BNF Grammar`[2] have been represented. The first `State`[2] object is created with a `Mark`[2] object for rule zero marked at the beginning:

```
$accept: ◊ start-symbol $eof;
```

`check()`[2] uses factory methods `state()`[2] and `mark()`[2] to create state 0:

```
this.states.push(this.state([this.mark(this.rules[0], 0)]));
```

A state is created with a list of one or more marked rules. Together they are called the *core* marks and they uniquely define the state. The state additionally contains the *closure*‡ of the core: all rules for all non-terminals which immediately follow a marker. Here is the factory method which is called with a list of `Mark`[2] objects:

```
state (core) {
  const coreLength = core.length;
  const messages = {};
  // compute closure: loop over core and add(ed) marks
    ...
  return new State(this, core, coreLength, messages);
}
```

The state stores the number of core marks because only cores have to be compared to determine if two states are equal.

Each state maps terminals and non-terminals to messages which control parsing:

message verb	**information**	**effect**
`shift`	next state	push onto state stack, advance in input.
`reduce`	rule	pop state stack by rule length, uncover state which expects the non-terminal.
`goto`	next state	push onto state stack.
`accept`		done!

The `State`[2] object contains a `messages` object:

- `messages` maps a terminal symbol which immediately follows a mark in the state either to a `shift` message which consumes the corresponding input and transition to another state, or to a `reduce` message which completes recognition of a rule.
- `messages` maps a non-terminal symbol which immediately follows a mark in the state to a `goto` message which transitions to another state.
- `messages` maps `$eof` if it immediately follows a mark in the state to an `accept` message which completes parsing.
- The keys of `messages` are exactly all symbols which immediately follow one or more marks in the state. All other terminal symbols would not be expected.

`messages` is set up when the rules in the closure are added to the `core` of the state:

```
  // compute closure: loop over core and add(ed) marks
  for (let c = 0; c < core.length; ++ c)
    // for each incomplete mark
    if (!core[c].complete) {
      // next symbol in a mark
      const s = core[c].rule.symbols[core[c].position];
      if (s instanceof NT && !(s.ord in messages))
        // add all rules for a new non-terminal, marked at 0
        s.rules.forEach(rule => core.push(this.mark(rule, 0)), this);
      // map this next terminal or non-terminal to null
      messages[s.ord] = null;
    }
```

This loop runs as long as new marked rules are added — until it reaches the eventual end of `core`.

`.complete` is true for a `Mark`[2] if the marker follows all symbols of the rule.

So far, all values in `messages` are `null` and the keys of `messages` are the symbols that immediately follow one of the marks in the state.

In a similar loop in the `check()`[2] method, all state objects are then asked to `advance()`[2], i.e., to create outgoing edges and more states as needed to terminate the edges, and to fill in all `messages` collections:

```
// tell each state to advance
// this creates new states which are also advanced
for (let s = 0; s < this.states.length; ++ s)
  this.states[s].advance(s);
```

The heavy lifting happens in the `advance()`[2] method:

```
advance (stateNumber) {
  // create reduce messages for complete rules
  this.marks.forEach(mark => {
    if (mark.complete) {
      ...
    } // done with every complete rule
  }, this);
```

The first step is to enter `reduce` messages into the `messages` collection for all `complete` marks. The details are discussed below.

Each literal, token, and non-terminal has a unique ordinal number, starting with 0. `messages` maps this number to a message. Once the `reduce` messages have been entered, all other values in

`messages` will have to be `shift` or `goto`, with one exception:

```
  // create accept/shift messages for each next symbol which has none
  for (let a in this.messages) {
    if (this.messages[a] == null) {
      if (a == this.grammar.lit().ord) {
        // special case: $eof
        this.messages[a] = this.grammar.accept();
        this.grammar.rules[0].reduced = true;
```

`$eof` only appears in rule zero and this symbol can only lead to a `reduce` or `accept` message. `accept` means that rule zero is successfully recognized.

If the symbol is not `$eof`, the message will be `shift` or `goto` and needs a target state which is computed by advancing the marker across the symbol in all `Mark`[2] objects where the marker is currently just before the symbol:

```
      } else {
        // create next core by advancing marker over one symbol
        const next = [ ];
        let symbol = null;
        this.marks.forEach(mark => {
          // find a as next symbol in all marks
          if (!mark.complete && a == mark.rule.symbols[mark.position].ord) {
            // remember symbol and push mark after symbol
            symbol = mark.rule.symbols[mark.position];
            next.push(mark.advance());
          }
        }, this);
```

`next` is a list, used to collect all new `Mark`[2] objects with the marker moved across the `symbol`. `advance()`[2] uses the factory method to create a new mark with the same rule but the marker in the next position. `next` will contain at least one `Mark`[2] object, otherwise `a` would not have been a key in `messages` for this state.

`next` is the core of the target state for the `shift` or `goto` message to be recorded for the key `a` in `messages`. If the core cannot be found among the known states, a new state needs to be created:

```
        // add new state with next as core, if any
        // shift/goto existent or new state
        if (!this.grammar.states.some((state, s) =>
            state.equals(next) ?
              (this.messages[a] = this.grammar.shift_or_goto(symbol, s),
                true) : false, this)) {
          this.messages[a] =
            this.grammar.shift_or_goto(symbol, this.grammar.states.length);
          this.grammar.states.push(this.grammar.state(next));
        }
```

This concludes creating all but the `reduce` messages, and it looks as if nothing can go wrong. Unfortunately, entering the `reduce` messages will show that there can be problems. This part of

the `advance()`[2] method is discussed next but it was already demonstrated above that not every grammar is suitable for stack-based parsing because there can be conflicts.

Conflicts

The previous section did not discuss how `reduce` messages are entered into a state's `messages` table before the other messages are filled in. A `reduce` message requires a `.complete` rule and it only makes sense to enter it for terminals which can follow the mark's rule's non-terminal:

```
  // create reduce messages for complete rules
  this.marks.forEach(mark => {
    if (mark.complete) {
      // rule we are in
      const rule = mark.rule;
      // for each terminal which can follow the rule in the grammar
      for (let t in rule.nt.follow) {  // ordinal number
        const f = rule.nt.follow[t];   // terminal which can follow
        if (!(t in this.messages)) {   // can it follow in this state?
          // if t is not in messages it cannot follow this state -> reduce
          rule.reduced = true;
          this.messages[t] = this.grammar.reduce(f, rule);
```

`.follow` contains all terminals which can follow a rule. Computing `.follow` for EBNF grammars was discussed in chapter four. For BNF grammars the computation is easier because BNF rules are simpler.

If a terminal in `.follow` is not yet a key in the state's `messages` table, it cannot lead from the state to another one; therefore, this terminal needs to be added with a `reduce` message.

However, if the terminal is already a key in the `messages` table, there is a *conflict*. If the terminal was associated with a `reduce` message previously, the terminal is in the `follow` set for some other complete rule, i.e., there are two different rules which can be reduced before the terminal is accepted:

```
        } else if (this.messages[t] != null) {
          // t is in messages and messages[t] is already set as a reduce
          const r2 = this.messages[t].info; // the other rule

          ++ this.grammar.rr;
          error('for', f.toString(), 'reduce/reduce conflict between',
            '(' + rule + ')', 'and', '(' + r2 + ')');

          // resolve for rule which is first in the grammar
          if (rule.index < r2.index) this.messages[t].info = rule;
```

Technically, this *reduce/reduce conflict* makes the grammar unsuitable for stack-based parsing because the state table's message is not unique. Traditionally, as a stopgap measure in such a case, the first rule in the grammar is selected to be reduced.

If there is no prior `reduce` message, the terminal is in the `messages` table because it will be associated with a `shift` message when the marker is moved across it, as discussed in the previous

section:

```
        else { // this.messages[t] == null
          ++ this.grammar.sr;
          error('shift/reduce conflict between',
            f.toString(), 'and rule', '(' + rule + ')');
        } // done with conflicts
      } // done with every t which can follow
    } // done with every complete rule
  }, this);
```

This is called a *shift/reduce conflict* and makes the grammar unsuitable for stack-based parsing, too. In this case the stopgap measure is to prefer the `shift` message.

A *shift/reduce* Conflict

Example 10/03[1] shows that preferring the `shift` action can be desirable, for example to resolve the classic *dangling* `else` problem in favor of the innermost `if` statement.

```
statement: 'if' Number statement;
statement: 'if' Number statement 'else' statement;
statement: Number;
```

The grammar above has a shift/reduce conflict for `else`.

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and check on the conflict.
- Toggle **build lists** and
- press **parse** to see that

```
if 1
  if 2 3 else 4
```

is recognized as

```
[ 'if' '1' [ 'if' '2' [ '3' ] 'else' [ '4' ] ] ]
```

i.e., the `else` is recognized as part of the innermost `if`.

- Toggle **actions** and
- press **parse** again to confirm:

```
STATE TUPLE       MESSAGE                          RETURNS
  2  (2) 'else'   reduce  statement: Number;       [ '3' ]
  2  (0) $eof     reduce  statement: Number;       [ '4' ]
  7  (0) $eof     reduce  statement: 'if' Number   [ 'if' '2' [ '3' ] 'else
  5  (0) $eof     reduce  statement: 'if' Number   [ 'if' '1' [ 'if' '2' [
[ 'if' '1' [ 'if' '2' [ '3' ] 'else' [ '4' ] ] ]
```

A *reduce/reduce* Conflict

The built-in resolution for a shift/reduce conflict, i.e., to `shift` rather than to `reduce`, can often be accepted. However, reduce/reduce conflicts are usually more serious and need to be investigated carefully.

Example 10/04[1] contains a grammar, hugely simplified, where an `expression` is based on a `condition` or arithmetic and where a `condition` is based on a comparison or arithmetic:

```
1  expression: condition;
2  expression: Number;
3  condition: sum '<' sum;
4  sum: Number;
```

- The very first button should show bnf. If not, click it until it does.
- Toggle **states** and press **new grammar** to represent and check the grammar.
- Check the `output` area to see the reduce/reduce conflict in state 1 between the rules in line 2 and 4 above which is resolved in favor of the rule in line 2.
- Toggle **parser** or **actions** and
- press **parse** to confirm that a single `Number` such as `0` can be recognized.
- Move the last rule into second place and press **new grammar** again to see that the conflict is now resolved in favor of that rule.
- Press **parse** to confirm that now a single `Number` such as `0` cannot be recognized:

```
1  (0) $eof is not allowed
2  expecting: '<'
3  irrecoverable error
```

LR(1)

As a final note, the implementation discussed here is a simplification of Knuth's LR(1)‡. BNF[2] was designed to illustrate the principle and to at least be capable of handling all the examples from the previous chapters.

The idea of using the `follow` set of a rule's non-terminal to avoid some conflicts when the complete rule is processed is convenient but not as restrictive as possible.

Each marked rule starts with a nonterminal in a known context and one could consider only those terminals which can follow in that context, i.e., each mark should include it's own `follow` set. This results in larger state tables but potentially fewer conflicts and, therefore, more grammars which are suitable for LR(1) parsing.

Precedence and Associativity

The LL(1) grammar for interpreting arithmetic expressions written in EBNF was a bit cumbersome. Operator associativity had to be implemented in the semantic actions, not in the grammar itself.

Example 10/05[1] contains a more convenient grammar:

```
sum:      product | add | subtract;
add:      sum '+' product;
subtract: sum '-' product;
product:  factor | multiply | divide;
multiply: product '*' factor;
divide:   product '/' factor;
factor:   power | term;
power:    term '**' factor;
term:     number | '(' sum ')';
number:   Number;
```

This time, left associativity for `sum` and `product` is expressed through left recursion, right associativity for `power` is expressed through right recursion. Operator precedence is encoded into the grammar because the non-terminals `sum`, `product`, `factor`, and `term` establish appropriate levels.

Both, recursive descent‡ and the stack-based parser, can apply an action when a rule is reduced. The translation from EBNF to BNF does not change the structures which are presented to the action. As a result, actions for immediate evaluation with this grammar are very intuitive, for example:

```
class Actions {
  // sum: product | add | subtract;
  sum (value) { return value; }

  // add: sum '+' product;
  add (sum, x, product) { return sum + product; }
    ...
  // term: number | '(' sum ')';
  term (... arg) {
    return arg.length > 1 ? arg[1] : arg[0];
  }

  // number: Number;
  number (number) { return parseInt(number, 10); }
}
```

Here is an expression which produces `2` as a result:

```
  1 + 2*3 - 45/(1 + 2**3**2 / 4**3)
```

The grammar in example 10/05[1] uses EBNF notation — because `|` is used to separate alternatives — but the grammar is left recursive and has to be processed using the stack-based parser:

- The very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to translate the grammar from EBNF to BNF, represent, and check it.
- Press **parse** to recognize and evaluate the expression in the `program` area.
- Toggle **parser** and
- press **parse** again to observe evaluation.

The actions for `sum`, `product`, `factor`, and `term`, each, receive one or three values and each action just extracts and returns the appropriate value.

The actions for `add`, `subtract`, etc., each, receive two values, separated by an operator, perform immediate evaluation, and return the result.

None of the actions has to deal with associativity or precedence.

Yet another grammar for arithmetic expressions, in example 10/06[1], is as short as it can get

```
expr:     add | subtract | multiply | divide | power
          | '(' expr ')' | number;
add:      expr '+' expr;
subtract: expr '-' expr;
multiply: expr '*' expr;
divide:   expr '/' expr;
power:    expr '**' expr;
number:   Number;
```

but this grammar lacks any information about operator precedence and associativity.

- The very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to translate the grammar to BNF and try to represent and check it.
- Unfortunately, there are 25 shift/reduce conflicts...

Following yacc‡, precedence and associativity are specified in example 10/07[1] as a sequence of precedence levels preceding the grammar rules:

```
%left '+' '-';
%left '*' '/';
%right '**';

expr:     add | subtract | multiply | divide | power
  ...
```

Precedence increases in the order of the level statements. Each level statement contains one or more terminal symbols. Terminal symbols in the same level statement have the same precedence and associativity. A precedence level can also start with `%nonassoc` to suppress associativity, e.g., for comparison operators.

The grammar rules remain unchanged for example 10/07[1] but the precedences make no sense in the context of recursive descent:

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar. Note that precedences are not allowed for recursive descent.
- Click the very first button until it shows stack-based.
- Press **new grammar** to represent and check the grammar. Note that there are no more conflicts *reported*.
- Press **parse** to confirm that the expression in the `program` area can be recognized and evaluated.
- Toggle **parser** and
- press **parse** again to observe evaluation as a result of the `reduce` messages.

Principles

Consider the grammar

```
expr: expr 'a' expr;
expr: expr 'b' expr;
expr: Number;
```

and the input

```
1 a 2 b 3
```

and two trees which could be built to suggest that `a` and `b` are mathematical operations on the inputs `1`, `2`, and `3`:

The left tree suggests that operator `a` has precedence over `b`, e.g., multiplication happens before addition, or — for equal precedence — that `a` is left-associative, e.g., addition and subtraction happen from left to right.

The right tree suggests that operator `b` has precedence over `a`, or — for equal precedence — that `b` is right-associative, e.g., exponentiation or assignment happen from right to left.

What happens if creating the state table has reached the following point:

```
expr a expr  ◊  b
```

Rule 1 in the grammar above is complete, and it can be followed by terminal `b`.

The left tree can be built if rule 1 is reduced first, the right tree requires a `shift` of `b` so that rule 2 can be reduced first.

Therefore:

1. If the last terminal in a complete rule (`a`) has higher precedence than the "follow" terminal (`b`), the rule should be reduced.
2. If the last terminal in a complete rule has lower precedence than the "follow" terminal, there should be a `shift`.
3. If the last terminal in a complete rule and the "follow" terminal have equal precedence, the rule should be reduced if the last terminal is left-associative, and there should be a `shift` if the last terminal is right-associative.
4. Otherwise, i.e., with equal precedence and no associativity, there remains a shift/reduce conflict.

This will be implemented below.

Hiding Conflicts

It should be noted that precedences *hide* conflicts, but they are a natural way to implement arithmetic. It is usually a good idea to check the state table before and after adding precedence information to make sure the precedences have the intended effect.

Example 10/08[1] is a "dangling else" without a visible conflict:

```
1  %right 'else';
2
3  statement: 'if' Number statement %prec 'else';
4  statement: 'if' Number statement 'else' statement;
5  statement: Number;
```

A BNF rule can include a trailing `%prec` clause to specify precedence and associativity explicitly by referencing a terminal on a precedence level defined earlier (lines 3 and 1 above).

For this grammar there is no message about the shift/reduce conflict demonstrated earlier and state 5 shows that this state table will still reduce `else` with the closest `if`:

```
1  state 5
2    statement: 'if' Number statement ◊ %prec 'else';
3    statement: 'if' Number statement ◊ 'else' statement %prec 'else';
4
5    $eof         reduce (statement: 'if' Number statement %prec 'else';)
6    'else'       shift 6
```

It is tempting to think that changing `%right` to `%left` would reduce `else` with an outer `if` but in fact the change renders the grammar useless:

```
1  rule 2 is never reduced
2    ...
3  state 5
4    statement: 'if' Number statement ◊ %prec 'else';
5    statement: 'if' Number statement ◊ 'else' statement %prec 'else';
6
7    $eof         reduce (statement: 'if' Number statement %prec 'else';)
8    'else'       reduce (statement: 'if' Number statement %prec 'else';)
9
10 errors: 1
```

Now the second grammar rule is never reduced, i.e., input cannot contain `else`.

Actually, we are in case (3) described above. The `%prec` clause assigns the same precedence to the rule which the "follow" terminal `else` has. Right associativity will cause a `shift` in this case, i.e., `else` will be reduced with the innermost `if`.

EBNF and Precedence

The state table is computed for a BNF grammar — or for an EBNF grammar internally translated into BNF. How should a `%prec` clause apply to EBNF to be translated to BNF?

The clause must always apply to a single BNF rule. EBNF brackets and braces have to be translated into several BNF rules because each BNF rule can only contain one symbol sequence. This suggests that a `%prec` clause can only apply to a symbol sequence in EBNF, i.e., to one (of perhaps several) alternatives.

Example 10/09[1] is a small but useful example which shows how to add a unary minus to the arithmetic example discussed above:

```
%left '-';

expr:     subtract | minus | number;
subtract: expr '-' expr;
minus:    '-' expr;
number:   Number;
```

With the additional semantic action

```
class Actions09 extends Ten.Actions07 {
  minus (x, a) { return - a; }
}
```

this will get `1` as the result of evaluating

```
-1 -- 2
```

The grammar enforces that the unary minus has precedence over the binary minus. Without a precedence level the grammar is ambiguous. `%left` defines associativity for binary minus and settles the conflicts; replacing `%left` by `%right` would cause a chain of subtractions to be evaluated right to left. Unary minus only "fits" the grammar at an implicitly higher precedence.

Example 10/10[1] adds another precedence level and a `%prec` clause to the EBNF grammar:

```
%right Number;
%left '-';

expr:     subtract | minus | number;
subtract: expr '-' expr;
minus:    '-' expr %prec Number;
number:   Number;
```

This — deliberately — gives the unary minus *less* precedence than the binary minus. The same input now evaluates to `-3`.

It is instructive to trace both evaluations. With the rule `%prec` clause,

```
5  (0) $eof         reduce  expr: minus;            -3
```

is the last `reduce` message, without it it is

```
4  (0) $eof          reduce  expr: subtract;          1
```

i.e., with lower precedence `minus` is the last operation, with higher precedence for `minus` the subtraction is last.

Both examples are EBNF grammars because they contain alternatives for `expr`, but both require the stack-based parser because of the precedence levels.

The `%prec` clause is not applied to an EBNF rule — which can have several alternatives — but to a symbol sequence such as `'-' expr`. Example 10/11[1] takes this to an extreme:

```
%right Number;
%left '-';

expr: expr '-' expr |
      '-' expr %prec Number |
      Number;
```

The action for this one-line grammar is just based on the number of arguments:

```
class Actions {
  expr (... arg) {
    switch (arg.length) {
    case 3: return arg[0] - arg[2];
    case 2: return - arg[1];
    case 1: return parseInt(arg[0], 10);
    }
  }
}
```

Why use `Number` in the precedence level statement for this example? All that is required is to connect the unary minus with a `%prec` clause to a terminal symbol which is — for fun — on a level below binary minus. Assigning a precedence to `Number` through a level statement does not affect the state table and does not introduce a new, otherwise unused terminal symbol.

Trick question: without a `%prec` clause and without changes to the grammar structure, how can

```
-1 -- 2 ---- 4
```

be evaluated as either `-7` or `5`?

Conflict Resolution

`Precedence`[2] objects represent precedence levels and contain associativity and a list of terminals which are at the same level of precedence. `Grammar` objects for either EBNF[2] or BNF[2] contain a list of `Precedence`[2] objects in order of increasing precedence; for EBNF[2] they can only be used when translating to BNF[2].

`BNF.Lit`[2] and `BNF.Token`[2] objects, `EBNF.Seq`[2] and `BNF.Rule`[2] objects can have a `.prec` property

with similar information which can be set up during construction because the precedence table has to precede the rules of a grammar.

The Grammars' Grammars for EBNF[2] and BNF[2] contain the appropriate syntax and the bootstrap process described in chapter nine includes the necessary actions to create the representations.

Thus, the stage is set to resolve some conflicts. They are detected when there are complete rules and `reduce` messages are about to be entered into the `messages` table for a state. Recall the code structure in the `advance()`[2] method set up above:

```
1  advance (stateNumber) {
2                       // create reduce messages for complete rules
3    this.marks.forEach(mark => {
4      if (mark.complete) {
5        const rule = mark.rule;                    // rule we are in
6        // for each terminal which can follow the rule in the grammar
7        for (let t in rule.nt.follow) {            // ordinal number
8          const f = rule.nt.follow[t];   // terminal which can follow
9          if (!(t in this.messages)) {
10              // if t is not in messages it cannot follow this state
11           ...                               // create reduce message
12         } else if (this.messages[t] == null) {
13             // if t is in messages -- there might be a s/r conflict
14           ...                                   // explained below
15         } else {
16           ...   // t already set as a reduce -- report r/r conflict
17         }                                      // done with this t
18       }                      // done with every t which can follow
19     }                                  // done with complete mark
20   }, this);                                // done with all marks
21                        // create shift messages, next state, etc.
```

Recall that `rule` is complete, ready to be reduced (line 5 above), `f` is one of the "follow" terminals (line 8), `t` is it's key in `messages` (line 7), and a message has not yet been entered (line 9).

If both, rule and "follow" terminal, actually have a precedence the shift/reduce conflict can be

resolved with the following code inserted above (at line 14):

```
            // if t is in messages -- there might be a s/r conflict
          if (rule.prec && f.prec.assoc) {
                      // rule and terminal have defined precedence
            if (rule.prec.level > f.prec.level) {
                        // rule's precedence is higher -> reduce
              ...
            } else if (rule.prec.level < f.prec.level) {
             // rule's precedence is lower -> fall through to shift
            } else                              // equal precedence
              switch (rule.prec.assoc) {
              case '%left':   // rule is left-associative -> reduce
                ...
              case '%right':  // rule is right-associative -> shift
                break;                              // fall through
              case '%nonassoc':         // non-associative -> error
                delete this.messages[t];
                          // i.e. f as input would be an error
              }
          } else {
              // no precedence available -- shift/reduce conflict
            ...
          }                   // resolved as a shift (fall through)
```

Conflict resolution through precedences is implemented just as the principles above require. In order to concentrate on the overall structure, calls on the factory methods for the messages and insertion in the state's `messages` table have been omitted in the code above. The complete code can be viewed from the documentation of the `advance()`[2] method.

Recognition

Given a state table owned by a `Grammar`[2] object, how is input recognized, how are values collected and presented to any `Action`[2] methods, and what needs to be done so that a recursive descent parser and a stack-based parser, both derived from the same EBNF grammar, can interact with the exact same `Action`[2] methods?

Recognition starts with a `Parser`[2] object which is constructed by the `Grammar`[2] object and owns a state stack which is simply a list of indices into the `Grammar`[2]'s list of states. The current state is the top element on the stack; therefore the stack initially contains just one element with value zero.

`parse()`[2]

The `parse()`[2] method takes a list of `Tuple`[2] objects which a `Scanner`[2] can prepare from a string. `null` in the list represents the end of all input, i.e., `$eof`; otherwise `parse()`[2] is called again with more input.

`parse()`[2] executes a loop which can be aborted:

```
1  try {
2    while (true) {
3      if (!this.current) this.next();          // lookahead as needed
4      if (this.current.t                           // expected input?
5          && this.current.t.ord in this.state.messages) {
6        if (this.process(this.current))
7          this.next();                                    // consumed
8      } else                // illegal character or unexpected input
9        this.recover();
10   }
11 } catch (outcome) {
12   if (outcome instanceof Array) return outcome[0];       // success
13   throw outcome;                // true: more input, else: failure
14 }
```

`this.current` is a `Tuple`[2] representing the current input terminal and `next()`[2] is called to move along — just not beyond `null`, i.e., `$eof`. By convention, `next()`[2] throws `true` to request more input.

If there is a message for the current input terminal in the current state `process()`[2] executes the message for the `Tuple`[2] and returns `true` if the terminal should be consumed, i.e., after a `shift` message (line 6 above). `false` would protect the terminal for another round, i.e., after `reduce` plus `goto`.

If there is no message the input is not expected and `recover()`[2] gets a chance to arrange for the loop to continue as discussed below (line 9).

If a value, wrapped into an `Array`, is thrown `parse()`[2] would unwrap and return it (line 12) — this is a hook for an observer to return a value from a semantic action. Any other `throw` (but `true`) will terminate recognition ungracefully (line 13).

`process()`[2]

The `process()`[2] method accepts an input `Tuple`[2], finds a message, if any, offers it to `observe()`[2],

and manages the state stack:

```
1  process (tuple) {
2                                  // get message and inform observer
3    const verb = this.state.messages[tuple.t.ord].message,
4      info = this.state.messages[tuple.t.ord].info,
5      result = this.observe(tuple, verb, info);
6    switch (verb) {                                 // dispatch message
7    case 'accept':
8      throw [ result ];                     // parse ends with success
9    case 'shift':
10     this.stack.push(info);                     // shift to new state
11     return true;                                // tuple is consumed
12   case 'reduce':
13           // pop the stack by the length of the rule, uncover state
14     this.stack.length -= info.symbols.length;
15                 // there always follows a goto for the non-terminal
16     const g = this.state.messages[info.nt.ord];
17     this.observe(tuple, g.message, g.info);     // observe the goto
18     this.stack.push(g.info);                    // goto to new state
19     return false;                             // tuple still available
20   }
21 }
```

The message is obtained from the current state (lines 3 and 4 above) and a call to `observe()`[2] with the message produces a result (line 5). As described above `observe()`[2] will invoke an `Action`[2] method, if there is one, or return the result `null` so that there always is a value.

If the message is `accept` the result is wrapped into an `Array` and thrown (line 8), to be returned from `parse()`[2].

For `shift` the message contains the next state which is pushed and the input terminal will be consumed (lines 10 and 11).

For `reduce` the message references the rule so that the state stack can be popped (line 14). In addition, a `goto` for the non-terminal is sent to `observe()`[2] and the corresponding state is pushed (lines 17 and 18). In this case the input terminal is not consumed (line 19).

`observe()`[2]

The `observe()`[2] method accepts a `Tuple`[2] and the corresponding message, asks the `build()`[2] method to compute a result in place of `null` if `Grammar.config.build`[2] is set or if `Action`[2] methods were supplied in the first call to `parse()`[2], asks the `trace()`[2] method to create and display a trace if the configuration option `Grammar.config.trace`[2] is set, and displays an error message if the incoming message is `error` with non-empty information.

`observe()`[2], `build()`[2], and `trace()`[2] are deliberate hooks for subclassing at different levels. For example, yacc‡ and bison‡ have a significantly different approach to collecting input and presenting it to semantic actions because they do not have to accommodate EBNF‡.

build()²

Chapter five explained how the recursive descent parser collects recognized input into nested lists which can be processed by Action² methods, if any. Similarly, the stack-based parser can use Action² methods if an observer collects input in response to `shift` messages and calls actions in response to `reduce` messages.

When the parse()² method is first called, if the Grammar.config.build² configuration option is set or if Action² methods are supplied a `value` stack is added to the Parser². The build()² method accepts a Tuple² and the corresponding message, manages the value stack to parallel the state stack, and presents values to Action² methods, if any, to potentially change the result of `reduce` messages:

```
1  build (tuple, verb, info) {
2    switch (verb) {
3    case 'shift':                        // shift: collect input text
4      this.values.push(tuple.value);
5      return null;
6    case 'reduce':                      // reduce: pop by rule length
7      const len = info.symbols.length;
8      let result = this.values.splice(- len, len);       // can be []
9      result = this.act(info.nt.name, result); // apply action if any
10     this.values.push(result);       // goto (follows!): push result
11     return result;
12   case 'accept':             // accept: return result of start rule
13     return this.values.pop();
14   }                          // actual goto can be silently ignored
15 }
```

build()² is called with the current input `tuple` and `verb` and `info` of the message triggered by the state and the input.

A `shift` message indicates that the parser transitions to a new state in response to some input and build()² pushes the input onto the value stack for the stacks to stay in sync (line 4 above).

A `reduce` message indicates that a rule has been completed and build()² has to pop values off the value stack to keep it in sync. The extra `info` references the rule to be reduced. The rule's number of symbols (line 7) determines how many values have to be popped off the value stack (line 8) to be presented to an Action² method, if any (line 9). Either the list of values or the result of the action is returned for `reduce` (line 11) and pushed as value associated with the immediately following `goto` message (line 10) which itself will be silently ignored (line 14).

The method act()² (line 9) is shared between the recursive descent and the stack-based parser. It takes a rule name and a list of values collected for the rule and tries to find and execute an action method with the rule name. act()² returns the result of the action method or the list of values.

An `accept` message indicates that the parser is done and build()² pops the top value off the value stack and returns it. Eventually, process()² sends it back as the parse()² method's result value. Success!

Translating EBNF

The `build()`[2] method seems to produce nested lists just like the recursive descent parser does. However, example 10/12[1] shows that there is a problem:

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar.
- Toggle **build lists** and
- press **parse** to see the result.

This BNF grammar uses left recursion to specify that a sentence should consist of one or more sequences, where a sequence consists of `a`, an optional `b`, and `c`:

```
some: seq;
some: some seq;
seq: 'a' opt 'c';
opt: 'b';
opt: ;
```

This program consists of two sequences:

```
a b c
a c
```

Here is the output, reformatted for clarity:

```
[
  [ [ 'a' [ 'b' ] 'c' ] ]
    [ 'a' [     ] 'c' ]
]
```

By default, the observer creates a list whenever a rule is reduced, i.e., rule 4 (above) will return a list containing `b` and rule 5 will return an empty list. Note that the recursive descent parser will instead return `null` when there is no input for an optional alternative — which is enclosed in brackets in the EBNF notation.

Rule 3 returns a list when a sequence is found — these are the brackets directly around the sequences in the output above. Rule 1 returns a list containing the list with the first sequence ever and rule 2 creates yet another — outermost — list whenever another sequence is found. The `Action`[2] method for `some` would have to count arguments to know which of the two rules for `some` is reduced in order to process it's arguments correctly.

The EBNF grammar in example 10/13[1]

```
some: { 'a' [ 'b' ] 'c' };
```

accepts the same input with two sequences shown above but it produces a more uniform output

(reformatted for clarity):

```
[
  [
    [ 'a' [ 'b' ] 'c' ]
    [ 'a'   null  'c' ]
  ]
]
```

This is independent of the choice of parser:

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the result for the recursive descent parser.
- Click the very first button to show stack-based,
- press **new grammar** to represent and check the grammar and
- press **parse** to see that the stack-based parser produces the same result.

A BNF grammar expresses alternatives as rules with the same non-terminal at left. EBNF uses `|` to separate alternatives and insists on unique non-terminals as rule names. Additionally, EBNF uses brackets for optional alternatives and braces for alternatives which can be repeated.

The EBNF rule in example 10/13[1] produces a list for the selected alternative. The braces produce a list containing one inner list for each alternative that was recognized. The brackets produce `null` or a list for the recognized alternative.

The two examples together hint at the translation from EBNF to BNF which the factory method `BNF.Grammar.fromEBNF()`[2] implements. Example 10/14[1] combines the two grammars and (almost) allows direct comparison between automatic and manual translation:

```
1  cases: ebnf bnf;
2
3  ebnf:  'ebnf' { 'a' [ 'b' ] 'c' };
4
5  bnf:   'bnf' ebnf4;
6
7  ebnf4: ebnf5 | ebnf4 ebnf5;
8  ebnf5: 'a' ebnf6 'c';
9
10 ebnf6: | 'b';
```

The two `cases` in this EBNF grammar accept the same sequences as before, prefixed with `ebnf` or `bnf` to select different rule sets for recognition.

The `ebnf` rule uses braces and brackets for iteration and the optional part (line 3 above).

The `bnf` rule (line 5) uses the left-recursive `ebnf4` rules to express iteration of one or more sequences (line 7), the `ebnf5` rule to express the sequence itself (line 8) — i.e., the single alternative enclosed by the braces in the `ebnf` rule — and the `ebnf6` rules (line 10) to express the optional `b` — i.e., the enclosed alternative and the brackets in the `ebnf` rule.

Preparing the grammar in Example 10/14[1] shows (more or less) that the manual translation agrees with the translation performed for the stack-based parser:

- The very first button should show stack-based. If not, click it until it does.

- Press **new grammar** to represent and check the grammar.
- The error is due to the fact that EBNF does not permit empty alternatives — regardless of parser. Remove the empty alternative of the `ebnf6` rule and
- press **new grammar** again to represent and check the grammar.

The BNF rules can be found near the beginning of the `output` area:

```
0 $accept: cases $eof;
1 cases: ebnf bnf;
2 $-6: ;
3 $-6: 'b';
4 $-5: 'a' $-6 'c';
5 $-4: $-5;
6 $-4: $-4 $-5;
7 ebnf: 'ebnf' $-4;
  ...
```

Technically, special characters cannot be part of non-terminal names in either grammar notation — they are used internally for BNF rule zero and for the additional rules resulting from the translation. BNF rule 2 (above) is simply copied from EBNF (line 1 below), BNF rules 2 to 7 are the translation of the `ebnf` rule (line 3 below). Rules 2 and 3 take care of the brackets (compare to line 5), rules 5 and 6 take care of the iteration with rule 4 for the content (compare to lines 6 and 7). Altogether, the BNF rules are the same as the explicit manual translation in the example's (almost) EBNF grammar:

```
cases: ebnf bnf;

ebnf:  'ebnf' { 'a' [ 'b' ] 'c' };
  ...
ebnf6: | 'b';
ebnf5: 'a' ebnf6 'c';
ebnf4: ebnf5 | ebnf4 ebnf5;
```

The translation of brackets and braces from EBNF to BNF according to the pattern suggested above requires additional BNF rules which should result in more deeply nested lists to be built. Why can the stack-based parser turn out the same lists as the recursive descent parser?

There have to be hidden actions which happen when the additional rules are reduced. The `reduce` case implemented in the `build()`[2] method turns out to be a bit more devious than

shown before:

```
1   case 'reduce':                      // reduce: pop by rule length
2     const len = info.symbols.length;
3     let result = this.values.splice(- len, len);       // can be []
4     if (this.grammar.ebnf &&  // extra rule to translate from EBNF?
5         info.nt.name.startsWith(this.grammar.config.uniq)) {
6       if (len == 2 && info.symbols[0] === info.nt)  // left recurse
7         result.splice(0, 1, ...result[0]); // don't add one more []
8       else if (!len)        // return null if brackets find nothing
9         result = null;
10    } else                                // rule copied from EBNF
11      result = this.act(info.nt.name, result);     // action if any
12    this.values.push(result);       // goto (follows!): push result
13    return result;
```

Additional rules are only present if the BNF grammar is translated from EBNF (which is noted in a property `.ebnf` of the grammar). The names of the additional BNF rules will always start with a prefix which can be configured as `Grammar.config.uniq`[2]. Those rules cannot have explicit `Action`[2] methods because they only exist if the BNF grammar is translated from EBNF as shown above:

```
1   0 $accept: cases $eof;
```

Rule 0 collects an extra list but the top-level `parse()`[2] method corrects that implicitly.

```
1   $-6: ;
```

Rules like the one above describe an optional alternative which recognizes nothing. If they are completed `null` must be returned rather than the empty list which was collected. This is taken care of in line 9 in the code above.

```
1   $-6: 'b';
2   $-5: 'a' $-6 'c';
3   $-4: $-5;
```

Rules like these three above collect the same lists which the recursive descent parser collects for alternatives, even rules like the last one which recognizes the first iteration within braces.

```
1   $-4: $-4 $-5;
```

This kind of a left-recursive rule handles further iterations of alternatives in braces. The rule pattern is recognized in line 6 in the code above and in line 7 the list collected for the second symbol is added (not appended!) to the list already collected by flattening this first list with the spread syntax†.

All other completed rules will invoke the same actions as for the recursive descent parser.

The hidden actions may look complicated but they preserve the Idioms for Actions described in chapter five which have worked well throughout.

Representing structured input always requires some kind of list and tree building. Iteration constructs in the grammar and the hidden actions avoid the list manipulations which would otherwise be needed to accommodate iteration implemented as recursion.

EBNF notation seems more convenient and the stack-based parser is more powerful. Thanks to the hidden actions this is a winning combination.

Error recovery

Most of the implementation of recognition was discussed before. The `parse()`[2] method executes the following loop:

```
1  try {
2    while (true) {
3      if (!this.current) this.next();          // lookahead as needed
4      if (this.current.t                           // expected input?
5          && this.current.t.ord in this.state.messages) {
6        if (this.process(this.current))
7          this.next();                                    // consumed
8      } else                 // illegal character or unexpected input
9        this.recover();
10   }
11 } catch (outcome) {
12   if (outcome instanceof Array) return outcome[0];       // success
13   throw outcome;                // true: more input, else: failure
14 }
```

`this.current` is the next input symbol to be recognized, represented as a `Tuple`[2]. `next()`[2] is called to advance in the input (line 3 above).

`this.state` is the current parser state, i.e., the top value on the parser's stack, and `this.state.messages` maps the unique ordinal values of expected input symbols to messages (line 5).

`process()`[2] is called (line 6) to send such a message to `observe()`[2], change the parser state, and request to consume input or return a recognition result as appropriate.

`recover()`[2] is called (line 9) to handle unexpected inputs in order to continue recognition if possible. `recover()`[2] could:

- discard input, i.e., call `next()`[2], or
- pop parser states and perhaps tell `observe()`[2], or
- select a combination of these.

Theoretically, expected input symbols could even be created and inserted; however, that might well result in a never ending recognition loop with a very long sentence-to-be...

`recover()`[2] first sends an `error` message to `observe()`[2] indicating (from the state table) which

input symbols were expected:

```
recover () {
  const error = this.grammar.token(),       // unique symbol $error
    eof = this.grammar.lit();                 // unique symbol $eof

  this.observe(this.current, 'error',
    `${this.current} is not allowed\nexpecting:` +
    Object.entries(this.state.messages).reduce((s, kv) => s +=
      kv[1].symbol instanceof Base.T && kv[1].symbol !== error ?
        ' ' + kv[1].symbol : '', ''));
```

Next, illegal input characters are silently dropped:

```
  while (!this.current.t)
    this.next();          // next tuple, throws true for more input
                          // now at tuple with $eof or actual input
```

Finally, `recover()`[2] has no choice but to pop the stack and/or to discard input. The choice can be fine tuned in the grammar: following yacc‡, there is a special `$error` token which can be used in rules to let a grammar have some control over error recovery. `$error` is used like any other token on the right-hand side of a rule and, therefore, ends up as a key in some `state.messages`:

```
   pop: while (this.stack.length > 0) {
     if (!(error.ord in this.state.messages)) { // $error unexpected
       this.observe(this.current, 'error', null); // pop value stack
       this.stack.pop();                          // pop state stack
       continue;
     }                                       // else $error expected
     if (this.process(this.grammar.tuple(this.current.lineno, error)))
       while (true)        // did shift $error, now search for input
         if (this.current.t?.ord in this.state.messages) {
           if (this.process(this.current)) {    // did shift current
             this.#current = null;                        // consume
             return;                  // recovery should be complete
           }          // else did reduce+goto before current, retry
         } else if (this.current.t === eof) {        // end of input
           this.grammar.message('terminating at ' + this.current);
           break pop;                              // cannot recover
         } else {               // current is not expected: discard
           this.grammar.message('discarding ' + this.current);
           this.next();
         }
     //   else did reduce before $error, process $error again or pop
   }
   throw 'irrecoverable error';
 }
```

If `$error` is not expected in `state.messages` (line 2 above) `recover()`[2] sends a special `error` message so that `observe()`[2] can pop a value stack, if any (line 3), and `recover()`[2] pops the state

stack (line 4).

If `$error` is never expected an `irrecoverable error` is eventually thrown (line 23) and aborts recognition.

Otherwise, once `$error` is expected, `process()`[2] deals with it, wrapped in an artificial `Tuple`[2] (line 7).

If there is a `shift` message for `$error` the search is on for a `shift` message for an input symbol (lines 8 to 20). If found, this input symbol is consumed (line 11) and `recover()`[2] returns (line 12).

If instead there was a `reduce` message before the input symbol the message is executed, followed by `goto` as usual, and the input symbol is retried (back to line 8).

If the input symbol is not expected (at line 14) end of input results in an `irrecoverable error` (line 15 to line 23) and other input is discarded (line 18 back to line 8).

Finally, if there is a `reduce` message before `$error` it is executed, followed by `goto` as usual, and either the new state expects `$error` or the new state is popped, too (line 7 back to line 1).

This is a tutorial implementation; therefore, all discarded symbols are reported (line 18). The algorithm could be modified to be less chatty and perhaps require more than one successful `shift` message, i.e., accepting more than one input symbol, before normal processing is resumed.

Example 10/15[1] implements error recovery for a `sentence` consisting of a list of one or more copies of the letter `a`:

```
1  sentence: some;
2  sentence: $error;
3  some: 'a';
4  some: some 'a';
5  some: $error;
6  some: some $error;
```

- The very first button should show bnf. If not, click it until it does.
- Toggle **states** and
- press **new grammar** to represent and check the grammar.

There is a reduce/reduce conflict in state 2:

```
1  state 2
2    sentence: $error ◊;
3    some: $error ◊;
4
5    $eof         reduce (sentence: $error;)
6    'a'          reduce (some: $error;)
7    $error       reduce (some: $error;)
8  for $eof reduce/reduce conflict between (some: $error;) and (sentence: $error;)
```

The conflict happens at the end of input and is resolved in favor of the earlier rule 2 (line 5). This

can be seen if input is simply the letter `b`:

- Toggle **parser** and press **parse**:

```
STATE TUPLE           MESSAGE                    RETURNS
  0  (1) "b"          error   -message-          null
error: at (1) "b": (1) "b" is not allowed
expecting: 'a'
  0  eof "" $error    shift   2                  null
  2  eof $eof         reduce  sentence: $error;  null
  0  eof $eof         goto    4                  null
  4  eof $eof         accept                     null
```

- Move rule 2 to the end of the grammar,
- press **new grammar** to prepare the modified grammar, and press **parse**:

```
STATE TUPLE           MESSAGE                    RETURNS
  0  (1) "b"          error   -message-          null
error: at (1) "b": (1) "b" is not allowed
expecting: 'a'
  0  eof "" $error    shift   2                  null
  2  eof $eof         reduce  some: $error;      null
  0  eof $eof         goto    3                  null
  3  eof $eof         reduce  sentence: some;    null
  0  eof $eof         goto    4                  null
  4  eof $eof         accept                     null
```

- Toggle **build lists** and press **parse** again to see that different lists are built depending on which rule catches the error.
- Try different inputs — including none — to see that an `irrecoverable error` cannot happen for this grammar.

Idioms for `$error`

The placement of `$error` symbols in BNF grammar rules is guided by the following, conflicting goals:

- as close as possible to the start symbol of the grammar.

  This way there is always a point to recover from because there should always be a state very low on the stack in which `$error` is expected. The parser then is unable to clear its stack early, i.e., to not `accept` the end of input from the scanner.

- as close as possible to each terminal symbol.

  This way only a small amount of input would be skipped on each error.

- without introducing conflicts.

  This may be quite difficult. In fact, accepting shift/reduce conflicts is reasonable as long as they serve to lengthen strings. E.g., one can continue parsing an expression past an error, rather then accepting the same error at the statement level, thus trashing the rest of the expression.

Given these goals, there are typical positions for `$error` symbols:

- in each recursive construct, i.e., in iterations such as statement lists in a program, argument lists in a function call, etc.
- preferably not at the end of a rule because that is more likely to cause conflicts.
- non-empty lists (*some*) require two `$error` symbols: one for a problem at the beginning of the list, and one for a problem at the current end of the list — see example 10/15[1].
- possibly empty lists (*many*) require an `$error` symbol in the empty branch — if this causes bad conflicts the symbol could be added to the places where the possibly empty list is referenced.

Example 10/16[1] is a blueprint for typical constructs:

```
example: ;
example: example 'many' many ';';
example: example 'some' some ';';
example: example 'list' list ';';
```

Input consists of any number of examples. An `example` starts with the name of a construct and ends with a semicolon. There is no error recovery at this level — e.g., calling for a non-existing construct causes an `irrecoverable error`. Each construct should recover from all errors so that the terminating semicolon can be found. The following action

```
class Actions {
  example (... arg) { if (arg.length) puts(`reduced ${arg[1]}`); }
}
```

will confirm successful recovery *within* each construct.

```
many: ;
many: many 'm';
many: many $error;
```

A `many` construct contains any number of `m` but no other letters. One additional rule with `$error` in place of `m` is sufficient for a full recovery. For example, in the following input

```
many;
many m m;
many A;
many B m;
many m C m;
many m D m E;
```

the upper-case letters are not allowed but the `many` construct is still reduced six times.

```
some: 's';
some: some 's';
some: $error;
some: some $error;
```

Similarly, `some` contains one or more `s` but no other letters and two rules with `$error` in place of

`s` are required for a full recovery. Again, in the following input

```
some s s;
some F;
some G s;
some s H s;
some s s I;
```

the upper-case letters are not allowed but the `some` construct is still reduced five times.

```
list: 'l';
list: list ',' 'l';
list: $error;
list: list $error;
list: list $error 'l';
```

Finally, `list` is defined as a comma-separated list of one or more `l`. For programming language grammars this is probably the most frequently used construct. Unfortunately, it is also the most complicated. The following input contains many errors such as unexpected upper-case letters and missing or superfluous commas:

```
list l;
list l, l;
list l,;
list J;
list l K;
list l L l;
list l, M;
list l, N l;
list l, O,;
list l, P, l;
list l, Q, l,;
```

The output shows that `list` is still reduced eleven times.

Example 10/16[1] contains the grammar and the input shown above. It is instructive to compare input and error messages:

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the error messages.

As expected, every upper-case character in the input triggers a message, but the examples for `list` additionally complain about three unexpected semicolons. The output does *not* end with an `irrecoverable error`.

In order to avoid message bursts, `recover()`[2] discards illegal characters silently, but discarded tokens and literals are reported.

- Add a token definition `{ Bad: /[A-Z]/ }` and a rule `example: Bad;` so that the token is used in the grammar and recognized by the scanner and
- press **new grammar** and **parse** to see the discard messages.

Example 10/17[1] contains the same grammar and input as before and adds the following actions

to record the numbers of the rules which are reduced:

```
class Actions {
  example (example, tag, rules) {
    if (!(typeof example == 'string')) return '';     // example: ;
    return `${example}${tag} ${rules.join(' ')}\n`; // example: ...
  }

  many (many, m) {
    if (!many) return [ 5 ];                            // many: ;
    many.push(m == 'm' ? 6 : 7); return many;    // many: many 'm';
  }

  some (some, s) {
    if (some == 's') return [ 8 ];                    // some: 's';
    if (!some) return [ 10 ];                       // some: $error;
    some.push(s == 's' ? 9 : 11); return some;   // some: some 's';
  }

  list (list, comma, l) {
    if (list == 'l') return [ 12 ];                   // list: 'l';
    if (!list) return [ 14 ];                       // list: $error;
    if (comma == ',') list.push(13);          // list: list ',' 'l';
    else if (l == 'l') list.push(16);     // list: list $error 'l';
    else list.push(15);                        // list: list $error;
    return list;
  }
}
```

An action is called when a rule is reduced and selected by the non-terminal name. The actions above are based on the following:

- a `shift` of a terminal pushes the input value onto the value stack,
- a `shift` of `$error` pushes `null`,
- a `goto` pushes the result of a rule's action, and
- a rule's action receives one value corresponding to each symbol on the right-hand side.

The output shows for each recognized construct the list of rule numbers which were reduced during recognition, for example:

input	output	rules
`many;`	`many 5`	`5 many: ;`
`many m m;`	`many 5 6 6`	`6 many: many 'm';`
`many A;`	`many 5 7`	`7 many: many $error;`
`many B m;`	`many 5 7 6`	
`many m C m;`	`many 5 6 7 6`	
`many m D m E;`	`many 5 6 7 6 7`	

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar,
- toggle **no args** to suppress messages about mismatched argument counts, and

- press **parse** to see the raw result data — how often is each rule used?

Example 10/18[1] can be used to analyze some interactions of error recovery strategies. The idea is to recognize one or more `a` followed by one or more `b`. Each can be done with and without error recovery:

```
as:  'a';
as:  as 'a';

aes: 'a';
aes: $error;
aes: aes 'a';
aes: aes $error;
```

Both sets of rules, `as` and `aes`, recognize one or more `a` but only `aes` includes error recovery. `bs` and `bes` are defined in the same fashion.

Finally, a `sentence` starts by selecting a strategy, from `1` to `4`, followed by some `a` and then some `b`. As a fallback, `sentence` includes `$error` but no provision to try again:

```
sentence: '1' as  bs  ';';
sentence: '2' aes bs  ';';
sentence: '3' as  bes ';';
sentence: '4' aes bes ';';
sentence: $error;
```

A typical (faulty) input is

```
1 b a b a;
```

- The very first button should show bnf. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see that all but the strategy input in the `program` area is discarded.
- Toggle **actions** and press **parse** again to see why there is no `irrecoverable error`.
- Toggle **parser** and press **parse** again to find out which single input symbol is recognized.
- Change `1` in the `program` area to `2`, etc., and repeat the experiments for the different recovery strategies.
- Which strategy results in an `irrecoverable error` and why?
- Does the resolution of the conflict in state 23 make any difference?
- How many `a` or `b` are ever passed to an action, if any?

Error recovery for EBNF

Example 10/19[1] recognizes all sentences but one which example 10/16[1] recognizes:

```
examples: { example };
example: many | some | list | $error;
many: 'many' [{ 'm' }] ';';
some: 'some' { 's' } ';';
list: 'list' 'l' { ',' 'l' } ';';
```

Which "sentence" for example 10/16[1] is no longer acceptable?

A simple action can show the list that each `example` recognizes:

```
class Actions {
  example (e) { puts('example', g.dump(e)); }
}
```

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see that the recursive descent parser can handle all examples involving `many`:

```
many;
many m m;
many A;
many B m;
many m C m;
many m D m E;

some s s;
some F;
  ...
```

The recursive descent parser succeeds as long as it drops illegal characters but it fails after dropping `F` because *check before you call* cannot enter the iteration in the rule for `some` and the recursive descent parser is not designed to recover.

- To work with the stack-based parser, click the very first button to show stack-based.
- Press **new grammar** to represent and check the grammar and
- toggle **actions** and press **parse** to see how the stack-based parser survives all examples.
- Remove `| $error` from the second rule and try again.

`$error` has no effect in the recursive descent parser. However, it is an alternative of the top-level iterative construct in this grammar and allows the stack-based parser to recover. The trace shows that a lot of meaningful input is discarded.

- Reload the example,
- click the very first button until it shows stack-based,
- toggle **insert $error** to insert `$error` into all iteration constructs during the translation to BNF, and
- press **new grammar** to represent and check the grammar again.

There are a number of reduce/reduce conflicts which are all caused by the explicit `$error` token in the second rule.

- Remove `| $error` from the second rule,
- press **new grammar** to represent and check the grammar again, and
- toggle **actions** and press **parse** to see that the stack-based parser again survives all examples.

The trace shows that with the semi-automatic error recovery produced by **insert $error** a lot more meaningful input is recognized.

In general, the stack-based parser is preferable because it can work with more grammars, e.g., with a shift/reduce conflict for a dangling `else`, and it usually recovers well if `$error` is automatically inserted into iterations expressed with braces in EBNF.

If a grammar is used to explain the features of a programming language, the effects of braces and brackets in EBNF are easier to understand than left recursion and empty alternatives in BNF. Actions are easier to design for EBNF and they do not have to be changed for the translation to BNF.

Quick Summary

Parser Types.

- LL(1)‡ refers to recognition from left to right searching for the left-most derivation with one symbol lookahead. This is known as *top-down* because the left-most derivation determines the root of the parse tree. The lookahead symbol is used for the *check before you call* approach discussed in chapter four.
- LR(1)‡ instead searches for the right-most derivation. This is known as *bottom-up* because it creates the parse tree from the leaves to the root.
- Grammars suitable for LL(1) parsing are suitable for LR(1) parsing. The opposite is not true; e.g., some left-recursive grammars are suitable for LR(1) parsing.

State-Transition Table.

- A state-transition table is computed from a BNF grammar by considering all acceptable input sequences.
- States are sets of marked rules. Rules and rule lengths are finite, i.e., the set of states is finite.
- Starting with a mark at the beginning of the start rule of the grammar, closure‡ is applied to each new state, i.e., if the mark precedes a non-terminal, all rules for the non-terminal are added to the state and marked at their beginnings.
- `shift` transitions for each combination of a state and an input symbol are defined by moving each mark across the next terminal in it's rule. `goto` transitions, instead, involve non-terminals. This finite process can result in new marked rules and then new states.
- Completed rules have the mark at their end. For simplified LR(1)‡, a `reduce` transition is defined if the lookahead is in the follow set of the rule.

Conflicts and Resolution.

- There is a reduce/reduce conflict if several different rules in the same state have the mark at their end and if the lookahead is in the follow set of each. Recognition would be based on whichever conflicting rule appears earlier in the grammar.
- There is a shift/reduce conflict if in the same state there is a mark at the end of one rule and the lookahead is in the follow set and in a different rule the mark can be moved across the lookahead. In this case, recognition will be based on `shift`, i.e., accept more input, which implements, e.g., proper dangling else recognition.

Explicit *shift/reduce* Conflict Resolution.

- A precedence table preceding a grammar assigns decreasing precedence levels and associativity to lists of terminals.
- A precedence clause at the end of a single BNF rule copies precedence level and associativity to the last terminal in the rule.
- Conflicts are hidden and resolved according to the following table:

last rule terminal	follow terminal	resolution
higher precedence	lower	`reduce`
lower	higher	`shift`
equal, `%left`	equal	`reduce`
equal, `%right`	equal	`shift`
equal, `%nonassoc`	equal	error

Recognition.

- Based on a state-transition table, a LR(1) parser consumes lookaheads and pushes it's new states onto a stack until the combination of state and lookahead indicates that a rule has been completed. The states corresponding to the rule are popped and a new state corresponding to the completed rule is pushed. End of input should cause a transition to a legal state.
- The parser can be observed. An observer maintains a stack of values parallel to the states and offers them to actions when rules are reduced.
- EBNF can be translated to BNF and hidden actions added so that identical actions can be used by a top-down and a bottom-up parser.

Error Recovery

- If there is an unexpected input symbol the parser will pop the state stack until the input symbol is expected or the stack is empty, i.e., recognition failed.
- For better control, grammar rules can contain `$error` tokens which accept no input but, if expected in a state, will stop the stack popping and result in a state transition. Next, unexpected input symbols are discarded until either recognition can continue with a `shift` transition or the end of input is reached.

- `$error` tokens are most effective when inserted into iteration constructs, especially, when EBNF is translated into BNF to construct a more powerful parser.

11. Compiling Revisited

What's in a Tree?
What's with a Tree?

Chapter six started the implementation of a little language with expressions and control structures which concluded with first-order functions in chapter eight. Chapter seven contained a short section on type-checking. Overall, the discussion focussed on semantic analysis and code generation as part of syntax analysis, i.e., carried out by the actions called when grammar rules were completed.

This approach is called one-pass compilation‡ because the source program is immediately rewritten in the target language — it is not converted to an intermediate representation to be processed more than once. One-pass compilation‡ should require less time and memory but the resulting code might not be as performant.

This chapter explores a different approach: the source program is represented as a tree and visitors‡ take care of type-checking and code generation. The result is much better separation of concerns‡ and reusability‡ of the components. Base classes provide infrastructure for tree building and visiting. Mix-ins‡ implement tree-building actions for recognition and tree-processing methods for visitors and are used to allow selective sharing of methods.

Classes and Mix-Ins

Classes can be extended by adding or overriding methods. If a method `m()` exists in both, the original superclass‡, and the subclass‡ resulting from extending the superclass‡, the subclass method can call the superclass method as `super.m()`†, i.e., overriding a method does not discard it's code.

Mix-ins‡ are functions which conceptually add methods to classes:

```
1  const Mix = superclass => class X extends superclass {
2    hello () { console.debug('hello world', super.constructor.name); }
3  };
4  try       { new class A { } () . hello(); }
5  catch (e) { console.log(e.message); }
6  finally   { new (Mix(class A { })) () . hello ();
7              new (Mix(Mix(class A { }))) () . hello(); }
```

This code produces the following output:

```
1  (intermediate value).hello is not a function
2  hello world A
3  hello world
```

- Class `A` has no method `hello()` (code lines 4 and 5, output line 1).

- `Mix(class A {})` creates a class which has `A` as the superclass and has the method `hello()` (code line 6, output line 2).
- Mix-ins can be cascaded. `Mix(Mix(class A {}))` creates a subclass of the subclass created by `Mix(class A {})` and these subclasses have no names (code line 7, output line 3).

The point is that mix-ins‡ can be applied to "add" methods, but there is no extra code if the mix-ins‡ are not applied, i.e., mix-ins‡ can be used to group functionality which can optionally be added to a class. Once added it will be inherited.

The code above shows that JavaScript does not need language elements to specifically support mix-ins‡. There are different ways to implement such a feature. The technique shown above is taken from a paper by Justin Fagnani†.

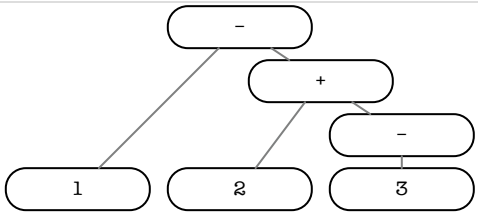
The examples in this chapter are largely cumulative. To avoid repetition, the practice page includes a module Eleven[2] which contains all classes and mix-ins‡ defined and used in the examples in this chapter. They are summarized in appendix D. The methods can be seen in the method browser[3].

Building a Tree

Chapter four looked at the arithmetic expression

```
1 - (2 + - 3)
```

which can be represented, e.g., by the following tree:



In JavaScript this tree can be represented using nested lists as follows:

```
[ 'subtract',
  [ 'number', 1 ],
  [ 'add',
    [ 'number', 2 ],
    [ 'minus',
      [ 'number', 3 ] ] ] ]
```

Each node of the diagram is an `Array` in JavaScript. The first element of each array is a tag, represented as a string with lower-case letters. The remaining elements are values and, in particular, further nodes represented as arrays.

Using Recursive Descent

Example 11/01[1] shows that it is relatively easy to create a tree for a list of arithmetic expressions.

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the tree which is built for the list of expressions in the `program` area.

Given the rules

```
number: Number;
term: number | '(' sum ')';
signed: [ '-' ] term;
```

the corresponding tree-building action methods would be something like

```
number (number) { return [ 'number', parseInt(number, 10) ]; }
term (...val) { return val.length == 1 ? val[0] : val[1] }
signed (minus, term) { return minus ? [ 'minus', term ] : term; }
```

and left associativity can be handled with an iteration in the grammar

```
product: signed [{ multiply | divide }];
multiply: '*' signed;
divide: '/' signed;
```

and by creating lists with "holes"

```
multiply (x, right) { return [ 'multiply', null, right ]; }
divide (x, right) { return [ 'divide', null, right ]; }
```

and assembling them with `reduce()`† from left to right as follows:

```
product (signed, many) { return (many ? many[0] : []).
  reduce((product, alt) => (alt[0][1] = product, alt[0]), signed);
}
```

Building a Tree for Arithmetic

`Build`[2] is the base class for the tree builders created in this chapter:

```
1  class Build {
2    /** Sets the property */
3    constructor (parser) { this.parser = parser; }
4
5    /** Tags node with source position as `.lineno` if available. */
6    _lineno (node) {
7      if (this.parser.current && this.parser.current.lineno)
8        node.lineno = this.parser.current.lineno;
9      return node;
10   }
11 }
```

`Build`[2] provides access to the `parser` (line 3 above), e.g., for error reporting and access to the source line numbers.

If errors are detected when trees are processed the error messages should refer back to the source program. Therefore, `_lineno()`[2] tries to add a property `.lineno` to a tree node (line 8) which references the current source seen by the parser during tree building.

`BuildRD()`[2] is a mix-in‡ which contains the actions to build a tree for arithmetic expressions based on the recursive descent grammar shown in the previous section. Combined as `Build_RD(Build)`, the superclass `Build`[2] and the mix-in `Build_RD()`[2] result in the subclass which should be handed to the `parse()`[2] method.

Example 11/01[1] uses the trick introduced in chapter six to accomplish this:

```
1  (() => {  // define and immediately use an anonymous function
2    class Build { ... }
3    const Build_RD = superclass => class extends superclass { ... };
4    return Build_RD(Build);
5  }) ()
```

The class and mix-in‡ definitions are placed into an anonymous function where `return` delivers the actual class for `parse()`[2] (line 4 above) and the anonymous function is called immediately after being defined (line 5).

- The very first button should show descent. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the tree which is built for the expression in the `program` area.
- Replace the entire text in the `actions` area by `Eleven.Build_RD(Eleven.Build)` to import the builder class from the module Eleven[2] and repeat the steps.

Using Precedences

The LL(1) grammar for arithmetic expressions has to use iterations and nested rules so that the resulting tree reflects the expected associativities and precedences. Instead, example 11/02[1] uses

explicit precedences and an ambiguous, very recursive grammar:

```
%left '+' '-';
%left '*' '/';
%right '**';
%right Number;

expr:     add | subtract | multiply | divide | power
          | minus | '(' expr ')' | number;
add:      expr '+' expr;
subtract: expr '-' expr;
multiply: expr '*' expr;
divide:   expr '/' expr;
power:    expr '**' expr;
minus:    '-' expr %prec Number;
number:   Number;
```

In example 11/02[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the tree which is built for the expression in the `program` area.
- Delete the code in the `actions` area and compare the previous output to the nested lists which are built without the actions — these lists represent the same nesting and information but would be much harder to process again.

Using an SLR(1) grammar with precedences hugely simplifies the class actions for building, e.g.,

```
const Build_Number = superclass => class extends superclass {
  expr (...values) { return values.length > 1 ? values[1] : values[0]; }
  add (a, x, b) { return this._lineno([ 'add', a, b ]); }
    ...
};
```

The anonymous function pattern can be reused from the previous example[1]:

```
(() => {  // define and immediately use an anonymous function
  const Build_Number = superclass => class extends superclass {
    ...
  };
  return Build_Number(Eleven.Build);
}) ()
```

The advantage of using mix-ins‡ is that `Build_Number(Eleven.Build)` only contains methods which match the structure of the SLR(1) grammar, i.e., if the grammar is changed, only the corresponding mix-ins‡ has to be adapted. Build actions very closely match their rules.

In spite of different parsers, different grammars, and different actions, both examples, 11/01[1] and 11/02[1], build the same trees for sentences which are recognized by both grammars. This means that code for further processing of the trees can be shared.

Visiting a Tree

Visitors‡ are objects which apply algorithms such as evaluation, type-checking, code-generation, etc., to data structures such as a tree constructed by the action methods described in the previous section. Visitors can be used to separate concerns‡ in a way very similar to the interplay between grammar rules and actions.

Example 11/03[1] implements expression evaluation using a visitor. `Visit`[2] is the base class for visitors:

```
1  class Visit {
2    trace = false;      // RegExp selects tags to display
3
4    visit (node, trace) {
5      if (trace instanceof RegExp) this.trace = trace;
6      // not a list: return it
7      if (!(node instanceof Array)) return node;
8      // visit
9      let result;
10     const show = this.trace instanceof RegExp &&
11       this.trace.test(node[0]) ? this._dump(node, 0) : false;
12     try {
13       return result = this.constructor.prototype[node[0]].call(this, node);
14     } finally {
15       if (show) puts(show, 'returns', this._dump(result, 1));
16     }
17   }
```

The most important method is `visit()`[2]. It is called to apply the visitor's algorithm to a `node`. An optional parameter can turn on tracing by setting the `.trace` property (line 7 above) to make the setting permanent. The `node` argument should be an `Array`, otherwise it is simply returned (line 9). A "real" node, i.e., an `Array`, contains a tag as first element which is used to select a method of the visitor (line 15). The method is called with the node as the only argument and the result is returned. There can be tracing, depending on the node's tag and the setting of the property `.trace` (line 13). The method might change the contents of the node; therefore, part of the display is computed before the method is called (line 13) and it is shown together with the result returned by the method (line 17).

`Visit`[2] has a few more methods. `_tree()`[2] acts as an assertion. It recursively walks a tree, nodes before subtrees, and checks if there are methods for all the node tags. If not it throws an error

message.

```
1   _tree (node) {                  // recursively validates a tree
2     if (!(node instanceof Array)) return;
3     if (!node.length) throw 'empty node';
4     if (typeof node[0] != 'string') throw 'node tag is not a string';
5     if (!node[0].length) throw 'empty node tag';
6     if (node[0] == 'visit') throw "'visit' cannot be a tag";
7     if (typeof this.constructor.prototype[node[0]] != 'function')
8       throw node[0] + ': unknown node tag';
9     node.slice(1).forEach(node => this._tree(node));
10  }
```

`_dump()`[2] recursively converts a tree into a string — up to a certain depth. If the argument is not a tree it is decoded (line 2 below). Otherwise the tag and — depending on depth — the other entries are shown (line 10). If present as `.lineno`, the source line number is appended to the node display (line 12). Similarly, information from `.type` would be shown (line 13).

```
1   _dump (node, shallow = -1) {    // recursively dumps a tree
2     if (!(node instanceof Array))
3       switch (typeof node) {
4       case 'boolean':
5       case 'number': return node;
6       case 'string': return "'" + node.replace(/(['\\\n])/g, "\\$1") + "'";
7       default:       return typeof node;
8       }
9
10    let result = '[ ' + (!shallow ? this._dump(node[0]) :
11      node.map(item => this._dump(item, shallow - 1)).join(' ')) + ' ]';
12    if ('lineno' in node) result += '.' + node.lineno;
13    if ('type' in node) result += ':' + node.type;
14    return result;
15  }
```

Finally, `_error()`[2] is used to report and count errors during a visit:

```
1   errors = 0;         // counts calls to _error()
2
3   _error (lno, ... s) {
4     if (typeof lno == 'number' && lno > 0) lno = `line ${lno}:`;
5     else lno = s.splice(0, 1)[0];
6     puts(`error ${++ this.errors}: ${lno}`, ... s);
7   }
8 };
```

Methods such as `_tree()`[2], `_dump()`[2], and `_error()`[2] should not be mistakenly used for node tags; therefore, these "private" method names start with an underscore. Because of it's importance for the concept, `visit()`[2] is a deliberate exception to this convention.

Interpreting Arithmetic Expressions

With the infrastructure provided by Visit[2], interpretation of an arithmetic expression amounts to a traversal of the tree, i.e., evaluation visits to the subtrees before an operation specific to a node is applied:

```
const Eval_Number = superclass => class extends superclass {
  add (node) { return this.visit(node[1]) + this.visit(node[2]); }
  subtract (node) { return this.visit(node[1]) - this.visit(node[2]); }
  multiply (node) { return this.visit(node[1]) * this.visit(node[2]); }
  divide (node) { return this.visit(node[1]) / this.visit(node[2]); }
  power (node) { return this.visit(node[1]) ** this.visit(node[2]); }
  minus (node) { return - this.visit(node[1]); }
  number (node) { return this.visit(node[1]); }
};
```

Each node has a tag defining the operation and one or two operand values or subtrees which have to be evaluated first. In the implementation shown above the evaluation order for the subtrees is defined by the implementation language, i.e., strictly left-to-right for JavaScript. This could be changed by temporarily storing the second subtree value in a local variable in each method.

Main Program

The mix-in‡ Main()[2] contains action methods which let new top-level rules for the grammar do the job of a main program — at the expense of not discarding the parser first. Here are the new rules, start rule first:

```
run:      main;
main:     dump;
dump:     expr;
expr:     add | subtract | ... ;
```

The action for dump[2] will display and return the tree built by `expr`:

```
const Main = (superclass, ...args) => class extends superclass {
  dump (tree) {
    puts(new Visit()._dump(tree));
    return tree;
  }
```

The action for main[2] (discussed below) arranges for one or more visits to the tree and returns the last visit — in this case expression evaluation — as a parameterless function and the action for run[2] executes this function and returns the result:

```
  run (funct) { return funct(); }
```

The rules for `dump` or `run` can be omitted and the other rules adjusted if there is no need to display the tree or if `main` is expected to create an executable which can be run more than once.

Classes or mix-ins‡ for visitors can be imported or the anonymous function pattern in the `actions` area can be used to define them. The call on the mix-in‡ `Main()`[2] determines in which order the visitors are applied:

```
1  (() => {  // define and immediately use an anonymous function
2    // base class with visitor methods
3    class Visit {
4      ...
5    }
6    // mix-in for expression evaluation
7    const Eval_Number = superclass => class extends superclass {
8      ...
9    };
10   // mix-in with top-level actions, runs visitors
11   const Main = (superclass, ...args) => class extends superclass {
12     dump (tree) { ... }
13     main (tree) { ... }
14     run (funct) { ... }
15   };
16   // result of anonymous function
17   return Main(Eleven.Build_Number(Eleven.Build), // builder actions
18     Eval_Number(Visit),                          // evaluation visitor
19     /./);                          // node selector for trace, if any
20 }) ()
```

The action for `main`[2] depends on a closure† over extra arguments of the mix-in‡ `Main()`[2]. `...args` consists of one or more visitor classes (lines 11 and 18 above) and optionally one regular expression to control tracing (line 19). The last visit, if any, is returned as a function:

```
1    main (tree) {
2      let [lastVisitor, lastTree, trace] = this._doVisits(tree, args);
3      return () => lastVisitor.visit(lastTree, trace);
4    }
```

The private method `_doVisits()`[2] is called with a tree and a list of extra arguments to consume. It returns a list containing the last visitor object, the tree to apply it to, and the tracing expression,

if any:

```
1   _doVisits (tree, args) {
2     let trace;                    // (first) trace pattern, if any
3     const visitors = args.filter(arg => {       // remove patterns
4         if (!(arg instanceof RegExp)) return true;
5         if (!trace) trace = arg;
6         return false;
7       }),
8       tail = visitors.splice(-1, 1);       // last visitor, others
9     if (!tail.length) throw 'main: no visitors';
10    let caller;          // each visitor is constructed with caller
11    [tree, caller] = visitors.reduce(([tree, caller], Visitor) => {
12      const visitor = new Visitor (caller);  // create next visitor
13      visitor._tree(tree);                        // validate tree
14      tree = visitor.visit(tree, trace);                 // visit
15      if (trace) { puts(visitor._dump(tree)); }    // trace, if any
16      if (visitor.errors) throw `${visitor.errors} error(s)`;
17      return [tree, visitor];           // done; next visit if any
18    }, [tree, this]);               // first caller is the builder
19    const lastVisitor = new tail[0](caller); // last visitor object
20    lastVisitor._tree(tree);                    // validate tree
21    return [ lastVisitor, tree, trace ];
22  }
```

First the arguments are split into a trace pattern, if any, a list of all but the last class, if any, and the last class (lines 2 to 9 above). One after another, a visitor is created from a class, each incoming tree is checked and visited, and each resulting tree is shown if requested (lines 11 to 15). The last (or possibly only) visitor is created, the last tree is checked, and the visitor, tree, and trace pattern are returned (lines 19 to 21).

In example 11/03[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the tree which is built for the expression in the `program` area, followed by the result of interpreting the expression.
- Remove the `run` rule,
- press **new grammar** to represent and check the new grammar,
- press **parse** to see the tree, and
- press **run** to interpret.
- Remove the `dump` rule, adjust the `main` rule, and repeat the steps.
- Finally, add a regular expression such as `/./` to the call to `Main()`[2] near the end of the `actions` area to trace interpretation.

Representing a Little Language

Example 11/04[1] adds variable names, comparisons, control structures, and a few other statements

to the grammar:

```
1  %nonassoc '=' '<>' '>' '>=' '<' '<=';
2  %left     '+' '-';
3    ...
4  stmts:    stmt [{ ';' stmt }];
5  stmt:     assign | print | loop | select;
6  assign:   Name '=' expr;
7  print:    'print' expr [{ ',' expr }];
8  loop:     'while' expr 'do' stmts 'od';
9  select:   'if' expr 'then' stmts [ 'else' stmts ] 'fi';
10
11 expr:     eq | ne | gt | ge | lt | le
12           | add | ... | number | name;
13 eq:       expr '=' expr;
14   ...
15 le:       expr '<=' expr;
16 add:      expr '+' expr;
17   ...
18 name:     Name;
```

Comparisons have lower precedence than arithmetic operators (line 1 above) and — different from JavaScript — are not associative, i.e., they cannot be chained. They are added to the rule for expressions (line 11). Any expression can be the condition for a `while` loop (line 8) or an `if` statement (line 9) — this will be restricted later with type checking.

A name can be a term in an expression (line 12). It can be used in an assignment statement (line 6).

Tree building reuses `Build_Number()`[2] to support arithmetic expressions and add the mix-ins‡ `Build_Cmps()`[2] for comparisons, `Build_Stmts()`[2] for statements, and `Build_Names()`[2] for names. Comparisons are represented just like arithmetic operations:

```
const Build_Cmps = superclass => class extends superclass {
  // eq: expr '=' expr;
  eq (a, x, b) { return this._lineno([ 'eq', a, b ]); }
  ...
};
```

A single statement is just returned, but a list of two or more statements is collected into one

`'stmts'` node:

```
const Build_Stmts = superclass => class extends superclass {
  // stmt: assign | print | loop | select;
  stmt (stmt) { return stmt; }

  // stmts: stmt [{ ';' stmt }];
  stmts (stmt, many) {
    return many == null ? stmt :
      this._lineno([ 'stmts',
        ...many[0].reduce(
          (stmts, alt) => { stmts.push(alt[1]); return stmts; },
          [ stmt ])
      ]);
  }
```

A `print` statement is represented as a `'print'` node with a list of expression subtrees:

```
  // print: 'print' expr [{ ',' expr }];
  print (x, expr, many) {
    return this._lineno([ 'print',
      ...(many ? many[0] : [ ]).reduce(
        (exprs, alt) => { exprs.push(alt[1]); return exprs; },
        [ expr ])
    ]);
  }
```

A `while` loop is represented as a `'loop'` node with a condition expression and a single statement or a list of statements:

```
  // loop: 'while' expr 'do' stmts 'od';
  loop (w, expr, d, stmts, o) {
    return this._lineno([ 'loop', expr, stmts ]);
  }
```

An `if` statement is represented as a `'select'` node with a condition and one or two dependent statements or lists:

```
  // select: 'if' expr 'then' stmts [ 'else' stmts ] 'fi';
  select (i, expr, t, left, opt, f) {
    const result = this._lineno([ 'select', expr, left ]);
    if (opt) result.push(opt[1]); return result;
  }
};
```

In this little language, names are simply `Name` tokens referencing variables; however, they could be references to functions or array elements, etc. Therefore, tree building actions for statements

and operands involving names are collected into a separate mix-in‡:

```
const Build_Names = superclass => class extends superclass {
  // assign: Name '=' expr;
  assign (name, x, expr) {
    return this._lineno([ 'assign', name, expr ]);
  }
  // name: Name;
  name (name) { return this._lineno([ 'name', name ]); }
};
```

As an example, Euclid's Algorithm‡

```
x = 36; y = 54;
while x <> y do
  if x > y
    then x = x - y
    else y = y - x fi od;
print x
```

is represented as the following tree:

```
[ 'stmts'
  [ 'assign' 'x' [ 'number' 36 ] ]
  [ 'assign' 'y' [ 'number' 54 ] ]
  [ 'loop' [ 'ne' [ 'name' 'x' ] [ 'name' 'y' ] ]
    [ 'select' [ 'gt' [ 'name' 'x' ] [ 'name' 'y' ] ]
      [ 'assign' 'x' [ 'subtract' [ 'name' 'x' ] [ 'name' 'y' ] ] ]
      [ 'assign' 'y' [ 'subtract' [ 'name' 'y' ] [ 'name' 'x' ] ] ] ] ]
  [ 'print' [ 'name' 'x' ] ] ]
```

In example 11/04[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the tree.
- Change the grammar so that assignment uses a `name` reference rather than a `Name` token. How does the tree change and what would be the consequence for evaluation?
- Add a start rule to `dump` the resulting tree with line numbers and use the mix-in `Eleven.Main()`[2] to provide the action.

Interpreting a Little Language

Example 11/05[1] reuses `Eleven.Main()`[2] for the top-level rules and the classes and mix-ins‡ discussed so far for the build actions and to interpret arithmetic expressions. It adds new mix-ins‡ `Eval_Cmps()`[2] to interpret comparisons, `Eval_Stmts()`[2] to interpret statements, and

`Eval_Names()`[2] to interpret names with a symbol table:

```
(() => {  // define and immediately use an anonymous function
  // ... new mix-ins ...

  // builder and interpreter-visitor
  return Eleven.Main(Eleven.Build_Stmts(
                       Eleven.Build_Names(
                         Eleven.Build_Cmps(
                           Eleven.Build_Number(Eleven.Build)))),
                     Eval_Stmts(
                       Eval_Names(
                         Symbols(
                           Eval_Cmps(
                             Eleven.Eval_Number(Eleven.Visit))))));
}) ()
```

Comparisons are interpreted just like arithmetic operations with a postorder traversal, i.e., the subtrees are visited and interpreted first and then the comparison is applied:

```
// mix-in with comparisons
const Eval_Cmps = superclass => class extends superclass {
  // [ 'eq' a b ]
  eq (node) { return this.visit(node[1]) == this.visit(node[2]); }
    ...
};
```

A list of statement nodes is interpreted one by one:

```
// mix-in with statements and list evaluation
const Eval_Stmts = superclass => class extends superclass {
  // [ 'stmts' stmt ... ]
  stmts (node) { node.slice(1).forEach(stmt => this.visit(stmt)); }
```

For a `'print'` node all expression subtrees are visited and interpreted and the results are displayed together, separated by blanks:

```
  // [ 'print' value ... ]
  print (node) { puts(...node.slice(1).map(value => this.visit(value))); }
```

A `'loop'` node is interpreted by repeatedly interpreting the condition subtree followed by the loop body subtree if the condition is true and returning as soon as the condition is false:

```
  // [ 'loop' cond stmt ]
  loop (node) { while (this.visit(node[1])) this.visit(node[2]); }
```

A `'select'` node is interpreted by evaluating the condition subtree followed by the `then` subtree

if the condition is true or the `else` subtree if the condition is false and there is one:

```
  // [ 'select' cond then else? ]
  select (node) {
    if (this.visit(node[1])) this.visit(node[2]);
    else if (node.length > 3) this.visit(node[3]);
  }
};
```

In this little language, names are simply `Name` tokens referencing variables; however, they could be references to functions or array elements, etc. Therefore, statements and operands involving names are collected into a separate mix-in which requires a symbol table.

The mix-in‡ `Symbols()`[2] uses a Map† to store objects with arbitrary properties as descriptions for names. If available, this Map† is imported from the preceding visitor so that a sequence of visitors can add more information (line 5 below):

```
// mix-in with symbol table
const Symbols = superclass => class extends superclass {
  constructor (prev, ... more) {
    super(prev, ... more);
    this.symbols = prev?.symbols ?? new Map ();
  }

  _alloc (name) {
    let symbol = this.symbols.get(name);         // check if exists
    if (!symbol)                                // create with ordinal
      this.symbols.set(name,
        symbol = { ord: this.symbols.size + 1 });
    return symbol;
  }
};
```

The mix-in‡ construction creates an anonymous class which can have an explicit constructor. If it does one has to be careful how arguments are managed — it is probably best to assume that all mix-ins‡ along the chain receive the same set of arguments (line 3 above).

A private method `_alloc()`[2] returns a description for a name and creates one if none exists. Each symbol receives a property `.ord` which labels them in order of creation, starting with `1` (line 12)

The mix-in‡ `Eval_Names()`[2] can use `Symbols()`[2] to interpret operations on a `name`:

```
1  // mix-in with name evaluation
2  const Eval_Names = superclass => class extends superclass {
3    // [ 'name' name ]
4    name (node) {
5      const symbol = this._alloc(node[1]);
6      if (!('value' in symbol)) symbol.value = 0;
7      return symbol.value;
8    }
9
10   // [ 'assign' name value ]
11   assign (node) { this._alloc(node[1]).value = this.visit(node[2]); }
12 };
```

When a reference to a name is interpreted, a description is located or created (line 5 above), initialized with zero if there is no value (line 6), and the current value is returned (line 7).

To interpret an `'assign'` node a description for the name is located or created and the value is computed by a visit to the subtree and stored in the description. Note that the evaluation order of the implementation language might play a role: should the name be defined and initialized before or after the value to be assigned is computed?

In example 11/05[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to create the executable.
- Press **run** to execute; supply different values for `x` and `y`.
- Add a top-level `run` rule to interpret immediately and repeat the steps.
- Remove the `dump` rule, adjust the `main` rule, and repeat the steps.
- Add a regular expression such as `/./` to the call to `Main()`[2] to trace interpretation.
- Finally, change the `actions` area so that all code is imported from module Eleven[2] (this just requires small changes to the `return` statement), recompile, and execute.

Rewriting a Tree

A visitor can copy a tree or modify it in place, e.g., when type checking a program. In chapter seven it was demonstrated in example 7/02[1] that type checking‡ is very similar to evaluation, with types replacing actual values. Result types are propagated from the leaves of the tree through the operator nodes to the statement nodes, and at each level the expectations are checked against the incoming types, e.g., a Boolean condition for a loop, or a string value for assignment to a variable declared with a string type. Mismatches can be reported as errors or corrected by applying implicit conversions.

Alternatively, for type inference‡, sets of types acceptable to operators are pushed to the leaves of the tree, pruned for literals, and act as constraints on variables.

Typed Expressions

Example 11/06[1] implements evaluation for expressions which include `bool`, `number`, and `string` values. The grammar is extended with the typical operations:

```
%left     'or';
%left     'and';
%nonassoc '=' '<>' '>' '>=' '<' '<=';
  ...
main:     expr;
expr:     or | and | eq | ... | minus | not | len | input | cast
          | '(' expr ')' | bool | number | string;
  ...
or:       expr 'or' expr;
and:      expr 'and' expr;
not:      'not' expr %prec Number;
bool:     'true' | 'false';

len:      'len' expr %prec Number;
input:    'input' [ String String ];
string:   String;

cast:     '(' type ')' expr %prec Number;
type:     'bool' | 'number' | 'string';
  ...
```

`String` is a new kind of token: a nonempty literal string value enclosed in single quotes, with single quotes, linefeeds, and backslashes escaped by backslashes:

```
String: /'(?:\\['\\\n]|[^'\\\n])+'/
```

Extending the grammar requires a few new actions for tree building:

```
const Build_Bool = superclass => class extends superclass {
  // or: expr 'or' expr;
  or (a, x, b) { return this._lineno([ 'or', a, b ]); }
  ...
```

`or()`[2], and similarly `and()`[2] and `not()`[2], build nodes for the Boolean operations. It is left up to interpretation or code generation whether or not `and` and `or` are short circuited‡.

```
  // bool: 'true' | 'false';
  bool (bool) { return this._lineno([ 'bool', bool == 'true' ]); }
};
```

`bool()`[2] represents one of the new literals `'true'` and `'false'` as a `'bool'` node with the

corresponding raw Boolean value.

```
1  const Build_String = superclass => class extends superclass {
2    // len: 'len' expr;
3    len (x, b) { return this._lineno([ 'len', b ]); }
```

`len()`[2] represents a unary operation which computes the length of a string as a `'len'` node.

```
1    // input: 'input' [ String String ];
2    input (i, opt) {
3      return (opt ? opt : [ ]).
4        reduce((r, s) =>
5          (r.push(s.slice(1, -1).replace(/\\(.)/g, '$1')), r),
6        [ 'input' ]);
7    }
```

`input()`[2] represents an input operation with optional prompt and default strings as an `'input'` node which contains the raw strings, if any, without the enclosing quotes and without backslash escapes (line 5 above).

```
1    // string: String;
2    string (string) {
3      return this._lineno([ 'string',
4        string.slice(1, -1).replace(/\\(.)/g, '$1') ]);
5    }
6  };
```

`string()`[2] represents a `String` literal as a `'string'` node which contains the raw string value.

```
1  const Build_Cast = superclass => class extends superclass {
2    // type: 'bool' | 'number' | 'string';
3    type (type) { return type; }
4    // cast: '(' type ')' expr;
5    cast (l, type, r, b) { return this._lineno([ 'cast', type, b ]); }
6  };
```

Finally, `cast()`[2] represents an explicit type cast with a type name and a value subtree as a `'cast'` node.

Interpreting Typed Expressions

The implementation language JavaScript has enough (and sometimes surprising) implicit type conversions so that

```
1  '2 ' == 2 && '2' * '1\\3'.length + String(true)
```

produces `6true`. Translated to conform to the little language grammar above, the expression

```
'2 ' = 2 and
  '2' * len '1\\3' +
    (string) true
```

is represented as

```
[ 'and'
  [ 'eq' [ 'string' '2 ' ].1 [ 'number' 2 ].1 ].1
  [ 'add'
    [ 'multiply' [ 'string' '2' ].2 [ 'len' [ 'string' '1\\3' ].2 ].2 ].2
    [ 'cast' 'string' [ 'bool' true ] ] ] ]
```

An evaluation visitor needs additional methods corresponding to the additional build actions:

```
const Eval_Bool = superclass => class extends superclass {
  // [ 'or' expr expr ]
  or (node) {
    return node.slice(1).reduce((result, tree) => {
      if (result) return result;  // short-circuit
      result = this.visit(tree);
      if (typeof result != 'boolean')
        this._error(node.lineno, "'or' non-boolean");
      return result;
    }, false);
  }
  ...
```

`or()`[2], and similarly `and()`[2] and `not()`[2], implement the Boolean operations. Both binary operations short-circuits‡ evaluation; e.g., in `or()`[2] subtrees are only visited until a true value is returned (line 5 above). The operations are only intended for Boolean values and report typing issues during evaluation (line 8).

```
  // [ 'bool' literal-value ]
  bool (node) {
    if (typeof node[1] != 'boolean')
      this._error(node.lineno, "'bool' non-boolean");
    return node[1];
  }
};
```

`bool()`[2] evaluates one of the new literals `'true'` and `'false'` which the build action has already

converted.

```
const Eval_String = superclass => class extends superclass {
  // [ 'len' expr ]
  len (node) {
    const val = this.visit(node[1]);
    if (typeof val != 'string')
      this._error(node.lineno, "'len' non-string");
    return val.length;  // undefined if not string
  }
```

`len()`[2] implements the unary `len` operation which computes the length of a string. The result is undefined — and reported — if the subtree value is not a string.

```
  // [ 'string' literal-value ]
  string (node) {
    if (typeof node[1] != 'string')
      this._error(node.lineno, "'string' non-string");
    return node[1];
  }
```

`string()`[2] evaluates a `String` literal where the build action has already elaborated the backslash escapes.

```
  // [ 'concat' a b ]
  concat (node) {
    const vals = node.slice(1).map(this.visit.bind(this));
    if (vals.some(val => typeof val != 'string'))
      this._error(node.lineno, "'concat' non-string");
    return vals[0] + vals[1];
  }
};
```

The little language is going to use `+` to designate both, number addition and string concatenation. Therefore, `concat()`[2] is available to interpret a `'concat'` node by interpreting the subtrees and concatenating the result.

```
const Eval_Cast = superclass => class extends superclass {
  // [ 'cast' type value ]
  cast (node) {
    switch (node[1]) {
    case 'bool':   return !! this.visit(node[2]);
    case 'number': return Number(this.visit(node[2]));
    case 'string': return String(this.visit(node[2]));
    default:       throw node[1] + ': not expected for cast';
    }
  }
```

Finally, `cast()`[2] implements an explicit type cast with a type and a value as a node. This method relies on conversions provided by the implementation language JavaScript.

As before, builder and interpreter for typed expressions are imported or defined and combined in the `actions` area:

```
(() => {  // define and immediately use an anonymous function
  // ... new mix-ins ...

  // builder and interpreter for typed expressions
  return Eleven.Main(Build_Cast(
                       Build_String(
                         Build_Bool(
                           Eleven.Build_Cmps(
                             Eleven.Build_Number(Eleven.Build))))),
                     Eval_Cast(
                       Eval_String(
                         Eval_Bool(
                           Eleven.Eval_Cmps(
                             Eleven.Eval_Number(Eleven.Visit))))));
}) ()
```

In example 11/06[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the error message and the result `'6true'`.
- Finally, add a regular expression such as /./ to the call to Main() near the end of the `actions` area to trace interpretation and try expressions like `true or 0` and `false and 1` to see that the Boolean operations only evaluate as far as they have to.

Checking Typed Expressions

JavaScript is *dynamically typed*: every value belongs to a small set of types, variables accept values of any type, and operators implicitly convert argument values so that the operation can be applied. In particular, values can be passed to functions that are not specifically designed to deal with them — often resulting in confusing runtime errors.

Statically typed languages try to avoid most implicit conversions and require that variable declarations include types. In particular, function parameters must be declared with types so that unexpected argument types can be detected during compilation.

Type checking‡ should be part of compilation and try to ensure that operations will only be applied to the type of values for which they are intended. It has a choice of silently inserting suitable `cast` operations or reporting errors to prevent execution.

Example 11/07[1] implements type checking for typed expressions and inserts implicit conversions so that, e.g., the evaluation methods in `Eval_Bool()`[2] will only receive Boolean values and the error message seen in example 11/06[1] is no longer triggered. The semantics of the typed expressions deliberately are defined to be different from the implementation language JavaScript, e.g., comparisons happen in the type of the left operand, they do not prefer the type 'number'.

The grammar for typed expressions, builder, and interpreter remain unchanged from example 11/06[1]. Type checking is implemented as a new visitor based on a new set of mix-ins‡. This visitor is applied after building and before interpretation, i.e., the `actions` area has the following

structure:

```
(() => {  // define and immediately use an anonymous function
  // base class for type checking
  class Check extends Eleven.Visit { ... }

  // ... type checking mix-ins ...

  // builder, type-checker, and interpreter for typed expressions
  return Eleven.Main(Eleven.Build_Cast(
                       Eleven.Build_String(
                         Eleven.Build_Bool(
                           Eleven.Build_Cmps(
                             Eleven.Build_Number(Eleven.Build))))),
                     Check_Cast(
                       Check_String(
                         Check_Bool(
                           Check_Cmps(
                             Check_Number(Check))))),
                     Eleven.Eval_Cast(
                       Eleven.Eval_String(
                         Eleven.Eval_Bool(
                           Eleven.Eval_Cmps(
                             Eleven.Eval_Number(Eleven.Visit))))));
}) ()
```

In example 11/07[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar and
- press **parse** to see the new result `false`.
- Add `/./` as last parameter to the call to `Main()`[2], specifically to trace evaluation.
- Analyze why `'2 ' = 2` produces `false`.
- Remove the type checking visitor from the call to `Main()`[2] and check out that `'2 ' = 2` now produces `true`.
- Consult the explanation of the JavaScript less than operator† to see why.

`Check`[2], the base class for type checking, inherits from `Visit`[2] and adds a few utility methods:

```
1  // base class for type checking
2  class Check extends Visit {
3    // [ 'bool' literal-value ] etc.
4    _literal (node) {
5      if (!(typeof node[1]).startsWith(node[0]))
6        this._error(node.lineno, `expected ${node[0]} literal`);
7      node.type = node[0]; return node;
8    }
```

`_literal()`[2] receives a node describing a literal value, makes sure that the literal value has the expected type (line 5 above), notes the type in the `.type` property for the node (line 7), and returns the typed node.

JavaScript passes arrays by reference. Therefore, like all other type checking methods, `_literal()`[2] can return the node it received and modified — type checking can rewrite the tree in place.

```
1   // [ tag expr ... ]
2   _toType (type, node, index) {
3     if (this.visit(node[index]).type != type)
4       (node[index] = [ 'cast', type, node[index] ]).type = type;
5     return node;
6   }
```

`_toType()`[2] ensures that a subtree returns a specific type. `_toType()`[2] receives the type name, a node — which it will return — and an index selecting a subtree. `_toType()`[2] visits the subtree to perform type checking (line 3 above). If the subtree type is unexpected `_toType()`[2] modifies the node by inserting a `cast` node on top of the subtree in place of the subtree (line 4) — an error could be reported instead.

```
1   // [ tag expr ... ]
2   _require (type, node) {
3     node.slice(1).forEach((_, n) => this._toType(type, node, n+1));
4     node.type = type;
5     return node;
6   }
```

`_require()`[2] receives a type name and a node and applies `_toType()`[2] to all subtrees to ensure that they return the type. It then notes the type in the `.type` property for the node, and returns the typed node.

`_require()`[2] and `_literal()`[2] implement type checking for all operations on numbers and Boolean values:

```
1   // mix-in to check arithmetic operations
2   const Check_Number = superclass => class extends superclass {
3     // [ 'add' expr expr ]
4     add (node) { return this._require('number', node); }
5     ...
6     // [ 'number' value ]
7     number (node) { return this._literal(node); }
8   };
9
10  // mix-in to check Boolean operations
11  const Check_Bool = superclass => class extends superclass {
12    // [ 'or' expr expr ]
13    or (node) { return this._require('bool', node); }
14    ...
15    // [ 'bool' value ]
16    bool (node) { return this._literal(node); }
17  };
```

Arithmetic operations such as `add` return a `number` and all operands have to produce numbers (line 4 above). Logic operations such as `or` are defined to return `bool` and the operands should

produce Boolean values (line 13).

Comparisons are defined to employ the type of the left operand and return a Boolean value:

```
1  // mix-in to check comparisons
2  const Check_Cmps = superclass => class extends superclass {
3    // [ compare expr expr ]
4    _cmp (node) {
5      const type = this.visit(node[1]).type;
6      this._toType(type, node, 2);
7      node.type = 'bool';
8      return node;
9    }
10   // [ 'eq' expr expr ]
11   eq (node) { return this._cmp(node); }
12   ...
13 };
```

`_cmp()`[2] uses `_toType()`[2] to convert the right operand if necessary (line 6 above) and marks that the node returns `bool` (line 7). `_cmp()`[2] implements type checking for all comparisons (line 11).

A string literal, input, and the `len` operation are straightforward to check:

```
1  // mix-in to check string operations
2  const Check_String = superclass => class extends superclass {
3    // [ 'string' value ]
4    string (node) { return this._literal(node); }
5    // [ 'input' prompt? default? ]
6    input (node) {
7      node.type = 'string'; return node;
8    }
9    // [ 'len' expr ]
10   len (node) {
11     this._require('string', node);
12     node.type = 'number'; return node;
13   }
```

String concatenation is more complicated because an `'add'` node should result in concatenation if a string value is involved and in addition if two numbers are involved:

```
1    // [ 'add' expr expr ]
2    add (node) {
3      const a = this.visit(node[1]), b = this.visit(node[2]);
4      if (a.type != 'string') {
5        if (b.type != 'string') return super.add(node);   // any  any
6        this._toType('string', node, 1);                  // any  string
7      } else if (b.type != 'string')
8        this._toType('string', node, 2);                  // string any
9      node[0] = 'concat';                                 // string string
10     node.type = 'string'; return node;
11   }
12 };
```

`add()`[2] overrides `Check_Number.add()`[2], i.e., the mix-in‡ `Check_String()`[2] has to be applied *after* the mix-in‡ `Check_Number()`[2].

`add()`[2] visits both subtrees (line 3 above) and if neither has a string value it defers to `super.add()`[2] to handle a `number` result (line 5). Otherwise, the first or second operand might have to be converted into a string value (line 6 or line 8). Finally, the operation is changed to `'concat'` (line 9) and the node is marked to produce a `string` value as result (line 10).

The mix-in‡ `Check_Cast()`[2] only has to deal with a `'cast'` node:

```
1  // mix-in to check a cast operation
2  const Check_Cast = superclass => class extends superclass {
3    // [ 'cast' type expr ]
4    cast (node) {
5      this.visit(node[2]);
6      node.type = node[1]; return node;
7    }
8  };
```

The subtree has to be visited but the node is marked with the explicit type (line 6 above) — silently assuming that any kind of conversion is supported.

Checking a Typed Little Language

Example 11/08[1] implements type checking for the little language introduced in example 11/05[1]. Variables have to be declared, assignments and `input` statements have to respect the types, `print` statements require strings, conditions should return Boolean values, and expressions include the Boolean, string, and cast operations introduced in example 11/06[1].

The grammars of the previous examples are merged and there is a small change at the top level to handle declarations:

```
1  run:      main;
2  main:     block;
3  block:    item [{ ';' item }];
4  item:     dcl | stmt;
5  dcl:      type Name [{ ',' Name }];
6  type:     'bool' | 'number' | 'string';
```

The `main` action will return type checking as a function which the action for `run` will execute. A sentence consists of one or more items, separated by semicolons (line 3 above). An item is a declaration or a statement (line 4). A declaration starts with one of the types (line 6) followed by one or more variable names, separated by commas (line 5).

A `block` is *not* permitted at the statement level, i.e., in the body of a loop or selection, but this would be the obvious hook to add block structure as described in chapter seven.

The grammar permits declarations and statements in any order, i.e., variables can be used before they are declared, but the `block()`[2] action will build a node which puts things in order,

declarations before statements:

```
1  // mix-in with building for 'block' and 'dcl'
2  const Build_Dcl = superclass => class extends superclass {
3    // block: item [{ ';' item }];
4    block (item, many) {
5      const items = (many ? many[0] : []).reduce(
6        (items, alt) => { items.push(alt[1][0]); return items; }, [ item[0] ]);
7      return this._lineno([ 'block' ].concat(
8        items.filter(item => item[0] == 'dcl'),
9        items.filter(item => item[0] != 'dcl')));
10   }
11   // dcl: type Name [{ ',' Name }];
12   dcl (type, name, many) {
13     return this._lineno([ 'dcl', type, name ].
14       concat(many ? many[0].map(alt => alt[1]) : []));
15   }
16 };
```

The `block()`[2] action collects all items into one list (line 5 above). There is no action for `item`, i.e., the `item` rule will return a list with a single declaration or statement node which `block()`[2] has to extract (line 6). The list of all items is then split into declarations (line 8) and statements (line 9) and both are combined into a `'block'` node (line 7) because `concat()`† flattens one level of arrays.

The `dcl()`[2] action builds a `'dcl'` node with the declared type and the list of names (line 13).

There is not much to do to check `'block'` and `'dcl'` nodes:

```
1  // mix-in with type checking for 'block' and 'dcl'
2  const Check_Dcl = superclass => class extends superclass {
3    // [ 'block' dcl... stmt... ]
4    block (node) {
5      node.slice(1).forEach((item, n) => node[n + 1] = this.visit(item));
6      return node;
7    }
8    // [ 'dcl' type name ... ]
9    dcl (node) {
10     node.slice(2).forEach(name => {
11       if (this.symbols.has(name))
12         this._error(node.lineno, name + ': duplicate');
13       this._alloc(name).type = node[1];
14     });
15     return node;
16   }
17 };
```

The `block()`[2] method visits each subtree (line 5 above), i.e., it executes declarations before it checks statements.

The `dcl()`[2] method needs a `symbols` `Map`†, e.g., from the `Symbols`[2] mix-in‡. It forbids that a name has already been declared (line 11) and creates a description with a `type` property (line 13).

Most statements require little checking and they have no useful node type upon return:

```
1  // mix-in with type checking for statements
2  const Check_Stmts = superclass => class extends superclass {
3    // [ 'stmts' stmt... ]
4    stmts (node) {
5      node.slice(1).forEach(stmt => this.visit(stmt)); return node;
6    }
7    // [ 'print' expr ... ]
8    print (node) { return this._require('string', node); }
9    // [ 'loop' expr body ]
10   loop (node) {
11     this.visit(node[2]);
12     return this._toType('bool', node, 1);
13   }
14   // [ 'select' expr then else? ]
15   loop (node) {
16     this.visit(node[2]);
17     return this._toType('bool', node, 1);
18   }
19   /** `[ 'select' cond then else? ]` condition cast to `bool`.
20       @param {Array} node - to check.
21       @memberof module:Eleven~Check_Stmts
22       @instance */
23   select (node) {
24     node.slice(2).forEach(node => this.visit(node));
25     return this._toType('bool', node, 1);
26   }
27 };
```

The `stmts()`[2] method visits each subtree (line 5 above).

The `print()`[2] method uses `_require()`[2] to ensure that all arguments are strings (line 8).

The `loop()`[2] method visits both subtrees and uses `_toType()`[2] to ensure that the loop condition returns a Boolean value (line 12).

The `select()`[2] method visits all subtrees and uses `_toType()`[2] to ensure that the condition returns a Boolean value (line 17).

Checking the use of variables is more complicated. Just like `dcl()`[2] the methods in

`Check_Names()`[2] require a `symbols` `Map`[†]:

```
1  // mix-in with type checking for names
2  const Check_Names = superclass => class extends superclass {
3    // [ 'name' name ]
4    name (node) {
5      node.type = this._type(node.lineno, node[1]);
6      return node;
7    }
8    // return the type of a name, or 'number'
9    _type (lineno, name) {
10     const symbol = this._alloc(name);
11     if (!('type' in symbol)) {
12       this._error(lineno, name + ': undefined');
13       symbol.type = 'number';
14     }
15     return symbol.type;
16   }
17   // [ 'assign' name expr ]
18   assign (node) {
19     return this._toType(this._type(node.lineno, node[1]), node, 2);
20   }
21 };
```

The `name()`[2] method sets the node type to the declared type of the variable (line 5 above). It uses `_type()`[2] to obtain the type from the symbol table (line 9). The type should have been declared!

The `assign()`[2] method uses `_toType()`[2] to visit the expression subtree and ensure that the expression delivers the type which is expected by the variable — this information is delivered by `_type()`[2].

The `actions` area defines the new mix-ins[‡] discussed above and uses `Main()`[2] to create the

builder and type checker:

```
(() => {  // define and immediately use an anonymous function
  // mix-ins with building actions
    ...
  // mix-ins with methods for type checking
    ...
  // builder and checker for a typed little language
  return Eleven.Main(Build_Dcl(
                       Eleven.Build_Stmts(
                         Eleven.Build_Names(
                           Eleven.Build_Cast(
                             Eleven.Build_String(
                               Eleven.Build_Bool(
                                 Eleven.Build_Cmps(
                                   Eleven.Build_Number(Eleven.Build)))))))),
                     Check_Dcl(
                       Check_Stmts(
                         Check_Names(
                           Eleven.Symbols(
                             Eleven.Check_Cast(
                               Eleven.Check_String(
                                 Eleven.Check_Bool(
                                   Eleven.Check_Cmps(
                                     Eleven.Check_Number(Eleven.Check)))))))))));
}) ()
```

Example 11/08[1] contains a (rather contrived) typed version of Euclid's Algorithm‡:

```
x = input 'x' '36'; y = input 'y' '54'; string x; number y;
eq = x = y; bool eq;
while not eq do
  if (number) x > y then
    x = x - y
  else
    y = y - x
  fi;
  eq = x = y
od;
print 'Greatest common divisor:', x
```

- The very first button should show stack-based. If not, click it until it does.
- Remove the `run` and `main` rules,
- press **new grammar** to represent and check the grammar, and

- press **parse** to see the original tree.

```
[ 'block'
  [ 'dcl' 'string' 'x' ] [ 'dcl' 'number' 'y' ] [ 'dcl' 'bool' 'eq' ]
  [ 'assign' 'x' [ 'input' 'x' '36' ] ]
  [ 'assign' 'y' [ 'input' 'y' '54' ] ]
  [ 'assign' 'eq' [ 'eq' [ 'name' 'x' ] [ 'name' 'y' ] ] ]
  [ 'loop' [ 'not' [ 'name' 'eq' ] ] [ 'stmts'
    [ 'select'
      [ 'gt' [ 'cast' 'number' [ 'name' 'x' ] ] [ 'name' 'y' ] ]
      [ 'assign' 'x' [ 'subtract' [ 'name' 'x' ] [ 'name' 'y' ] ] ]
      [ 'assign' 'y' [ 'subtract' [ 'name' 'y' ] [ 'name' 'x' ] ] ] ]
    [ 'assign' 'eq' [ 'eq' [ 'name' 'x' ] [ 'name' 'y' ] ] ] ] ]
  [ 'print' [ 'string' 'Greatest common divisor:' ] [ 'name' 'x' ] ] ]
```

- Note that the `block` action has sorted the declarations before the statements.
- Restore the rules,
- press **new grammar** to represent and check the grammar, and
- press **parse** to see the tree after it was modified by the type checker:

```
[ 'block'
  [ 'dcl' 'string' 'x' ] [ 'dcl' 'number' 'y' ] [ 'dcl' 'bool' 'eq' ]
  [ 'assign' 'x' [ 'input' 'x' '36' ] ]
  [ 'assign' 'y' [ 'cast' 'number' [ 'input' 'y' '54' ] ] ]
  [ 'assign' 'eq'
    [ 'eq' [ 'name' 'x' ] [ 'cast' 'string' [ 'name' 'y' ] ] ] ]
  [ 'loop' [ 'not' [ 'name' 'eq' ] ] [ 'stmts'
    [ 'select'
      [ 'gt' [ 'cast' 'number' [ 'name' 'x' ] ] [ 'name' 'y' ] ]
      [ 'assign' 'x' [ 'cast' 'string'
        [ 'subtract'
          [ 'cast' 'number' [ 'name' 'x' ] ] [ 'name' 'y' ] ] ] ]
      [ 'assign' 'y'
        [ 'subtract'
          [ 'name' 'y' ] [ 'cast' 'number' [ 'name' 'x' ] ] ] ] ]
    [ 'assign' 'eq'
      [ 'eq' [ 'name' 'x' ] [ 'cast' 'string' [ 'name' 'y' ] ] ] ] ]
  [ 'print' [ 'string' 'Greatest common divisor:' ] [ 'name' 'x' ] ] ]
```

- Remove or duplicate a declaration to see error messages from type checking.
- Change the variables' types to see how that changes what casts are inserted.

Interpreting a Typed Little Language

Example 11/09[1] adds the necessary methods to interpret the typed version of the little language. The `actions` area creates the new interpreter and uses `Main()`[2] to create a new `main` action

which will return the interpreter as a function:

```
(() => {  // define and immediately use an anonymous function
  // mix-in for blocks and declarations
    ...
  // builder, checker, and interpreter for a typed little language
  return Eleven.Main(Eleven.Build_Dcl(
                       Eleven.Build_Stmts(
                         Eleven.Build_Names(
                           Eleven.Build_Cast(
                             Eleven.Build_String(
                               Eleven.Build_Bool(
                                 Eleven.Build_Cmps(
                                   Eleven.Build_Number(Eleven.Build)))))))),
                     Eleven.Check_Dcl(
                       Eleven.Check_Stmts(
                         Eleven.Check_Names(
                           Eleven.Symbols(
                             Eleven.Check_Cast(
                               Eleven.Check_String(
                                 Eleven.Check_Bool(
                                   Eleven.Check_Cmps(
                                     Eleven.Check_Number(Eleven.Check)))))))),
                     Eval_Dcl(
                       Eleven.Eval_Stmts(
                         Eleven.Eval_Names(
                           Eleven.Symbols(
                             Eleven.Eval_Cast(
                               Eleven.Eval_String(
                                 Eleven.Eval_Bool(
                                   Eleven.Eval_Cmps(
                                     Eleven.Eval_Number(Eleven.Visit)))))))));
}) ()
```

Most of the interpreter was developed in example 11/05[1] above. Type checking adds `'block'`

and `'dcl'` nodes. Therefore, just two visitor methods have to be added to the interpreter:

```
1   // mix-in to interpret 'block' and 'dcl'
2  const Eval_Dcl = superclass => class extends superclass {
3   // [ 'block' dcl ... stmt ... ]
4   block (node) {  node.slice(1).forEach(item => this.visit(item)); }
5   // [ 'dcl' type name ... ]
6   dcl (node) {
7     node.slice(2).forEach(name => {
8       if (this.symbols.has(name) && 'value' in this.symbols.get(name))
9         this._error(node.lineno, name + ': duplicate');
10      switch (node[1]) {
11      case 'bool':    this._alloc(name).value = false; break;
12      case 'number':  this._alloc(name).value = 0; break;
13      case 'string':  this._alloc(name).value = ''; break;
14      default:        this._error(node.lineno, node[1] + ": not in 'dcl'");
15      }
16    });
17  }
```

The `block()`[2] method simply visits the subtrees, i.e., the declarations followed by the statements (line 4 above).

The `dcl()`[2] method initializes the variables. The test for duplicates (line 8) should not be necessary because the interpreter uses the symbol table which was created by the type checker, i.e., duplicate variables would have been detected before and `Main()`[2] does not continue from one visitor to the next if errors are reported. Variables are initialized to "zero", but the value is type-specific (line 10).

In example 11/09[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar,
- press **parse** to create the executable, and
- press **run** to interpret the program.

```
1  && 'value' in this.symbols.get(name)
```

- Remove the phrase above from the body of the `dcl` method and press **parse** and **run** again to see three error messages because `x`, `y`, and `eq` are already in the symbol table. This demonstrates that the symbol table is copied from the type checker.

```
1  bool b; print b
```

- Change the program to the one shown above and check and interpret it. The output should be `false`.
- Remove the body of the `dcl` method and check and interpret again. The output is `0` because there is no initialization, the reference to a name defaults a missing value to zero, and the cast to a string for printing uses `String()`† which in JavaScript converts anything to a string.

Generating Code

Code generation for a stack machine using action methods was introduced starting in chapter six with examples 6/09[1] and 6/10[1] for arithmetic expressions and example 6/11[1] for control structures. If a program is represented as a tree code generators are visitors which can apply some code optimization. The stack machines can be reused and extended with new instructions, e.g., for exponentiation or to operate on strings and Boolean values.

Compiling Arithmetic Expressions

Example 11/10[1] implements code generation for trees of arithmetic expressions which include exponentiation. `Code`[2] is the base class for code generators. It extends `Visit`[2] and manages stack machines. In particular, it reuses machine generators such as `Six.Machine10`[2] and it implements an extension mechanism to add instructions:

```
1  class Code extends Visit {
2    // machine generator class, allows replacement
3    get Machine () { return this.#Machine ??= Six.Machine10; }
4    #Machine;
5    // instructions mix-in, allows extensions
6    get Instructions () {
7      return this.#Instructions ??= superclass => superclass;
8    }
9    #Instructions;
10   // machine generator, should not change
11   get machine () {
12     return this.#machine ??= new (this.Instructions(this.Machine)) ();
13   }
14   #machine;
15   // for 'compile' rule, overwrite to match machine generator
16   get executable () { return this.machine.run(0); }
17   // visit subtrees, generate 'op' instruction, returns end address
18   _postfix (node, op) {
19     node.slice(1).forEach(node => this.visit(node));
20     return this.machine.gen(op);
21   }
22 }
```

The stack machine generators in chapter six and chapter seven form a linear inheritance hierarchy and define the machine instructions. `Code`[2] defines getters† as hooks which can be overwritten in mix-ins‡ to change which machine generator class will be used (line 3 above), to add new instructions in a cumulative fashion (line 6), and to access the executable as befits the underlying machine generator (line 16).

The private method `_postfix()`[2] generates code for many operators. Given a node, the subtrees are visited to generate code to push values onto the runtime stack and then an instruction corresponding to the node is generated to operate on the values (lines 19 and 20).

The mix-in‡ `Code_Number()`[2] extends the `Instructions()` mix-in‡ to add a `Power` instruction

(lines 3 to 11 below) and it contains the methods to visit the trees built for arithmetic expressions:

```
1  const Code_Number = superclass => class extends superclass {
2    // overwrite instructions mix-in, add 'Power'
3    get Instructions () {
4      return this.#Instructions ??=
5        superclass => class extends super.Instructions(superclass) {
6          /** `stack: ... a b -> ... a**b` */
7          Power (memory) {
8            memory.splice(-2, 2, memory.at(-2) ** memory.at(-1));
9          }
10       };
11   }
12   #Instructions;
13   ...
14   // [ 'power' a b ]
15   power (node) { return this._postfix(node, 'Power'); }
16   // [ 'minus' a ]
17   minus (node) { return this._postfix(node, 'Minus'); }
18   // [ 'number' n ]
19   number (node) {
20     if (typeof node[1] != 'number')
21       this._error(node.lineno, "'number' non-number");
22     return this.machine.gen('Push', node[1]);
23   }
24 };
```

For nodes like `'power'` or `'minus'` `_postfix()`[2] visits the subtrees from left to right for code generation and then generates the instruction required for the node (lines 15 and 17 above). For a `'number'` node `'Push'` is generated to load a constant onto the runtime stack (line 22).

Finally, the mix-in‡ `Compile()`[2] works exactly like the mix-in‡ `Main()`[2] discussed above:

```
1  const Compile = (superclass, ...args) => class extends superclass {
2    // compile: expr;
3    compile (tree) {
4      const [lastVisitor, lastTree, trace] = this._doVisits(tree, args);
5      lastVisitor.visit(lastTree, trace);
6      if (trace) puts(lastVisitor.machine.toString());
7      return lastVisitor.executable;
8    }
9  };
```

It defines a build action for a `compile` rule which expects a tree, runs a list of visitors such as type checking and code generation, and returns the `executable` from the last visitor.

The `actions` area contains the compiler for arithmetic expressions:

```
(() => {  // define and immediately use an anonymous function
  // mix-in to run the visitors
  const Compile ...

  // base class to generate code
  class Code ...

  // mix-in to generate code for arithmetic expressions
  const Code_Number ...

  // builder and code generator for arithmetic expressions
  return Compile(
           Eleven.Main(Eleven.Build_Number(Eleven.Build)),
           Code_Number(Code));
}) ()
```

`Main()`[2] is needed in the inheritance chain of mix-ins‡ to supply `_doVisits()`[2] which does the actual work for the `compile()`[2] build action. The construction still supports `dump` and `run` rules to display the tree and immediately run the executable returned by `compile()`[2].

The grammar for this example starts with:

```
compile:  expr;
expr:     add | subtract | multiply | divide | power
          | minus | '(' expr ')' | number;
```

The precedence table, `expr`, and the remaining rules are unchanged from from example 11/02[1].

In example 11/10[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar,
- press **parse** to create the executable, and
- press **run** to execute the generated code.
- Insert `dump` between the `compile` and `expr` rules to display the expression tree.
- Add a regular expression such as `/./` to the call to `Compile()`[2] to trace code generation, i.e., display the successive last addresses and then dump code memory.

- Add `run` as the start rule to immediately execute the generated code:

```
> run = g.parser().parse(program, actions)
[ 'subtract' [ 'add' [ 'number' 1 ].1 ...
[ 'number' ].1 returns 1
[ 'number' ].1 returns 2
  ...
[ 'subtract' ] returns 20
0: memory => this.Push(1)(memory)
1: memory => this.Push(2)(memory)
  ...
19: memory => this.Subtract(memory)
[ -10 ]
```

Compiling a Little Language

Example 11/11[1] extends code generation to comparisons, variables, constrol structures, and a few other statements. It reuses the grammar and build actions from example 11/04[1]. The `actions` area contains the compiler for a little language:

```
(() => {  // define and immediately use an anonymous function
  // mix-in to generate stack machine code for comparisons
  const Code_Cmps = ...

  // mix-in to generate stack machine code for names
  const Code_Names = ...

  // mix-in to generate stack machine code for statements
  const Code_Stmts = ...

    // builder and code generator for a little language
    return Eleven.Compile(
             Eleven.Main(Eleven.Build_Stmts(
                           Eleven.Build_Names(
                             Eleven.Symbols(
                               Eleven.Build_Cmps(
                                 Eleven.Build_Number(Eleven.Build))))),
             Code_Stmts(
               Code_Names(
                 Code_Cmps(
                   Eleven.Symbols(
                     Eleven.Code_Number(Eleven.Code))))));
  }) ()
```

Code generation for comparisons is implemented using `_postfix()`[2]:

```
const Code_Cmps = superclass => class extends superclass {
  // [ 'eq' a b ]
  eq (node) { return this._postfix(node, 'Eq'); }
  ...
```

The necessary instructions were implemented in `Six.Machine11`[2] which the mix-in‡ `Code_Cmps()`[2] has to request:

```
  // [override] use Six.Machine11
  get Machine () { return this.#Machine ??= Six.Machine11; }
  #Machine;
```

This stack machine generator significantly redefines `run()`[2], therefore, access to the executable has to be overwritten:

```
  // [override] size memory, check for 'trace' variable
  get executable () {
    const trace = this.symbols.get('trace');
    return this.machine.run(this.symbols.size, 0,
      trace ? trace.ord - 1 : false);
  }
};
```

As a consequence, `Code_Cmps()`[2] silently assumes that the mix-in‡ `Symbols()`[2] has been included.

The mix-in‡ `Code_Names()`[2] implements code generation for name references:

```
const Code_Names = superclass => class extends superclass {
  // [ 'name' name ]
  name (node) {
    return this.machine.gen('Load', this._alloc(node[1]).ord - 1);
  }
  // [ 'assign' name value ]
  assign (node) {
    this.visit(node[2]);
    this.machine.gen('Store', this._alloc(node[1]).ord - 1);
    return this.machine.gen('Pop');
  }
};
```

The instructions `Load`, `Store`, and `Pop` were implemented in `Six.Machine11`[2] and variable addresses are managed by `_alloc()`[2] which is part of the mix-in‡ `Symbols()`[2].

Generating optimized code for statements is a bit more interesting. First, the mix-in‡ `Code_Stmts()`[2] adds a `Bnzero` instruction which pops the runtime stack and branches if the

popped value was non-zero (lines 7 to 9 below).

```
1  const Code_Stmts = superclass => class extends superclass {
2    // [ extend ] add 'Bnzero' to optimize 'while' loops
3    get Instructions () {
4      return this.#Instructions ??=
5        superclass => class extends super.Instructions(superclass) {
6          /** `stack: ... bool -> ... | pc: bool? a` */
7          Bnzero (a) {
8            return memory => { if (memory.pop()) memory.pc = a; }
9          }
10       };
11   }
12   #Instructions;
```

Referring to `super.Instructions` ensures that the `Power` instruction defined by `Code_Number()`[2] remains available (line 5 above).

Code generation itself deals with `'stmts'`, `'print'`, `'select'`, and `'loop'` nodes:

```
1    // [ 'stmts' stmt ... ]
2    stmts (node) {
3      return node.slice(1).reduce((end, stmt) => this.visit(stmt), 0);
4    }
5    // [ 'print' value ... ]
6    print (node) {
7      node.slice(1).forEach(value => this.visit(value));
8      return this.machine.gen('Print', node.length - 1);
9    }
```

`stmts()`[2] visits each subtree in turn to generate code for each statement (line 3 above). Like all of code generation it returns the end address of the generated code.

`print()`[2] visits each subtree in turn to generate code to push all values onto the runtime stack (line 7) and then generates a `Print`[2] instruction (line 8) which will print the values together and remove them from the stack.

```
1    // [ 'select' expr stmt stmt? ]
2    select (node) {
3      const a = this.visit(node[1]);          //      cond
4      this.machine.code.push(null);           // a:   Bzero b
5      let b = this.visit(node[2]), end = b;   //      then
6      if (node.length > 3) {                  // b:end:
7        this.machine.code.push(null);    //          Branch end
8        end = this.visit(node[3]);            // b:   else
9                                              // end:
10       this.machine.code[b ++] = this.machine.ins('Branch', end);
11     }
12     this.machine.code[a] = this.machine.ins('Bzero', b); // fixup
13     return end;
14   }
```

`select()`[2] visits the first subtree to generate code for the condition of an `if` statement (line 3 above), leaves room for a conditional branch (line 4), and visits the second subtree to generate code for the `then` part (line 5). If there is no `else` part there is only one repair: the conditional branch has to bypass the `then` part (line 12). Otherwise there has to be room for an unconditional branch (line 7) followed by the code for the `else` part generated by a visit to the third subtree (line 8). In this case there are two repairs: the conditional branch has to reach the `else` part (line 10) and the unconditional branch has to bypass the `else` part (line 12).

Either way, `select()`[2] returns the next code address (line 13). The comments above indicate with symbolic labels such as `a:`, `b:`, etc., how convenient this is to generate the branch instructions for control structures.

`select()`[2] generates the same code as example 6/11[1] but the tree representation collects branch address management into a single method rather than having to distribute it over a number of rules and actions.

```
1   // [ 'loop' expr stmt ]
2   loop (node) {
3     const a = this.machine.code.push(null) - 1,  // a:   Branch b
4       b = this.visit(node[2]);                    // a+1: stmt
5     this.visit(node[1]);                          // b:   cond
6     this.machine.code[a] = this.machine.ins('Branch', b); // fixup
7     return this.machine.gen('Bnzero', a + 1);     //      Bnzero a+1
8   }
9 };
```

The tree representation makes it possible for `loop()`[2] to generate code for the loop body (line 4 above) ahead of the code for the condition (line 5). An unconditional `Branch`[2] instruction is inserted *before* the loop body (line 6) and is executed just once to initially transfer control to the condition. The conditional `Bnzero`[2] instruction follows the condition (line 7) and transfers control each time back to the loop body.

If the condition is false initially it takes two branch instruction cycles (rather than one in example 6/11[1]) to bypass the loop, but once the loop takes place almost half the branch instruction cycles are saved!

In example 11/11[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar,

- press **parse** to create the executable for Euclid's algorithm‡:

```
trace = -1;
input x, y;
while x <> y do
  if x > y then
    trace = 1; x = x - y; trace = -1
  else y = y - x
  fi
od;
print x
```

- Add a regular expression such as `/./` to the call to `Compile()`[2] to trace code generation, i.e., display the successive last addresses and then dump code memory.
- Check out the `loop` optimization:

```
  ...
10: memory => this.Branch(33)(memory)
11: memory => this.Load(1)(memory)      // if x > y
12: memory => this.Load(2)(memory)
13: memory => this.Gt(memory)
14: memory => this.Bzero(28)(memory)
  ... then ...
27: memory => this.Branch(33)(memory)
  ... else ...
33: memory => this.Load(1)(memory)      // while x <> y
34: memory => this.Load(2)(memory)
35: memory => this.Ne(memory)
36: memory => this.Bnzero(11)(memory)
  ...
```

Compiling a Typed Little Language

Example 11/12[1] reuses the grammar, tree building actions, and type checker from example 11/08[1] and extends stack machine and code generation from example 11/11[1] to produce the last

compiler in this book. The `actions` area contains the compiler for a typed little language:

```
(() => {  // define and immediately use an anonymous function
  // additional mix-ins to generate code for Boolean expressions
  ...
  // builder, type checker, and code generator for a typed little language
  return Eleven.Compile(Eleven.Main(Eleven.Build_Dcl(
                           Eleven.Build_Stmts(
                             Eleven.Build_Names(
                               Eleven.Symbols(
                                 Eleven.Build_Cast(
                                   Eleven.Build_String(
                                     Eleven.Build_Bool(
                                       Eleven.Build_Cmps(
                                         Eleven.Build_Number(Eleven.Build)))))))))
                         Eleven.Check_Dcl(
                           Eleven.Check_Stmts(
                             Eleven.Check_Names(
                               Eleven.Symbols(
                                 Eleven.Check_Cast(
                                   Eleven.Check_String(
                                     Eleven.Check_Bool(
                                       Eleven.Check_Cmps(
                                         Eleven.Check_Number(Eleven.Check)))))))))
                         Code_Dcl(
                           Eleven.Code_Stmts(
                             Eleven.Code_Names(
                               Code_Cast(
                                 Code_String(
                                   Code_Bool(
                                    Eleven.Code_Cmps(
                                      Eleven.Symbols(
                                        Eleven.Code_Number(Eleven.Code)))))))))));
}) ()
```

The binary Boolean operations are a bit tricky: short-circuit evaluation‡ turns them into something akin to control structures but they still have to push a value onto the runtime stack:

```
const Code_Bool = superclass => class extends superclass {
  // [ 'or' a b ]
  or (node) {
    this.visit(node[1]);                        //    push a
    const x = this.machine.code.push(null) - 1; // x: IfTrue y
    this.machine.gen('Pop');                    //    pop a
    const y = this.visit(node[2]);              //    push b
                                                // y:
    this.machine.code[x] = this.machine.ins('IfTrue', y); // fixup
    return y;
  }
```

`or()`[2] generates code to evaluate the left operand (line 4 above) and leaves room for a branch

instruction (line 5). If the left operand leaves `false` on top of the runtime stack, this value has to be removed (line 6) and there has to be code to evaluate the right operand (line 7) which will leave the ultimate result of the `or` operation on the stack. Therefore, the branch instructions `IfTrue` and `IfFalse` should check the value on the stack but not remove it:

```
1   // [extend] add short-circuit branches and 'Not'
2   get Instructions () {
3     return this.#Instructions ??=
4       superclass => class extends super.Instructions(superclass) {
5         /** `stack: ... bool -> ... bool | pc: bool? a` */
6         IfTrue (a) {
7           return memory => { if (memory.at(-1)) memory.pc = a; };
8         }
9         /** `stack: ... bool -> ... bool | pc: !bool? a` */
10        IfFalse (a) {
11          return memory => { if (!memory.at(-1)) memory.pc = a; };
12        }
13        /** `stack: ... a -> ... !a` */
14        Not (memory) { memory.splice(-1, 1, !memory.at(-1)); }
15      };
16  }
17  #Instructions;
```

`and()`[2] follows the same pattern as `or()`[2] but uses `IfFalse` instead:

```
1   // [ 'and' a b ]
2   and (node) {
3     this.visit(node[1]);                        //    push a
4     const x = this.machine.code.push(null) - 1; // x: IfFalse y
5     this.machine.gen('Pop');                    //    pop a
6     const y = this.visit(node[2]);              //    push b
7                                                 // y:
8     this.machine.code[x] = this.machine.ins('IfFalse', y); // fixup
9     return y;
10  }
```

The comments again illustrate the advantage of generating control structure code in a single method for a tree node.

```
1   // [ 'not' a ]
2   not (node) { return this._postfix(node, 'Not'); }
```

`not()`[2] uses `_postfix()`[2] to generate code to push a Boolean value onto the stack and generates

`Not` to complement it.

```
  // [ 'bool' a ]
  bool (node) {
    if (typeof node[1] != 'boolean')
      throw `[ 'bool' ${node[1]} ]: not boolean`;
    return this.machine.gen('Push', node[1]);
  }
};
```

Finally, `bool()`[2] has a Boolean literal in the node and generates `Push` to push it onto the runtime stack.

The rest of the additional code generation for the typed little language follows the usual pattern. `cast()`[2] visits the value subtree to generate code to push the value onto the stack (line 4 below) and generates a `Cast` instruction:

```
1  const Code_Cast = superclass => class extends superclass {
2    // [ 'cast' type b ]
3    cast (node) {
4      this.visit(node[2]);
5      return this.machine.gen('Cast', `'${node[1]}'`, `'${node[2].type}'`);
6    }
7    // [extend] add 'Cast' instruction
8    get Instructions () {
9      return this.#Instructions ??=
10       superclass => class extends super.Instructions(superclass) {
11         /** `stack: ... a -> ... cast a` */
12         Cast (to, from) {
13           let cast;
14           switch (to + '-' + from) {
15           case 'bool-number':   cast = x => !!x; break;
16           case 'bool-string':   cast = x => /^\s*true\s*$/i.test(x); break;
17           case 'number-bool':
18           case 'number-string': cast = Number; break;
19           case 'string-bool':
20           case 'string-number': cast = String; break;
21           default: throw `Cast ${to} ${from}: illegal cast`;
22           }
23           return memory =>
24             memory.splice(-1, 1, cast(memory.at(-1)));
25         }
26       };
27   }
28   #Instructions;
29 };
```

The `Cast` instruction is added to the stack machine generator as usual (lines 8 to 28 above). It selects one of several functions — based on on JavaScript conversions — depending on the combination of `to` and `from` types (lines 13 to 20) and creates an instruction which applies the selected function to the top of the runtime stack (line 24). The implementation verifies that the

function is only applied to a value of the intended JavaScript type (line 21).

Code generation for string nodes again uses `_postfix()`[2] but there have to be new machine instructions:

```
1  const Code_String = superclass => class extends superclass {
2    // [ 'concat' value value ]
3    concat (node) { return this._postfix(node, 'Concat'); }
4    // [ 'len' value ]
5    len (node) { return this._postfix(node, 'Len'); }
6    // [ 'string' literal ]
7    string (node) {
8      if (typeof node[1] != 'string')
9        throw `[ 'string' ${node[1]} ]: not string`;
10     return this.machine.gen('Push', this._escape(node[1]));
11   }
12   // [ 'input' prompt? default? ]
13   input (node) {
14     return this.machine.gen('InputString',
15       this._escape(node[1] ?? "''"), this._escape(node[2] ?? "''"));
16   }
17   // [extend] add 'InputString', 'Len', and 'Concat' instructions
18   get Instructions () {
19     return this.#Instructions ??=
20       superclass => class extends super.Instructions(superclass) {
21         /** `stack: ... a b -> ... a+b` */
22         Concat (memory) {
23           memory.splice(-2, 2, memory.at(-2) + memory.at(-1));
24         }
25         /** `stack: ... a -> ... a.length` */
26         Len (memory) {
27           memory.splice(-1, 1, memory.at(-1).length);
28         }
29         /** `stack: ... -> ... val` */
30         InputString (prmpt, dflt) {
31           return memory => memory.push(prompt(prmpt, dflt));
32         }
33       };
34   }
35   #Instructions;
```

`concat()`[2] and `len()`[2] call `_postfix()`[2] to generate code (lines 3 and 5 above) using the new instructions `Concat`[2] (line 22) and `Len`[2] (line 26). It is assumed that a string requires just one memory slot, just like a number. Therefore, `string()`[2] can use `Push` to push a string constant onto the runtime stack (line 10). Finally, `input()`[2] generates an `InputString`[2] instruction (line 14) which is implemented using `prompt()`† (line 30).

There is one glitch, however. The method `gen()`[2] was introduced in chapter six. It uses `eval()`† so that the instructions can be displayed with meaningful names and argument values. Therefore, string arguments have to be escaped before they are handed to `eval()`†. Assuming that backslash is only needed to escape backslashes, single quotes, and newlines, this can be handled

by replace()†:

```
  _escape (s) { return `'${s.replace(/['\n\\]/g, '\\$&')}'`; }
```

A more comprehensive method would be Tuple.escape()².

The stack machine's memory contains the variable values and the runtime stack. JavaScript will distinguish numbers and Boolean values for a trace or a memory dump. However, here too, string values need to be escaped:

```
  get Machine () {
    const escape = this._escape.bind(this);
    return this.#Machine ??= class extends super.Machine {
        /** Show strings in memory. */
        get Memory () {
            return this.#Memory ??= class extends super.Memory {
              toString () {
                return '[ ' + this.map(
                    v => typeof v == 'string' ? escape(v) : v
                  ).join(' ') + ' ]';
              }
            };
        }
        #Memory;
      };
  }
  #Machine;
};
```

Memory² is defined as a subclass of Array and toString() needs to be replaced (line 7 to 11 above). The actual work can be handled by _escape()²; however, _escape()² is a method in a mix-in‡ and cannot be static because there is no class name. It can be used as a function with bind()† (line 2).

Finally, the mix-in Code_Dcl² handles 'block' and 'dcl' nodes and requires no new

instructions:

```
1  const Code_Dcl = superclass => class extends superclass {
2    // [ 'block' dcl ... stmt ... ]
3    block (node) {
4      return node.slice(1).reduce((end, node) => this.visit(node), 0);
5    }
6    // [ 'dcl' type name ... ]
7    dcl (node) {
8      return node.slice(2).reduce((end, name) => {
9        const addr = this._alloc(name).ord - 1;
10       switch (node[1]) {
11       case 'number': return this.machine.code.length;
12       case 'bool':   this.machine.gen('Push', false); break;
13       case 'string': this.machine.gen('Push', "''"); break;
14       }
15       this.machine.gen('Store', addr);
16       return this.machine.gen('Pop');
17     }, 0);
18   }
19 };
```

Just as in the type checker, `block()`[2] visits each subtree in turn, declarations before statements (line 4 above).

`dcl()`[2] assumes that there is a symbol table and calls `_alloc()`[2] for each name (line 9). This will allocate a memory slot for new names if there is no type checking. Memory is initialized to zero, but variables of other types are initialized to `false` and empty strings, respectively (lines 12 and 13).

In example 11/12[1]

- the very first button should show stack-based. If not, click it until it does.
- Press **new grammar** to represent and check the grammar,
- press **parse** to create the executable for the typed version of Euclid's algorithm‡ used before, and
- press **100** to run the program.

The execution output shows how zero-filled memory is allocated for the `string` variable `x`, the `number` variable `y`, and the `bool` variable `eq`, and how the variables `x` and `eq` are initialized to

values of proper type:

```
> memory = run(null, 100)
[ 0 0 0 ]
[ 0 0 0 '' ] 0: memory => this.Push('')(memory)
[ '' 0 0 '' ] 1: memory => this.Store(0)(memory)
[ '' 0 0 ] 2: memory => this.Pop(memory)
[ '' 0 0 false ] 3: memory => this.Push(false)(memory)
[ '' 0 false false ] 4: memory => this.Store(2)(memory)
[ '' 0 false ] 5: memory => this.Pop(memory)
[ '' 0 false '36' ] 6: memory => this.InputString('x', '36')(memory)
[ '36' 0 false '36' ] 7: memory => this.Store(0)(memory)
[ '36' 0 false ] 8: memory => this.Pop(memory)
  ...
[ '18' 18 true 'Greatest common divisor:' ] 48: memory => this.Push('Greatest com
[ '18' 18 true 'Greatest common divisor:' '18' ] 49: memory => this.Load(0)(memor
Greatest common divisor: 18
[ '18' 18 true ] 50: memory => this.Print(2)(memory)
[ '18' 18 true ]
```

From that point on until the end of execution the types of the variables' memory cells don't change anymore.

And More...

In example 11/12[1] the nonsensical program

```
if not trace or false then print 'ok' fi; bool trace
```

is compiled into

```
0: memory => this.Push(false)(memory)     // bool trace = false
1: memory => this.Store(0)(memory)
2: memory => this.Pop(memory)
3: memory => this.Load(0)(memory)         // if not trace
4: memory => this.Not(memory)
5: memory => this.IfTrue(8)(memory)       //    or
6: memory => this.Pop(memory)
7: memory => this.Push(false)(memory)     //    false
8: memory => this.Bzero(11)(memory)
9: memory => this.Push('ok')(memory)      // then print 'ok
10: memory => this.Print(1)(memory)
```

and produces the expected `ok`. However, the code could be optimized: the `IfTrue` branch at address 5 should not end up at the `Bzero` branch at address 8, it should directly branch into `then` at address 9. More generally, if short-circuit evaluations[‡] are at the top level of a control structure their branches should be integrated into the control structure.

An intermediate representation such as tagged nested lists has many advantages. Code can be optimized by moving parts of the program — e.g., the condition of `loop`. Branching around

nested function can be avoided. Type checking, interpretation, and code generation can be reused by targeting an existing intermediate representation from a new grammar.

It is all about the separation of concerns‡: the frontend uses a grammar to deal with recognition, the intermediate language is convenient for the visitor pattern‡ and different processing steps such as type checking, interpretation, optimization, and code generation can be implemented as separate visitors.

Quick Summary

- One-pass compilation‡ uses the semantic actions executed when rules are reduced to immediately perform type checking and interpretation or code generation, all in the same action.
- Alternatively, there can be an intermediate representation of the recognized input, designed to be easy to create in the actions and to process in separate passes for type checking, memory allocation, interpretation, code generation, optimization, etc.
- In JavaScript, trees created from nested lists with leading, unique string tags are easy to create because they match grammar and program structure and they are easy to modify in place.
- The tags can be dispatched to methods of visitor‡ classes which facilitates separation of concerns‡, i.e., a divide and conquer‡ approach, and is very similar in spirit to the way semantic actions are selected by rules.
- Dispatching tags to methods creates a loose but precise coupling, i.e., it is easy to replace an implementation, e.g., to replace an interpreter by a code generator, retarget a code generator, or map a language to a processor.
- Singleton objects are very useful to provide a common context, e.g., access to a table of symbol descriptions, for visits to different nodes in a tree.
- Classes can be extended to add or replace functionality; however, overridden methods are not deleted even if they remain unused.
- Mix-ins‡ are a way to add methods to classes, e.g., to package support for subsets of nodes, and they can be selectively added.
- In JavaScript, one way to implement a mix-in† is as a function which extends a class — creating an anonymous subclass. In this case, the new methods close over the function parameters, i.e., different invocations of the mix-in can create different methods.
- The technique can lead to a set of building blocks for compilers; however, creating a flexible and comprehensive architecture is still a challenge.

A: The Practice Page

The Model
Scripting and Buttons Explained

Execution of the examples in this book is implemented as a model[2] which represents the state of an example and manipulates it, e.g., by using the parser generators or executing a function produced by action methods[2].

The model can be controlled by scripting with a Node.js† program[2] or through the graphical interface in the Practice Page[1].

Scripting

`script.js`... is a Node.js† module which reads command words from standard input, separated by white space, and applies them to a current `Model`[2]. As such the code of this script illustrates how the EBNF[2] and BNF[2] modules are used as parser generators. For example

```
 $ node script.js << 'end'
   model eg/02/11.eg       new parse   stack new parse
   ebnf  tests/02-11a.eg   new parse   stack new parse
end
```

creates a `Model`[2], loads the initial texts from an example file, represents and checks the grammar and performs syntax analysis first with the LL(1)‡ generator implemented by EBNF[2] and next with the SLR(1)‡ generator implemented by BNF[2], overwrites some texts using a different file, and performs syntax analysis with each generator again.

The command words have the following effects:

word	effect
`model`	resets all flags and variables
`load`	resets global strings
`ebnf`, `stack`, and `bnf`	clear all flags and select a parser generator
`ebnf`	clears all flags and selects the LL(1) generator[2]
`greedy`	toggles a flag to use the `expect()` rather than `check()` algorithm
`shallow`, `deep`, and `follow`	toggle flags to trace the corresponding algorithms
`lookahead`, `parser`, and `actions`	toggle flags to trace the corresponding operations
`noargs`	toggles a flag to suppress argument count checking for actions
`stack`	clears all flags and selects the SLR(1) generator[2] using translation from EBNF
`error`	toggles a flag to insert `$error` when translating from EBNF
`sets` and `states`	toggle flags to add information to the grammar description
`bnf`	clears all flags and selects the SLR(1) generator[2] using strict BNF
`build`	toggles a flag to create lists when parsing with the SLR(1) generator[2]
`new`	represents and checks the grammar
`scan`	performs lexical analysis on the program
`parse`	performs syntax analysis on the program, calling actions if any
`run`	runs the executable, if any
`1`, `10`, and `100`	execute a number of instructions in a stack machine, if any
`exit`	terminates the script

Any other word is interpreted as a path to a file from which to load texts. In the file each text is preceded by `%%` and one of the names `actions`, `grammar`, `output`, `program`, or `tokens`, enclosed in white space, to overwrite the corresponding global string.

Graphical User Interface

The Practice Page[1] contains resizable text areas which contain grammar rules in the `grammar` area, token definitions as object properties defined in JavaScript in the `tokens` area, optionally a class or an object with action methods[2] defined in JavaScript in the `actions` area, program text in the `program` area, and output from the model. The page also contains buttons which are functionally equivalent to most of the scripting command words described above.

Each text area can be resized by dragging the bottom right corner, and all but the output area can be maximized relative to the others by clicking on the label near the top right of an area; shift-click restores the original layout.

An alt-, meta-, or command-click on the label of a non-empty text area opens a pop-up window containing the text, with syntax highlighting by Sunlight† for the tokens and actions areas.

Text in the output area at the bottom can be selected and copied. All other text areas can be edited and their content is available in global variables for programming.

Button states are reflected in their background color, where white indicates that a condition is not set or that a function cannot be executed yet. Buttons are inactive while a text area has focus.

The state of the very first button and the color of all buttons indicates which grammar notation and parser are used:

button state	implementation	parsing technique	grammar notation
descent	EBNF[2]	recursive descent LL(1)	EBNF
stack-based	BNF[2]	stack-based SLR(1)	EBNF
bnf	BNF[2]	stack-based SLR(1)	strict BNF

The label at the top right of each text area indicates which global string the text area corresponds to.

If a stored example is loaded, the book button near the top right of the output area connects to this book, near the first reference to the example. Otherwise the button connects to this appendix.

The remaining buttons have the following effects:

button	effect
no check	if set, configures new grammar creation to not use the `check()`[2] algorithm, i.e., to suppress checking for ambiguity; the recursive descent parser will be greedy
shallow	if set, configures new grammar creation to trace the `shallow()`[2] algorithm and display the resulting sets
deep	if set, configures new grammar creation to trace the `deep()`[2] algorithm and display the resulting sets
follow	if set, configures new grammar creation to trace the `follow()`[2] algorithm and display the resulting sets
insert $error	if set, configures new grammar creation to insert `$error` alternatives when translating braces to BNF; likely to cause shift/reduce conflicts (which are usually benign)
sets **sets**	if set, displays the lookahead sets when a BNF grammar is processed
states **states**	if set, displays the states when a BNF grammar is processed
new grammar	represents and checks a `Grammar` and assigns it to `g`; all but white space in the `grammar` area is significant
scan	applies the scanner created from `g`, if any, to `program`; output is a list of tuples[2]
lookahead	if set, traces scanner progress during parsing
parser	if set, traces parsing `program`
actions	if set, traces calls to `actions`, if any, while parsing `program`
no args	if set, suppresses argument count checking for actions while parsing `program`
build lists	if set, adds an observer to collect input into lists when parsing with a BNF[2] grammar
parse	applies the parser created from `g`, if any, to `program`; output can contain nested lists of terminals or results of actions; a function will be assigned to `run`
run	executes `run`, if any, and displays the result
1, **10**, and **100**	execute and trace a number of instructions in the stack machine, if any

Globals

Constructing a `Model`[2] creates some global variables which operations on the `Model`[2] will affect. A question mark after a type below indicates that the global will not be overwritten if it already exists.

name	type	content
`actions`	`string`	defines the class and action methods[2] to be called by a parser
`g`	`Grammar`	represents `grammar` if not `null`
`grammar`	`string`	rules of a grammar, argument for the construction of `g`
`newOutput`	`function?`	displays it's arguments, blank-separated and marked as a new section
`program`	`string`	should be a sentence conforming to `grammar`, argument for recognition
`prompt`	`function?`	displays a prompt string, returns input or a default string, else error
`puts`	`function?`	displays it's arguments, blank-separated
`run`	`function`	`null` or an executable compiled from `program` by the `actions`; a stack machine has two arguments, other executables have none
`tokens`	`string`	defines an object with pattern properties defining the tokens used in `grammar` and compiled into `g`. A property with an empty key overwrites white space as text to be skipped in *both*, `grammar` and `program`

The parser generator modules and the modules with classes from the examples are also available through global variables.

`tokens` are evaluated when `grammar` is processed by a parser generator. `actions` are evaluated and used, if available, when a `program` is parsed. Errors are reported to the output area. Malicious content should be avoided...

Examples and Local Storage

Many examples are available… and can be loaded into the Practice Page[1] using a search string, e.g., from the following list:

search string	example explained
`?eg=interpret`[1]	interpret arithmetic expressions with numbers.
`?eg=compile`[1]	compile arithmetic expressions into functions.
`?eg=postfix`[1]	compile arithmetic expressions into postfix.
`?eg=stack`[1]	compile arithmetic expressions for a stack machine.
`?eg=little`[1]	compile a little language for a stack machine.
`?eg=little_fn`[1]	compile a little language into functions.
`?eg=typing`[1]	compile a typed little language for a stack machine.
`?eg=recursion`[1]	compile mutually recursive functions.
`?eg=functions`[1]	compile functions with parameters and local variables.
`?eg=scopes`[1]	compile block scopes.
`?eg=nesting`[1]	compile nested functions.
`?eg=first_glob`[1]	compile global first-order functions.
`?eg=fn_parameter`[1]	compile nested functions as parameters.
`?eg=first`[1]	compile first-order functions.
`?eg=curry`[1]	currying.
`?eg=compose`[1]	function composition.
`?eg=bootstrap`[1]	compile the EBNF parser generator.
`?eg=extend`[1]	extend EBNF notation.

With an `eg=` parameter the search string can select any file with an extension `.eg` in the folder `eg…` and subfolders, relative to the page. If the search fails the Practice Page[1] is reset and contains brief hints as to the use of the different text areas.

Additionally, the search string can contain a `mode=` parameter with the values `ebnf`, `stack`, or `bnf` corresponding to the command words described above.

If possible, the current state of the text areas is saved in local storage† with the key `EBNF/state` once the practice page loses visibility.

This state is normally reloaded once the practice page is visible again. It is destroyed if the search string is invalid.

Stored examples and local storage use the file format described above.

A Note on `eval()`†

The Practice Page[1] critically depends on the use of `eval()`† and thus can fail if destructive text is entered into the text areas.

`tokens` and `actions` are compiled with `eval?.()`†. Interestingly, the executable `run` is not — because it is created by the compiled `actions`.

The rules for specifying `grammar` could be changed to require that tokens be specified as part of the grammar specification, e.g., as assignments of pattern strings to token names before or after the grammar rules. This would eliminate the need for the `tokens` area.

However the action methods[2] themselves have to be compiled from the text in the `actions` area before they can be called during parsing. Once they are compiled with `eval?.()`† — just before parsing — they can be called during parsing to create any kind of JavaScript data. This,

specifically, includes functions. The executable `run` is the result of the actions during parsing and as such is a function created by the actions, i.e., it needs no further compilation.

B: The Stack Machine

Memory Architecture
Instructions

The Stack Machine uses JavaScript functions to simulate machine instructions. Beginning with `Machine10`[2] the machine classes generate and store the instructions in an array `code[]` which is considered immutable once filled.

To simplify visual inspection of `code[]`, the instructions are implemented through methods of the machine classes. Methods for simple instructions such as `Add` directly use `memory` as an argument, i.e., they are instruction functions. Other methods such as `Branch` require a memory address, etc., as arguments and return the actual instruction functions. All of the instruction functions manipulate `memory` in some way.

There are two interpreters which call the instruction functions:

- The *stack machine*‡ uses one JavaScript array `memory` for frames and the value stack.
- The *garbage-collected stack machine*‡ uses `memory` for global information followed by the value stack, and individually allocated arrays for the frames. These arrays have a property `.id` with a unique value for tracing; `.id` is initialized from `memory.id`.

To enable single step execution the interpreters return `memory` with a property `.continue` indicating if the program is suspended (`true`) or has terminated (`false`).

Example 6/10[1]: Arithmetic Expressions

`memory`	use
`0` ...	values of global variables
size ...	value stack

`Six.Machine10`[2]

```
Add (memory)                          // stack: ... a b -> ... a+b
Divide (memory)                       // stack: ... a b -> ... a/b
Input (dflt)                            // stack: ... -> ... input
Load (addr)                         // stack: ... -> ... memory[addr]
Minus (memory)                          // stack: ... a -> ... -a
Multiply (memory)                     // stack: ... a b -> ... a*b
Pop (memory)                            // stack: ... val -> ...
Push (result)                         // stack: ... -> ... result
Puts (memory)                   // stack: ... val -> ... | puts(val)
Store (a)             // stack: ... val -> ... val  | memory[a]: val
Subtract (memory)                     // stack: ... a b -> ... a-b
```

Example 6/11[1]: Control Structures

memory	use
`.pc` *register*	next address in `code` to execute
`0` ...	values of global variables
size ...	value stack

`Six.Machine11 extends Six.Machine10`[2]

```
1  Branch (a)                              // stack: ... -> ... | pc: a
2  Bzero (a)                    // stack: ... bool -> ... | pc: !bool? a
3  Eq (memory)                           // stack: ... a b -> ... a == b
4  Ge (memory)                           // stack: ... a b -> ... a >= b
5  Gt (memory)                            // stack: ... a b -> ... a > b
6  Le (memory)                           // stack: ... a b -> ... a <= b
7  Lt (memory)                            // stack: ... a b -> ... a < b
8  Ne (memory)                           // stack: ... a b -> ... a != b
9  Print (n)                                  // stack: ... n*val -> ...
```

Example 7/04[1]: Functions

memory	use
`.pc` *register*	next address in `code` to execute
`0` ...	values of global variables
frame	
`+0`	result value of function call
`+1`	return address in `code` for function call
...	stack

`Seven.Machine04 extends Six.Machine11`[2]

```
1  // stack: ... -> ... old-pc | pc: addr
2  Call (addr)
3  // stack: ... old-pc -> ,,, 0 old-pc
4  Entry (memory)
5  // stack: ... old-pc -> ... | pc: old-pc
6  Return (memory)
7  // stack: ... x old-pc result -> ... result old-pc result
8  ReturnValue (memory)
```

Example 7/06[1]: Local Variables

`memory`	**use**
`.pc` *register*	next address in `code` to execute
`.fp` *register*	base address of current frame in `memory`
`0` ...	values of global variables
frame	
`+0` ...	argument values for the parameters
`+`*parms*	return address in `code` for function call
`+`*parms*`+1`	*dynamic link*, i.e., address in `memory` of previous frame
`+`*parms*`+2`	result value of function call
`+`*parms*`+3` ...	values of local variables
...	stack

`Seven.Machine06 extends Seven.Machine04`[2]

```
1  // stack: ... arguments old-pc
2  //     -> ... arguments old-pc old-fp result locals
3  Entry (parms, size)
4  // stack: ... arguments old-pc old-fp result locals
5  //     -> ... result old-pc
6  Exit (parms)
7  // stack: ... -> ... frame[addr]
8  LoadFP (addr)
9  // stack: ... val -> ... val | frame[addr]: val
10 StoreFP (addr)
```

Example 7/13[1]: Nested Functions

`memory`	**use**
`.pc` *register*	next address in `code` to execute
`.fp` *register*	base address of current frame in `memory`
`.dp` *register*	base address of current display in `memory`
`0` ...	values of global variables
frame	
`+0` ...	values of parameter arguments
`+`*parms*	return address in `code` for function call
`+`*parms*`+1`	address in `memory` of previous frame
`+`*parms*`+2`	address in `memory` of previous display
`+`*parms*`+3`	result value of function call — `memory.dp` points here
`+`*parms*`+3+1` ...	base addresses of visible frames
`+`*parms*`+3+`*depth*	base address of this frame — at *depth*
`+`*parms*`+4+`*depth* ...	values of local variables
...	stack

`Seven.Machine13 extends Seven.Machine06`[2]

```
// stack: ... arguments old-pc
//     -> ... arguments old-pc old-fp old-dp result display locals
Entry (parms, depth, size)
// stack: ... arguments old-pc old-fp old-dp result display locals
//     -> ... result old-pc
Exit (memory)
// stack: ... -> ... frame[depth][addr]
LoadDP (addr, depth)
// stack: ... val -> ... val | frame[depth][addr]: val
StoreDP (addr, depth)
```

Example 8/01[1]: Global first-order Functions

Same memory layout as Local Variables above.

mix-in `Eight.Machine01()`[2]

```
// stack: ... addr -> ... old-pc | pc: addr`
CallValue (memory)
// stack: ... x-len n*val -> ... n*val x-len`
Rotate (n, len = 1)
```

Example 8/08[1]: Functions as Argument Values

Same memory layout as Nested Functions above.

This machine requires *two* slots to represent a function value: display pointer followed by function start address. This does not affect the `Call`[2], `CallValue`[2], and `Return`[2] instructions.

mix-in `Eight.Machine08()`[2]

```
// stack: ... arguments dp old-pc
//        -> ... arguments old-pc old-fp old-dp result display locals`
Entry (args, depth, vars)
// stack: ... arguments old-pc old-fp old-dp result display locals
//        -> ... result old-pc`
Exit (args)
// stack: ... -> ... dp`
PushDP (memory)
```

Example 8/14[1]: Nested first-order Functions

This machine manages frames dynamically: frames are separate arrays subject to JavaScript's garbage collection.

This machine requires *two* slots to represent a function value: frame pointer followed by function start address. This does not affect the `Call`[2], `CallValue`[2], and `Return`[2] instructions.

`memory`	**use**
`.pc` *register*	next address in `code` to execute
`.fp` *register*	`null` or `Array` of current frame
`0` ...	values of global variables
...	stack
`frame`	**use**
`0`	return address in `code` for function call
`1`	`null` or `Array` of previous frame
`1+1` ...	arrays of visible frames
`1+`*depth*	`Array` of this frame (at *depth*)
`2+`*depth*	result value of function call
`3+`*depth*	extra slot, exactly if result value is function value
...	argument values
... *frame size*`-1`	local variable values

mix-in `Eight.Machine14()`[2]

```
// stack: ... arguments fp old-pc
//        -> ... | frame: old-pc old-fp display result arguments locals`
Entry (args, depth, result, vars)
// stack: ... | frame: old-pc old-fp display result ...
//        -> ... result old-pc | fp: old-fp | frame unchanged`
Exit (depth, result)
// stack: ... -> ... frame[depth][addr]`
LoadGC (addr, depth)
// stack: ... -> ... fp`
PushFP (memory)
// stack: ... val -> ... val | frame[depth][addr]: val`
StoreGC (addr, depth)
```

Example 11/10[1]: Compiling Arithmetic Expressions

This mix-in for code generation extends `Six.Machine10`[2] with a `Power` instruction to support exponentiation.

mix-in `Eleven.Code_Number()`[2]

```
Power (memory)                        // stack: ... a b -> ... a**b
```

Example 11/11[1]: Compiling a Little Language

This mix-in for code generation extends `Six.Machine11`[2] with a `Bnzero` instruction to optimize code generation for `while` loops. It requires `Eleven.Symbols`[2].

mix-in `Eleven.Code_Stmts()`[2]

```
Bnzero (a)                  // stack: ... bool -> ... | pc: bool? a
```

Example 11/12[1]: Compiling a Typed Little Language

These mix-ins for code generation extend `Six.Machine11`[2] with instructions to support code generation for Boolean and string values and conversions.

mix-in `Eleven.Code_Bool()`[2]

```
IfTrue (a)             // stack: ... bool -> ... bool | pc: bool? a
IfFalse (a)           // stack: ... bool -> ... bool | pc: !bool? a
Not (memory)                             // stack: ... a -> ... !a
```

mix-in `Eleven.Code_String()`[2]

```
Concat (memory)                        // stack: ... a b -> ... a+b
Len (memory)                          // stack: ... a -> ... a.length
InputString ('prompt', 'default')       // stack: ... -> ... string
```

mix-in `Eleven.Code_Cast()`[2]

```
Cast ('to', 'from')                   // stack: ... a -> ... cast a
```

C: The One-Pass Compilers

The Method Browser
Rules and Actions

Symbol Table

Appendix B summarized the different versions of the stack machine. This appendix contains tables which index the stack machine compilers for the little language developed in chapters six, seven, and eight. The table entries are linked to the examples, to the documentation, and from there to the sources.

These compilers generate code during recognition, i.e., in the action methods. The action classes extend from example 6/10[1] (`Arithmetic10`[2]) to example 6/11[1] (`Control11`[2]), 7/4[1] (`Functions04`[2]), 7/6[1] (`Parameters06`[2]), and 7/9[1] (`Blocks09`[2]).

To implement nested functions example 7/13[1] defines the mix-in‡ `Nest13`[2] to extend `Blocks09`[2] and support function definitions at greater depth — which requires a static link‡.

To implement global first-order functions example 8/1[1] defines the mix-in `Global01`[2] to extend `Blocks09`[2] and primarily support type checking with function types for variables, functions, parameters, and argument values.

To implement nested functions as argument values example 8/8[1] defines the mix-in `Pass08`[2] to extend `Blocks09`[2] and `Nest13`[2]; the language forbids using functions and results or variables with function values.

Finally, to implement nested first-order functions, example 8/14[1] defines the mix-in `First14`[2] to extend the implementation in example 8/1[1] and support dynamic memory management for frames.

The Method Browser

The method browser[3] can be used to study the evolution of the action classes and methods and, in particular, to see where a method is first defined.

By default it will load the sources of the implementation of the little language in chapters six through eight, but `file` parameters can be used in a search string to load JavaScript files from the `modules` directory.

Patterns are used to extract module, class or mix-in, and method names, and the source text of the methods. The patterns assume a particular coding style, mostly based on white space and nesting.

The names are displayed in a table with columns for modules, classes or mix-ins, methods, and nested methods. Names can be selected by clicking in each column. Multiple selections within a column are considered alternatives, selections across columns must all be satisfied. A column without selections is a wildcard.

Selected names have a darker blue background, names that could be added to a selection have a

lighter blue background.

Each column can be scrolled vertically. If the mouse hovers over a column (indicated by a focus border) a keypress will scroll to names with the corresponding initial and the *delete* key will clear all selections in the column.

A combination of names selects methods which can be displayed, sorted alphabetically by method name, or by class or mix-in. A single method is displayed as soon as it is selected.

An initial selection can be defined by specifying one or more `module`, `item`, `method`, and `op` parameters in a search string for the page. A `show` parameter requests immediate display of the selection.

`op` refers to *nested methods*, i.e., methods within classes which are nested into top-level classes, e.g., the classes which are used to represent types, variables, and functions in the implementation of the little language. The nested methods are also located by patterns based on coding style.

Grammar Rules and Actions

This table shows how the grammar rules and actions are defined or overridden. Numbers reference the example where a rule was used and ⊕ references action method definitions or overrides.

rules										
list: stmt [{ ';' stmt }];	6/10[1]									
prog: stmts;		6/11[1]	⊕[2]							
prog: [vars] funs;				7/4[1]	⊕[2]	7/6[1]		7/9[1]		7 1
prog: [typedcls] [vars] funs;										
typedcls: { 'type' typedcl [{ ',' typedcl }] ';' };										
typedcl: Name '(' [types] ')' [':' typename];										
typedcl: Name '(' [types] ')' [':' number'];										
types: typename [{ ',' typename }];										
typename: Name | 'number';										
vars: 'var' names ';';				7/4[1]		7/6[1]		7/9[1]		7 1
vars: 'var' varname [{ ',' varname }] ';';										
varname: Name [':' type];										
varname: Name;										
type: Name | 'number';										
names: Name [{ ',' Name }];				7/4[1]	⊕[2]	7/6[1]		7/9[1]		7 1
funs: { fun };				7/4[1]		7/6[1]		7/9[1]		7 1
fun: head ['begin' stmts 'end'] ';';				7/4[1]	⊕[2]					
fun: head parms [block] ';';						7/6[1]	⊕[2]	7/9[1]		7 1
head: 'function' Name;				7/4[1]	⊕[2]	7/6[1]		7/9[1]		7 1
parms: '(' [names] ')';						7/6[1]	⊕[2]	7/9[1]		7 1
parms: '(' [names] ')' [':' Name];										
block: 'begin' [vars] stmts 'end';						7/6[1]				
block: begin [vars] stmts 'end';								7/9[1]	⊕[2]	
block: begin body 'end';										7

rules											
											1
`begin: 'begin';`									7/9[1]	⊕[2]	7 1
`body: [ vars ] [ funs ] stmts;`											7 1
`stmts: stmt [{ ';' stmt }];`			6/11[1]		7/4[1]		7/6[1]		7/9[1]		7 1
`stmt: sum;`	6/10[1]	⊕[2]									
`stmt: assign | print | loop | select;`			6/11[1]	⊕[2]							
`stmt: assign | print | return | loop | select;`					7/4[1]		7/6[1]				
`stmt: assign | print | return | block | loop | select;`									7/9[1]		7 1
`assign: Name '=' sum;`			6/11[1]	⊕[2]							
`assign: Name [ '=' sum ];`					7/4[1]	⊕[2]					
`assign: symbol action;`							7/6[1]	⊕[2]	7/9[1]		7 1
`action: store | call;`							7/6[1]		7/9[1]		7 1
`store: '=' sum;`							7/6[1]	⊕[2]	7/9[1]		7 1
`call: args;`							7/6[1]		7/9[1]		7 1
`call: { args };`											
`args: '(' [ sums ] ')';`							7/6[1]	⊕[2]	7/9[1]		7 1
`print: 'print' sums;`			6/11[1]	⊕[2]	7/4[1]		7/6[1]		7/9[1]		7 1
`sums: sum [{ ',' sum }];`			6/11[1]	⊕[2]	7/4[1]		7/6[1]		7/9[1]		7 1
`return: 'return' [ sum ];`					7/4[1]	⊕[2]	7/6[1]		7/9[1]		7 1
`loop: While cmp Do stmts 'od';`			6/11[1]	⊕[2]	7/4[1]		7/6[1]				
`loop: While cmp Do [ vars ] stmts 'od';`									7/9[1]	⊕[2]	
`loop: While cmp Do body 'od';`											7 1
`While: 'while';`			6/11[1]	⊕[2]	7/4[1]		7/6[1]		7/9[1]		7 1

rules								
`Do: 'do';`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	⊕[2] 7/11[1]
`select: 'if' cmp Then stmts [ Else stmts ] 'fi';`			6/11[1]	⊕[2]	7/4[1]	7/6[1]		
`select: 'if' cmp then [ else ] 'fi';`							7/9[1]	⊕[2] 7/11[1]
`then: Then [ [ vars ] stmts ];`							7/9[1]	⊕[2]
`then: Then [ body ];`								7/11[1]
`else: Else [ vars ] stmts ;`							7/9[1]	⊕[2]
`else: Else body ;`								7/11[1]
`Then: 'then';`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	⊕[2] 7/11[1]
`Else: 'else';`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	⊕[2] 7/11[1]
`cmp: sum rel;`			6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]
`rel: eq | ne | gt | ge | lt | le;`			6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]
`eq: '=' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`ne: '<>' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`gt: '>' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`ge: '>=' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`lt: '<' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`le: '<=' sum;`			6/11[1]	⊕[2]	7/4[1]	7/6[1]	7/9[1]	7/11[1]
`sum: 'let' Name '=' sum | product [{ add | subtract }];`	6/10[1]	⊕[2]						
`sum: product [{ add | subtract }];`			6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]
`add: '+' product;`	6/10[1]	⊕[2]	6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]
`subtract: '-' product;`	6/10[1]	⊕[2]	6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]
`product: signed [{ multiply | divide }];`	6/10[1]		6/11[1]		7/4[1]	7/6[1]	7/9[1]	7/11[1]

rules							
`multiply: '*' signed;`	6/10[1]	⊕[2]	6/11[1]	7/4[1]		7/6[1]	7/9[1]
`divide: '/' signed;`	6/10[1]	⊕[2]	6/11[1]	7/4[1]		7/6[1]	7/9[1]
`signed: [ '-' ] term;`	6/10[1]	⊕[2]	6/11[1]	7/4[1]		7/6[1]	7/9[1]
`term: input | number | name | '(' sum ')';`	6/10[1]		6/11[1]	7/4[1]		7/6[1]	7/9[1]
`input: 'input' [ Number ];`	6/10[1]	⊕[2]	6/11[1]	7/4[1]		7/6[1]	7/9[1]
`number: Number;`	6/10[1]	⊕[2]	6/11[1]	7/4[1]		7/6[1]	7/9[1]
`name: Name;`	6/10[1]	⊕[2]	6/11[1]	7/4[1]	⊕[2]		
`name: symbol [ args ];`						7/6[1]	⊕[2] 7/9[1]
`name: symbol [{ args }];`							
`symbol: Name;`						7/6[1]	⊕[2] 7/9[1]

Symbol Table Entries

Symbol table entries have a common base class `Symbol`[2] which ensures that every symbol has a `name` and can reference the `owner`, i.e., the action class which defines the symbol's class; therefore, the symbol classes are inner classes‡ of an action class as the `owner`.

Getters† in the action classes are used to access the symbol classes because getters can be overwritten in subclasses to silently return extended symbol classes. It should be noted, however, that a reference to a getter can be confused with a reference to a method with the same name, i.e., given the class definition

```
1  class X {
2    get x () { /* ... */ }
3    x () { /* ... */ }
4  }
```

the reference `(new X()).x` is ambiguous; e.g., the Node.js† JavaScript compiler will return the method function and not note an error.

property	`Symbol`[2]	usage
`.owner`	7/4[1]	outer class' object
`.name`	7/4[1]	symbol's name

Variables

Variable descriptions have a common base class `Var`[2] which is extended by the various action classes. The table links the properties of the descriptions to the classes and examples where they are defined or overridden. `()` indicates a method. `+` indicates that an override references `super`.

property	`Var`[2]	`Var`[2]	`Var`[2]	`Var`[2]	`Var`[2]	`Var`[2]	usage
`.addr`	7/4[1]						memory address or offset
`.depth`		7/6[1]	7/13[1]				(static) level
`.type`				8/1[1]			variable's type
`.load()`	7/4[1]	7/6[1]	7/13[1]		8/8[1]	8/14[1]	generates `Load*`
`.storeOk()`	7/4[1]			8/1[1]	+8/8[1]		`true` unless `store()` is in error
`.store()`	7/4[1]	7/6[1]	7/13[1]			8/14[1]	generates `Store*`
`.call()`				8/1[1]			generates `Call*`
`.toString()`	7/4[1]	7/6[1]	7/13[1]	+8/1[1]			text representation

Functions

Function descriptions have a common base class `Fun`[2] which is extended by the various action classes. The table links the properties of the descriptions to the classes and examples where they are defined or overridden. `[]` indicates that a property is a list. `()` indicates a method. `+` indicates that an override references `super`.

property	Fun[2]	Fun[2]	Fun[2]	Fun[2]	Fun[2]	Fun[2]	Fun[2]	usage
`.start`	7/4[1]							`false` or code address
`.calls[]`	7/4[1]							slots for `Call*`
`.returns[]`	7/4[1]							slots for `Branch` exit
`.parms`		7/6[1]						number of parameters
`.addr`		7/6[1]						result value offset
`.locals` get/set		7/6[1]	7/9[1]					maps local names to descriptions
`.size` get/set		7/6[1]	7/9[1]					next address in frame
`.blocks[]`			7/9[1]					`.locals`/`.size` stack
`.frameSize`			7/9[1]					frame size
`.depth`				7/13[1]				(static) level
`.scope`				7/13[1]				block containing definition
`.type`					8/1[1]			function's type
`.loads[]`					8/1[1]			slots for `Push*`
`.entry()`	7/4[1]	7/6[1]		+7/13[1]				sets `start`, reserves `Entry`
`.undo()`	7/4[1]	+7/6[1]						undoes `entry()`
`.setParms()`		7/6[1]		+7/13[1]	8/1[1]	8/8[1]	8/14[1]	sets `.parms` `.addr`, types
`.call()`	7/4[1]					+8/8[1]	+8/14[1]	generates `Call*`
`.return()`	7/4[1]							generates `Branch` exit
`.storeOk()`	7/4[1]			7/13[1]	+8/1[1]			`true` unless `store()` is in error
`.store()`	7/4[1]	7/6[1]		7/13[1]			8/14[1]	generates `Store*` result, types
`.load()`					8/1[1]	+8/8[1]	+8/14[1]	generates `Push*`
`.push()`			7/9[1]					pushes `.blocks[]`
`.pop()`			7/9[1]	+7/13[1]				pops `.blocks[]`, sets frame size
`.end()`	7/4[1]		+7/9[1]		+8/1[1]			fixes forward references, `exit()`
`.exit()`	7/4[1]	7/6[1]	7/9[1]	7/13[1]		8/8[1]	8/14[1]	generates `Entry`, `Exit`, `Return`
`.toString()`	7/4[1]	7/6[1]	7/9[1]	+7/13[1]	+8/1[1]			text representation

Blocks

A `Block`[2] represents a scope of names to support block structure.

property	Block[2]		usage
`.locals`	7/9[1]		maps names in scope to descriptions
`.size`	7/9[1]		next address in scope
`.toString()`	7/9[1]		text representation
`.bypass`		7/13[1]	address of branch bypassing functions

Types

There is just one type description class `Type`[2] and there is a separate, global symbol table `typeSymbols`[2] for types to allow matching of functions and types on their names. Two type values have reserved names: `number`[2] describes number values and, for convenience, `main`[2] describes a function without arguments which returns a number.

property	Type[2]	usage
`.parms[]`	8/1[1]	`null` or list of parameter types
`.returns`	8/1[1]	`null` or result type
`.isFun`	8/1[1]	`true` if function type
`.toString()`	8/1[1]	text representation

D: The Compiler Kit

Tree Builders
Visitors

This appendix contains tables which index the tree builders, interpreters, type checkers, and stack machine code generators for the little language developed in chapter eleven. The table entries are linked to the examples, to the documentation, and from there to the sources.

All code is structured as mix-ins‡ which are mostly independent of each other to be combined for different types of language processing. All classes and mix-ins are contained in module `Eleven`[2].

The method browser[3] described in appendix C can be used to compare the action methods for the tree builders and the processing methods in the visitors.

Tree Builders

Trees are built during recognition, i.e., in action methods. Tree builder mix-ins are applied to the base class `Build`[2] which provides a small amount of infrastructure.

Tree building employs an SLR(1) grammar‡ with a precedence table which allows a symmetric definition of the operations in expressions. This table links the precedence levels to the builders and examples which require them:

precedence	builder mix-in	used in
`%left 'or';` `%left 'and';`	`Build_Bool`[2]	06[1] 07[1] 08[1] 09[1] 12[1]
`%nonassoc '=' '<>' '>' '>=' '<' '<=';`	`Build_Cmps`[2]	04[1] 05[1] 06[1] 07[1] 08[1] 09[1] 11[1] 12[1]

`%left '+' '-';`
`%left '*' '/';`
`%right '**';`
`%right Number;`
	`Build_Number`[2]	02[1] 03[1] 04[1] 05[1] 06[1] 07[1] 08[1] 09[1] 10[1] 11[1] 12[1]

The following table links tree builder mix-ins to the examples which use them, lists their grammar rules and links them to the builder actions and from there to the sources, and specifies the resulting tree nodes or values. The table in the next section of this appendix relates the tree nodes to visitor mix-ins which can process them and to the examples which use the visitors.

The `Main`[2] mix-in supports a build rule `main:` *tree*`;` which applies a list of visitors, specified as mix-in arguments, to the *tree* and returns a function which will apply the last visitor. Convenience methods support build rules `dump:` *tree*`;` to dump a tree and `run:` *function*`;` to execute the function.

The `Compile`[2] mix-in supports a build rule `compile:` *tree*`;` which applies a list of visitors, specified as mix-in arguments, to the *tree* and returns the executable created by the last visitor which must be a code generator.

Operations on names require a symbol table which is supported by the `Symbols`[2] mix-in and forwarded from one visitor to the next.

Compile[2], **used in** 10[1] 11[1] 12[1]	**action**	**returns**
`compile:` *tree*`;`	`compile()`[2]	`(memory, steps) => StackMachine(`
Main[2], **used in** 03[1] 05[1] 06[1] 07[1] 08[1] 09[1] 10[1] 11[1] 12[1]	**action**	**returns**
`run:` *function*`;`	`run()`[2]	*function's result*
`main:` *tree*`;`	`main()`[2]	`()` => *last visitor*`(`*tree*`)`
`dump:` *tree*`;`	`dump()`[2]	*tree*
Build_Dcl[2], **used in** 08[1] 09[1] 12[1]	**action**	**returns**
`block: item [{ ';' item }];`	`block()`[2]	`[ 'block' dcl ... stmt ... ]`
`item: dcl | stmt;`		
`dcl: type Name [{ ',' Name }];`	`dcl()`[2]	`[ 'dcl' type Name ... ]`
Build_Stmts[2], **used in** 04[1] 05[1] 08[1] 09[1] 11[1] 12[1]	**action**	**returns**
`stmts: stmt [{ ';' stmt }];`	`stmts()`[2]	`stmt | [ 'stmts' stmt ... ]`
`stmt: print | ...;`	`stmt()`[2]	*tree*
`print: 'print' expr [{ ',' expr }];`	`print()`[2]	`[ 'print' expr ... ]`
`loop: 'while' expr 'do' stmts 'od';`	`loop()`[2]	`[ 'loop' expr stmts ]`
`select: 'if' expr 'then' stmts [ 'else' stmts ] 'fi';`	`select()`[2]	`[ 'if' expr stmts stmts? ]`
Build_Names[2], **used in** 04[1] 05[1] 08[1] 09[1] 11[1] 12[1]	**action**	**returns**
`assign: Name '=' expr;`	`assign()`[2]	`[ 'assign' Name expr ]`
`name: Name;`	`name()`[2]	`[ 'name' Name ]`
Build_Cast[2], **used in** 06[1] 07[1] 08[1] 09[1] 12[1]	**action**	**returns**
`type: 'bool' | 'number' | 'string';`	`type()`[2]	*typename*
`cast: '(' type ')' expr %prec Number;`	`cast()`[2]	`[ 'cast' type expr ]`
Build_String[2], **used in** 06[1] 07[1] 08[1] 09[1] 12[1]	**action**	**returns**
`input: 'input' [ String String ];`	`input()`[2]	`[ 'input' String? String? ]`
`len: 'len' expr %prec Number;`	`len()`[2]	`[ 'len' expr ]`
`string: String;`	`string()`[2]	`[ 'string' String ]`
Build_Bool[2], **used in** 06[1] 07[1] 08[1] 09[1] 12[1]	**action**	**returns**
`or: expr 'or' expr;`	`or()`[2]	`[ 'or' expr expr ]`
`and: expr 'and' expr;`	`and()`[2]	`[ 'and' expr expr ]`
`not: 'not' expr %prec Number;`	`not()`[2]	`[ 'not' expr ]`
`bool: 'true' | 'false';`	`bool()`[2]	`[ 'bool' Boolean ]`
Build_Cmps[2], **used in** 04[1] 05[1] 06[1] 07[1] 08[1] 09[1] 11[1] 12[1]	**action**	**returns**
`eq: expr '=' expr;`	`eq()`[2]	`[ 'eq' expr expr ]`
`ne: expr '<>' expr;`	`ne()`[2]	`[ 'ne' expr expr ]`
`gt: expr '>' expr;`	`gt()`[2]	`[ 'gt' expr expr ]`
`ge: expr '>=' expr;`	`ge()`[2]	`[ 'ge' expr expr ]`
`lt: expr '<' expr;`	`lt()`[2]	`[ 'lt' expr expr ]`
`le: expr '<=' expr;`	`le()`[2]	`[ 'le' expr expr ]`
Build_Number[2], **used in** 02[1] 03[1] 04[1] 05[1] 06[1] 07[1] 08[1] 09[1] 10[1] 11[1] 12[1]	**action**	**returns**
`expr: add | ... | '(' expr ')';`	`expr()`[2]	*tree*

Compile[2], used in 10[1] 11[1] 12[1]	action	returns
`add: expr '+' expr;`	`add()`[2]	`[ 'add' expr expr ]`
`subtract: expr '+' expr;`	`subtract()`[2]	`[ 'subtract' expr expr ]`
`multiply: expr '+' expr;`	`multiply()`[2]	`[ 'multiply' expr expr ]`
`divide: expr '+' expr;`	`divide()`[2]	`[ 'divide' expr expr ]`
`power: expr '+' expr;`	`power()`[2]	`[ 'power' expr expr ]`
`minus: '-' expr %prec Number;`	`minus()`[2]	`[ 'minus' expr ]`
`number: Number;`	`number()`[2]	`[ 'number' Number ]`

Visitors

The visitor pattern‡ is useful to process trees. Module `Eleven`[2] implements three kinds of visitors‡:

- Interpreters visit trees and evaluate the nodes — basically in postorder‡. Interpreter mix-ins are applied to the base class `Visit`[2] which implements the visitor pattern‡ and provides tracing.
- Type Checkers visit, annotate, and modify trees in place prior to interpretation or code generation. Type checker mix-ins are applied to the base class `Check`[2] which extends `Visit`[2] and implements convenience methods.
- Code generators visit trees, perhaps modified by type checking. Code generators use a stack machine generator ultimately based on `Machine10`[2] to generate code and create a stack machine executable. Code generator mix-ins are applied to the base class `Code`[2] which extends `Visit`[2], implements convenience methods, and by default imports `Machine10`[2].

Visiting names requires a symbol table which is supported by the `Symbols`[2] mix-in and forwarded from one visitor to the next.

The following table links visitor mix-ins to the examples which use them, lists their tree nodes and links them to the visitor methods and from there to the source code, and specifies the results.

[node]	method	Eval_Dcl[2] 09[1] returns	Check_Dcl[2] 08[1] 09[1] 12[1] sets .type	Code_Dcl[2] 12[1] generates
'block' dcl... stmt...	block()	undefined[2]	not set[2]	[2]
'dcl' type Name ...	dcl()	undefined[2]	sets each Name[2]	[2] Push[2] Store[2] P
[node]	method	Eval_Stmts[2] 05[1] 09[1]	Check_Stmts[2] 08[1] 09[1] 12[1]	Code_Stmts[2] 11[1] 12[1]
'stmts' stmt ...	stmts()	undefined[2]	not set[2]	[2]
'print' value ...	print()	undefined[2]	not set[2]	[2] Print[2]
'loop' cond stmt	loop()	undefined[2]	not set[2]	[2] Branch[2]
'select' cond then else?	select()	undefined[2]	not set[2]	[2] Branch[2]
[node]	method	Eval_Names[2] 05[1] 09[1]	Check_Names[2] 08[1] 09[1] 12[1]	Code_Names[2] 11[1] 12[1]
'name' Name	name()	Name's value[2]	Name's type[2]	[2] Load[2]
'assign' Name value	assign()	undefined[2]	not set[2]	[2] Store[2] Pop[2]
[node]	method	Eval_Cast[2] 06[1] 07[1] 09[1]	Check_Cast[2] 07[1] 08[1] 09[1] 12[1]	Code_Cast[2] 12[1]
'cast' type expr	cast()	type'ed value[2]	type[2]	[2] Cast[2]
[node]	method	Eval_String[2] 06[1] 07[1] 09[1]	Check_String[2] 07[1] 08[1] 09[1] 12[1]	Code_String[2] 12[1]
'input' prompt? default?	input()	String[2]	'string'[2]	[2] InputString[2]
'concat' a b	concat()	String[2]	'string'[2]	[2] Concat[2]
'len' a	len()	Number[2]	'number'[2]	[2] Len[2]
'string' String	string()	String[2]	'string'[2]	[2] Push[2]
[node]	method	Eval_Bool[2] 06[1] 07[1] 09[1]	Check_Bool[2] 07[1] 08[1] 09[1] 12[1]	Code_Bool[2] 12[1]
'or' a b	or()	Boolean[2]	'bool'[2]	[2] IfTrue[2]
'and' a b	and()	Boolean[2]	'bool'[2]	[2] IfFalse[2]
'not' a	not()	Boolean[2]	'bool'[2]	[2] Not[2]
'bool' Boolean	bool()	Boolean[2]	'bool'[2]	[2] Push[2]
[node]	method	Eval_Cmps[2] 05[1] 06[1] 07[1] 09[1]	Check_Cmps[2] 07[1] 08[1] 09[1] 12[1]	Code_Cmps[2] 11[1] 12[1]
'eq' a b	eq()	Boolean[2]	'bool'[2]	[2] Eq[2]
'ne' a b	ne()	Boolean[2]	'bool'[2]	[2] Ne[2]
'gt' a b	gt()	Boolean[2]	'bool'[2]	[2] Gt[2]
'ge' a b	ge()	Boolean[2]	'bool'[2]	[2] Ge[2]
'lt' a b	lt()	Boolean[2]	'bool'[2]	[2] Lt[2]
'le' a b	le()	Boolean[2]	'bool'[2]	[2] Le[2]
[node]	method	Eval_Number[2] 03[1] 05[1] 06[1] 07[1] 09[1]	Check_Number[2] 07[1] 08[1] 09[1] 12[1]	Code_Number[2] 10[1] 11[1] 12[1]
'add' a b	add()	Number[2]	'number'[2]	[2] Add[2]
'subtract' a b	subtract()	Number[2]	'number'[2]	[2] Subtract[2]

[node]	method	Eval_Dcl[2] 09[1] returns	Check_Dcl[2] 08[1] 09[1] 12[1] sets .type	Code_Dcl[2] 12[1] generates
'multiply' a b	multiply()	Number[2]	'number'[2]	Multiply[2]
'divide' a b	divide()	Number[2]	'number'[2]	Divide[2]
'minus' a	minus()	Number[2]	'number'[2]	Minus[2]
'number' n	number()	Number[2]	'number'[2]	Push[2]