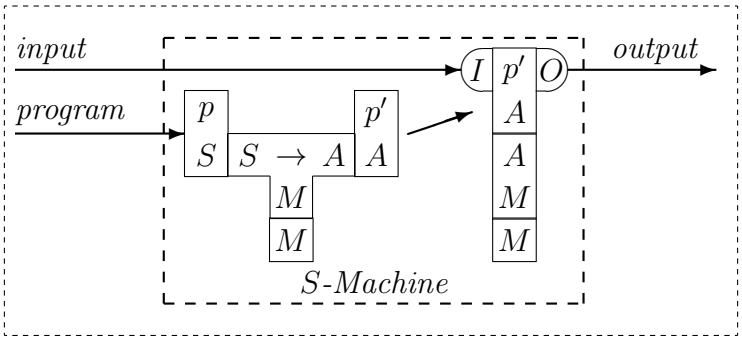
Berthold Hoffmann
hof@informatik.uni-bremen.de

Compiler Construction

Principles of Programming Language Implementation

Course in Sommer 2014
(Notes for an E-learning Course)



Fachbereich Mathematik und Informatik – Universität Bremen

Abstract. This course is about everything that every student of computer science should know about the implementation of high-level programming languages like C++, JAVA, and Haskell. Only a few students will later implement an programming language, but all of them should know how programs are translated and interpreted, in order to estimate how efficiently certain programs are executed. Also, many students will, at some time in their professional life, implement or extend compilers or interpreters for special application languages.

The Contents in a Nutshell

1. Introduction (implementation of programming languages, structure of compilers, and preliminaries)
2. Lexical analysis (regular expressions, finite automata, `lex`)
3. Syntax analysis (context-free grammars, context-free parsers, `yacc`)
4. Contextual analysisis (declaration and type analysis)
5. Code generation for imperative and object-oriented languages

In the optional practical work, the contents of the course can be consolidated. Here, students shall extend the compiler for the tiny programming language OOPS step by step until, finally, it will translates an object-oriented programming language with single inheritance, dynamic binding of methods, and automated garbage collection. The basic compiler, and the interpreter for its target language are given. The system shall be developed in the object-oriented programming language JAVA.

Contents

1

Introduction

This chapter takes a bird's eye view of the implementation of programming languages with interpreters and compilers, and sketch the typical structure of compilers.

1.1 Machines, Interpreters, Compilers

The implementation of programming languages starts from the abstract notion of a *machine*. Such a machine is *programmable*, since it can execute arbitrary *programs* of a language L, where every program reads some *input* data and produces some *output* data. The machine is equated with the language it accepts, and called an L-machine. It is *universal*, since one assumes the language L to be *computationally complete* in the sense that it allows to express every computable function. Figure 1.1 shows a *universal programmable computing machine* as a *black box*. In Figure 1.2 it is shown how we visualize the execution of an L-program p on an L-machine. So a "machine" is what is nowadays called a platform: the computer hardware with the operating system running on it.

Abstractly, implementing a programming language L aims at constructing a universal programmable machine for L. Although "*construction*" could be taken virtually as the task of building the machine *concretely*, in hardware, this is typically not the case.

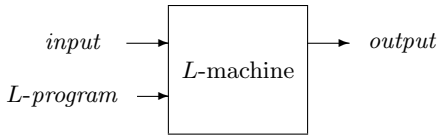


Fig. 1.1. A (universal, programmable) L-machine

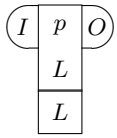


Fig. 1.2. Execution of a program

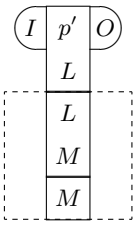


Fig. 1.3. Interpretation of a program

Instead, an L-machine can be realized as an *abstract machine*: One writes, in the language M of some concrete machine, a program that can execute arbitrary L-programs, taking their input to produce the output. Typically, the language M is low-level: machine language or assembler of a concrete hardware, or a low-level programming language that can be executed on some concrete machine. The program executing L-programs is called an *interpreter*. With interpreters, one can *simulate* machines for several languages L_1 to L_k on the same concrete machine M. The interpretation of a program with an abstract machine is drawn as in Figure 1.3. An interpreter has the drawback that it executes the program and its input simultaneously. This causes *overhead*. Nevertheless, interpreters may be a good choice for implementing languages that are either simple – so that no overhead arises – or very complicated – so that the overhead is acceptable.

Another way of implementing programming languages by software is a *compiler*. A compiler is a program written in an *implementation language* M that first translates a program p written in its *source language* L into an equivalent program p' in its *target language*, let it equal its implementation language M. An L-program is then processed in two stages:

- First, at *compile time*, the compiler is executed on machine M, translating p to p'.
- Second, at *run time*, the translated program p' is executed on M.

The processing of an L-program is drawn in Figure 1.4. A compiler is harder to write than an interpreter, because a complete program p' has to be constructed that captures the entire semantics of p. However, this way of processing has the advantage over interpreters that the program p is processed alone, and only once; afterwards the translated program p' can be executed many times, for different inputs, without the overhead occurring in interpreters. Typically, the languages related to a compiler have the following properties:

- Its source language S is a higher programming language
- Its target language T is a low-level language like binary code, assembler or C.
- Its implementation language M is a low-level language.

Some special compilers are the following

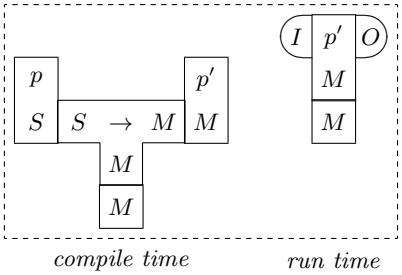


Fig. 1.4. Processing a program with a compiler

- A *native* compiler is a compiler producing code for the machine on which it runs.
- In a *cross-compiler*, the target language M and the implementation language M' are different machine languages.
- An *assembler* is a native compiler for a low-level source language A.
- In a *source-to-source compiler*, not only the source language S is a high-level programming language, and the target language S' as well. If S' is "higher" than S, such a compiler may be used to re-engineer software by migrating it to a language that is "more modern", and easier to maintain.

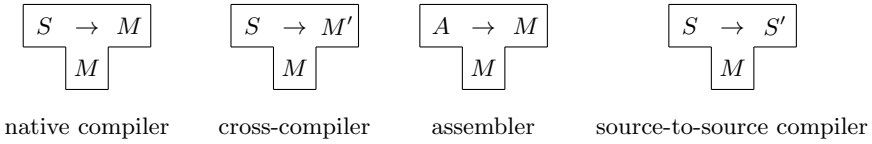


Fig. 1.5. Special kinds of compilers

1.2 Re-Targeting

Implementing a programming language for just one machine is not sufficient. It should be feasible to move a compiler to other machines. This process is called *re-targeting*.

In general, all programming languages should run on every machine on the market. This causes much implementation effort: For implementing k languages on n machines, we need $k \cdot n$ compilers altogether. A designer of a new language has to provide n new compilers, one for every machine. A producer of a new machine needs k compilers. Such an effort is inacceptable in practice. Fortunately, compiler writers have come up with ideas to reduce this effort.

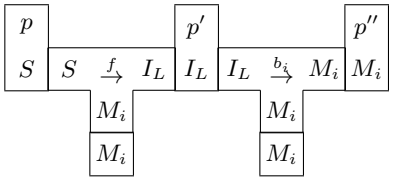


Fig. 1.6. Compiler frontend and backend

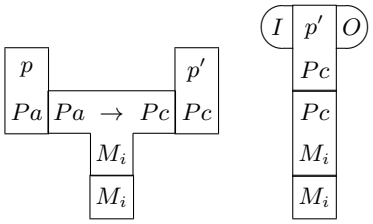


Fig. 1.7. The portable Pascal compiler

A very general attempt has already been started in the mid-fifties: a project in the US aimed at designing a *universal intermediate language for compilers*, called UNCOL. With such a language, k compilers for all languages to UNCOL, plus n compilers from UNCOL to all machines suffice to make all programming languages universally available. However, the project failed to design this language so that all compilers to and from UNCOL are *efficient*.

Later, a less ambitious attempt (illustrated in Figure 1.6) turned out to be more successful. The idea is to design an intermediate language I_L dedicated to some source language L, and two divide a compiler into two parts:

- A *frontend* f translates the source language L into I_L.
- Several *backends* b_i translate I_L to all the machines M_i (from $1 \leqslant i \leqslant n$).

I_L must allow efficient compilation from the source language L, efficient compilation to all machines M_i, and should be as primitive as possible, quasi an *abstract machine language*. Dedicated intermediate languages reduce the effort for implementing a new language on all machines to at least $\frac{1}{2} + \frac{n}{2}$; the ratio may be even $\frac{2}{3} + \frac{n}{3}$ or $\frac{3}{4} + \frac{n}{4}$ if the backend is easier to implement than the frontend.

It is feasible to design dedicated intermediate languages meeting these requirements. Actually, the backend can be implemented by an interpreter, which still reduces the effort for the backends.

Example 1.1 (The Portable Zurich Pascal Compiler). The portable Pascal compiler by Urs Ammann consists of a compiler from Pascal to *P-code*, which is a platform-independent assembly language designed for translating Pascal programs easily, and an interpreter for P-code [5].

On a machine M, the execution of a Pascal program p consists of a translation phase and an interpretation phase, as shown in Figure 1.7.

Such an architecture has also been used for Java, with Java Byte Code (JBC) as the intermediate language. The .Net architecture is even more ambitious: it is used for different source languages, like C#, Eiffel, and F#.

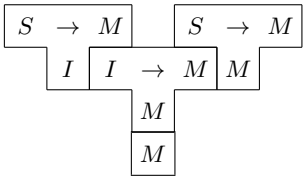


Fig. 1.8. Compiling a compiler

1.3 Implementation

What a language should be used to implement a compiler? A compiler is a big, modular program, managing complex data (trees, graphs, tables), using recursion intensively. And, it shall be portable to different machines. So, compilers are nowadays developed in a high-level programming language I, and have to be translated themselves to be executable, as shown in Figure 1.8. That is, the implementation relies on the existence of a compiler for its implementation language I.

As a consequence, the compiler can be maintained only as long as a compiler for I is still available. How can such a commitment to another high-level language be avoided?

1.3.1 Bootstrapping for Independence

The answer – somewhat surprising – is to develop the compiler in its own source language S. How can this work? We do it in a process called *bootstrapping*, where we need to start with two compilers, see Figure 1.9:

- A *master compiler* m is written in its source language S. It is developed with great care so that it executes fast and generates code of good quality. It is often expressed only in a sub-language S' of the source language S. (E.g, if S supports concurrency, this will most likely not be used in m.)
- A *one-way compiler* o is developed for the machine M as simply as possible. It may execute inefficiently, may generate inefficient code, and be restricted to the sub-language S' used in the master compiler m.

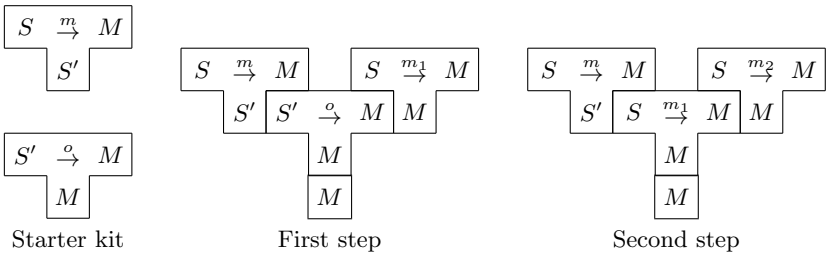


Fig. 1.9. Bootstrapping for independence

In a first step, the master compiler m is ported onto machine M by compiling it with the one-way compiler o. The resulting compiler m_1 is, however, inefficient as it has been compiled with o; the code produced with m_1 is as good as that of m. In the second step, the translated compiler m_1 is used to compile the master compiler m again. The resulting compiler m_2 is now both efficient, and produces good code.

The one-way compiler o can be developed in a high-level language I, and then be compiled. Since it is used just for the first step, this does not imply a commitment on I. Later revisions of m can always be compiled with m_1.

Actually, we could start the bootstrap by writing a compiler s for the sub-language S' in S', and derive from it the one-way compiler for S' in I almost mechanically. Later, we can incrementally extend s to the full master compiler m without ever using o again.

1.3.2 Bootstrapping for Re-Targeting

Bootstrapping can also be used to implement a compiler for another target machine. Assume the master compiler m has been bootstrapped on M so that we have the compiler m_2. Now we can re-target m for a different machine M'. We can then compile the re-targeted compiler m' on M and get a cross-compiler m'_1. The programs translated with m'_1 can be transferred to M' and be executed there.

As soon as m is stable, we can use the cross compiler m'_1 to compile it on M, getting a native compiler m'_2 for M'. See Figure 1.10.

1.3.3 Bootstrapping a Compiler-Interpreter System

Bootstrapping can also be used to re-target compilers consisting of a frontend compiler and a backend interpreter.

Example 1.2 (Portable Zurich Pascal Implementation). The portable Zurich Pascal compiler mentioned in Example 1.1 has been developed in its own source language Pascal. It was distributed with the three components shown

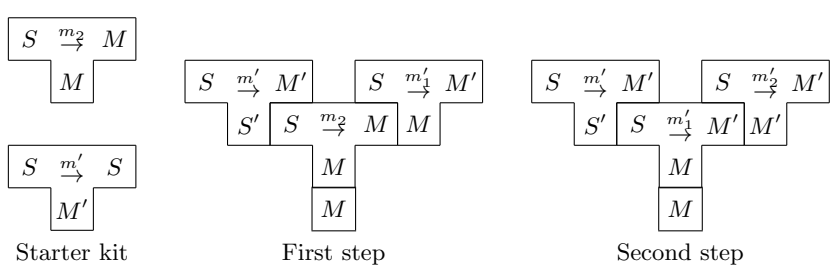


Fig. 1.10. Bootstrapping for re-targeting

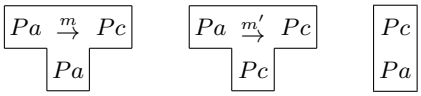


Fig. 1.11. The starter kit of the portable Zürich Pascal implementation

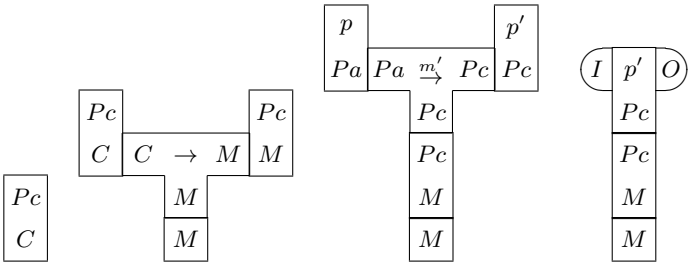


Fig. 1.12. Producing an interpreted Pascal compiler

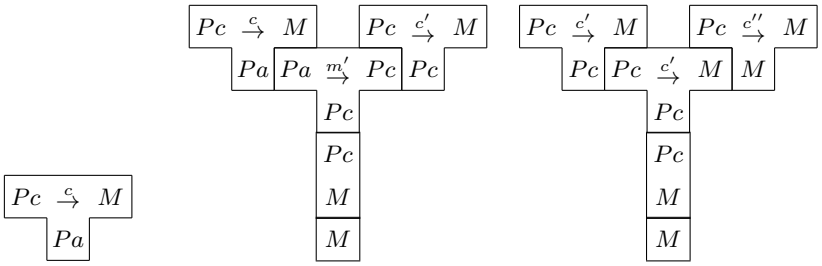


Fig. 1.13. Generating a native P-code compiler

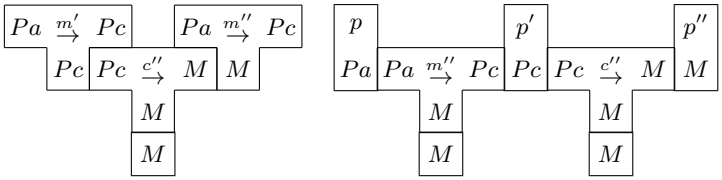


Fig. 1.14. Bootstrapping the PASCAL compiler

in Figure 1.11: the master compiler m, its translation into P-code m', and the P-Code interpreter in Pascal.

In order to implement the system on a machine M, we rewrite the P-code interpreter in a language available on M (like C) and compile it on M. This interpreter can be combined with m' to compile Pascal programs on M, and can be used to execute them on M. Compilation will be slow as the compiler is interpreted. (The master compiler m is not used here.)

To make the compiler more efficient, we write a compiler c of P-code to machine code of M in Pascal, and compile it with the interpreted compiler on M. The resulting compiler c' is "semi-native", wrt. its target language. Now

compile the P-code compiler c' with itself, and get a native P-code compiler c''. (See Figure 1.13.)

The compiler c'' is used to compile the Pascal compiler m' into a native Pascal compiler m''. The result in a native two-step compiler $(m''; c'')$ on M. (See Figure 1.14.)

1.4 Structure of Compilers

We now take a somewhat closer look at compilers. In the long history of compiling programming languages, certain principles for structuring them have got commonly accepted. This resulted in the phase model for compilers shown in Figure 1.15. A *phase* is a part of a compiler devoted to a particular subtask.

A compiler is divided into two major phases:

- The *analysis* determines the structure of the source program, and checks whether it is well-defined.
- The *synthesis* is concerned with the construction of an equivalent program in the target language.

The analysis is divided into three sub-phases:

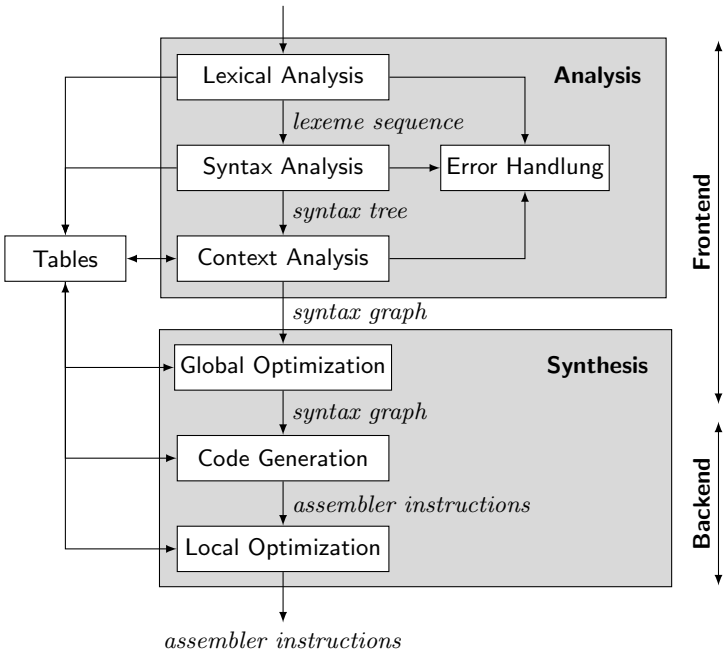


Fig. 1.15. The phase model of compilation

- The *lexical analysis* reads the character string of the source program, groups it into a sequence of lexemes such as operators, identifiers, delimiters, comments etc.
- The *syntax analysis* parses the lexeme sequence according to the hierarchical structure, and delivers a syntax tree, consisting of declarations, statements, expressions etc.
- The *contextual analysis* inspects properties beyond hierarchical structure: it identifies the declaration valid for its names, and checks the types of expressions etc. The syntax tree is extended by cross references between uses and declarations of names, expressions and their type (declaration) etc.

Whereas the structure of analysis is very much standard, the sub-phases of synthesis may show more variety:

- The kernel is *code generation*, mapping the data of the source program into storage cells, variables onto addresses, control structures to jumps, and operations to instructions of the target language. In most cases, the output is a sequence of instructions in an assembler-like language.
- A sub-phase that is optimistically called *global optimization* attempts to remove redundancies in order to improve the structure of the syntax graph, using full knowledge of the source program. It may transform the syntax graph into a representation dedicated to its purpose, like data flow graphs or control flow graphs.
- *Local optimization* (another an optimistic name) may improve the efficiency of the assembler program emitted by code generation, using knowledge about the target language. A typical technique is *peephole optimization* where a "looking glass" is moved over the instructions to replace short subsequences by more others that are more efficient.

A slightly different division of a compiler is motivated by the problem of re-hosting and re-targeting discussed earlier:

- The *frontend* contains all components that are independent of the target machine; this includes analysis and global optimization (if present). This phase may also include the generation of a (platform-independent) program representation.
- The *backend* comprises only the platform-dependent sub-phases, like code generation and local optimization (if present).

Another structuring notion for compilers is a *pass*. It originated in the old days of compiling main storage was too precious to keep the program in it permanently; frequently, the results of phases were written to the file system in intermediate representations – also called *intermediate languages*. At that time, almost every phase run as a separate pass, which then performed a complete inspection of the program from left to right (start to end). Nowadays, the notion of passes is still used to express the complexity of a language wrt.

compilation: languages like Pascal and Ada have been designed so that a single pass – inspecting the program once – suffices to compile it, whereas many modern languages like Java Haskell need two passes, one for collecting information about declarations, and another one to use it for context analysis and code generation.

So nowadays, the intermediate languages need no longer be textual: They can be sequences (lists), trees, graphs (object structures).

Even if the phases of a compiler need no longer process the intermediate representations sequentially from left to right, they can still be thought of as executing one after the other, reading the representation of the preceding phase, and writing a representation to the following phase. Practically, their execution may be interleaved, like in a UNIX pipe; this is often done with lexical and syntax analysis, and with code generation and peephole optimization. An important principle is that the data flow in a compiler should be strictly forward: no phase should need informations computed by later phases. (In some old-fashioned languages like PL/1, lexical analysis needed information from context analysis, causing backtracking, which is very complicated and inefficient.)

Besides the intermediate representations, compilers often store informations in tables and pass them forward. Particularly for the informations associated with identifiers, like textual representation, declaration, or storage address, special care is taken so that they can be accessed efficiently.

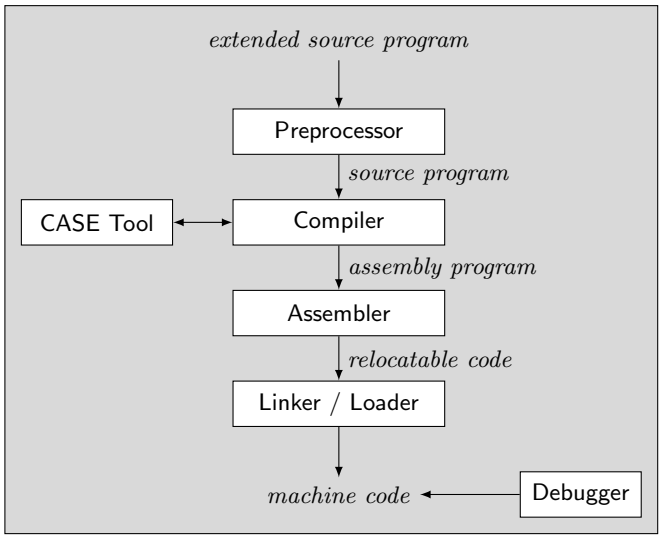


Fig. 1.16. The environment of a compiler

1.4.1 The Context of a Compiler

Every compiler interacts with the computer hardware via the operating system, i.e., with its platform. Furthermore, it is often embedded into a CASE tool, and uses auxiliary systems like preprocessors (notorious for C and C++), assemblers to produce relocatable code, the linker and loader that produces machine code, and with a debugger. See Figure 1.16.

Bibliographical Notes

For playing "compiler tetris", we follow [20, Chapter 2]. The structure of compilers is described in a similar way as in [3, Section 1.2].

Excercises

Solutions to some of these exercises can be found on page 147 *ff*.

Exercise 1.1. A property P of a programming language is called

- *static* if it can be checked already while the program is compiled, and
- *dynamic* if it can, in general, only be checked at the time when the compiled program is executed.

Please consider the programming languages C, JAVA, and Haskell (or other languages that you know). Find out which of the following properties is static, and dynamic in these languages, respectively:

1. The *binding* of names (identifiers) to the declarations introducing them.
2. The type of a variable.
3. The bounds of an array variable.
4. The value of a variable.

Please compare the languages wrt. your findings.

Exercise 1.2. Why did the UNCOL project (mentioned in Section 1.2) fail in designing a universal intermediate language for compilers that is *efficient* for all languages and machines?

Exercise 1.3. The language JAVA is implemented by a compiler and an interpreter:

- The compiler transforms Java programs into *Java Byte Code* (JBC).
- JBC-programs are executed by an interpreter, known as the *Java Virtual Machine* (JVM).

Portable implementations of Java – like the *Java Developer Kit* by SUN systems are based on the following components:

(a) A master compiler from Java to JBC, written in Java itself.
(b) A compiler from Java to JBC, written JBC, which is derived from the master compiler. (You get this compiler for free as soon as you are able to execute Java on some platform.)
(c) A *Java Virtual Machine* (JVM), an interpreter for JBC, written in Java.

The master compiler can be assumed to use only a small portion of the JVM library.

Tasks

1. *Port to machine M.* How can Java be ported to a new machine M, given the components above? Do this by implementing a component that is as simple and small as possible.
   You may assume that a native C-compiler, producing M machine code, and "written" in M machine code, exists for M. The resulting implementation need not run efficiently – it may involve an interpreter.
2. *Bootstrap on machine M.* How can the compiler be made native? Again, do this by implementing a small component.

2

Lexical Analysis

The lexical analyzer shall divide the source program, given as a character string that is usually read from a file, into a sequences of lexemes. "Lexeme" is a notion from linguistic, for the lexical atoms of a language. In natural languages, these are the words of a sentence.

Like many other phases of a compiler, the lexical analysis consists of two sub-phases:

1. The *scanner* analyzes the source program.
2. The *screener* translates lexemes into a representation used in subsequent phases.

The scanner can be considered as the analysis phase, and the screener as the synthesis phase of the lexical analyzer.

In this chapter, we first discuss character sets and lexemes, introduce regular expressions for describing lexemes, and define how lexemes can be recognized by (deterministic) finite automata. Finally we discuss several aspects of screening, in particular the representation of identifiers, and other practical implementation details, before we describe the generation of lexical analyzers with the Unix tool `lex`.

2.1 Characters

The character string forming the source program is built over a character set like UNICODE. For our examples, ASCII is sufficient, and abstractly, we just assume that there is some vocabulary V, as defined in Section A.2.1.

Usually, the lexical analyzer distinguishes the following kinds of characters, see Figure 2.1:

- *Letters*, sometimes divided into upper-case and lower-case.
- *Digits*, mostly the decimals ones from `0` to `9`.
- *Special characters*, including punctuation like `;`, brackets like (, and operators like `+`.

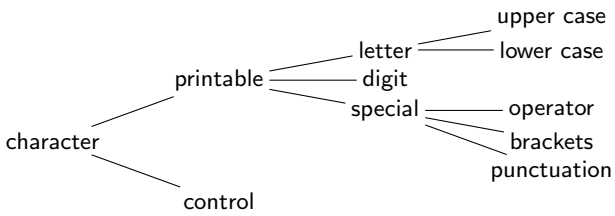


Fig. 2.1. Classification of characters

- "Not printing" *control characters* like `ESC` and `DEL`.

2.2 Lexemes

Lexemes are the basic building blocks of a program text. Other terms are *symbol* and *token*. They are atomic in the sense that they are important for the semantics of the language, but only as a whole. An identifier like `Pete` is a lexeme, because it is always considered as a whole – it is not necessary to partition it into its letters `p`, `e`, `t`, and `e` in order to define the semantics. In contrast to that, a qualified name like `Myclass.hugo` is not a lexeme, because its must be partitioned into its components – `Myclass` and `hugo` – in order to define its semantics. However, a string like `13.4E-20` is considered as a lexeme (for a floating point literal), although it has components that are lexemes: the fixed point literal `13.4` and the integer literal `20`. Because, once a floating point number has been recognized, it need not be partitioned again.

2.2.1 Typical Kinds of Lexemes

Which kind of lexemes do typically occur in a programming language?

- *Identifiers* like `Pete` can be used as names for entities in a program, such as types, classes, variables, functions, methods etc.
- *Operators* like `=`, `<=`, and **or** name *infix* function that are written between their arguments.
- *Delimiters* like `;`, `(`, and **begin** structure the program text for the reader, in particular the compiler, similar to the punctuation marks in ordinary text.
- *Literals* denote explicit ("manifest") values of built-in types like character strings (`"hallo"`) or numbers (`13.4E-20`).
- *Keyword* are identifiers that have a fixed, unalterable meaning in a program text. They can name types like, **boolean** and **int**, operators like **or**, or delimiters like **begin**. In modern languages they are *reserved* for this purpose, and cannot be used by the programmer to denote other entities. (If they are *not* reserved, this may cause severe difficulties, see Section 2.6.1.)

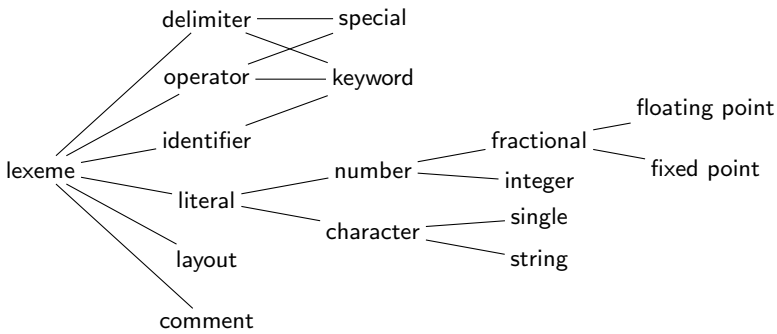


Fig. 2.2. Typical lexemes occurring in programming languages

- *Layout* consisting of blanks, tabulators, newlines serves to "format" the program text, and may also serve as a delimiter between lexemes, as in **int** `n`. In most languages, one layout character is as good as many. However, in languages like Python and Haskell, the indentation of text, i.e., the number of blanks after a newline, is used to specify syntactic structure, in order to avoid delimiters like {, }, and `;`.
- *Comments* shall document the program, but are irrelevant for its semantics.

The border between operators and delimiters is not sharp: In `Myclass.hugo`, the period delimits the identifiers, but also denotes an operation, namely the access to the feature of a class. See Figure 2.2 for an overview of lexemes.

2.3 Regular Expressions and Definitions

Regular expressions and regular definitions are used to describe the form of lexemes. They are equivalent wrt. descriptive power to the *regular grammars* known from formal language theory (see Section A.2.3) but are more convenient to use.

Definition 2.1 (Regular Expression). Let V be a vocabulary, and $M_{\mathcal{R}} = \{\varepsilon, |, ^?, ^+, ^*, (,)\}$ be the *meta-symbols* for regular expression, disjoint with V.

Regular expressions are the least language $\mathcal{R}$ over the vocabulary $V \uplus M_{\mathcal{R}}$, which can be constructed with the following rules:

1. A symbol v is a regular expression, for every $v \in V$.
2. The meta-symbol ε is a regular expression, denoting the *empty word*.
3. If $R_1, R_2, \ldots, R_n$ are regular expressions, so are
   a) the *alternative* $R_1 \mid R_2 \mid \cdots \mid R_n$ and
   b) the *concatenation* $R_1\, R_2 \cdots R_n$.
4. If R is a regular expression, so is
   a) the *option* $R^?$ ("maybe"),
   b) the *non-empty iteration* R^+ ("many"),

c) the *iteration* R^* ("any"), and
d) the *grouping* (R).

The *language* generated by a regular expression is defined recursively over their structure:

1. $\mathcal{L}(v) = \{v\}$ for all $v \in V$.
2. $\mathcal{L}(\varepsilon) = \{\varepsilon\}$.
3. For all regular expressions $R_1, R_2, \ldots, R_n$ holds:
   a) $\mathcal{L}(R_1 \mid R_2 \mid \cdots \mid R_n) = \bigcup_{i=1}^{n} \mathcal{L}(R_i)$.
   b) $\mathcal{L}(R_1\, R_2 \cdots R_n) = \mathcal{L}(R_1) \cdot \cdots \cdot \mathcal{L}(R_n)$.
4. For a regular expression R holds:
   a) $\mathcal{L}(R^?) = \mathcal{L}(R) \cup \{\varepsilon\}$,
   b) $\mathcal{L}(R^+) = \mathcal{L}(R)^+$
   c) $\mathcal{L}(R^*) = \mathcal{L}(R)^*$ und
   d) $\mathcal{L}((R)) = \mathcal{L}(R)$. [1]

We call two regular expressions $R, Q \in \mathcal{R}$ *equivalent*, and write $R \equiv Q$, if they generate the same language: $\mathcal{L}(R) = \mathcal{L}(Q)$.

Lemma 2.1. *The following equivalences hold for arbitrary regular expressions $Q, R, S \in \mathcal{R}$:*

$R \mid R \equiv R$		*idempotence*
	$Q \mid R \equiv R \mid Q$	*commutativity*
$(Q \mid R) \mid S \equiv Q \mid (R \mid S)$		*associativity*
$(Q\, R)\, S \equiv Q\, (R\, S)$		*associativity*
$(Q \mid R)\, S \equiv Q\, S \mid R\, S$	$Q\, R \mid Q\, S \equiv Q\, (R \mid S)$	*distributivity*
$\varepsilon\, R \equiv R$	$R\, \varepsilon \equiv R$	*neutral elemenet*
$R^? \equiv R \mid \varepsilon$		*option*
$R^+ \equiv R \mid R\, R^+$		*non-empty iteration*
$R^* \equiv \varepsilon \mid R\, R^*$		*iteration*

Proof. By straight-forward application of the definitions. □

For practical use of regular expressions, we allow to bind them to variable names, and use these names in other regular expressions.

Definition 2.2 (Regular Definition). Let X be a set of variable names that are disjoint to V and to the meta-symbols $M_{\mathcal{R}}$.

We extend regular expressions by allowing the one additional construction rule, as follows:

2a. A variable x is a regular expression, for every $x \in X$.

[1] In these definitions, $L \cdot L'$ denotes the concatenation of languages, and L^+ and L^* denote their transitive and transitive-reflexive closure, respectively. See Section A.2.2.

A *regular definition* has the form

$$\begin{aligned} x_1 &\triangleq R_1 \\ &\vdots \\ x_{n-1} &\triangleq R_{n-1} \\ x_n &\triangleq R_n \end{aligned}$$

where, for $1 \leqslant i \leqslant n$, $x_i \in X$ and $R_i \in \mathcal{R}$ so that R_i contains only variables x_j with $j < i$.

Let $R[x/Q]$ denote the *substitution* of a variable $x \in X$ by the regular expression (Q), at all occurrences of x in R. Then, it holds:

$$R_i \equiv R[x_{i-1}/R_{i-1}] \ldots [x_1/R_1] \qquad (1 \leqslant i \leqslant n)$$

We use further abbreviations in regular expressions.

Definition 2.3. A regular expression R, and a variable x with the definition $x \triangleq R$ define a *character class* if $\mathcal{L}(R) \subseteq V$. We extend regular expressions by another rule:

5. If R is a character class definition (and $x \triangleq R$ for a variable $x \in X$) then
   a) $\widetilde{R} \in \mathcal{R}$, and
   b) $\widetilde{x} \in \mathcal{R}$.

The *language* for these forms of expressions, which are called *complements*, is defined as follows:

5. For a character class definition R,
   a) $\mathcal{L}(\widetilde{R}) = V \setminus \mathcal{L}(R)$.
   b) $\mathcal{L}(\widetilde{x}) = V \setminus \mathcal{L}(R)$ if $x \triangleq R$.

It is a good idea to name all character classes that are used in the definition of a lexeme.

Symbols and Meta-Symbols

In regular expressions and definitions, the vocabulary V, the meta-symbols $M_{\mathcal{R}}$, and the variables X must be distinguishable from one another.

In a regular expression "(a)*", a symbol like "*" oder "(" could be a meta-symbol, or just a symbol in V; a word "digit" could be in V^5, or denote a variable from X. In the scanner generator `lex`, all symbols $v \in V$ may be written as `"`v`"`, and meta-symbols can "*escape*" their usual meaning by preceeding it with a *backslash*. Thus the meta-symbol `|` can be distinguished from the symbol `"|"` (or `\|`) in V. The escape symbol itself can escape its usual meaning in the same way: `\\` denotes a backslash in V.

Like in C, C++, and Java, `lex` allows to denote control characters by certain letters following a backslash: "`\n`" denotes newline, "`\t`" the horizontal

tabulator etc. In `lex`, the regular expressions `^`R and R`$` specify more abstractly that the regular expression R should appear at the beginning and at the end of a line, respectively.

In our examples of regular expressions and definitions, we shall underline symbols from V, and enclose variable names in angle brackets. Then "$\underline{\text{data}}$", "$\underline{*}$",and "$\underline{(}$" are in V^*, whereas "$\langle\text{data}\rangle$" is from X, and $*$ and (are from $M_{\mathcal{R}}$.

Example 2.1 (Integer Literals and Identifiers). Integer literals and simple identifiers can be defined by the regular definitions:

$$
\begin{aligned}
\langle\text{digit}\rangle &\triangleq \underline{0}|\underline{1}|\underline{2}|\underline{3}|\underline{4}|\underline{5}|\underline{6}|\underline{7}|\underline{8}|\underline{9}\\
\langle\text{letter}\rangle &\triangleq \underline{a}|\underline{b}|\underline{c}|\underline{d}|\underline{e}|\underline{f}|\underline{g}|\underline{h}|\underline{i}|\underline{j}|\underline{k}|\underline{l}|\underline{m}|\underline{n}|\underline{o}|\underline{p}|\underline{q}|\underline{r}|\underline{s}|\underline{t}|\underline{u}|\underline{v}|\underline{w}|\underline{x}|\underline{y}|\underline{z}\\
&\quad\ |\ \underline{A}|\underline{B}|\underline{C}|\underline{D}|\underline{E}|\underline{F}|\underline{G}|\underline{H}|\underline{I}|\underline{J}|\underline{K}|\underline{L}|\underline{M}|\underline{N}|\underline{O}|\underline{P}|\underline{Q}|\underline{R}|\underline{S}|\underline{T}|\underline{U}|\underline{V}|\underline{W}|\underline{X}|\underline{Y}|\underline{Z}\\
\langle\text{letgit}\rangle &\triangleq \langle\text{letter}\rangle \mid \langle\text{digit}\rangle\\
\langle\text{integer}\rangle &\triangleq \langle\text{digit}\rangle^+\\
\langle\text{identifier}\rangle &= \langle\text{letter}\rangle\ \langle\text{letgit}\rangle^*
\end{aligned}
$$

Variables like $\langle\text{digit}\rangle$ and $\langle\text{number}\rangle$ may not be used recursively. Such definitions would allow to generate context-free languages!

2.4 Finite Automata

Automata are abstract devices that recognize (formal) languages. For recognizing regular languages, as they are generated with regular expressions, we need automata that have a finite number of states (and no internal storage).

Definition 2.4 (Finite Automaton). A finite automaton (FA, for short) $A = (V, Q, \Delta, q_0, F)$ has components as follows:

- V is a *vocabulary*.
- Q is a finite set of *states*.
- $\Delta \subseteq Q \times (V \cup \{\varepsilon\}) \times Q$ is a (finite) set of *state transitions*.
- $q_0 \in Q$ is the distinguished *start state*.
- $F \subseteq Q$ is a set of *final states*.

Direct transitions of a finite automaton are defined as the least relation $\vdash_\Delta \subseteq (Q, V^*) \times (Q, V^*)$ obtained by the following rule:

- For a symbol $a \in V$, a word $w \in V^*$, and a transition $(q, a, q') \in \Delta$, $(q, aw) \vdash_\Delta (q', w)$. Equivalently, this can be specified with an inference rule:

$$\frac{a \in V, w \in V^*, (q, a, q') \in \Delta}{(q, aw) \vdash_\Delta (q', w)}$$

As usual, $\vdash^*_\Delta$ denotes the transitive-reflexive closure of this relation, which is called the *transition* relation of A.

The *language* accepted by A is given as the words read while performing transitions from the start state to a final state:

$$\mathcal{L}(A) = \{w \in V^* \mid (q_0, w) \vdash^*_\Delta (q, \varepsilon), q \in F\}$$

A finite automaton is *deterministic* (short: "a DFA") if the following conditions hold:

1. There is no state transition under the empty word ε.
2. Different state transitions from a state q are under different symbols: if (q, v, q') and $(q, v', q'') \in \Delta$, $v = v'$ implies $q' = q''$.

Otherwise, the automaton is *non-deterministic* (short: "an NFA").

A finite automaton reads words (from left to right). ...

A finite automaton can be represented as a directed graph.

Definition 2.5 (Transition Graph of a Finite Automaton). A transition graph is a directed graph, with edges labeled by $V \cup \{\varepsilon\}$.

Let $A = (V, Q, \Delta, q_0, F)$ be a finite automaton. The *transition graph* $T(A)$ of A is defined as follows:

1. The states Q are the nodes of $T(A)$.
2. Every transition $(q, a, q') \in \Delta$ is an edge from q to q' labeled with a.
3. An arrow points to the node q_0 representing the start state.
4. Final states $q \in F$ are drawn with double lines.

For convenience, we inscribe numbers to the nodes of a transition graph, and allow a set $\{(q, a_1, q'), \dots (q, a_n, q')\} \subseteq \Delta$ of transitions to be represented by a single edge labeled with $a_1, \dots, a_n$, or with the variable x if x defines the character class $\{a_1, \dots, a_n\}$.

We write "$q \rightarrow_a q'$" if there is a transition labeled with $a \in V \cup \{\varepsilon\}$ from q to q', and extend this notion to paths: we write $q \rightarrow^*_\varepsilon q$ for empty paths, and $q \rightarrow^*_{vw} q'$ if there is an intermediate state q'' so that $q \rightarrow_v q'' \rightarrow^*_w q'$.

A word $w \in V^*$ is accepted by the automaton is $q_0 \rightarrow^*_w q$ for some final state $q \in F$.

Notation. In the transition graphs of finite automata, we allow that an edge is labeled with a character class, in order to avoid sets of parallel edges.

Intuitively, the transition graph can be used to determine whether w is accepted by A as follows: Try to follow a path from the start state q_0 that is labeled with w; if there is a path ending in a final state, w is accepted; otherwise, if there is a path ending in an intermediate state, w is the prefix of some word accepted by A; if there is no such path, only a (possibly empty) prefix of w is accepted by A.

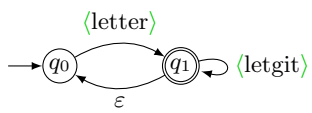


Fig. 2.3. The transition graph of the finite automaton A_0

Example 2.2 (A Finite Automaton for Identifiers). Consider the finite automaton $A_1 = (V, Q_1, \Delta_1, q_0, F)$, where

- the vocabulary V contains letters and digits,
- The set of states is $Q1 = \{q_0, q_1\}$,
- the set of state transitions is

$$\Delta_1 = \{(q_0, \langle\text{letter}\rangle, q_1), (q_1, \langle\text{letgit}\rangle, q_1), (q_1, \varepsilon, q_0)\}$$

- q_0 is the start state,
- $F = \{q_1\}$ is the singleton set of final states.

A_0 is nondeterministic, due to the third transition. There is a sequence of direct transitions

$$(q_0, \underline{\text{Xy1}}) \vdash_{\Delta_0} (q_1, \underline{\text{y1}}) \vdash_{\Delta_0} (q_0, \underline{\text{y1}}) \vdash_{\Delta_0} (q_1, \underline{1}) \vdash_{\Delta_0} (q_0, \varepsilon)$$

Thus $\underline{\text{Xy1}} \in \mathcal{L}(A_0)$. Note that the transition sequence

$$(q_0, \underline{\text{Xy1}}) \vdash_{\Delta_0} (q_1, \underline{\text{y1}}) \vdash_{\Delta_0} (q_1, \underline{1}) \vdash_{\Delta_0} (q_0, \underline{1})$$

gets stuck in a non-final state although the word is accepted (via the transitions discussed earlier).

The transition graph $T(A_0)$ of A_0 is shown in Figure 2.3. $T(A)$ contains paths

$$q_0 \xrightarrow{\underline{\text{X}}} q_1 \quad q_0 \xrightarrow{\varepsilon} q_0 \quad q_0 \xrightarrow{\overset{+}{\underline{\text{X}}}} q_0 \quad q_0 \xrightarrow{\overset{+}{\underline{\text{Xy1}}}} q_1 \quad q_0 \xrightarrow{\overset{+}{\underline{\text{Xy1}}}} q_0$$

It is easy to see that A_0 accepts the language $\mathcal{L}(\langle\text{identifier}\rangle)$. This would also be the case if the third (ε) transition would be removed. Then the automaton would be deterministic.

2.5 Transforming Regular Expressions into DFAs

Finite automata accept just the languages that can be generated with regular expressions (or with regular grammars). Moreover, every regular expression R can be systematically transformed into a deterministic finite automata A_R that is equivalent, i.e. $\mathcal{L}(R) = \mathcal{L}(A_R)$. There are (at least) two ways to define the transformation, see Figure 2.4.

The first one is described in every textbook and has three steps:

1. Apply graph transformation rules to construct the equivalent NFA of a regular expression.
2. Make the NFA deterministic, by the power set construction.
3. Minimize the resulting DFA, by removing unreachable states and by identifying equivalent states.

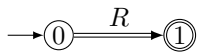
This construction is explained in Section 2.5.1. Alternatively, regular expressions can be directly transformed into a DFA, where every state represents a regular sub-expression that is transformed into quasi-regular form in order to determine the transitions. It is described in Section 2.5.4.

2.5.1 Transforming Regular Expressions into NFAs

Every regular expression can be transformed into an equivalent (non-deterministic) finite automaton. We specify it by graph transformation rules on *abstract* transition graphs that are extended in so far as their edges may be labeled with arbitrary regular expressions, not just with symbols or the empty word. An edge labeled with a regular expression R is called *abstract*, and drawn with double lines. It shall represent a sub-automaton (still to be defined) that accepts R. The transformation rules gradually refine abstract edges until all edges are labeled with symbols or the empty word – if this is the case for all edges, the automaton is finite. (See [21, Abschnitt 7.2.2])

Definition 2.6 (NFA of Regular Expression). Figure 2.5 shows graph transformation rules NFA on abstract transition graphs. They are applied by matching an abstract edge of one of the left-hand sides in a source graph G, removing the edge, and gluing nodes s and t on the right-hand side of this rules with the corresponding nodes of the match, producing a target graph H.

The NFA transition graph $A(R)$ of a regular expression R is obtained by applying the transformation rules to the start graph

$S =$ 

as long as possible.

It is easy to see that NFA generates finite automata.

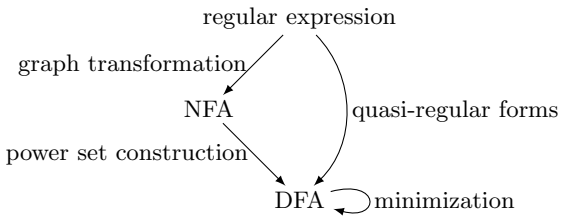


Fig. 2.4. Generating deterministic finite automata for regular expressions

Fact 1 *1. Whenever*

$$\rightarrow (0) \xRightarrow{R} (1) \Rightarrow^*_{NFA} G \text{ so that } G \not\Rightarrow_{NFA} G'$$

G is a finite automaton.
2. G is equivalent to R, i.e., it accepts the language generated by R.
3. G is nondeterministic in general.

Proof. 1. Figure 2.5 contains a left-hand side for every form of regular expression, so that every abstract edge can be transformed. Since the abstract edges on the right-hand side of the rules are proper regular sub-expression of that on the left-hand side, this process is bound to terminate. The resulting graph G does not contain an abstract edge, and is thus (the transition graph of) a finite automaton.
2. By inspection of the rules it is clear that the labels of every path from s to t on the right-hand side spell a regular expression that is equivalent to that of the abstract edge on the left-hand side. Thus every graph generated from S is equivalent to the regular expression on S.
3. Obvious, as the transformation rules for sequences and options introduce ε-transitions, and the transformation of a regular expression "$a\,R \mid a\,Q$" generates an automaton with ambiguous transitions under the symbol a. □

Example 2.3 (NFA for Integer Literal and Identifier). Literals of integer literals and identifiers are defined in Example 2.1.

With the rules NFA in Figure 2.5 we generate the following automata:

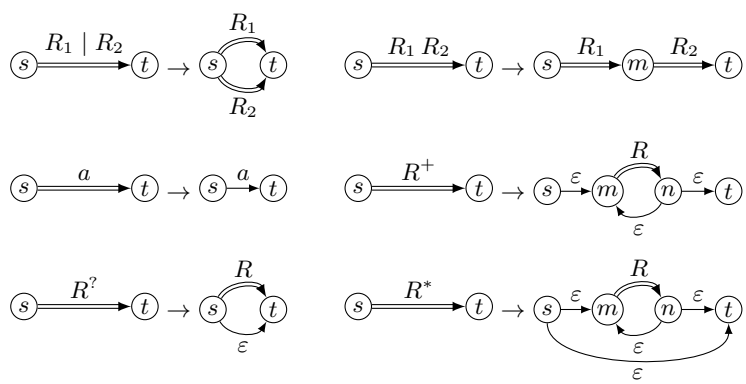
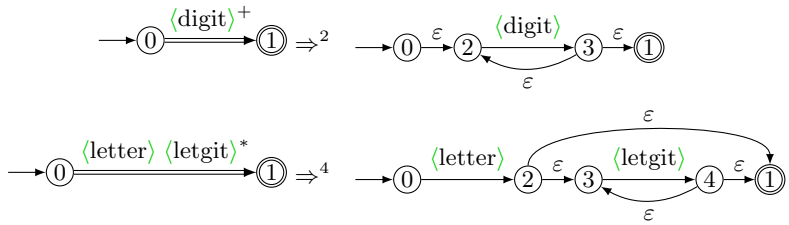


Fig. 2.5. Rules transforming regular expressions into NFAs. (See [21, Sect. 7.2.2])



2.5.2 Transformaing NFAs into DFAs

For every NFA, an equivalent DFA can be obtained by the *power-set construction*. All states reachable from the a start state q_0 with ε transitions are considered to be equivalent to q_0, and form a power-set state P_0. If some state $q \in P_0$ has a transition $q \rightarrow_a q'$, the non-empty set of follower states of P_0 that can be reached from some of its states via paths labeled with a are considered to be equivalent successor states $S(P_0, a)$, and form a power-set state with a transition $P_0 \rightarrow_a S(P_0, a)$. The construction continues for all follower states until no more power-set states are constructed.

Algorithm 2.1 (Power-Set Construction). *Input*: A (non-deterministic) finite automaton $A = (V, Q, \Delta, q_0, F)$.
Output: The *power-set automaton* $P(A) = (V, \mathcal{Q}, \Delta_{\mathcal{Q}}, P_0, F_{\mathcal{Q}})$, where every power-set state $P \in \mathcal{Q}$ is a non-empty subset of Q.
Construction: The start state is the power-set state $P_0 = \{q' \in Q \mid q_0 \rightarrow_\varepsilon^* q'\}$.
Set $i \leftarrow 0$; define $\mathcal{Q}_0 = \{P_0\}$ and $\Delta_0 = \emptyset$.
Repeat

1. For every $P \in \mathcal{Q}_i$ and every $a \in V$, construct the successor states

$$S(P, a) = \{q' \in Q \mid q \rightarrow_a^* q', q \in P\}.$$

2. Set $i \leftarrow i + 1$. Define $\mathcal{Q}_i \leftarrow \mathcal{Q}_{i-1} \cup \{S(P, a) \mid P \in Q_{i-1}, a \in V, S(P, a) \neq \emptyset\}$.
Define $\Delta_i \leftarrow \Delta_{i-1} \cup \{P \rightarrow_a S(P, a) \mid P \in Q_{i-1}, S(P, a) \in \mathcal{Q}_i\}$.

Until $\mathcal{Q}_i = \mathcal{Q}_{i-1}$.
Set $\mathcal{Q} \leftarrow Q_{i-1}$, $\Delta_Q \leftarrow \Delta_{i-1}$, and $F_{\mathcal{Q}} = \{P \in \mathcal{Q} \mid P \cap F \neq \emptyset\}$.

Theorem 2.1. *Algorithm 2.1 terminates, and is correct.*

Proof. Since the finite set Q of states has only finitely many different power sets, the repetition will terminate.

For correctness, we have to show that $P(A)$ is deterministic, and equivalent to $P(A)$.

It is easy to see that the construction preserves the invariant

$$P \rightarrow_a P' \text{ iff there are } q \in P, q' \in P' \text{ so that } q \rightarrow_a^* q'$$

This equivalence implies

$P \rightarrow_w^* P'$ iff there are $q \in P, q' \in P'$ so that $q \rightarrow_w^* q'$

This holds for the start state P_0 and some final state $P \in F_Q$ in particular so that A and $P(A)$ recognize the same language.

Since the construction does not insert ε-transitions, and constructs successor states under distinct symbols, $P(A)$ is deterministic. □

Example 2.4 (DFA for Integer Literal and Identifier). We compute the following power-set states for the NFA for integer literals in Example 2.3:

1. $P_0 = E(0) = \{0, 2\}$.
2. $P_1 = F(P_0, \langle\text{digit}\rangle) = \{1, 2, 3\}$.
3. $P_2 = F(P_1, \langle\text{digit}\rangle) = \{1, 2, 3\} = P_1$.

We obtain the DFA in Figure 2.6.

For the NFA for identifiers in Example 2.3 we obtain the following power-set states:

1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, \langle\text{letter}\rangle) = \{1, 2, 3\}$.
3. $P_2 = F(P_1, \langle\text{letgit}\rangle) = \{1, 3, 4\}$.
4. $F(P_2, \langle\text{letgit}\rangle) = \{1, 3, 4\} = P_2$.

Then the DFA is as in Figure 2.7.

2.5.3 Minimizing Deterministic Finite Automaton

Automata – deterministic or not – may contain states that are *useless*. This is the case if these states do not occur on paths from the start state to a final state. Such states are like "dead code" in a program; they can be removed, with their incident transitions. The resulting automaton is called *reduced*.

An automaton may also contain sets of states that are equivalent, i.e., represent the same behavior. They can be found by "guessing" an equivalence relation on nodes.

Definition 2.7 (Minimizing Finite Automata). An equivalence relation $\equiv \subseteq Q \times Q$ defines a partition of states as follows:

- $[q]^{\equiv} = \{q' \in Q \mid q' \equiv q\}$.
- $[Q]^{\equiv} = \{[q]^{\equiv} \mid q \in Q\}$.

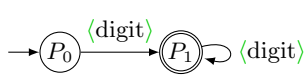


Fig. 2.6. DFA for ⟨integer⟩

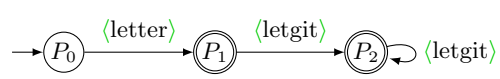


Fig. 2.7. DFA for ⟨identifier⟩

Let $\equiv_1$ and $\equiv_2$ be equivalence relation on states. Then $\equiv_1$ is *more general than* $\equiv_2$ if $q \equiv_2 q'$ implies $q \equiv_1 q'$ for all states $q, q' \in Q$, and if $|[Q]^{\equiv_1}| < |[Q]^{\equiv_2}|$.

An equivalence $\equiv$ on states defines *behavioral equivalence* if every two different states q_1 and q_2 with $q_1 \equiv q_2$ satisfy the following properties:

1. For every $a \in V \cup \{\varepsilon\}$, we have $q_1 \rightarrow_v q_1'$ if and only if $q_2 \rightarrow_v q_2'$ with $q_1' \equiv q_2'$.
2. $q_1 \in F$ if and only if $q_2 \in F$.

Let $A = (V, Q, \Delta, q_0, F)$ be a finite automaton. The most general behavioral equivalence for Q defines a minimal automaton

$$[A]^{\equiv} = (V, [Q]^{\equiv}, [\Delta]^{\equiv}, [q_0]^{\equiv}, [F]^{\equiv})$$

with

$$[\Delta]^{\equiv} = \{[q]^{\equiv} \rightarrow_a [q']^{\equiv} \mid q \rightarrow_a q' \in \Delta\}$$

and $[F]^{\equiv} = \{[q]^{\equiv} \mid q \in F\}$.

The algorithm for minimization can be found in [21, Abschnitt 7.2.5]. In general, the "guessing" of $\equiv$ is not easy. In examples, it is often straight-forward.

Example 2.5 (Minimal DFA for Integer Literal and Identifier). The DFAs for integer-literal in Example 2.4 and for identifier in Example 2.4 are reduced.

The DFA for integer-literal is minimal. Its states have an equivalent transition relation, but only one of the states is final.

In the DFA for identifier, the least behavioral equivalence makes $P_1 \equiv P_2$. The minimal automaton looks as in Figure 2.8.

2.5.4 Direct Construction of Deterministic Finite Automata

Regular expressions can easily be translated into deterministic finite automata after transforming regular expressions into quasi-regular form.

Definition 2.8 (Quasi-Regularity). A regular expression R is *quasi-regular* if

1. it has the form $R = R_1 \mid \cdots \mid R_n$ (with $n \geqslant 1$), and
2. if, for $1 \leqslant i \leqslant n$, the alternatives R_i either take the form $R_i = \varepsilon$, or $R_i = a\, R_i'$, with $a \in V$ and $R_i' \in \mathcal{R}$, and
3. if, for $1 \leqslant i, j \leqslant n$, $R_i = R_j$ implies that $i = j$.

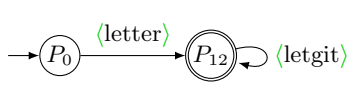


Fig. 2.8. The minimal DFA for ⟨identifier⟩

Lemma 2.2. *Every regular expression has an equivalent expression in quasi-regular form, which can be effectively constructed, and is unique up to commutativity of alternatives.*

Proof. We apply the equivalences in Lemma 2.1 as transformation rules in the following way:

1. (*Unfolding*) Apply the rules in the left column from left to right, i.e., by matching their left-hand side, and replacing them by their right-hand side. Since every form of regular expression occurs as a left-hand side, this eventually leads to an expression consisting of $n \geqslant 1$ alternatives R_i which are either empty, or start with a symbol.
2. (*Grouping*)Now apply commutativity (carefully) so that alternatives are adjacent whenever they are either both empty, or both start with the same symbol. Obviously, this form can be achieved as well.
3. (*Factorization*) Now use the rule for idempotence to identify empty adjacent alternatives, and the rule for distributivity appearing in the right column in order to factorize alternatives starting with the same symbol. The resulting regular expression is quasi-regular.

The result R' is unique up to switching alternatives. Since this does not influence the language being generated, we can choose $R' = \mathrm{qrf}(R)$. $\square$

This lemma is illustrated with an example.

Example 2.6. Quasi-Regularization of a Regular Expressions for Fractions] Consider the regular definition for (decimal) faction literals:

$$\langle\text{fraction}\rangle = \langle\text{digit}\rangle^*\underline{.}\,\langle\text{digit}\rangle^+$$

During the transformation, we treat the character class $\langle\text{digit}\rangle$ like a single symbol.

$$\begin{aligned}
&\langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\\
\equiv\ &(\varepsilon \mid \langle\text{digit}\rangle\langle\text{digit}\rangle^*)\,\underline{.}\langle\text{digit}\rangle^+\\
\equiv\ &\varepsilon\underline{.}\langle\text{digit}\rangle^+ \mid \langle\text{digit}\rangle\langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\\
\equiv\ &\underline{.}\,\langle\text{digit}\rangle^+\\
\mid\ &\langle\text{digit}\rangle\langle\text{digit}\rangle^*\underline{.}\,\langle\text{digit}\rangle^+
\end{aligned}$$

In this unfolded expression, alternatives are already grouped and unique so that they need not be factorized.

Lemma 2.2 is an essential ingredient for the construction of DFAs without the power set construction:

Algorithm 2.2 (DFA for Regular Expressions).

Input: R;
$i \leftarrow 0$; $Q_i \leftarrow \{R\}$; $Q \leftarrow Q_i$; $\Delta \leftarrow \{\}$
repeat $i \leftarrow i+1$; $Q_i \leftarrow \{\}$
  for all $R \in Q_{i-1}$
  do $R_1 \mid \cdots \mid R_n \leftarrow \mathrm{qrf}(R)$
    if $\exists i, 1 \leqslant i \leqslant n : R_i = \varepsilon$ **then** $F \leftarrow F \cup \{R\}$
    $\Delta \leftarrow \Delta \cup \{(R, a, R'_i) \mid R_i = a\, R'_i, 1 \leqslant i \leqslant n\}$
    $Q_i \leftarrow Q_i \cup \{R'_i \mid R_i = a\, R'_i, R'_i \notin Q, 1 \leqslant i \leqslant n\}$
  end for
  $Q \leftarrow Q \cup Q_i$
until $Q_i = \{\}$
Output: $A = (V, Q, \Delta, q_0 = R, F)$

Theorem 2.2. *Algorithm dDEA computes an automaton A with $\mathcal{L}(A) = \mathcal{L}(R)$.*

Proof. • The algorithm terminates since only the applications of rules for R^+ and R^* make the expression longer. These are repetitive, i.e., of the form $R\,R^+$ and $R\,R^*$, respectively, so that one returns to the original state after some transformations expanding R.
• All transitions from a state are deterministic by construction.
• It is easy to show the following invariant: in every state $R' \in Q$, the sub-automaton $A = (V, Q, \Delta, R', F)$ recognizes the language $\mathcal{L}(R')$.
• Then the final states are correctly determined. □

This construction is similar to the construction of $SLR(0)$ items described in Section 3.3.1.

Example 2.7 (A DFA for Simple Identifiers). For simple identifiers as discussed in Example 2.1, the states of the DFA are determined as follows:

• $Q_0 \leftarrow \{q_o\}$ with $q_0 = \mathit{letter}\ \langle\mathrm{letgit}\rangle^* = \mathrm{qrf}(q_0)$.
• $\Delta \leftarrow \{(q_0, \mathit{letter}, q_1)\}$ with $q_1 = \langle\mathrm{letgit}\rangle^* \equiv \varepsilon \mid \mathit{letgit}\ \mathit{letgit}^* = \mathrm{qrf}(q_1)$.
• $Q_1 \leftarrow \{q_1\}$.
• $F \leftarrow \{q_1\}$.
• $\Delta \leftarrow \{(q_0, \mathit{letter}, q_1), (q_1, \langle\mathrm{letgit}\rangle, q')\}$ with $q' = \langle\mathrm{letgit}\rangle^* = q_1$.
• $Q_2 \leftarrow \{\}$.
• $A = (V, Q = \{q_0, q_1\}, \Delta = \{(q_0, \mathit{letter}, q_1), (q_1, \langle\mathrm{letgit}\rangle, q_2)\}, q_0, F = \{q_1\})$.

The transition graph of automaton equals that in Figure 2.8.

Example 2.8 (A DFA for Fraction Literals). Fraction literals can be defined as follows (see Example 2.6):

$$\langle\mathrm{fraction}\rangle = \langle\mathrm{digit}\rangle^*.\langle\mathrm{digit}\rangle^+$$

We show, in Table 2.1, how Algorithm 2.2 works. The rows correspond to steps of Algorithm 2.2: Next to state q_i we show its quasi-regular form $\mathrm{qrf}(q_i)$. The

Table 2.1. States, their quasi-regular form, and successors in the DFA for fraction literals.

No.	State	Quasi-Regular Form	Successor
q_0	$\langle\text{digit}\rangle^*\,\underline{.}\,\langle\text{digit}\rangle^+$	$\underline{.}\,\langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\rangle\langle\text{digit}\rangle^*\,\underline{.}\,\langle\text{digit}\rangle^+$	q_1 q_2
q_1	$\langle\text{digit}\rangle^+$	$\langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon\\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_3
q_2	$\langle\text{digit}\rangle^*\,\underline{.}\,\langle\text{digit}\rangle^+$	$\underline{.}\,\langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\rangle\langle\text{digit}\rangle^*\,\underline{.}\,\langle\text{digit}\rangle^+$	q_1 q_2
q_3	ε $\mid \langle\text{digit}\rangle^+$	ε $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon\\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_3

right column shows the successor state for every alternative $a_i\ R'_i$ of $\text{qrf}(q_i)$, as it is determined in the **for**-loop. This gives the transitions, as the start a_i of the corresponding alternative is the symbol read under the transition.

The algorithm starts with $Q_0 = \{q_0\}$, and determines the successor states $Q_1 = \{q_1, q_2\}$ in the first step. Thus the first line gives the transitions $(q_0, \underline{.}, q_1)$ and $(q_0, \langle\text{digit}\rangle, q_2)$.

The next iterations of **repeat** calculate $Q_2 = \{q_3, q_4\}$ and $Q_3 = \{q_5, q_6\}$, with the corresponding transitions. Then all successors of Q_3 are already known, and the algorithm terminates.

The empty alternatives ε in the quasi-regular forms of the states determines the final states of the automaton as the singleton set $F = \{q_3\}$. Its transition graph is shown in Figure 2.9.

Example 2.9 (A DFA for Float Literals). Float literals (with exponent part) may have forms like `14.2E3`, `.2E-3`, `14.2E+3`, and also `14E-3`. They can be defined using the definitions of integer and fraction literals (see Example 2.1 and 2.8) :

$$\langle\text{exp}\rangle \triangleq \underline{\text{E}}\ (\underline{+} \mid \underline{-})^?\ \langle\text{digit}\rangle^+$$
$$\langle\text{float}\rangle \triangleq (\langle\text{integer}\rangle \mid \langle\text{fraction}\rangle)\ \langle\text{exp}\rangle$$

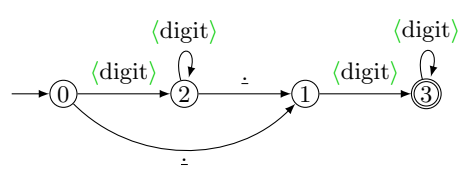


Fig. 2.9. A DFA for fraction literals

Table 2.2. States, their quasi-regular form, and successors in the DFA for float literals.

No.	State	Quasi-Regular Form	Successor
q_0	$(\langle\text{digit}\rangle^*\underline{.})^?\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$	$\underline{.}\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$ $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{digit}\rangle^*\underline{.}\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle \\ \mid \langle\text{exp}\rangle \\ \mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle\end{array}\right)$	q_1 q_2
q_1	$\langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$	$\langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{exp}\rangle \\ \mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle\end{array}\right)$	q_3
q_2	$\langle\text{digit}\rangle^*\underline{.}\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$ $\mid \langle\text{exp}\rangle$ $\mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$	$\underline{.}\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$ $\mid \underline{\text{E}}\ (+ \mid -)^?\ \langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{digit}\rangle^*\underline{.}\ \langle\text{digit}\rangle^+\langle\text{exp}\rangle \\ \mid \langle\text{exp}\rangle \\ \mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle\end{array}\right)$	q_1 q_4 q_2
q_3	$\langle\text{exp}\rangle$ $\mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle$	$\underline{\text{E}}\ (+ \mid -)^?\ \langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{exp}\rangle \\ \mid \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle\end{array}\right)$	q_4 q_3
q_4	$(+ \mid -)^?\ \langle\text{digit}\rangle^+$	$\langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon \\ \mid \langle\text{digit}\rangle^+\end{array}\right)$ $\mid + \langle\text{digit}\rangle^+$ $\mid - \langle\text{digit}\rangle^+$	q_5 q_6 q_6
q_5	ε $\mid \langle\text{digit}\rangle^+$	ε $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon \\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_5
q_6	$\langle\text{digit}\rangle^+$	$\langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon \\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_5

The definition can be "massaged" as follows:

$$\begin{aligned}\langle\text{float}\rangle &\triangleq (\langle\text{integer}\rangle \mid \langle\text{fraction}\rangle)\ \langle\text{exp}\rangle \\ &\equiv \left(\langle\text{digit}\rangle^+ \mid \langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\right)\ \langle\text{exp}\rangle \\ &\equiv (\langle\text{digit}\rangle^*\underline{.})^?\ \langle\text{digit}\rangle^+\ \langle\text{exp}\rangle\end{aligned}$$

Table 2.2 is organized as in Example 2.8: The rows describe a state q_i with the regular sub-expression to be recognized, its quasi-regular form $\text{qrf}(q_i)$, and its successor state for every alternative $a_i\ R'_i$ of $\text{qrf}(q_i)$.

Its resulting transition graph is shown in Figure 2.10.

2.5.5 Programming DFAs

There are two ways to implement a deterministic finite automata as a program:

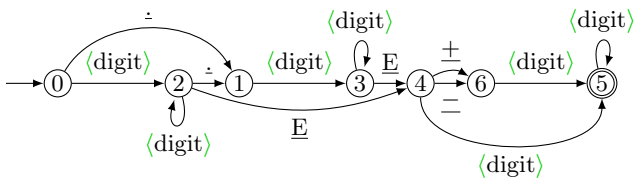


Fig. 2.10. A DFA for float literals

- as a table that is interpreted with a universal driver, or
- as simple imperative programs with jumps or loops.

Both implementations have the following assumptions in common:

1. The character to be read next is in variable *ch*.
2. The variable *ch* contains a possible starter of the lexeme when the automaton starts (a letter in case of ⟨identifier⟩, a quote in case of ⟨string literal⟩ etc.).
3. The implementation reads all characters that belong to the lexeme, i.e., the *longest matching prefix* of the input.
4. After recognition, the variable *ch* contains a character that does not belong to the lexeme.

Scanner Tables. A two-dimensional table has rows for every state, and columns for every symbol of the DFA. At position (q, t), the table contains the state q' if the automaton has a transition $q \rightarrow_t q'$. For all places (q, t) without a transition, the table is set to *accepting* $= -1$ if q is a final state, and to $\perp = -2$ for all intermediate states.

The driver for this table looks as follows:

```
begin
  var ch: Terminal;
  var Trans : array[State, Terminal] of State
              := ( ... );                          — Initialize with DFA
  acc: State = constant −1;
  q : State := 0; — start state
  while q >= 0
  loop
    q := Trans(q, ch);
    next(ch);
  end loop;
  if q = accepting
  then return
  else ... — error handling
  end if;
end
```

Table 2.3. Scanner table for identifiers

State	no.	a	...	z	A	...	Z	0	...	9	.	...	*
P_0	0	1	...	1	1	...	1	-2	...	-2	-2	...	-2
P_1	1	1	...	1	1	...	1	1	...	1	-1	...	-2

Example 2.10 (Scanner Table for Identifiers). Table 2.3 shows the scanner table for the DFA in Figure 2.8.

The scanner table is sparse, as it has many equal entries (namely, *acc* or *error*). So, the table should be represented according to one of the known techniques, for instance as an array of linked lists of type **array**[*State*] **of** *StateList* containing pairs of symbols (or of symbol classes) and successor states in decreasing probability.

Implementation as a table is beneficial if the automaton has been generated by a scanner generator like `lex`, as only the table has to be freshly initialized for each scanner whereas the program – the driver – stays the same.

Programs with Jumps and Loops. Every DFA can be implemented as a procedure. In general, the structure will be as follows:

- Every state corresponds to a place in the procedure that has a label.
- In every state, a case distinction inspects the variable *ch*.
- Every transition corresponds to a case; in every case, a symbol is read, a jump is performed to the (label of the) state.
- In a final state, the procedure returns if no transition is possible; otherwise, a lexical error has to be reported and handled.

Such procedures are poorly structured. As a matter of fact, they are close to assembly code. In many cases, a scanner procedure can also be formulated in a more structured way, with loops etc.

Example 2.11 (Scanner for Identifier). A scanner for ⟨identifier⟩ as a goto-program looks as follows:

```
procedure identifier is
begin
  <<q_0>> next(ch);
          goto <<q_1>>;
  <<q_1>> if letter(ch) or else digit(ch)
          then next(ch);
             goto <<q_1>>;
          end;
   return;
end identifier;
```

It can also be formulated with loops:

```
procedure identifier is
begin
  next(ch);
  while letter(ch) or else digit(ch)
  loop next(ch);
  end loop;
  return;
end identifier;
```

Such an implementation is recommended if the scanner is implemented "by hand", as this code is fairly readable.

2.6 Assembling the Scanner

Until now we have defined single lexemes, constructed the DFAs accepting them, and generated the programs scanning them. The scanner of a programming language has to read *all* lexemes of the source language. If the lexemes of the source language have been defined with regular expressions R_1 to R_n, the scanner must analyze the regular expression

$$R_1 \mid \cdots \mid R_n$$

Nowadays, programming languages are often defined in such a way that the combination of the lexeme definition does not cause any problems.

Example 2.12 (Lexeme von LOOP*).* The language LOOP to be implemented in the practical has the following lexemes:

$$\begin{aligned}
\langle\text{identifier}\rangle &= \langle\text{letter}\rangle\ (\langle\text{letter}\rangle \mid \langle\text{digit}\rangle)^* \\
\langle\text{integer}\rangle &= \langle\text{digit}\rangle\ \langle\text{digit}\rangle^* \\
\langle\text{operator}\rangle &= \underline{:=} \mid \underline{=} \mid \underline{\#} \mid \underline{<} \mid \underline{>} \mid \underline{<=} \mid \underline{>=} \mid \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/} \mid \underline{.} \\
\langle\text{delimiter}\rangle &= \underline{;} \mid \underline{(} \mid \underline{)} \mid \underline{:} \mid \underline{,} \\
\langle\text{keyword}\rangle &= \underline{\text{class}} \mid \underline{\text{extends}} \mid \underline{\text{is}} \mid \underline{\text{end}} \mid \underline{\text{method}} \mid \underline{\text{begin}} \mid \underline{\text{read}} \mid \underline{\text{write}} \\
&\quad \mid \underline{\text{if}} \mid \underline{\text{then}} \mid \underline{\text{elseif}} \mid \underline{\text{else}} \mid \underline{\text{while}} \mid \underline{\text{do}} \mid \underline{\text{return}} \\
&\quad \mid \underline{\text{or}} \mid \underline{\text{and}} \mid \underline{\text{mod}} \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \underline{\text{null}} \mid \underline{\text{self}} \mid \underline{\text{base}} \mid \underline{\text{new}} \\
\langle\text{layout}\rangle &= \ldots \\
\langle\text{comment}\rangle &= \underline{\{}\ldots\underline{\}} \mid \underline{|}\ldots\underline{\backslash\text{n}}
\end{aligned}$$

Keywords are a subset of the identifiers. Except for keywords, all lexemes start with disjoint characters. If we combine the DFAs of these regular expressions, the resulting automaton is deterministic again.

In the final states of the combined automaton, it does no longer suffice to report "success", since the automaton recognizes different kind of lexemes. If the automaton reaches a final state, it thus reports *which* lexeme

has been scanned. This information can be represented by values of an enumeration type containing all lexemes, e.g., *Identifier* , *AssignmentOperator*, *LessEqualOperator*, *ClosingParanthesis* etc.

Example 2.13 (Combining Number Literals). Sometimes, the assembly of the scanner is more difficult. Consider a language with three kinds of number literals:

- integer literals as in Example 2.1, with a DFA as in Figure 2.6,
- fraction literals as in Example 2.8, with a DFA as in Figure 2.9, and
- float literals as in Example 2.9, with a DFA as in Figure 2.10.

The regular expression

$$\langle\text{number}\rangle \triangleq \langle\text{integer}\rangle \mid \langle\text{fraction}\rangle \mid \langle\text{float}\rangle$$

can be merged to a concise regular definition

$$\begin{aligned}
\langle\text{exp}\rangle &\triangleq \underline{\text{E}}\,(\underline{+} \mid \underline{-})^{?}\ \langle\text{digit}\rangle^{+} \\
\langle\text{number}\rangle &\triangleq \langle\text{integer}\rangle \mid \langle\text{fraction}\rangle \mid \langle\text{float}\rangle \\
&\equiv \langle\text{integer}\rangle \mid \langle\text{fraction}\rangle \mid (\langle\text{integer}\rangle \mid \langle\text{fraction}\rangle)\ \langle\text{exp}\rangle \\
&\equiv (\langle\text{integer}\rangle \mid \langle\text{fraction}\rangle)\ \langle\text{exp}\rangle^{?} \\
&\equiv \big(\langle\text{digit}\rangle^{+} \mid \langle\text{digit}\rangle^{*}\underline{.}\langle\text{digit}\rangle^{+}\big)\ \langle\text{exp}\rangle^{?} \\
&\equiv (\langle\text{digit}\rangle^{*}\underline{.})^{?}\ \langle\text{digit}\rangle^{+}\langle\text{exp}\rangle^{?}
\end{aligned}$$

The DFAs cannot be merged so easily: If the start states of the automata are glued, this gives a non-deterministic automaton. We have to apply the power set construction in Algorithm 2.1 to make it deterministic. Application of Algorithm 2.2 to ⟨number⟩ would yield the same automaton (see Exercise 2.7). Its final states accept different kinds of numbers.

Actually, the scanner shall divide the whole source program into a sequence of lexemes, corresponding to the regular expression

$$(R_1 \mid \cdots \mid R_n \mid \langle\text{end-of-program}\rangle)^{+}$$

where ⟨end-of-program⟩ is a distinguished end marker, e.g., the EOF character. If iterated, the scanning will often be ambiguous: the digit sequence 12345 can be scanned as a sequence of one to five integer literals.

This very simple form of ambiguity can be resolved by the policy of *longest match*: the automata always try to read *as much as possible*. Then 12345 is scanned as a single integer literal.

2.6.1 Problems and Pathology with Lexemes

Sometimes, the policy of longest match does mislead, as shown by the following example in Pascal.

Example 2.14 (Intervals and Numbers in PASCAL*).* In PASCAL, the text 3..14 defines an interval of integer numbers, denoted with three lexemes (3, .., 14). Floating point number literals can be written in the usual decimal notation, e.g. as 3.14.

If the automaton for numbers does longest match "blindly", it would read the first period of the text 3..14, and report an error since the fractional part is missing. One could require that intervals are written as `3␣..␣14`. Then the problem is gone since the blank determines the end of the integer literal 3.

A better solution – that does not change the definition of the source language – if to apply longest match not always, but only if it is reasonable. We introduce a new regular operator for this purpose, called *lookahead*: R_1/R_2 generates the regular expression R_1 only if the rest of the string is built according to the regular expression R_2.

The regular expression for number literals could then be redefined as follows:

$$(\langle\text{digit}\rangle^*(\underline{.}/\langle\text{digit}\rangle))^? \ \langle\text{digit}\rangle^+ \ \langle\text{exp}\rangle^?$$

Then the period is read only if a digit follows; otherwise, only an integer literal is recognized.

Another problem may occur in enclosed lexemes like string literals or comments. If such lexemes may contain newlines, a missing closing character may lead to the situation that the entire rest of the program is recognized as a comment or string literal. Nowadays this problem – not for the scanner itself, but for the programmer using it – is avoided by banning newlines from such lexemes.

Two design options for the lexical analysis may lead to pathological situations:

- Keywords are not reserved, but may be used, at the same time, for identifiers of variables, procedures etc. (Fortran, PL/1).
- Layout is not used as a delimiter, but ignored completely; programs can be written without a single blank or newline (Algol, Fortran).

Example 2.15 (Lexical Problems in Fortran and PL/1). The following lines of Fortran differ just in one character, leading to a completely different lexical analysis of the texts:

```
DO 5 I = 1.25        | Assignment "DO5I := 1.25"
DO 5 I = 1,25        | loop "for i:= 1 to 25 do ..."
```

Since layout is ignored, and keywords like `DO` are not reserved, the recognition of `DO` is difficult. (The legend says that the tiny difference has sent a NASA-orbiter rather far off its intended route to Saturn.)

The following lines of PL/1 also differ just in one character:

```
DECLARE (A, B, C, D);        /* call a procedure DECLARE */
DECLARE (A, B, C, D)=...     /* start of a declaration   */
```

Again, the semantic difference is big. To resolve it, long lookahead is needed, or feedback from the contextual analysis, saying whether `DECLARE` is a procedure or not.

2.6.2 Lexical Errors

The scanner may detect errors. They arise whenever the automaton is unable to do a transition from a state that is not final. Examples include:

- Characters that may not occur in the program at all, or only enclosed in string literal or in comments. The start state of the combined automaton will not contain a transition under these characters.
  Sequences of such characters can be skipped; a typical error message would be: "*unexpected characters:... skipped*".
- Characters that are needed to recognize a lexeme, like the digit following the decimal point "`.`" or the exponent sign "`E`" in number literals.
  Here, the scanning should be terminated with a message like "*digit in exponent part expected: inserted*". In this case, the missing characters are ("virtually") inserted into the source text, a digit in this example.
- Terminating characters of lexemes, like the closing quote of string literals, or the closing characters of comments.
  Here, again, the missing character should be ("virtually") inserted into the source text.

Altogether there are two basic mechanisms of lexical error handling:

- *skipping* illegal characters, and
- *inserting* a character ("virtually").

Advanced scanners for languages with many keywords could furthermore apply a mechanism of "spelling correction": If the scanner delivers an identifier x instead of a keyword k expected by the syntax analysis, the scanner could check whether x is *similar* to k. This would be defined to be the case if x equals k after deleting / inserting one character, or after interchanging two adjacent characters.

The keyword <u>else</u>, e.g., is similar to the identifiers <u>lse</u>, <u>ese</u>, <u>ele</u>, <u>els</u> (deleting), <u>eelse</u>, <u>elste</u> etc. (inserting), <u>lese</u>, <u>esle</u>, <u>eles</u> (interchanging). This way of (assumed) error "correction" should be handled with care, however: short keywords as <u>if</u> are similar to, e.g., <u>i</u>, <u>f</u>, (deleting), <u>iif</u>, <u>ift</u> etc. (inserting), <u>fi</u> (interchanging). This is not a good basis for modification. Also, spelling correction violates a principle of compiler organization: There should is information flow from one phase (syntax analysis) back to an earlier one (lexical analysis). This is banned, as it may lead to back-tracking, which is time-consuming.

2.7 Screening

The scanner recognizes lexemes, and produces the following data for a lexeme:

- Its *kind* (identifier, string literal ...) The kind is a value of an enumeration type.
- Its *representation*, a string.
- Its *source position*. This is used for error messages or (later) for debugging. A source position has the general form (f, l, c), where f is the name of the source file, l the line number, and c the column number at which the lexeme occurs.

The screener has the task to process this data so that it can be used by the subsequent phases, syntax and context analysis, and code generation.

- For most of the lexemes, their kind is needed for the syntax analysis. However, some lexemes, like comments and layout, are just ignored in subsequent phases.
- The source position is needed in all parts of the analysis, in order to give error messages.
- For literals, their representation is needed for generating code that represents the values of these literals on the target machine.
- For identifiers, their representation is needed for context analysis, which will frequently check whether two identifiers have equal representations. Special care has to be taken to make this operation efficient.

The screener stores representations of literals and identifiers in a *representation table*, and passes indexes into that table to later phases. Identifiers are stored in an "hashed" *identifier table*, and the unique key of an identifier is passed on to contextual analysis so that checks for equality boil down to equality of their keys. In the identifier table, keys have an associated slot for information. Into this slot, the context analysis will insert declaration information, and the code generation will insert address information.

We will not describe here how hashing works – this should be known. For this particular purpose, a simple scheme like *hashing and chaining* is adequate.

2.7.1 Recognition of Keywords

Keywords usually have the form of identifiers, and are usually reserved for their purpose to denote a delimiter or operator of the source language. They can be recognized in two ways:

- The *scanner* may recognize them. Then the regular expression for identifiers has to be merged with those for the keywords. If the language is rich in keywords, this will blow up the automata considerably.
- The *screener* can determine which of the identifiers recognized by the scanner is actually a keyword. This can be done as follows:
  - The identifier table is initialized with all keywords, and the kind of the keywords is associated with their keys, e.g., "begin symbol", "and operator" .
  - The scanner recognizes identifiers and delivers their representation.

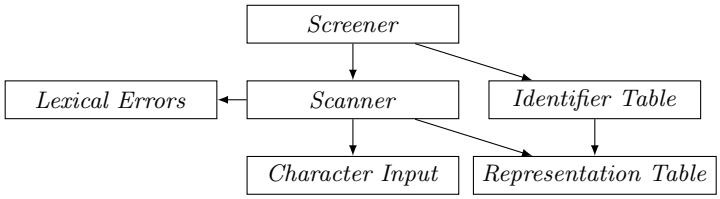


Fig. 2.11. The modules of a lexical analyzer

- The screener enters the identifier into the identifier table. If it was already stored, the kind stored with it is returned. Otherwise, it is entered with the kind "identifier".

This way of recognition keeps scanner automata small, and it is efficient. Moreover, it is flexible: new keywords can be added by just extending the initialization procedure.

2.8 The Structure of Lexical Analysis

A lexical analyzer should be divided into modules as follows (see Figure 2.11):

1. *Character Input* reads the source program from file. This is done blockwise in order to minimize file accesses. For the scanner, two variables are made available, e.g., `thisCharacter` and `nextCharacter`. They contain the actual and the next character, respectively. A method `readCharacter` reads one character and re-fills the buffer if necessary.
2. *Representation Table* stores representations of lexemes as literals and identifiers.
3. *Lexical Errors* contains methods that report and handle errors.
4. The *scanner* implements the finite automaton. It has a method `readLexeme` that recognizes the next lexeme.
5. *Identifier Table* stores identifiers in a hash table.
6. The *screener* calls the scanner, and translates the lexemes recognized by the scanner: representations of literals and identifiers are stored in the *Representation Table*, identifiers are entered into the *Identifier Table*. Two variables `thisLexeme` and `nextLexeme` are made available to the syntax analysis. They contain the actual and the next lexeme, respectively. A method `readLexeme` reads the next lexeme.

2.9 Generating Scanners with `lex` and `flex`

The scanner generator `lex`[2] is a part of the UNIX operating system. Its GNU version `flex`[3] offers a slightly enhanced functionality. Both systems generate scanners in C. Now there exist variants of these tools that generate code for other target languages, e.g., for Java.[4]

Each of these generators provides libraries for character input, error handling, string table handling, and generates scanners for regular definitions, and screeners from transformation rules.

These tools support more general regular definitions than those discussed so far: Not only character classes, complements, and lookahead operators are provided, but also an n-to-m iteration operator, and the option to distinguish states in which the scanner behaves differently (e.g., inside or outside a comment or a string literal). The meta-language is powerful enough to scan even complicated languages; it may even cope with the problems discussed in Section 2.6.1.

The tools have a particular syntax for regular definitions, and screener transformation rules. See [17] for details.

Example 2.16 (Flex Specification for a Pascal Subset). Figure 2.12 shows a `flex` scanner definition for a Pascal subset.[5] The brackets `%{ ... %}` enclose preprocessor instructions for the C compiler.

Then follow auxiliary regular definitions, of *digit* and *Id* in this example. An expression $[x{-}y]$ defines a character class ranging from character x to y. The $*$ denotes iteration, as usual.

After the "`%%`" follow screener transformations. Each of them consists of a regular expression, and a C-block containing the action taken when this lexeme has been recognized. Note that auxiliary definitions must be bracketed, as in { *digit* }. The unbracketed regular expression *digit* defines the five-letter string consisting of the characters *D*, *I*, *G*, *I*, and *T*! In this simple example, the transformations just report the recognized lexeme on standard output.

After the "`%%`" follow auxiliary C definitions. In this case, a procedure *main* is defined that calls the lexical analysis (*yylex* ();).

If the specification is stored in file `Pascmini.l`, calling `flex Pascmini.l` generates a scanner in the file `lex.yy.c`. Compiling it with a C compiler, e.g., by `gcc lex.yy.c`, yields a scanner in the file `a.out` that scans the standard input (unless a file name is given as an argument) and writes to the standard output. [6]

Missing: `lex` notation of regular expressions.

[2] dinosaur.compilertools.net/lex
[3] www.gnu.org/software/flex
[4] E.g., www.cs.princeton.edu/~appel/modern/java/JLex or jflex.de
[5] Found under the URL `http://www.cs.princeton.edu/ appel/modern/c/software/flex/`.
[6] This can be tried out with the file `Pascmini.l` that has been uploaded to `stud.ip`.

```
%option noyywrap
/* scanner for a toy Pascal-like language */
%{ /* need this for the call to atof() below */
#include <math.h>
%}

letter    [a-zA-Z]
digit     [0-9]
Id        {letter} ({letter} | {digit})*

%%

{digit}+"."{digit}*          {
            printf( "A float: %<<<<s (%g)\n", yytext,
                    atof( yytext ) );
            }

{digit}+     {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

if|then|begin|end|procedure|function          {
            printf( "A keyword: %s\n", yytext );
            }

{Id}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"     printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"       /* eat up one-line comments */

[ \t\n]+            /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {++argv, --argc;  /* skip over program name */
     if ( argc > 0 ) yyin = fopen( argv[0], "r" );
     else            yyin = stdin;
     yylex();
    }
```

Fig. 2.12. Flex specification of a Pascal subset

Bibliographical Notes

The division of the lexical analyzer into a scanner and a screener has been proposed by Frank DeRemer [8]. The Lex tool is described in [17]; a tutorial is [19].

Excercises

Solutions to some of these exercises can be found on page 150 *ff*.

Exercise 2.1. Give a regular definition for *Identifiers*, which shall be formed according to the following rules:

1. Identifier are strings (of arbitrary length) that consist of letters, digits, and "_" (underscore).
2. Identifiers start with a letter.
3. An underscore must not be the start or end of an identifier, and may not occur next to another underscore in an identifier. [7]

Examples: `x` `Hallo_World` `Identifier_may_be_quite_long`
Counterexamples: `1x` `H_a_l_l_o__W_o_r_l_d` `Name_`

Exercise 2.2. Give a regular definition for *String Literals*, which shall be defined as follows:

1. String literals are enclosed in quotes "$\underline{"}$".
2. They consist of an arbitrary number of *printable characters*.
3. They must not contain a newline character.
4. If one quote shall be made part of a string literal, it has to appear twice in it.[8]

Examples: The string literals `""`, `" Hallo, Welt!"`, and `""" Hallo, Welt!"""` contain 0, 12, and 14 characters, respectively.

Tips

- The regular expression "\n" shall represent the newline character.
- The regular expression "$\overline{[c_1, \ldots, c_n]}$" shall denote the set of characters that contains all printable characters (including newline), except for c_1 bis c_n; i.e., the set $\mathsf{PrintableChar} \setminus \{c_1, \ldots, c_n\}$.

Exercise 2.3. The right-hand sides of the transformation rule for R^+ in Figure 2.5 may appear too complicated at first glance, with all the empty transitions around (which – very inconvenient – make the automata non-deterministic.

[7] These are the rules for identifiers in the programming language Ada.
[8] These are the rules for string literals in the programming language Pascal.

1. Could the pairs $(s, m), (n, t)$ of nodes in the rule be identified, and the ε-transitions between them be removed?
   If not, construct an example where the "simplified" rule destroys equivalence.
2. Could the pair (m, n) of nodes in the rule be identified, and the ε-transition between them be removed?
   Why? Or Why not?

Exercise 2.4. Construct deterministic finite automata for the regular definitions of identifiers and string literals given in Exercise 2.1 and Exercise 2.2 as follows:

1. Generates a non-deterministic automaton according to Definition 2.6,
2. Construct the equivalent deterministic automaton, by the quotient set construction Algorithm 2.1, and
3. construct the minimal deterministic automaton according to Definition 2.7.

Exercise 2.5. Use Algorithm 2.2 to construct a DFA for string literals as defined in Exercise 2.2.

Exercise 2.6. *Comments* shall be enclosed in `/*` and `*/`; they may include newlines.

1. Define comments by a regular definition.
2. Construct a minimal finite automaton for comments from this definition. You may do this "intuitively", without taking the formal steps discussed in Section 2.5.
3. Check whether your definition gets simpler if you use a *lookahead operator* R_1/R_2 mentioned in Example 2.14.

Exercise 2.7. Construct a DFA for ⟨number⟩ as defined in Example 2.13. You can do it in one of the ways discussed there:

1. Use Algorithm 2.2 on the regular definition for ⟨number⟩.
2. Apply Algorithm 2.1 to the NFA obtained by gluing the start state of the DFAs for integer literals in Figure 2.6, fraction literals in Figure 2.9, and float literals in Figure 2.10.

Exercise 2.8. The language LOOP, which is used in the compiler practical, has the following lexemes:

- non-empty sequences of layout characters *blank*, *tabulator*, and *newline*,
- comments, which may extend over several lines, are enclosed in `(*` and `*)`,
- identifiers,
- integer literals,
- string literals,
- the delimiters `.`, `,`, `:`, `;`, `(`, and `)`,
- the operator symbols `<=`, `>=`, `:=`, `*`, `/`, `+`, `-`, `=`, `#`, `<`, and `>`, as well as

- the keywords `AND`, `ATTRIBUTE`, `BEGIN`, `CLASS`, `ELSE`, `ELSEIF`, `END`, `EXTENDS`, `IF`, `IS`, `METHOD`, `MOD`, `NEW`, `NOT`, `OR`, `OVERRIDE`, `READ`, `RETURN`, `THEN`, `WITH`, `WHILE`, and `WRITE`.

Tasks

1. Define the scanner for LOOP in `lex`. Here, we are not interested in the translation of lexemes into tokens, so the `return` statements can be omitted.)
   The regular definitions developed in the course can be used for the lexemes.
2. Produce two versions of the scanner: one without keywords, and one including them (Then their definitions should go *before* that of identifiers.)
3. Compare the size of the scanner tables.
   Should be keywords rather be recognized by the screener?

More exercises: Limp-lexer.

3

Syntax Analysis

Syntax analysis shall recognize the hierarchical structure of the source program, and represent this structure as a tree.

Usually, the syntax of the source language is given; in most cases it is defined with a context-free grammar or with a similar formalism. Syntax analysis parses a source program according to the syntax, and represents the result as an abstract syntax tree. If the program cannot be parsed, errors are reported, and handled in a way so that further syntax errors can be reported during the same run of the analysis.

Example 3.1 (Syntax Analysis). A simple conditional command can be represented with an abstract syntax tree as follows.

$$\textbf{if } id_1 < id_2 \textbf{ then } id_2 := id_2 - id_1 \rightsquigarrow \begin{array}{c} \textbf{if-then-else} \\ \diagup \quad | \quad \diagdown \\ \textbf{bop} \quad \textbf{ass} \quad \textbf{skip} \\ \diagup | \diagdown \quad \diagup \diagdown \\ id_1 < id_2 \quad id_1 \ \textbf{bop} \\ \diagup | \diagdown \\ id_2 - id_1 \end{array}$$

Here, **if**, id_i, **then**, "<", ":=", and "−" are lexemes, which are recognized by lexical analysis, and **if-then-else**, **bop**, **ass**, and **skip** are node constructors for abstract syntax trees.

In this chapter, we shall first recall some properties of context-free grammars and their languages. In particular, we discuss transformations of grammars that modify the rules of a grammar while preserving the language generated by it. Transformations are often needed before a grammar can be used for parsing. Then we discuss major parsing techniques. *Top-down Parsing* – e.g., $LL(k)$ – can be programmed "by hand" in a systematic way. *Bottom-up parsing* – e.g., $LR(k)$ – is used by many *parser generators*; so we explain the $LR(k)$ technique, in order to understand how parser generators work, and what their error messages mean if the parser cannot be generated.

3.1 Context-free Grammars

Context-free Chomsky grammars are widely used to define the syntax of programming languages. Here, we just recall some properties of context-free grammars and languages that are useful for parsing. First, we consider special forms of the derivations defined in Section A.2.3.

Definition 3.1 (Canonical Derivations). The derivation relation $\Rightarrow_R$ of a set R of context-free rules can be restricted in such a way that the left-most, or rightmost nonterminal in the word is replaced.

$$\frac{w \in V^*, \beta \in (N \cup V)^*, A \to \alpha \in R}{wA\beta \overset{\text{lm}}{\Rightarrow}_R w\alpha\beta} \qquad \frac{\beta \in (N \cup V)^*, w \in V^*, A \to \alpha \in R}{\beta Aw \overset{\text{rm}}{\Rightarrow}_R \beta\alpha w}$$

Leftmost and rightmost derivations suffice to generate all words of a context-free language.

Theorem 3.1. *For some context-free grammar $G = (V, T, R, z)$ and some word $w \in T^*$:*

$$z \Rightarrow_R^* w \text{ if and only if } z \overset{\text{lm}}{\Rightarrow}_R^* w \text{ if and only if } z \overset{\text{rm}}{\Rightarrow}_R^* w.$$

This is why leftmost and rightmost derivations are called *canonical*.

Definition 3.2 (Derivation Trees). A (directed, ordered) *tree* B over a vocabulary M consists of a finite set $\dot{B}$ of *nodes* with a distinguished *root* $rt_B \in \dot{B}$. Every node $v \in \dot{B}$ is labeled with an element $\ell_B(v) \in M$, and has an ordered sequence $kids_B(v) = v_1 \dots v_k$ of *kids*; whenever this sequence is empty, we call v a *leaf*; otherwise v is called a *branch*.

A tree B over $V \cup \{\varepsilon\}$ is a *derivation tree* of the context-free grammar $G = (V, N, R, z)$ if:

1. Every branch v von B is labeled with a nonterminal.
2. Whenever a branch v has ordered successors $kids_B(v) = v_1, \dots, v_k$, (drawn from left to right in figures, $k > 0$), the labels $\ell_B(v) \to \ell_B(v_1) \cdots \ell_B(v_k)$ are a rule of R.
3. Whenever a node is labeled with ε, it is a leaf, and the only kid of some node v so that $\ell(v) \to \varepsilon$ is a rule in R.
4. B is *complete* if its root is labeled with z.
5. B is *terminal* if all its leaves are labeled with $T \cup \{\varepsilon\}$.

The *frontier* of a derivation tree consists of the labels at its leaves, which are connected from left to right, i.e., it is a word in V^*:

$$front(B) = front_B(rt_B)$$

$$front_B(v) = \begin{cases} front_B(v_1) \dots front_B(v_k) & \text{if } kids_B(v) = v_1, \dots, v_k,\ k > 0 \\ \ell_B(v) & \text{otherwise (if } v \text{ is a leaf)} \end{cases}$$

In what follows, $\mathcal{B}_G$ is the set of derivation trees for G. It is easy to see that every node v in a derivation tree B satisfies $\ell_B(rt_B) \Rightarrow_R^* \mathit{front}(B)$; if B is complete and terminal, $\mathit{front}(B) \in \mathcal{L}(G)$.

Definition 3.3. A context-free Grammar is *ambiguous* if its language contains words that have different leftmost derivations, different rightmost derivations, or different derivation trees, respectively. Otherwise it is called *unambiguous*.

Example 3.2 (A Context-free Grammar for Simple Expressions). Let a vocabulary $V = N \cup T$ be given with nonterminals $N = \{S, E\}$ and terminals $T = \{-, *, \underline{\mathrm{i}}, (,)\}$. ($\underline{\mathrm{i}}$ represents integer literals.) The *rule set* shall be given as

$$R = \left\{ \begin{array}{ll} S \to E & (1) \\ E \to E - E & (2) \\ E \to E * E & (3) \\ E \to \underline{\mathrm{i}} & (4) \\ E \to (E) & (5) \end{array} \right\}.$$

Then $G_0 = (V, N, R, S)$ is a context-free grammar for simple expressions.

Consider two derivation trees and leftmost derivations for an expression: Figure 3.1.

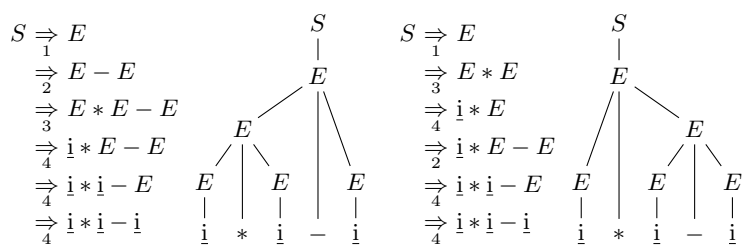


Fig. 3.1. Two leftmost derivations in G_0, and their derivation trees

Both derivations are leftmost, and their derivation trees are complete and terminal. The derivations and their trees are different, but generate the same word, and have the same frontier, respectively. Thus G_0 is ambiguous.

The example shows that ambiguous grammars are unsuited for programming languages. Only the leftmost derivation groups the expression (and the corresponding derivation tree) according to the usual precedences of operators.

Theorem 3.2. *Ambiguity is undecidable for context-free grammars.*

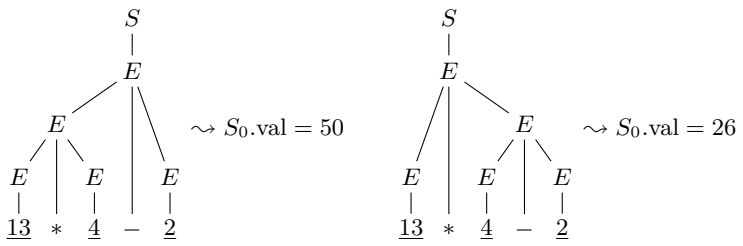


Fig. 3.2. Interpretation of an ambiguous expression

3.1.1 Syntax-Directed Transduction

Why should the syntax of programming languages be defined by unambiguous context-free grammars? First of all, parsing of unambiguous grammars is linear, in contrast to cubic effort for ambiguous grammars. Another aspect is more important: the syntax of the language provides the skeleton for all subsequent tasks, whether the language is compiled, or interpreted.

According to this principle, called *syntax-directed transduction*, the syntax of a language – more concretely: the rules of its context-free-grammar, – are associated with *semantic rules* that define its interpretation, or its translation into a target language. This kind of definition is unique only if the syntax is unambiguous.

Example 3.3 (Syntax-directed Interpretation and Translation). Figure 3.2 shows two syntax-directed transductions for the grammar G_0 for expressions in Example 3.2.

Occurrences of nonterminals in a context-free rule are numbered from left to right, in order to be distinguishable in semantic rules. We have substituted the terminal $\underline{\mathrm{i}}$, which is delivered by lexical analysis, by their representations (integer numbers) so that we can compute real results.

Table 3.1. Syntax-directed interpretation and translation of expressions

Syntax	Semantics	
	Interpretation	Translation
$S_0 \to E_1$	$S_0.\mathrm{val} \leftarrow E_1.\mathrm{val}$	$S_0.\mathrm{code} \leftarrow E_1.\mathrm{code} \mathbin{+\!\!+} \mathbf{ret}$
$E_0 \to E_1 - E_2$	$E_0.\mathrm{val} \leftarrow E_1.\mathrm{val} - E_2.\mathrm{val}$	$E_0.\mathrm{code} \leftarrow E_1.\mathrm{code} \mathbin{+\!\!+} E_2.\mathrm{code} \mathbin{+\!\!+} \mathbf{sub}$
$E_0 \to E_1 * E_2$	$E_0.\mathrm{val} \leftarrow E_1.\mathrm{val} * E_2.\mathrm{val}$	$E_0.\mathrm{code} \leftarrow E_1.\mathrm{code} \mathbin{+\!\!+} E_2.\mathrm{code} \mathbin{+\!\!+} \mathbf{mul}$
$E_0 \to \underline{\mathrm{i}}$	$E_0.\mathrm{val} \leftarrow str2int(i.\mathrm{repr})$	$E_0.\mathrm{code} \leftarrow \mathbf{push} \mathbin{+\!\!+} \underline{\mathrm{i}}.\mathrm{repr}$
$E_0 \to (E_1)$	$E_0.\mathrm{val} \leftarrow E_1.\mathrm{val}$	$E_0.\mathrm{code} \leftarrow E_1.\mathrm{code}$

In the column in the middle, a value val is computed, which is the result of evaluating the expression, thus defining an interpreter. In the column on the right, we define the translation of expressions into primitive stack machine instructions, thus defining a translation into a sequence code of instructions. The code can be executed by an "abstract machine" (a simple interpreter), which can be defined as follows:

$$
\begin{aligned}
\mathit{run}(c_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} c_n) &= \mathbf{int}[100]\mathit{Store};\ \mathbf{int}\ \mathit{top} \leftarrow 0;\ \mathit{exec}(c_1);\ \ldots;\ \mathit{exec}(c_n);\\
\mathit{exec}(\mathbf{push}\ i) &= \mathit{top} \leftarrow \mathit{top} + 1;\ \mathit{Store}[\mathit{top}] \leftarrow i\\
\mathit{exec}(\mathbf{sub}) &= \mathit{Store}[\mathit{top} - 1] \leftarrow \mathit{Store}[\mathit{top} - 1] - \mathit{Store}[\mathit{top}];\ \mathit{top} \leftarrow \mathit{top} - 1\\
\mathit{exec}(\mathbf{mul}) &= \mathit{Store}[\mathit{top} - 1] \leftarrow \mathit{Store}[\mathit{top} - 1] * \mathit{Store}[\mathit{top}];\ \mathit{top} \leftarrow \mathit{top} - 1\\
\mathit{exec}(\mathbf{ret}) &= \mathbf{return}\ \mathit{Store}[\mathit{top}]
\end{aligned}
$$

The evaluation of the expression gives the same result as the execution of its translation:

$$S_0.\mathrm{val} = \mathit{run}(S_0.\mathrm{code})$$

Since the grammar is ambiguous, its interpretation (and translation) may deliver different results, for the same expression, as shown in Fig. 3.2. The left tree respects the mathematical conventions for interpreting expressions.

3.1.2 Transformation of Context-Free Grammars

In order to parse words according to a context-free grammar, the grammar may be required to have certain properties.

Definition 3.4 (Properties of Context-free Grammars). For a context-free grammar $G = (V, N, R, z)$, we consider some properties of rules and nonterminals:

1. A rule $n \to \varepsilon$ is called *empty*, and a rule $n \to m$ is called a *chain rule*.
2. A nonterminal $n \in N$ is
   a) *productive* if $n \Rightarrow_R^+ w$ for some $w \in T^*$, and
   b) *useful* if $z \Rightarrow_R^+ \alpha n \beta$ for some $\alpha, \beta \in V^*$.
3. G is *reduced* if every nonterminal is useful and productive.
4. A rule $n \to \alpha n \beta$ is *directly recursive*. In particular, it is
   (a) *directly left-recursive* if $\alpha = \varepsilon$,
   (b) *directly right-recursive* if $\beta = \varepsilon$,
   (c) *directly cyclic* if $\alpha = \beta = \varepsilon$, and
   (d) *directly self-embedding* otherwise – then $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.
5. A nonterminal $n \in N$ is *recursive* if $n \Rightarrow_R^+ \alpha n \beta$. In particular, it is
   (a) *left-recursive* if $\alpha = \varepsilon$,
   (b) *right-recursive* if $\beta = \varepsilon$,
   (c) *cyclic* if $\alpha = \beta = \varepsilon$, and
   (d) *self-embedding* otherwise – then $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$.

G is *left-recursive* (*right-recursive*, *self-embedding*) if some of its nonterminals have the respective property.

Example 3.4 (Properties of G_0). In the grammar G_0, rules $E \to E\underline{-}E$ and $E \to E\underline{*}E$ are both directly left-recursive, and directly right-recursive. This makes G_0 left-recursive, and right-recursive. It is easy to see that G_0 is reduced. With an additional rule $X \to X\underline{\mathrm{i}}$, the fresh nonterminal X would be useless (because it is not used in one of the other rules) and unproductive (because there is no terminal rule for X allowing to derive a terminal word).

Definition 3.5 (Grammar Transformation). A computable function t from a grammar class $\mathcal{G}_1$ to a grammar $\mathcal{G}_2$ is a *transformation* if $\mathcal{L}(G) = \mathcal{L}(t(G))$ for every grammar $G \in \mathcal{G}_1$.

By Theorem 3.2, there is no general transformation of context-free grammars into unambiguous context-free grammars. In particular, there are context-free languages that are *inherently ambiguous*.

Example 3.5 (An Ambiguous Context-free Language). The language

$$\{a^k b^m c^n \mid k, m, n > 0, k = m \text{ or } m = n\}$$

is context-free. E.g., it is generated by the grammar

$$G_{\mathrm{a}} = \left\{ \begin{array}{ll} S \to A'C & (1\ ) \\ S \to AB' & (2\ ) \\ A' \to ab & (3\ ) \\ A' \to aA'b & (4\ ) \\ B' \to bc & (5\ ) \\ B' \to bB'c & (6\ ) \\ C \to c & (7\ ) \\ C \to cC & (8\ ) \\ A \to a & (9\ ) \\ A \to aA & (10) \end{array} \right\}.$$

Rule 1 generates the words $a^k b^k c^n$, and rule 2 the words $a^k b^m c^m$. Every word $a^k b^k c^k$ is generated ambiguously, both via rule 1 and 2.

It is easy to see that this is the case for every context-free grammar generating this language.

Definition 3.6 (Context-free Rule Transformation). In the following transformations of context-free rules, $\bar{n}$ is a "fresh" nonterminals that does not occur in the original grammar:
1. *Factorization* isolates common parts in rules:

$$\begin{bmatrix} n \to \alpha\beta_1\gamma \\ \vdots \\ n \to \alpha\beta_k\gamma \end{bmatrix} \rightsquigarrow \begin{bmatrix} n \to \alpha\bar{n}\gamma \\ \bar{n} \to \beta_1 \\ \vdots \\ \bar{n} \to \beta_k \end{bmatrix}$$

If $\beta = \varepsilon$, we call this *left-factorization*; if $\alpha = \varepsilon$, we call it *right-factorization*.
2. The inverse of factorization is *unfolding*:

$$\begin{bmatrix} m \to \alpha n \gamma \\ n \to \beta_1 \\ \vdots \\ n \to \beta_k \end{bmatrix} \rightsquigarrow \begin{bmatrix} m \to \alpha n \gamma \\ m \to \alpha \beta_1 \gamma \\ \vdots \\ m \to \alpha \beta_k \gamma \end{bmatrix}$$

If n has only the k rules mentioned, the original rule $m \to \alpha n \gamma$ can be removed. If we furthermore apply unfolding to all rules $m \to \alpha n \gamma$ using n, the rules for n become useless, and can be removed.

Theorem 3.3. *The rules in Definition 3.6 are grammar transformations.*

Proof. Let $G = (V, N, R, z)$ be the original grammar, and $G = (V', N', R', z)$ the transformed grammar.
1. $n \Rightarrow_R \alpha\beta_i\gamma$ if and only if $n \Rightarrow_{R'} \alpha\bar{n}\gamma \Rightarrow_{R'} \alpha\beta_i\gamma$.
2. $m \Rightarrow_R \alpha n\gamma \Rightarrow_R \alpha\beta_i\gamma$ if and only if $n \Rightarrow_{R'} \alpha\beta_i\gamma$.
These equivalences allow to replace steps with "old" rules by steps with "new" rules. □

Example 3.6 (Factoring Binary Operators). In grammar G_0, we can apply factorization to the rules defining binary operations, introducing a new non-terminal B. These rules are easier to extend by further operator symbols (+, /), giving a grammar for richer expressions. These rules can be transformed by unfolding.

$$\begin{bmatrix} E \to E - E \\ E \to E * E \end{bmatrix} \rightsquigarrow \begin{bmatrix} E \to EBE \\ B \to - \\ B \to * \end{bmatrix} \qquad \begin{bmatrix} E \to EBE \\ B \to - \\ B \to + \\ B \to * \\ B \to / \end{bmatrix} \rightsquigarrow \begin{bmatrix} E \to E - E \\ E \to E + E \\ E \to E * E \\ E \to E/E \end{bmatrix}$$

Example 3.7 (Another Grammar for Expressions). The context-free grammar G_2 is given with the following rules:

$$R_2 = \left\{ \begin{array}{ll} S \to E & (1\ ) \\ E \to T & (2a) \\ E \to E - T & (2b) \\ T \to F & (3a) \\ T \to T * F & (3b) \\ F \to \underline{\mathrm{i}} & (4\ ) \\ F \to (E) & (5\ ) \end{array} \right\}.$$

G_2 is unambiguous. This is achieved by distinguishing three kinds of expression: outside brackets, factors F may contain neither $\underline{-}$, nor $\underline{*}$, and terms

may not contain $\underline{-}$. Thus an expression containing $\underline{-}$ may never occur as on operand of $\underline{*}$.

As the recursive rules (2*b*) and (3*b*) for E and T are left-recursive but not right-recursive, derivations make that adjacent operations $\underline{-}$ or $\underline{*}$ are always grouped according to left-associativity. See Figure 3.3 as an example.

Theorem 3.4. $\mathcal{L}(G_1) = \mathcal{L}(G_2)$, *and* G_2 *is unambiguous.*

Proof Sketch. To show that G_1 is equivalent to G_2, we define rules that transform a derivation tree T_1 of G_1 into a derivation tree T_2 of G_2 with $\mathit{front}(T_1) = \mathit{front}(T_2)$.

The rules in Figure 3.4 shuffle the derivation trees of G_1 until neighboring operator subtrees are left-associative, and follow the precedence of $\underline{*}$ over $\underline{-}$.

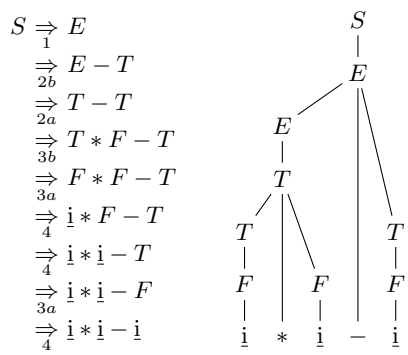


Fig. 3.3. A derivation (tree) for grammar G_2

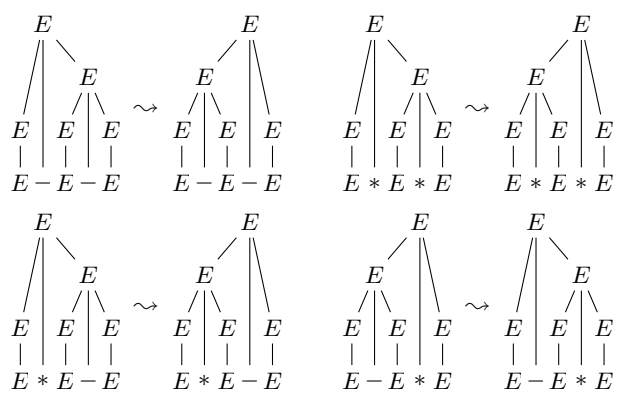


Fig. 3.4. Rules for restructuring derivation trees of G_1

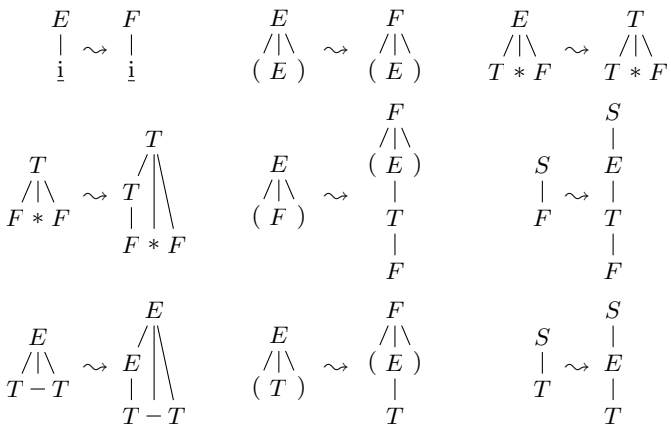


Fig. 3.5. Rules turning derivation trees of G_1 into trees for G_2

Then labels at the branches of the tree have to be changed to fit the nonterminals in G_2; also, subtrees for chain rules $E \to T$ and $T \to F$ have to be inserted. These rules are shown in Figure 3.5. Now an easy induction on the structure of derivation trees shows that when the leaves of the left-hand sides of the relabeling rules are built according to G_2, so will the right-hand sides of the rules. As the rules cover all possible combinations of labels, they will eventually transform the G_1-tree into a G_2-tree.

G_2 is unambiguous, since it allows only one derivation for expressions of the form

$$x * y - z = (x * y) - z,\ x - y - z = (x - y) - z), \text{ and } x * y * z = (x * y) * z$$

that are ambiguous in G_1. $\square$

For compilers, we do not use context-free grammars for deriving arbitrary words, but to answer the question whether a particular word $w \in V^*$ belongs to the language of a grammar, or not. This question is known as the *word problem*. A standard result of formal language theory says:

Theorem 3.5. *The word problem is decidable for context-free grammars.*

A program solving the word problem is called a *recognizer*. As mentioned before, there are recognizers that are cubic, and quadratic for unambiguous grammars.

A recognizer is not quite what we need in a compiler, because it is a boolean function that answers *true* whenever the inspected word is in the language. For compilation, we need a *parser*: a recognizer that does not only answer *true* for words in the language, but constructs a representation for its

derivation(s) whenever this is the case. Derivations can be represented by left-most or right-most derivations, by derivation trees, or by trees, called *abstract syntax trees*, which are similar to derivation trees.

3.1.3 Stack Automata

Context-free grammars have a relation to recognizing automata, like regular grammars and definitions have to finite-state automata: A language can be derived wit ha context-free grammar if and only if it can be recognized by a push-down automaton.

Definition 3.7 (Push-Down Automaton). A *push-down automaton* (*PDA*, for short) is a seven-tuple

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

1. Q is a finite set of *states* (symbols),
2. Σ is a finite *input vocabulary*.
3. Γ is a finite set of *stack symbols*,
4. δ is a *transition mapping* from $(\Sigma \times (\Sigma \cup \varepsilon) \times \Gamma$ to subsets of $Q \times \Gamma^*$,
5. $q_0 \in Q$ is the *initial state*,
6. $Z_0 \in \Gamma$ is the *initial stack symbol*, and
7. $F \subseteq Q$ is a set of *final states*.

A triple $(q, w, \alpha) \in (Q \times \Sigma^* \times \Gamma^*)$ is called a *configuration* of A, where

1. q represents the *current state*,
2. w represents the *unread input*, and
3. α represents the *stack*.

Moves of A are denoted by a binary relation $\vdash_A$ (or simply $\vdash$ if A is clear). Moves are defined by the inference rule

$$\frac{q \in Q, a \in (\Sigma \cup \varepsilon), w \in \Sigma^*, Z \in \Gamma, \alpha \in \Gamma^*, (q', \gamma) \in \delta(q, a, Z)}{(q, aw, Z\alpha) \vdash_A (q', w, \gamma\alpha)}$$

A move as above is an *empty move* if $a = \varepsilon$. Empty moves may change state and manipulate the stack; other moves may also change state and manipulate the stack, but read the first symbol of the input as well. Empty moves are possible while the input is empty, but no move is possible if the stack is empty.

As usual, $\vdash_P^+$ and $\vdash_P^*$ denote that there are move sequences of length $n > 0$ and $n \geqslant 0$, respectively. An *initial configuration* of A takes the form (q_0, w, Z_0), where w is the word to be recognized. *Final configurations* take the form (q, ε, α), where $q \in F$ and $\alpha \in \Gamma^*$.

The *language* accepted by A is defined as

$$L(A) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha), , q \in F, \alpha \in \Gamma^*\}$$

A pushdown automaton A as above is *deterministic* (a *dPDA* for short) if

1. $\delta(q, a, Z)$ contains at most one element for each $a \in \Sigma$ and $\delta(q, \varepsilon, Z) = \emptyset$, or
2. $\delta(q, a, Z) = \emptyset$ for each $a \in \Sigma$ and $\delta(q, \varepsilon, Z)$ contains at most one element.

Pushdown automata recognize exactly the context-free languages:

Theorem 3.6. *A language L is context-free if and only of it is accepted by some push-down automata.*

The proof of this theorem can be found in every book on formal language theory.

3.2 Top-Down Parsing

Top-down parsers attempt to construct a left-most derivation for a word (or its derivation tree) by constructing it from the top downwards, from the root to its leaves, matching the word as it is read, from left to right.

Definition 3.8 (Top-Down Parser). The *configurations* of a top-down parser are denoted as $w{\bullet}\alpha$, where α is the stack of the automaton, containing the hypothesis used for parsing, and w is the suffix of the word that has still to be parsed according to the hypothesis.

The *start configuration* of a parser is $z{\bullet}w$. This means: the hypothesis is the start symbol z, and w is the entire word to be parsed.

An *accepting configuration* of the parser has the form $\varepsilon{\bullet}\varepsilon$: the word w has been read completely, and the hypothesis has been confirmed completely.

The parser performs two kinds of actions:

1. *Matching* of a terminal symbol $t \in T$:

$$t\alpha{\bullet}tw \vdash_t \alpha{\bullet}w$$

   If the first terminal symbol to be read matches the first symbol in the hypothesis, it is removed from both, the input and the hypothesis.
2. *Expansion* of a rule $r = n \to \beta \in R$:

$$n\alpha{\bullet}w \vdash_r \beta\alpha{\bullet}w$$

   If the hypothesis starts with some nonterminal n, this symbol can be replaced by the right-hand side of a rule for n.

The parser satisfies the following *invariant*:

$$\forall w, \bar{w} \in T^* \; \forall \alpha \in V^* : z{\bullet}w\bar{w} \vdash_R^* \alpha{\bullet}\bar{w} \Leftrightarrow z \Rightarrow_R^* w\alpha$$

In particular, every expansion satisfies the invariant:

$$\forall w \in T^* \; \forall \alpha, \beta \in V^* : \alpha{\bullet}w \vdash_r \beta{\bullet}w \Leftrightarrow \alpha \overset{\text{lm}}{\Rightarrow}_r \beta$$

$$
\begin{array}{ll}
 & S\bullet 13 * 4 - 2 \\
\vdash_1 & E\bullet 13 * 4 - 2 \\
\vdash_2 & E - E\bullet 13 * 4 - 2 \\
\vdash_3 & E * E - E\bullet 13 * 4 - 2 \\
\vdash_4 & \underline{\mathrm{i}} * E - E\bullet 13 * 4 - 2 \\
\vdash_{\underline{\mathrm{i}}} & *E - E\bullet * 4 - 2 \\
\vdash_* & E - E\bullet 4 - 2 \\
\vdash_4 & \underline{\mathrm{i}} - E\bullet 4 - 2 \\
\vdash_{\underline{\mathrm{i}}} & -E\bullet - 2 \\
\vdash_- & E\bullet 2 \\
\vdash_4 & \underline{\mathrm{i}}\bullet 2 \\
\vdash_{\underline{\mathrm{i}}} & \varepsilon\bullet\varepsilon
\end{array}
$$

Fig. 3.6. A top-down parse for G_1

So every successful recognition performs a successful left-most derivation of a word in the language, as requested for a parser.

The parser may run into "blind alleys" if , at some point, a wrong rule has been chosen for expansion, or if the word does not belong to the language:

- The input does not match the hypothesis: $t\alpha\bullet t'w$ with $t \neq t'$.
- The input is too long: $\varepsilon\bullet t'w$.
- The input is too short: $v\alpha\bullet\varepsilon$.

In such situations, the parser must unroll actions by *backtracking*:

- Go back to the latest previous expansion, undo the expansion, and remember that the rule has been tried. Then expand the next untried rule.
- If every rule of the nonterminal has been tried at this point, backtrack to the latest previous expansion.
- When all expansions in the parse have been tried, terminate with failure: the word does not belong to the language, as all leftmost derivations have been tried.

Example 3.8 (Top-down Parsing of Expressions). Figure 3.6 shows a top-down parse for an expression, according to the (ambiguous) grammar G_1. Note that the parse constructs a left-most derivation of the word:

$$S \Rightarrow_1 E \Rightarrow_2 E - E \Rightarrow_3 E * E - E \Rightarrow_4 \underline{\mathrm{i}} * E - E \Rightarrow_4 \underline{\mathrm{i}} * \underline{\mathrm{i}} - E \Rightarrow_4 \underline{\mathrm{i}} * \underline{\mathrm{i}} - \underline{\mathrm{i}}$$

Deterministic Top-down Parsing

The top-down parsers are non-deterministic since the expansion operation may choose an arbitrary rule for the nonterminal to be expanded. How can the decision between different rules $n \to \beta$ and $n \to \gamma$ be made deterministic? A classical way is *lookahead* on the input: if the first $k > 0$ symbols of the remaining input indicate which of the rules for the n has to be expanded.

Definition 3.9 ($LL(k)$-Grammar and Parser). Consider a context-free grammar $G = (V, N, R, z)$.

The *first k terminals* (of length $k > 0$) of a word $\alpha \in V^*$ are defined as

$$Fi^k(\alpha) = \{u \in T^* \mid \alpha \Rightarrow_R^* u\beta, u \in T^k \text{ and } \beta \in V^*, \text{ or } |u| \leqslant k \text{ and } \beta = \varepsilon\}$$

A context-free grammar $G = (V, N, R, z)$ is *deterministically top-down parseable with k symbols of lookahead*, $LL(k)$ for short, if for every parse $z\bullet w \vdash^* n\alpha\bullet w$:

$$\text{for all } n \to \beta, n \to \gamma \in R \text{ with } \beta \neq \gamma : Fi^k(\alpha\beta) \cap Fi^k(\alpha\gamma) = \varepsilon$$

An $LL(k)$-parser for an $LL(k)$-grammar performs the following expand actions:

$$\forall r = n \to \beta \in R : n\alpha\bullet uw \vdash_{r,u} \beta\alpha\bullet uw \text{ if and only if } u \in Fi^k(\beta\alpha)$$

Theorem 3.7. *$LL(k)$-parsers are deterministic.*

Proof. Since the expansion steps of a top-down parsers construct left-most derivations, the deterministic expansion steps of an $LL(k)$-parser constructs at most one left-most derivation for every word. Thus the grammar is unambiguous. □

The $LL(k)$-condition is *sufficient* to give unambiguity. It is not *necessary*, since there are unambiguous grammars that fail to be $LL(k)$, e.g., G_2.

Corollary 3.1. *$LL(k)$-grammars are unambiguous.*

It is quite complicated and complex to decide the $LL(k)$-condition for a grammar, as the prefixes of all parses have to be checked for disjoint first symbols. So we devise a definition of prefixes that relies on rules alone:

Definition 3.10 ($SLL(k)$-Condition). The *prefixes* of a rule $n \to \alpha$ are defined as the first symbols, concatenated to follower symbols of n if α derives words shorter than k:

$$Pref^k(n \to \alpha) = \{w\bar{w} \mid w \in Fi^k(\alpha), \bar{w} \in Fo^{\bar{k}}(n), \bar{k} = k - |w|\}$$

$$Fo^k(n) = \begin{cases} \square^k & \text{if } n = z \\ \bigcup_{\bar{n} \to \alpha n \beta \in R} Pref^k(\bar{n} \to \beta) & \text{otherwise.} \end{cases}$$

Then the *strong $LL(k)$-condition* ($SLL(k)$-condition, for short) is:

$$\begin{aligned} &\forall n \in N\ \forall n \to \alpha, n \to \beta \in R \text{ with } \alpha \neq \beta : \\ &\qquad Pref^k(n \to \alpha) \cap Pref^k(n \to \beta) = \emptyset \end{aligned} \tag{3.1}$$

A grammar satisfying this condition is called *$SLL(k)$-grammar*.

Theorem 3.8. *$SLL(k)$-grammars are $LL(k)$.*

Definition 3.11 (Error Configuration of $SLL(k)$-Parsers). $SL(k)$-parsers have the following error configurations:
• The input does not match the hypothesis: $t\alpha\bullet t'w$ with $t \neq t'$. (As before.)
• No proper expansion is possible:

$$n\alpha\bullet uw, \text{ but } u \notin \bigcup_{n\to\beta\in R} Pref^k(n \to \beta).$$

• The input is too long: $\varepsilon\bullet t'w$. (As before.)
The handling of errors will be discussed later.

For practical parsing, prefixes and followers of length one turn out to be sufficient in most cases.

Algorithm 3.1 ($First$ and $Follow$ of Context-free Grammars).

Input: A grammar $G = (V, N, R, z)$.
Output: The functions $Fi^1, Fo^1, Pref^1 \colon R \to \wp(V \cup \{\Box\})$.
Method: Apply the following inference rules until the sets $Pref^1(r)$ are stable, for every $r = n \to \alpha \in R$.

$$\frac{t \in T}{t \in Fi^1(t)} \qquad \frac{n \to \alpha \in R}{Fi^1(\alpha) \subseteq Fi^1(n)}$$

$$\frac{}{\varepsilon \in Fi^1(\varepsilon)} \qquad \frac{}{Fi^1(v) \subseteq Fi^1(v\alpha)} \qquad \frac{\varepsilon \in Fi^1(n)}{Fi^1(\alpha) \subseteq Fi^1(n\alpha)}$$

$$\frac{}{\Box \in Fo^1(z)} \qquad \frac{n \to \alpha m\beta \in R}{Fi^1(\beta) \subseteq Fo^1(m)} \qquad \frac{n \to \alpha m\beta \in R, \varepsilon \in Fi^1(\beta)}{Fo^1(n) \subseteq Fo^1(m)}$$

$$\frac{n \to \alpha \in R}{Fi^1(\alpha) \subseteq Pref^1(n \to \alpha)} \qquad \frac{n \to \alpha \in R, \varepsilon \in Fi^1(\alpha)}{Fo^1(n) \subseteq Pref^1(n \to \alpha)}$$

Example 3.9 (Constructing an $SLL(1)$-Grammar for Expressions). In Table 3.2, rules of the grammar G_2 are shown in the first column, and the corresponding first symbols are shown in the second column. (Follower sets are not needed here because the grammar does not contain empty rules.) This grammar does not satisfy (3.1). Because, rules $(2b)$ and $(3b)$ are left-recursive. In this example, we can just swap the nonterminals in these rules and get a grammar G_3, which can be seen in the third column.[1]

Still, condition (3.1) does not hold, as the right-hand sides of rule pairs $((2a), (2b))$ and $((3a), (3b))$ start with the same nonterminals so that the first sets are as for G_2.

Left-factorization (defined in Definition 3.6) of rule pairs $((2a), (2b))$ and $((3a), (3b))$ in G_3 gives a grammar G_4, with additional nonterminals E' and

[1] Unfortunately, G_3 does no longer respect associativities. We shall deal with this problem later, when constructing trees.

Table 3.2. First and Follow sets for grammars G_2 and G_3

G_2	Fi^1	Fo^1	G_3	Fi^1	Fo^1
$S \to E$	$\underline{\mathrm{i}}, ($	$\square$	$S \to E$	$\underline{\mathrm{i}}, ($	$\square$
$E \to T$	$\underline{\mathrm{i}}, ($	$\square)$	$E \to T$	$\underline{\mathrm{i}}, ($	$\square$
$E \to E - T$	$\underline{\mathrm{i}}, ($	$-*$	$E \to T - E$	$\underline{\mathrm{i}}, ($	$)$
$T \to F$	$\underline{\mathrm{i}}, ($	$\square)$	$T \to F$	$\underline{\mathrm{i}}, ($	$\square$
$T \to T * F$	$\underline{\mathrm{i}}, ($	$-*$	$T \to F * T$	$\underline{\mathrm{i}}, ($	$)-$
$F \to \underline{\mathrm{i}}$	$\underline{\mathrm{i}}$	$\square)$	$F \to \underline{\mathrm{i}}$	$\underline{\mathrm{i}}$	$\square)$
$F \to (E)$	$($	$-*$	$F \to (E)$	$-*$	$($

Table 3.3. First, Follow, and Prefix sets for grammar G_4

G_4	Fi^1	Fo^1	$Pref^1$
$S \to E$	$\underline{\mathrm{i}}, ($	$\square$	$\underline{\mathrm{i}}, ($
$E \to TE'$	$\underline{\mathrm{i}}, ($	$-, \square$	$\underline{\mathrm{i}}, ($
$E' \to \varepsilon$	ε	$), \square$	$), \square$
$E' \to -TE'$	$-$		$-$
$T \to FT'$	$\underline{\mathrm{i}}, ($	$-,), \square$	$\underline{\mathrm{i}}, ($
$T' \to \varepsilon$	ε	$-,), \square$	$-,), \square$
$T' \to *FT'$	$*$		$*$
$F \to \underline{\mathrm{i}}$	$\underline{\mathrm{i}}$	$*, -,), \square$	$\underline{\mathrm{i}}$
$F \to (E)$	$($		$($

$$
\begin{aligned}
& S\bullet 13 * 4 - 2 \vdash_1 E\bullet 13 * 4 - 2 \vdash_{2'} TE'\bullet 13 * 4 - 2 \vdash_{3'} FT'E'\bullet 13 * 4 - 2 \\
\vdash_4 \quad & \underline{\mathrm{i}} * FT'E'\bullet 13 * 4 - 2 \text{ with } \big(\underline{\mathrm{i}} \in \mathit{Pref}^1(F \to \underline{\mathrm{i}})\big) \\
\vdash_{\underline{\mathrm{i}}} \quad & *FT'E'\bullet * 4 - 2 \vdash_* FT'E'\bullet 4 - 2 \vdash_4 \underline{\mathrm{i}}T'E'\bullet 4 - 2 \text{ with } \big(\underline{\mathrm{i}} \in \mathit{Pref}^1(F \to \underline{\mathrm{i}})\big) \\
\vdash_{\underline{\mathrm{i}}} \quad & T'E'\bullet - 2 \vdash_{3b'} E'\bullet - 2 \text{ with } \big(\underline{-} \in \mathit{Pref}^1(E' \to -TE')\big) \\
\vdash_{2a'} \quad & -TE'\bullet - 2 \text{ with } \big(\underline{-} \in \mathit{Pref}^1(E' \to -TE')\big) \\
\vdash_{\underline{-}} \quad & TE'\bullet 2 \vdash_{3'} FT'E'\bullet 2 \vdash_4 \underline{\mathrm{i}}T'E'\bullet 2 \text{ with } \big(\underline{\mathrm{i}} \in \mathit{Pref}^1(F \to \underline{\mathrm{i}})\big) \\
\vdash_{\underline{\mathrm{i}}} \quad & T'E'\bullet\varepsilon \vdash_{3a'} E'\bullet\varepsilon \text{ with } \big(\square \in \mathit{Pref}^1(T' \to \varepsilon)\big) \\
\vdash_{2a'} \quad & \varepsilon\bullet\varepsilon \text{ with } \big(\square \in \mathit{Pref}^1(E' \to \varepsilon)\big)
\end{aligned}
$$

Fig. 3.7. A deterministic $SLL(k)$ parse

T', which have empty rules. That is why also follower sets are computed in Table 3.3.

This grammar – finally! – satisfies the $SLL(k)$-condition for $k = 1$. For this grammar, expansion can be made deterministic, by inspecting the first symbol of the remaining suffix. See Fig. 3.7.

3.2.1 Extended Backus-Naur Form

Backus-Naur-Form has been used to define the syntax of Algol-60 for the first time; later it has been extended by abbreviations similar to operators in regular expressions: "$[B]$" is equivalent to "$B^?$", and "$\{B\}$" is equivalent to "B^*",, and "$B\ \{B\}$" is equivalent to "B^+".

Definition 3.12 (Extended Backus-Naur Form). A syntax definition in *extended Backus-Naur form* (Ebnf) consists of a set of definitions of the form

"$\langle n \rangle \triangleq B$.", where the left-hand sides $\langle n \rangle$ are *nonterminals*, and the right-hand sides B are EBNF *expressions* taking one of the following forms:

1. a choice "$B_1 \mid \cdots \mid B_n$" of EBNF expressions $B_1, \ldots, B_n$,
2. a *concatenation* "$B_1 \ldots B_n$" of EBNF expressions $B_1, \ldots, B_n$,
3. the (invisible) *empty string* "" (sometimes denoted ass "ε", for clarity),
4. a *terminal symbol* "$\underline{\mathrm{t}}$" of the language,
5. a *nonterminal* denoted as "$\langle \mathrm{n} \rangle$",
6. an optional expression "$[B]$",
7. an iterative expression "$\{B\}$", or
8. a bracketed expression "(B)".

Here concatenation binds stronger than choice; thus "$B_1 \mid B_2 B_3$" is equivalent to "$B_1 \mid (B_2 B_3)$".

Without the forms 6-8, the rules are in *simple Backus-Naur form.* [2]

An extension "(B)", "$\{B\}$" , and "$[B]$" can be removed by adding a fresh nonterminal "$\langle \bar{n} \rangle$", and inserting a new definition as for it as follows:

1. For a bracketed expression "(B)", insert the definition "$\langle \bar{n} \rangle \triangleq B$".
2. For an option "$[B]$", insert the definition "$\langle \bar{n} \rangle \triangleq \varepsilon \mid B$".
3. For an iteration "$\{B\}$", insert the definition "$\langle \bar{n} \rangle \triangleq \varepsilon \mid B \langle \bar{n} \rangle$".

The definition of First- and Follow-sets can be extended to EBNF rules.

Algorithm 3.2 (*First* and *Follow* of EBNF). For a grammar with EBNF operations, Algorithm 3.1 can be extended by the folwing rules:

$$\frac{}{\varepsilon \in Fi^1(\{B\})} \quad \frac{}{\varepsilon \in Fi^1([B])} \quad \frac{t \in Fi^1(B)}{t \in Fi^1(\{B\})}$$

$$\frac{t \in Fi^1(B)}{t \in Fi^1([B])} \quad \frac{t \in Fi^1(B_i), 1 \leqslant i \leqslant k}{t \in Fi^1(B_1 \mid \cdots \mid B_k)} \quad \frac{t \in Fi^1(B)}{t \in Fi^1((B))}$$

$$\frac{}{\square \in Fo^1(z)} \quad \frac{n \to \alpha m \beta \in R}{Fi^1(\beta) \subseteq Fo^1(m)} \quad \frac{n \to \alpha m \beta \in R, \varepsilon \in Fi^1(\beta)}{Fo^1(n) \subseteq Fo^1(m)}$$

$$\frac{n \to \alpha \in R}{Fi^1(\alpha) \subseteq Pref^1(n \to \alpha)} \quad \frac{n \to \alpha \in R, \varepsilon \in Fi^1(\alpha)}{Fo^1(n) \subseteq Pref^1(n \to \alpha)}$$

*Example 3.10 (*EBNF*-Definitions of Conditional Commands).* In ADA, conditional commands can be defined in EBNF as follows:

[2] Note that the forms "$[B]$"and "$\{B\}$" correspond to regular expressions of the form "$B^?$"and "B^*".

$\langle$statements$\rangle \triangleq \langle$statement $\rangle\{\underline{;}\ \langle$statement$\rangle\}$

$\langle$statement$\rangle \triangleq$ **if** $\langle$expression $\rangle$
then $\langle$statements $\rangle$
$\{$ **elseif** $\langle$expression$\rangle\}$
then $\langle$statements$\rangle\}$
$[$ **else** $\langle$statements$\rangle]$
end if

Optional, iterated, and bracketed expressions can be removed as follows:

$\langle$statements$\rangle \triangleq \langle$statement$\rangle\ \langle$statements rest$\rangle$

$\langle$statements rest$\rangle \triangleq \varepsilon$
$\mid \underline{;}\ \langle$statement$\rangle\ \langle$statements rest$\rangle$

$\langle$statement$\rangle \triangleq$ **if** $\langle$expression$\rangle$
then $\langle$statements$\rangle$
$\langle$else if part$\rangle$
$\langle$else part$\rangle$
end if

$\langle$else if part$\rangle \triangleq \varepsilon$
$\mid$ **elseif** $\langle$expression$\rangle$
then $\langle$statements$\rangle$
$\langle$else if part$\rangle$

$\langle$Else_Part$\rangle \triangleq \varepsilon$
$\mid$ **else** $\langle$statements$\rangle$

In this case, the new rules can be simplified:

$\langle$statements$\rangle \triangleq \langle$statement$\rangle$
$\mid \langle$statement$\rangle\ \underline{;}\ \langle$statements$\rangle$

$\langle$statement$\rangle \triangleq$ **if** $\langle$expression$\rangle$
then $\langle$statements$\rangle$
$\langle$else if part$\rangle$
end if

$\langle$else if part$\rangle \triangleq \varepsilon$
$\mid$ **else** $\langle$statements$\rangle$
$\mid$ **elseif** $\langle$expression$\rangle$
then $\langle$statements$\rangle$
$\langle$else if part$\rangle$

Example 3.11 (An EBNF *grammar for Expressions).* In EBNF, the grammar G_4 looks as follows:

$\langle$expression$\rangle \triangleq \langle$term$\rangle\ \{\underline{-}\ \langle$term$\rangle\}$

$\langle$term$\rangle \triangleq \langle$factor$\rangle\ \{\underline{*}\ \langle$factor$\rangle\}$

$\langle$factor$\rangle \triangleq \underline{\text{int}} \mid \underline{(}\langle$expression$\rangle\underline{)}$

3.2.2 Recursive Descent Parsing

For $SLL(1)$-Grammars, parsers can be constructed in a systematic way. They perform *recursive descent*. The construction can be done for arbitrary EBNF definitions.

Definition 3.13 (Constructing Recursive-Descent Parsers). Recursive descent works as follows:

1. For every definition $\langle n \rangle \triangleq B$ of a nonterminal by an EBNF expression B, a procedure n is generated that parses B.
2. The variable ℓ contains the next lexeme to be read.
3. There are two operations on ℓ:
   a) *scan* reads the next lexeme (overwriting the previous one).
   b) match(t) reads ℓ if ℓ and t are equal, and signals an error otherwise.

$$\text{match}(t) \triangleq \textbf{if } \ell = t \textbf{ then } \textit{scan} \textbf{ else } \textit{Error} \textbf{ end if}$$

4. Before n is called, variable ℓ contains a possible start of n.
5. After the call of n, ℓ contains a possible successor of n (or the program is erroneous; details follow).

The parsers are defined recursively over the structure of EBNF-expressions:

$$\begin{aligned}
[\![\langle n \rangle \triangleq B]\!] &\to \textbf{procedure } n \textbf{ is begin } [\![B]\!] \textbf{ end } n; \\
\forall t \in T : [\![t]\!] &\to \begin{cases} \textit{scan} & \text{if } t \text{ is the start of choice,} \\ & \quad \text{option, or iteration} \\ \textit{match}(t) & \text{otherwise} \end{cases} \\
\forall n \in N : [\![\langle n \rangle]\!] &\to n \\
[\![B_1 B_2 \cdots B_k]\!] &\to [\![B_1]\!]; [\![B_2]\!]; \ldots; [\![B_k]\!] \\
[\![B_1 \mid B_2 \cdots \mid B_k]\!] &\to \textbf{if } \ell \in \textit{Pref}^1(n \to B_1) \textbf{ then } [\![B_1]\!] \\
&\quad \textbf{elsif } \ell \in \textit{Pref}^1(n \to B_2) \textbf{ then } [\![B_2]\!] \\
&\quad \vdots \\
&\quad \textbf{elsif } \ell \in \textit{Pref}^1(n \to B_k) \textbf{ then } [\![B_k]\!] \\
&\quad \textbf{else } \text{Error } \textbf{end if} \\
[\![\{B\}]\!] &\to \textbf{while } \ell \in \textit{Fi}^1(B) \textbf{ loop } [\![B]\!] \textbf{ end loop} \\
[\![[B]]\!] &\to \textbf{if } \ell \in \textit{Fi}^1(B) \textbf{ then } [\![B]\!] \textbf{ end if} \\
[\![(B)]\!] &\to [\![B]\!]
\end{aligned}$$

Example 3.12 (A Recursive-Descent Parser for Expressions). The parser functions for G_4 in Example 3.9 can be generated as shown in Fig. 3.8.

3.2.3 Tree Construction During Recursive Descent

The recursive-descent parsers constructed so far just consume the input without producing anything except error messages. However, syntax analysis has to deliver an abstract syntax tree of the program. A tree can be constructed during parsing, by turning a parsing procedure into a function that returns a tree value.

Assuming that every alternative of a rule for some nonterminal n are numbered from 1 to m, the definition of a parser has to be changed as follows:

$$
\begin{aligned}
[\![\langle n\rangle \triangleq B]\!] &\to \textbf{procedure } n \textbf{ return } A_n \textbf{ is}\\
&\quad\ \textbf{begin } [\![B]\!] \textbf{ end } n;\\[1ex]
\forall n \in N : [\![\langle n\rangle]\!] &\to \textbf{var } a_i := n\\[1ex]
[\![B_1 \mid B_1 \cdots \mid B_n]\!] &\to \textbf{if } \ell \in \mathit{Pref}^1(n \to B_1)\\
&\quad\ \textbf{then } [\![B_1]\!];\ \textbf{return } \mathit{node}_1(a_{1,1}, \ldots, a_{1,m_1})\\
&\quad\ \textbf{elsif } \ell \in \mathit{Pref}^1(n \to B_2)\\
&\quad\ \textbf{then } [\![B_2]\!];\ \textbf{return } \mathit{node}_2(a_{2,1}, \ldots, a_{2,m_2})\\
&\quad\ \ldots\\
&\quad\ \textbf{elsif } \ell \in \mathit{Pref}^1(n \to B_n)\\
&\quad\ \textbf{then } [\![B_n]\!];\ \textbf{return } \mathit{node}_2(a_{n,1}, \ldots, a_{n,m_n})\\
&\quad\ \textbf{else } \text{Error};\ \textbf{return } \mathit{node}_\perp \textbf{ end if}
\end{aligned}
$$

$$
\begin{aligned}
[\![E \triangleq T\{\underline{-}\,T\}]\!] &\rightsquigarrow \textbf{procedure } E \textbf{ is begin } [\![T\{\underline{-}T\}]\!];\ \textbf{end } E;\\
[\![T\{\underline{-}\,T\}]\!] &\rightsquigarrow [\![T]\!]; [\![\{-T\}]\!]\\
&\rightsquigarrow T; \textbf{while } \ell = \underline{-} \textbf{ loop } [\![-T]\!] \textbf{ end loop}\\
&\rightsquigarrow T; \textbf{while } \ell = \underline{-} \textbf{ loop } \mathit{scan}; T \textbf{ end loop}\\[1ex]
[\![T \triangleq F\{\underline{*}\,F\}]\!] &\rightsquigarrow \textbf{procedure } T \textbf{ is begin } [\![F\{\underline{*}F\}]\!];\ \textbf{end } T;\\
[\![F\{\underline{*}\,F\}]\!] &\rightsquigarrow F; \textbf{while } \ell = \underline{*} \textbf{ loop } \mathit{scan}; F \textbf{ end loop}\\[1ex]
[\![F \triangleq \underline{\mathrm{i}} \mid (E)]\!] &\rightsquigarrow \textbf{procedure } F \textbf{ is begin } [\![\underline{\mathrm{i}} \mid (E)]\!];\ \textbf{end } F;\\
[\![\underline{\mathrm{i}} \mid (E)]\!] &\rightsquigarrow \textbf{if } \ell = \underline{\mathrm{i}} \textbf{ then } [\![\underline{\mathrm{i}}]\!] \textbf{ elsif } \ell = \underline{(} \textbf{ then } [\![(E)]\!] \textbf{ else } \text{Error} \textbf{ end if}\\
&\rightsquigarrow \textbf{if } \ell = \underline{\mathrm{i}} \textbf{ then } [\![\underline{\mathrm{i}}]\!]\\
&\qquad\ \textbf{elsif } \ell = \underline{(} \textbf{ then } [\![\underline{(}]\!]; [\![E]\!]; [\![\underline{)}]\!]\\
&\qquad\ \textbf{else } \text{Error} \textbf{ end if}\\
&\rightsquigarrow^4 \textbf{if } \ell = \underline{\mathrm{i}} \textbf{ then } \mathit{scan}\\
&\qquad\ \textbf{elsif } \ell = \underline{(} \textbf{ then } \mathit{scan}; E; \mathit{match}(\underline{)})\\
&\qquad\ \textbf{else } \text{Error} \textbf{ end if}
\end{aligned}
$$

Fig. 3.8. Constructing a recursive-descent parser for expressions

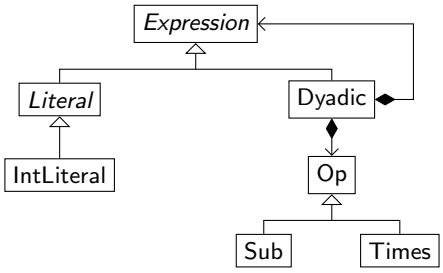


Fig. 3.9. Classes of an abstract syntax for expressions

The functions $node_i$ are constructors for the alternative i. If an alternative contains an option, the respective subtrees may be empty; if it contains an iteration, the corresponding subtrees are lists.

The syntax tree constructed in this way does still resemble a derivation tree. For every rule of the grammar, a node is built from the nodes built for the nonterminals and compound symbols (like identifiers and literals). For the subsequent phases of compilation, however, we need a tree representing the *semantic structure* of the program rather than its concrete syntax structure. Such a tree is called an *abstract syntax tree*; it abstracts from syntax aspects such as ease of parsing.

The grammar G_{E} in Example 3.2 has be transformed several times:

- We have made it unambiguous, by defining sub-expressions (terms and factors) in Example 3.7.
- We have eliminated left recursion in Example 3.9.
- Finally, we have left-factorized the grammar in Example 3.9 so that it fullfils the $SLL(k)$-condition.

Furthermore, we have used operators for the extended Backus-Naur form, in order to make the grammar more compact. All these steps have changed the rules considerably so that they do no longer represent the semantic structure of expressions, as the original grammar G_{E} did:

- *Bracketing* of expressions is represented by the hierarchical composition of syntax trees. That is why the bracketing rule needs not be represented explicitly in the syntax tree.
- The rules for expressions and terms are now right-associative whereas the subtraction and multiplication are *left-associative*.

Abstract syntax trees should rather represent the rules of G_1 than those of the grammar G_5 that has been used for parsing.

Example 3.13 (A Tree-Constructing Recursive-Descent Parser for Expressions). We need two constructor functions for representing abstract syntax trees of expressions:

```
[[E ≜ T {_ T}]] ⇝ procedure E return Exp is
                  begin var right : Exp;
                  var left : Exp := term;
                  while ℓ = _
                  loop scan; right := term;
                     left := Dyadic(left, minus, right);
                  end loop;
                  return left
                end E;

[[T ≜ F {* F}]] ⇝ procedure term return Exp is
                  begin var right : Exp;
                  var left : Exp := F;
                  while ℓ = *
                  loop scan; right := F;
                     left := dyad(left, times, right);
                  end loop;
                  return left
                  end term;

[[F ≜ i | (E)]] ⇝ procedure F return Exp is
                  begin  var e : Exp;
                  if ℓ = i then e := Intliteral(ℓ.val); scan;
                  elsif ℓ = ( then scan; e := E; match());
                  else Error... end if;
                  return e;
                  end F;
```

Fig. 3.10. Tree-constructing recursive descent parser for expressions

1. $Dyadic(e_1, op, e_2)$ constructs a dyadic expression with two operands e_1, e_2, and an operator Op that is just an enumeration type containg the Enumerands Sub and $Times$.
2. $IntLiteral(v)$ generates a literal with value v. (The value of the integer symbol.)

In Figure 3.9 we show the abstract syntax of expressions.

The tree-constructing parser functions for expressions may look as in Figure 3.10. In the loops of the parsers for E and T, the syntax trees are constructed so that left-associativity of the operators is respected.

3.2.4 Error Handling with Recursive-Descent Parsing

We have seen that an $SLL(k)$-parser detects an error before the first "wrong" symbol is read. Thus it is easy to give a meaningful error message, with the

```
procedure F return Exp is
  e : Exp;
  begin
  if ℓ = i then scan; e := literal(ℓ.val);
  elsif ℓ = ( then scan; e := E; match());
  else skipTo({*, −, ), □}); e := ErrorExp
  end if;
  return e
  end F;
```

Fig. 3.11. The recursive parser for factors, with error handling

exact position where the first erroneous symbol is seen. (This does not mean that thir is the real "reason" for the error – this may be a missing /superfluous symbol read earlier.)

The recovery from errors is based on operations that we know from the recovery of lexical errors, in Section 2.6.2:

- *Skipping* an unexpected symbol (sequence). Often, symbols are skipped until the parser finds a symbol at which the parser "knows" how to continue.
- *Insert* a symbol that is expected at this point.

Recovery can be inserted systematically into our framework for constructing recursive-descent parsers:

- When parsing some alternatives without finding a symbol that may be the first of some of these, symbols can be skipped up to a symbol that may be a follower of the alternatives.
- The method *match* inserts a symbol – virtually – if it is not found as expected.

Example 3.14 (Error Handling for the Recursive-Descent Parser of G_5). Only one rule – that for factors F – needs an extra error handling, see Figure 3.11. Let us assume that the method skipTo reads all symbols up to a set of symbols. We skip upto the symbol set $Fo^1(F)$, as it is determined in Table 3.3.

3.3 Bottom-up Parsing

Bottom-up parsers attempt to construct a reverse rightmost derivation (or a derivation tree) by constructing it from the bottom upwards, from the leaves to the root, matching the word as it is read, from left to right.

Definition 3.14 (Abstract Bottom-up Parser). The configurations of a bottom-up parser are denoted as $\alpha\bullet w$ where α is the stack of the parser,

containing the hypothesis used for parsing, and w is the suffix of the word that has still to be read.

The *start configuration* $\varepsilon\centerdot w_0$ has an empty stack, and the entire word w_0 is still to be read.

An *accepting configuration* $z\centerdot\varepsilon$ has read the entire input, and obtained the start symbol as a hypothesis.

The parser performs two kinds of actions:

1. *Shift* the first unread terminal symbol onto the stack:

$$\forall t \in T : \alpha\centerdot tw \vdash_t \alpha t\centerdot w$$

2. *Reduce* the right-hand side of a rule, which is on top of the stack, by its left-hand side:

$$\forall r = n \to \beta \in R : \alpha\beta\centerdot w \vdash_r \alpha n\centerdot w$$

Error configurations of bottom-up parsers take the form $\alpha\centerdot\varepsilon$ where the input is read entirely, but the stack does not start with the right-hand side of a rule. Then *backtracking* has to be done as follows:

1. Undo shift operations upto the latest configuration where a reduction has been possible. If a shift has been done in that configuration, do that reduction and continue. If a reduction has been done, mark it as unsuccessful, and do a reduction for another rule if it is possible. Continue backtracking otherwise.
2. If all reductions are marked as unsuccessful, in all configurations, the word is not in the language.

The parser satisfies the following invariant:

$$\forall w, \bar{w} \in T^* \; \forall \alpha \in V^* : \varepsilon\centerdot w\bar{w} \vdash^* \alpha\centerdot\bar{w} \Leftrightarrow \alpha \Rightarrow_R^* w$$

The reductions of the parser construct reverse rightmost derivations:

$$\forall w, \bar{w} \in T^* \; \forall \alpha, \beta \in V^* : \alpha\centerdot w \vdash_r \beta\centerdot w \Leftrightarrow \beta \overset{\text{rm}}{\Rightarrow}_r \alpha$$

Example 3.15 (Bottom-Up Parsing of Expressions). We return once more to the unambiguous grammar G_2 for expressions in Example 3.7: A bottom-up parse of the well-known expression $13 - 4 * 2$ may be as follows:

$$\begin{array}{cccccccc}
 & \varepsilon\centerdot 13 - 4 * 2 & \vdash_{\underline{\mathrm{i}}} & \underline{\mathrm{i}}\centerdot - 4 * 2 & \vdash_4 & E\centerdot - 4 * 2 & \vdash_- & E - \centerdot 4 * 2 \\
\vdash_{\underline{\mathrm{i}}} & E - \underline{\mathrm{i}}\centerdot * 2 & \vdash_4 & E - E\centerdot * 2 & \vdash_* & E - E * \centerdot 2 & \vdash_{\underline{\mathrm{i}}} & E - E * \underline{\mathrm{i}}\centerdot\varepsilon \\
\vdash_4 & E - E * E\centerdot\varepsilon & \vdash_3 & E - E\centerdot\varepsilon & \vdash_2 & E\centerdot\varepsilon & \vdash_1 & S\centerdot\varepsilon
\end{array}$$

If read from the end to the beginning, this constructs a rightmost derivation:

$$S \overset{\text{rm}}{\Rightarrow}_1 E \Rightarrow_2 E - E \overset{\text{rm}}{\Rightarrow}_3 E - E * F \overset{\text{rm}}{\Rightarrow}_4 E - E * \underline{\mathrm{i}} \overset{\text{rm}}{\Rightarrow}_4 E - \underline{\mathrm{i}} * \underline{\mathrm{i}} \overset{\text{rm}}{\Rightarrow}_4 \underline{\mathrm{i}} - \underline{\mathrm{i}} * \underline{\mathrm{i}}$$

Its is not obvious how bottom-up parsers can be made deterministic. Many configurations allow more than one operation to proceed. We distinguish two kinds of conflicts:

- In a configuration $\alpha\beta\centerdot aw$, the parser is in a *shift-reduce conflict* if it may reduce a rule $n \to \beta$, or shift the terminal symbol a onto the stack. This conflict exists as long as the input is not exhausted.
- In a configuration $\alpha\beta\gamma\centerdot w$, the parser is in a *reduce-reduce* conflict, if two different rules, $n \to \beta\gamma$ and $m \to \gamma$ can be reduced. In grammar G_2 of Example 3.7, this is the case in configurations $\alpha E - T\centerdot w$ and $\alpha T * F\centerdot w$, since then rules 2a and rule 2b (3a and 3b, resp.) can both be used. (In these cases, it is easy to see that only the first of these choices will lead to a successful parse.)

Shifting is always possible as long as there is further input; then, a reduction should be considered if the next symbol is a possible follower of the rule's right-hand side. This strategy has been applied in the parse in Example 3.15.

3.3.1 Simple $LR(k)$-Parsing

Deterministic bottom-up parsing algorithms for unambiguous context-free grammars go back to (*canonical*) $LR(k)$- *parsing* devised by D. E. Knuth [15]. The basic idea of this algorithm and its many derivates is the following:

- Distinguish *states* of the bottom-up parser by the set of hypotheses being followed in them.
- Follow these hypotheses *in parallel* until a choice of one of them is inevitable. This is the case if at least one hypothesis calls for a reduction.
- In such a state, it must be possible to take the decision between a shift and a reduction, or between different reductions in a deterministic way.

A single hypothesis is represented by an *item* of the form $n \to \alpha \centerdot \beta$, where $n \to \alpha\beta$ is a rule of the grammar in question, and the dot indicates that the part α of the rule's right-hand side has already been recognized. The items of a state are divided into its *kernel* and its *closure*. Roughly, the states of an $LR(k)$-parser are defined as follows:

1. The kernel of the start state of the parser consists of the item $z \to \centerdot\alpha\Box$. We assume that the start symbol has a single rule $z \to \alpha$; $\Box$ denotes the end of text.
2. Whenever some state contains an item $n \to \alpha \centerdot m\beta$ with a nonterminal m, items $m \to \centerdot\gamma$ are included in the closure of the state, for every rule of the form $m \to \gamma$.
3. Whenever some state contains an item $n \to \alpha \centerdot v\beta$ with a symbol $v \in V$, this state has a transition (under v) into a *goto-state*, the kernel of which contains the item $n \to \alpha v \centerdot \beta$.
4. The state containing the item $z \to \alpha \centerdot \Box$ is an *accepting* state.

The states, together with transitions, define what is called the *characteristic finite automaton* of a context free-grammar.

Definition 3.15 (Items and Characteristic Finite Automaton). Let $G = (V, N, R, z)$ be a context-free grammar wherein the start symbol z is defined by a single rule of the form $z \to \alpha\Box$, where the terminal symbol $\Box$ (that does not occur in other rules of R) shall represent the *end-of-text*.

Then $I_G = \{n \to \alpha \centerdot \beta \mid n \to \alpha\beta \in R\}$ is the set of *simple LR items* of G. For every set $\bar{I} \subset I_G$ of items, the *closure* I is obtained by applying the following rules until I is stable:

$$\frac{}{\bar{I} \subseteq I} \qquad \frac{n \to \alpha \centerdot m\beta \in I, m \to \gamma \in R}{m \to \centerdot \gamma \in I}$$

In the *characteristic finite state automaton* (*CFA*, for short) $A_G = (V, Q, \Delta, q_0, F)$, every state $q \in Q$ is defined by a set of items, i.e. $q \subseteq I_G$. It is defined as follows:

1. The start state is given as the closure of the singleton item set $\bar{q}_0 = \{z \to \centerdot \alpha\Box\}$.
2. If some $q \in Q$ contains an item $n \to \alpha \centerdot v\beta$ for some $v \in V$, Q contains a *goto-state* $q' = Goto(q, v)$ which is the closure of all items $\bar{q}' = \{n \to \alpha v \centerdot \beta \mid n \to \alpha \centerdot v\beta \in q\}$.
3. A state q containing a *final item* $z \to \alpha \centerdot \Box$ is in the set F of final states.

Note that the computation of closures and goto-states terminates since I_G is finite. So after some time, goto-states will be ones that have already been determined before. The simple CFA will not be directly used for recognition, but as an intermediate step to generate the parsers. Before that, it can be used to determine whether a parser can be generated at all.

Definition 3.16 (*SLR*(1) Conflicts). Let $A_G = (V, Q, \Delta, q_0, F)$ be the simple CFA of a context-free grammar G.

A state $q \in Q$ indicates a

- *shift-reduce* conflict if q contains items $n \to \alpha \centerdot a\beta$ and $m \to \gamma \centerdot$ so that $a \in Fo^1(m)$;
- *reduce-reduce* if q contains items $n \to \alpha \centerdot$ and $m \to \beta \centerdot$ so that $Fo^1(n) \cap Fo^1(m) \neq \emptyset$.

Then we say that G *has* $SLR(1)$ *conflicts*, and there is no $SLR(1)$ parser for G. Otherwise, G is an $SLR(1)$ grammar.

The classical way to defines parsers for $SLR(1)$ grammars is *table-driven*: Its CFA is used to generate two tables:

1. The *action table* A has rows labeled with states, columns labeled with terminal symbols. $\mathsf{A}[q, a]$ contains, for every state q and every terminal symbol a, one of the following entries:
   - "S q'" for the *shift* to state q';
   - "R r" for *reduce* according to rule r;
   - "Acc" for *accept*;
   - nothing otherwise – this indicates an error.

2. The *goto table* G has rows labeled with states, columns labeled with nonterminal symbols. $\mathsf{G}[q, n]$ contains, for every state q and every nonterminal symbol n, a successor state q' or nothing – again this indicates errors.

These tables are filled as defined below.

Definition 3.17 (*SLR*(1) Action and Goto Table, and Table Driver). Let $A_G = (V, Q, \Delta, q_0, F)$ be the simple CFA of a context-free grammar G.

The action table is filled according to the following rules:

$$\frac{n \to \alpha \,\bullet\, a\beta \in q, n \to \alpha a \,\bullet\, \beta \in q'}{\mathsf{A}[q, a] \leftarrow \mathrm{S}\, q'} \qquad \frac{n \to \alpha \,\bullet\, \in q, a \in Fo^1(n)}{\mathsf{A}[q, a] \leftarrow \mathrm{R}\,(n \to \alpha)} \qquad \frac{z \to \alpha \,\bullet\, \Box \in q}{\mathsf{A}[q, a] \leftarrow \mathrm{Acc}}$$

The goto table is filled according to the following rule:

$$\frac{n \to \alpha \,\bullet\, m\beta \in q, n \to \alpha m \,\bullet\, \beta \in q'}{\mathsf{G}[q, m] \leftarrow q'}$$

The parser works as follows, for every $SLR(k)$ table:

```
push(q0); scan;
while A[top, a] ≠ Acc
do  case A[top, a]
      of S q' ⇒ push(a); push(i); scan;
       | R r ⇒ for i := 1 to 2 * length(rhs(r_n)) do pop;
          z := top; push(lhs(r_n)); push(G[z, lhs(r_n)])
      otherwise syntaxerror("···");
      end case
end while;
if A[top, a] = Acc ∧ a = □
then print("yes!") else print("no!")
end if.
```

The definition of parsers can be extended to inspect up to $k > 0$ symbols of lookahead easily. However, $SLR(k)$-parsers require action tables of size $|Q| \cdot |T|^k$ which is inpractical, even if the action table is represented as a sparse matrix, e.g., an array indexed by states, which contains a linked list of actions.

The behavior of an $SLR(1)$-parser can be simulated by refining the configurations and moves of the general bottom-up parser of Definition 3.14.

Definition 3.18 (Abstract *SLR*(1)-parsers). In the configurations $\alpha \bullet w$ of an $SLR(k)$-parser, the stack α has states of the simple CFA in between the symbols that defined the stack of the general bottom-up parser. I.e., $\alpha \in Q \cdot (V \cdot Q)^*$.

In the *start configuration* $q_0 \bullet w_0$, the stack contains the start state q_0, and the entire word w_0 is still to be read.

An *accepting configuration* $q_0 z q \bullet \varepsilon$ has read the entire input, and obtained the start symbol as a hypothesis in a final state q.

The actions of the parser are refined as follows:

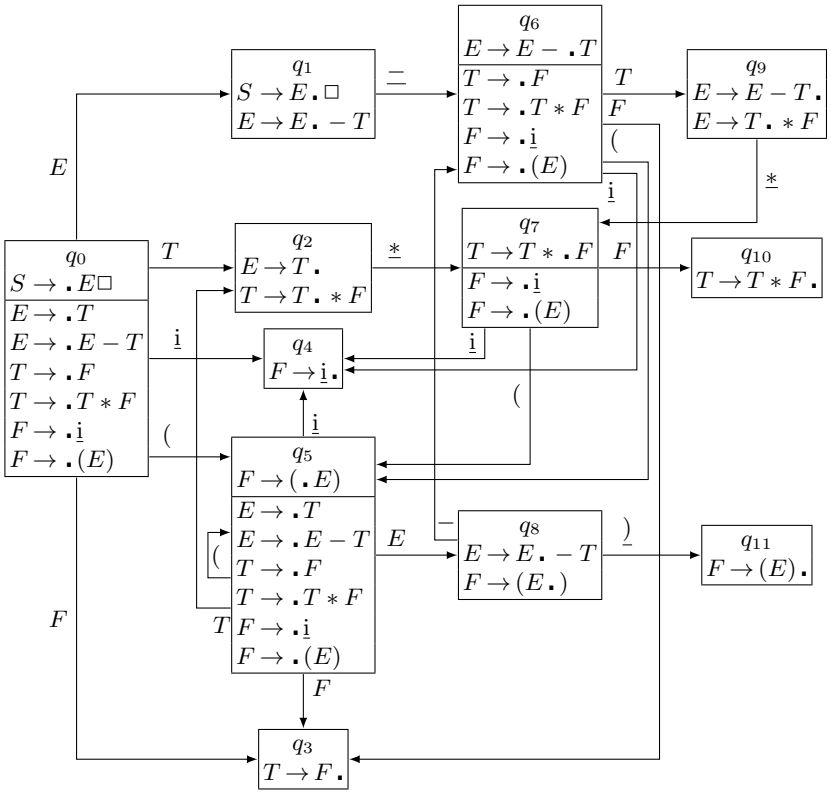


Fig. 3.12. Diagram of the simple CFA for G_2

1. *Reduce* the right-hand side of a rule, which is on top of the stack, by its left-hand side:

$$\forall r_i = n \to v_1 \ldots v_k \in R : \alpha\, q\, v_1 q_1 \ldots v_k q_k . tw \vdash_r \alpha\, q\, n\, q' . tw \quad \text{if } \mathsf{A}[q,t] = \mathsf{R}\ (r_i) \text{ and } q' = \mathsf{G}[q,n]$$

2. *Shift* the first unread terminal symbol onto the stack:

$$\forall t \in T : \alpha\, q . tw \vdash_t \alpha\, q\, t\, q' . w \text{ if } \mathsf{A}[q,t] = \mathsf{S}\ q'$$

Error configurations of $SLR(k)$- parsers take the form $\alpha q . aw$ where $\mathsf{A}[q,a]$ indicates an error.

We construct an $SLR(1)$-parser for grammar G_2 of expression see Theorem 3.4.

Example 3.16 (An $SLR(1)$-parser for G_2). Grammar G_2 has the start configuration

Table 3.4. An action and goto table for parsing expressions with G_2

State	Action						Goto		
q_i	$\underline{\square}$	$\underline{-}$	$\underline{*}$	$\underline{(}$	$\underline{)}$	$\underline{\mathrm{i}}$	E	T	F
0				S 5		S 4	1	2	3
1	Acc	S 6							
2	R 2a	R 2a	S 7		R 2a				
3	R 3a	R 3a	R 3a		R 3a				
4	R 4	R 4	R 4		R 4				
5				S 5		S 4	8	2	3
6				S 5		S 4		9	3
7				S 5		S 4			10
8		S 6			S 11				
9	R 2b	R 2b	S 7		R 2b				
10	R 3b	R 3b	R 3b		R 3b				
11	R 5	R 5	R 5		R 5				

$$q_0 = \left\{ \begin{array}{l} S \to \,.E\square, \\ E \to \,.T, \\ E \to \,.E - T, \\ T \to \,.F, \\ T \to \,.T * F, \\ F \to \,.\underline{\mathrm{i}}, \\ F \to \,.(E), \end{array} \right\}$$

The construction yields an automaton with twelve states; its transition diagram is shown in Figure 3.13.

G_2 is $SLR(1)$. For, in state q_9, there is no shift-reduce conflict, as $* \notin Fo^1(E) = \{\square, -,)\}$,[3] and analogously for states q_1 and q_2. (As in G_1, there are no candidates for reduce-reduce conflicts.)

So action and goto table can be constructed as shown in Table 3.4. (In the table entries, states and rules are represented by their numbers.)

$SLR(1)$-parsers have the simplest algorithm for constructing CFAs and action/goto tables, and they define the smallest subclass of grammars that can be deterministically parsed in a bottom-up fashion. The construction rejects many grammars (due to shift-reduce and reduce-reduce conflicts) that are unambiguous, and could be parsed deterministically.

The construction of $SLR(1)$-parsers can be generalized and refined in several ways. Then bigger subclasses of unambiguous context-free grammars can

[3] As determined in Table 3.2 of Example 3.9.

be parsed, but the constructions are more complicated, and may generate considerably bigger CFAs and action/goto tables.

(1) *Canonical LR-parsers* have states made from items $n \to \alpha \centerdot \beta \wr L$ where $L \subseteq T$ indicates the *lookahead* expected with the item. Then the closure of an item $n \to \alpha \centerdot m\beta \wr L$, will contain an item $m \to \centerdot \gamma \wr L'$ for the lookahead $L' = Fi^1(\beta) \cdot L$. Whenever a state contains an item $n \to \alpha \centerdot v\beta \wr a$, the kernel of its goto-state contains an item $n \to \alpha v \centerdot \beta \wr L'$, for every lookahead $L' = Fi^1(\beta) \cdot L$. Then, in a state q containing an extended item $n \to \alpha \centerdot \wr L$, a reduction of $n \to \alpha$ will only be inserted for the symbols L, which in general is a subset of $Fo^1(n)$ used with SLR-parser construction. So this construction will cause less conflicts in general, at the cost of a considerably bigger set of items, states, and tables – typically by a factor of 100 for realistic programming languages.

(2) $LALR$-parsers are best described as a compactification of LR-parsers:[4] they identify states that have the same action table entries. This reduces the number of states considerably, typically by a factor of 10 for realistic programming languages. However, they reject more grammars – for instance, if the goto table entries of identified states are in conflict.

(3) For every class of parsers, the lookahead can be increased to $k > 1$ symbols. This increases the effort for constructing the parser (determination of follower sets is expensive!), and also leads to bigger parsers.

It has turned out that longer lookahead is not worth its price; it has been shown, that every $LR(k)$ grammar can be transformed into an $LR(1)$ grammar. (This does not hold for $LARL$ and SLR-parsers, however!) The step from $SLR(1)$ to $LALR(1)$ or $LR(1)$ is more useful for grammars of programming languages. In practice, $LALR(1)$-parser construction is the method of choice; it used by most generators of bottom-up parsers because it has moderate space requirements and reasonable generality.

3.3.2 Operator-Precedence Parsing of Ambiguous Grammars

Deterministic bottom-up parsing may be generalized in another way that is practically more attractive: neither does it make the construction more complex, nor does it lead to bigger parse tables.

The idea is to associate priorities with the rules of a grammar. Let $r \rhd \bar{r}$ denote that r has a priority higher than $\bar{r}$. Whenever a conflict arises in the CFA, it is solved by inspecting priorities:

- In a shift-reduce conflict between items $n \to \alpha \centerdot a\beta$ and $m \to \gamma \centerdot$ (with $a \in Fo^1(m)$),
  - choose the shift if $(n \to \alpha a\beta) \rhd (m \to \gamma)$, and
  - choose the reduction of $m \to \gamma$ otherwise.
- In a reduce-reduce conflict between items $n \to \alpha \centerdot$ and $m \to \beta \centerdot$ (where $Fo^1(n) \cap Fo^1(m) \neq \emptyset$), choose the rule with the higher priority for reduction.

[4] The real construction directly proceeds from the grammar, without doing the expensive LR construction. Rather, it extends the CFA constructed for an SLR-parser by the lookahead sets.

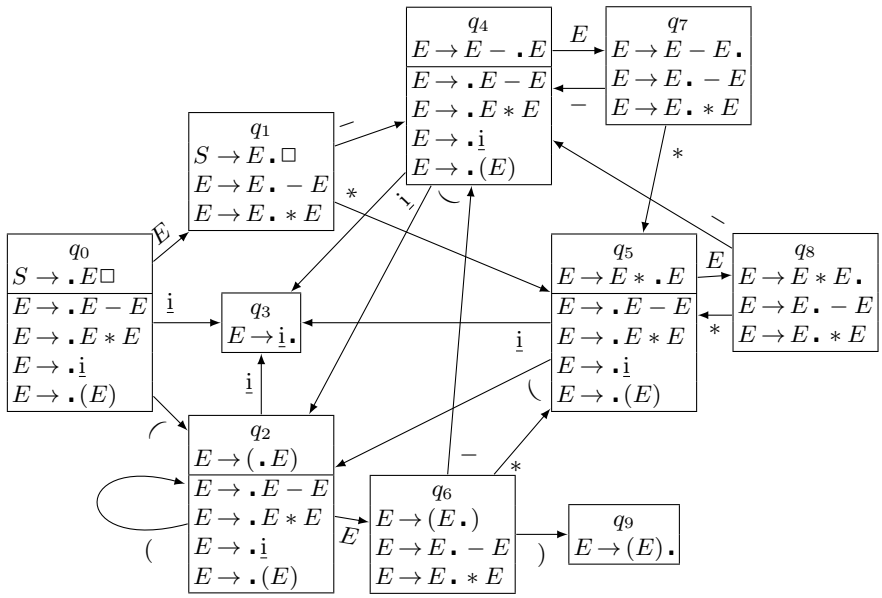


Fig. 3.13. Diagram of the CFA for G_1

To solve all conflicts of a grammar, the priorities must be defined for all rules causing conflicts. Then parsers can be constructed even for ambiguous grammars.

Example 3.17 (Characteristic Finite Automaton for G_1). For grammar G_1 of Example 3.2, the kernel of the start state is $\bar{q}_o = \{S \to .E\}$. This gives the closure

$$q_0 = \{S \to .E\square, E \to .E - E, E \to .E * E, E \to .\underline{i}, E \to .(E)\}$$

Successor states can be reached by reading symbols E, $\underline{(}$ and $\underline{i}$, resp.):

$$q_1 = \{S \to E., E \to E. - E, E \to E. * E\}$$
$$q_2 = \{E \to (.E), E \to .E - E, E \to .E * E, E \to .\underline{i}, E \to .(E)\}$$
$$q_3 = \{E \to \underline{i}.\}$$

By reading $\underline{-}$ and $\underline{*}$, resp., q_1 leads to following states:

$$q_4 = \{E \to E - .E, E \to .E - E, E \to .E * E, E \to .\underline{i}, E \to .(E)\}$$
$$q_5 = \{E \to E * .E, E \to .E - E, E \to .E * E, E \to .\underline{i}, E \to .(E)\}$$

By reading E in state q_2, one reaches the state

$$q_6 = \{E \to (E.), E \to E. - E, E \to E. * E\}$$

and, by reading $\underline{(}$ and $\underline{i}$, resp., one reaches states q_2 and q_3, which have already been determined before. By reading q_4 and q_5, resp., one reaches the state:

$$q_7 = \{E \to E - E\,\bullet, E \to E\,\bullet - E, E \to E\,\bullet * E\}$$
$$q_8 = \{E \to E * E\,\bullet, E \to E\,\bullet - E, E \to E\,\bullet * E\}$$

From these states, reading $\underline{-}$ and $\underline{*}$, resp., reaches the states q_4 and q_5 that are already known. Finally, reading $\underline{)}$ in state q_6 leads to the state

$$q_9 = \{E \to (E)\,\bullet\}$$

Now the states are complete. Figure 3.13 shows the transition diagram of the automaton.

G_0 is not $SLR(1)$; as a matter of fact, it cannot be $LR(k)$ (for any $k > 0$) because we have seen in Example 3.2 that it is ambiguous. State q_7 has a shift-reduce conflict: With lookahead $\underline{-}$ and $\underline{*}$, a shift to q_4 and q_5, resp., is possible, as well as a reduction of rule $E \to E - E$. A similar conflict occurs in state q_8, where rule $E \to E * E$ could be reduced.

We define the following priorities for the rules occurring in these conflicts:
- $(E \to E * E) \rhd (E \to E - T)$.
- $(E \to E * E) \rhd (E \to E * T)$.
- $(E \to E - E) \rhd (E \to E - T)$.[5]

Now all conflicts can be solved, and the action and goto tables can be constructed as in Table 3.5. Furthermore, they are solved in a way respecting the priorities of the operations, and their left-associativity. Thus the parse trees are "semantically correct".

The parser generator `yacc` and its descendants constructs $LALR$ parsers with priorities. Without going into details of its specification language, grammar G_1 could be specified in `yacc` as follows:

```
%token int

%left "-"
%left "*"

%%
E : E "-" E      { }
  | E "*" E      { }
  | "(" E ")"    { }
  | int          { }
  ;
```

In `yacc`, associativity and priorities are specified for operators (or keywords, in general) that occur in rules. Here, both operators are left-associative. The order of these associativity specifications defines their priorities, in ascending order.

[5] Thus $\rhd$ is not an order relation because it is not reflexive.

Table 3.5. An action and goto table for parsing expressions with G_1

State	Action						Goto
q_i	$\underline{\square}$	$\underline{-}$	$\underline{*}$	$\underline{(}$	$\underline{)}$	$\underline{\mathrm{i}}$	E
0				S 2		S 3	1
1	Acc	S 4	S 5				
2				S 2		S 3	6
3	R 3	R 3	R 3		R 3		
4				S 2		S 3	7
5				S 2		S 3	8
6		S 4	S 5		S 9		
7	R 1	R 1	S 5		R 1		
8	R 2	R 2	R 2		R 2		
9	R 4	R 4	R 4		R 4		

3.3.3 Constructing Abstract Syntax Trees

Construction of syntax trees in bottom-up parsers is easily done using an extra *semantic stack* that holds the (root nodes of the) trees of the symbols on its parse stack. Whenever the parser performs a reduction, it can pop the trees of the right-hand side symbols, and use them to construct a node for the rule that is then pushed onto the semantic stack.

Example 3.18 (Tree Construction for Expressions). In `yacc`, the parser for expressions specified in Example 3.16 could be extended by tree-constructing directives as follows.

```
%token id

%% /* Syntax     { semantic actions          } */

E : T             { $$ = $1;                  }
  | E "-" T       { $$ = Dyadic($1, Minus, $2);  }
  ;

T : F             { $$ = $1;                  }
  | T "*" F       { $$ = Dyadic($1, Times, $2); }
  ;

F : "(" E ")"     { $$ = $1;                  }
  | intliteral    { $$ = IntLiteral($1);                }
  ;

%% /* C functions defining semantic actions */
```

Here *Dyadic*, *IntLiteral* , *Minus*, and *Times* are constructors of the abstract syntax specified in Figure 3.9. The notation *$i* refers to the (root of the) tree of the ith right-hand side symbol, and *$$* is the tree corresponding to the left-hand side nonterminal.

3.3.4 Error Handling in *LR*-Parsers

An *LR*-parser recognizes an error if a state does not permit a shift or reduce action. At such a point, a reasonable error message can be given in any case. The items of the state, and the symbol to be read provide enough information to report the problem.

Error handling – in order to continue parsing, after inserting a symbol, or after skipping some symbols – is a bit more complicated, as the parser construction algorithm has to be extended by mechanisms for specifying error handling. See the `yacc` specification [13] for *error productions* which allow to specify error handling.

Bibliographical Notes

[12] is a standard introduction to automata theory and formal languages. Chapter 5 treats context-free grammars and languages, and Chapter 6 treats pushdown automata. The schemata for generating recursive descent parsers has been taken from [20, Section 4.3.3].

(*Canonical*) $LR(k)$ *parsing* has been devised by D. E. Knuth [15]. Franklin D. DeRemer has proposed *simple* $SLR(k)$ parsing [10] and $LALR(k)$ parsing [9]. Aho, Johnson and Ullman have improved these algorithms by priority specifications so that even ambiguous grammars can be parsed [2]. This is the method used in `yacc` and its many successors.

Excercises

Solutions to some of these exercises can be found on page 158 *ff*.

Exercise 3.1. Consider the following context-free rules, where terminals are underlined, and R is the start symbol:

$$
\begin{array}{lll}
R \rightarrow & R\,\underline{|}\,R & \\
\mid & R\,R & \\
\mid & R\,\underline{+} & \mid\ R\,\underline{*}\ \mid\ R\,\underline{?} \\
\mid & \underline{(}\,R\,\underline{)} & \\
\mid & \underline{\mathtt{a}}\ \mid\ \underline{\mathtt{b}}\ \mid\ \underline{\mathtt{c}} &
\end{array}
$$

Tasks

1. Which language is defined here?
2. Show that the grammar is ambiguous.
3. Make it unambiguous. Derivations and derivation trees shall respect that the precedences of the operators increase from top to bottom. The operators $\underline{*}$, $\underline{+}$ and $\underline{?}$ have the same precedence.

Exercise 3.2. The (unrevised) definition of Algol-60 contains rules for conditional commands as follows:

$$C \triangleq \underline{\mathrm{r}} \mid \textbf{if } E \textbf{ then } C \mid \textbf{if } E \textbf{ then } C \textbf{ else } C$$
$$E \triangleq \underline{\mathrm{e}}$$

Here the terminals $\underline{\mathrm{r}}$ and $\underline{\mathrm{e}}$ abstract from the rules for the remaining commands and for expressions, respectively.

Tasks

1. Show that the grammar is ambiguous.
2. Make it unambiguous. Derivations and derivation trees shall group an else part "dangling" between two preceeding then parts with the closest preceding one:

$$\textbf{if } E \textbf{ then if } E \textbf{ then } C \textbf{ else } C = \textbf{if } E \textbf{ then } (\textbf{if } E \textbf{ then } C \textbf{ else } C)$$

Exercise 3.3. Let the syntax of commands be defined as follows:

$$\begin{aligned}
\mathsf{program} &\triangleq \mathsf{statement}\ \underline{\square} \\
\mathsf{statements} &\triangleq \mathsf{statement} \\
&\quad \mid \mathsf{statement}\ \underline{;}\ \mathsf{statements} \\
\mathsf{statement} &\triangleq \textbf{if}\ \mathsf{expression}\ \textbf{then}\ \mathsf{statements}\ \textbf{end if} \\
&\quad \mid \textbf{if}\ \mathsf{expression}\ \textbf{then}\ \mathsf{statements}\ \textbf{else}\ \mathsf{statements}\ \textbf{end if} \\
&\quad \mid \textbf{while}\ \mathsf{expression}\ \textbf{do}\ \mathsf{statements}\ \textbf{end while} \\
&\quad \mid \mathsf{id}\ \textbf{:=}\ \mathsf{expression} \\
&\quad \mid \mathsf{id} \\
\mathsf{expression} &\triangleq \underline{\mathrm{e}}
\end{aligned}$$

Tasks

1. Check whether the grammar satisfies the $SLL(1)$ condition.
2. Determine the *First* and *Follower* sets for that purpose.
3. If the condition is violated, transform the grammar by left factorization until it does satisfy it.

Exercise 3.4. Consider the $SLL(1)$ syntax for commands from the previous exercise.

Tasks

1. Generate a recursive descent parser.
2. Construct an abstract syntax for commands.
3. Add tree-constructing instructions to the parser.
4. Incorporate error handling into the parser, based on deletion and insertion of symbols.

Exercise 3.5. Construct the CFA for the grammar with the following rules

$$S \to L \underline{=} R, S \to R, L \to \underline{*}\, R, L \to \underline{\mathrm{x}}, R \to L$$

(L and R are rudely simplified left-hand and right-hand sides of assignments in a C-like language.)

1. Identify the states with conflicts.

2. Identify which lookahead symbols may *really* occur in the reduction that causes the shift-reduce conflict. (This is the idea underlying $LALR$ parser construction.)

Exercise 3.6. Use Table 3.5 for grammar G_1 to parse, with the abstract LR parser in Definition 3.14, the words

- $a - b * c\Box$
- $(a - b) * c\Box$
- $(a - b)c\Box$
- $(a - b) * -c\Box$

Here a, b, c are numbers.

Exercise 3.7. Variables are expressions that may occur on the left-hand side of assignments (called *lvalues* in C). The evaluation of a variables yields the address of a value. In a grammar for a typical imperative programming language, the grammar G_7 for variables could be given by the following rules:

$$\begin{array}{ll}
S \to X & (1)\\
X \to \underline{\mathrm{x}} & (2)\\
X \to X\,\underline{\uparrow} & (3)\\
X \to X\,\underline{.}\,\underline{\mathrm{X}} & (4)\\
X \to X\,\underline{[}\,E\,\underline{]} & (5)\\
X \to X\,\underline{(}\,A\,\underline{)} & (6)\\
A \to E & (7)\\
A \to E\,\underline{,}\,A & (8)\\
E \to \underline{\mathrm{e}} & (9)
\end{array}$$

Tasks

1. Why is the grammar not $SLL(k)$?
2. Transform it into an equivalent $SLL(k)$ grammar
3. Generate a recursive descent parser for it.
4. Use the abstract top-down parser of Definition 3.8 to parse the variables
   - $x[e] \uparrow .s$

- $x(e, e).s$

5. Design an abstract Syntax, based on the original rules, and extend the recursive-descent parser so that it constructs abstract syntax trees.
6. Rewrite the (original) rules so that they are accepted by `yacc`. Is the grammar $SLR(k)$?
7. Use the abstract $LR(k)$-parser of Definition 3.14 to parse the variables in item 4.

Exercise 3.8. Turn the grammar for variables (*lvalues*) in Exercise 3.7 into a definition for `yacc` (or `bison`). Is the grammar $LALR(1)$?

Exercise 3.9. Turn the (original) grammar for statements in Exercise 3.3 into a definition for `yacc` (or `bison`).

The grammar is not $LALR(1)$ as it is ambiguous. Resolve ambiguity so that an **else** always is associated with the *latest* **then**. (`yacc`-parsers choose rules for reduction in the order in which they appear in the grammar.)

4

Context Analysis

Context analysis is the last sub-phase of the analysis phase of a compiler. This sub-phase has to check all rules concerning the source language that could be checked, neither by lexical analysis, nor by syntax analysis, but which can be determined before the program is (compiled and) executed. The *contextual constraints* of a language are called *static semantic rules* in some textbooks. This is because the formal semantics of programming language is often defined over the (context-free) syntax as it is determined in the syntax analysis; the part of the semantics that can be determined before execution is called static.

The contextual constraints of a language can be described with *attribute grammars*, which associate (static semantic) values to the nodes of syntax trees. An attribute grammar definition can be translated into an attribute evaluator that computes these values in an efficient way, by traversing the syntax tree. The number of traversals needed, and their direction, depends on the definition of contextual constraints.

In this chapiter, we first identify typical tasks of context analysis, introduce the terminology used for these tasks (Section 4.1), and introduce attribute grammars informally (Section 4.2). Then we describe two tasks in detail:

- declaration analysis (Section 4.3) and
- type analysis (Section 4.5).

In between we deal with the efficient implementation of declaration tables (Section 4.4), which are needed for declaration analysis, and finally, we describe how attribute grammar definitions are translated into evaluators, in functional and object-oriented languages (Section 4.6).

4.1 The Task

We use a small example to explain the tasks of context analysis.

Example 4.1 (Contextual Constraints for an Assignment). Consider the following assignment in an imperative language:

$\ldots a[i] := f(1); \ldots$

This piece of program text is *well-formed* only if the following contextual constraints are satisfied:

1. The identifiers a, i, and f must have a valid declaration at this point.
2. More specifically, a must be bound to an array variable, not to a constant, as it is updated. Let the *type* of a be $E[I]$, where I is the *index type*, and E the *element type* of a.
3. The identifier i must be bound to a variable or constant. Its type must be *compatible* to the index type I of a.
4. The index type I must be compatible with the type of the literal 1 (which will be something like Integer).
5. The identifier f must denote a function with one parameter. Its type has the form $P \to R$, where P is the parameter type, and R is the result type.
6. The parameter type P must be compatible to the type of the literal 1 (presumably Integer as well).
7. Finally, the element type E of a must be compatible with the result type R of f.

This example demonstrates two main tasks of context analysis:

- *Declarations analysis* – also called *identification* – checks whether every *use* of a name x has a corresponding *declaration* that *binds* x to some program entity.
- *Type analysis* checks the type rules of the language, i.e., whether the arguments of operations and procedures are compatible with the types of their formal parameters.

Programming languages are block-oriented. *Blocks* are program units that are nested (like packages, modules, classes, bodies of procedures, methods, functions, loops, choices, or structures), in which *local* declarations can be introduced. A declaration is said to *bind* an entity to a name. In a block, names can be declared anew, even if they are bound in the surrounding block. Then the local declaration *hides* the outer declaration of this name.

Programming languages are *statically typed*. Every entity (constant, variable, parameter, procedure etc.) is declared with a type, so that these entities have or yield values of this type during program execution.[1] In the expressions of a program, it must be checked whether the type constraints are satisfied.

Context analysis extends the abstract syntax tree of a program with cross references, e.g., from uses of names to the declaration binding it, and from expressions to their type definitions. The resulting *syntax graph* may well be cyclic if the program contains recursive types and procedures.

[1] In some languages, e.g., in object-oriented ones, the situation may be more complicated: there, the *dynamic type* of a value may be an arbitrary extension (subtype) of the *static type* of the entity that contains or yields it.

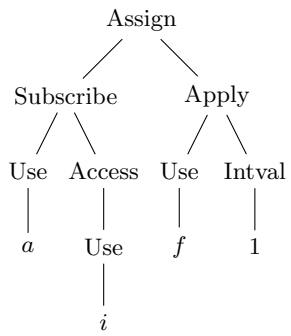


Fig. 4.1. Syntax tree...

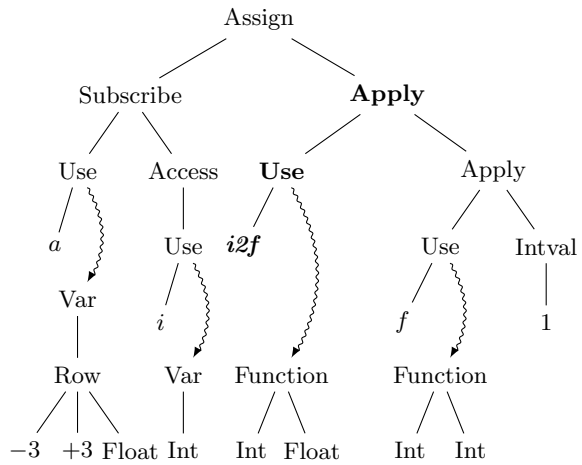


Fig. 4.2. ...and syntax graph for $a[i] := f(1);$

The underlying abstract syntax tree may also be extended in the context analysis: a subtree representing the expression "$2 + 3.14$" will presumably be extended to a subtree representing "$i2f(2) + 3.14$" ; i.e., the implicit type conversion of the integer 2 into a floating point number 2.0 is made explicit.

Example 4.2 (Context Analysis for an Assignment). For the assignment "$a[i] := f(1);$", syntax analysis could yield the subtree shown in Figure 4.1. Declaration analysis adds, to every Use node, a cross reference to its declaration (in the context of the code fragment). In our example, this could yield the following bindings: $I = \mathsf{Int}$, $P = \mathsf{Int}$, $R = \mathsf{Int}$ and $E = \mathsf{Float}$.

Type analysis uses these cross references to check whether the type rules of the languages are satisfied. This is the case, since we assume that integers (1) can be assigned to a variable ($a[i]$) containing floating point numbers. For the translation of the assignment, an explicit type conversion ("*i2f*") is added this subtree, which now represents the assignments "$a[i] := i2f(f(1))$".

This results in a subgraph shown in Figure 4.2.

Further transformations of the syntax tree, e.g., the replacement of a subexpression "$2+3.14$" by the literal "6.24", are the task of *global code improvement* and is not covered in this text.

Resource allocation is closely related to declaration analysis; it determines addresses and numbers needed for code generation, like the size of the values of some type, the address of a variable, or the size of the entities declared in a block. This will be discussed in the chapters of code generation, chapter 5 and chapter 6.

4.2 Attribute grammars

Attribute grammars can be used to specify syntax-directed transductions as discussed in Figure 3.2, and to generate *attribute evaluators* implementing these transductions. In an attribute grammar, the underlying context-free grammar specifies the abstract syntax of the language to be translated, and the nodes of the abstract syntax are attributed with values, which are computed according to semantic equations that are associated with the rules if the abstract syntax.

4.2.1 The Semantic Basis

The semantic equations of an attribute grammar are defined using a semantic basis. In implementations, the basis will be given as a collection of libraries, written in the language of the attribute evaluator.

Abstractly, they are given as a many-sorted algebra.

Definition 4.1 (Algebra). Let $\Sigma = \langle S, \Omega \rangle$be a *signature*, consisting odff afinite set S of sorts, and of a family of *operators* $\Omega = (\Omega_{w,s})_{w \in S^*, s \in S}$ typed over the sorts S. An operator *op* from $\Omega_{w,s}$ will be specified as $op\colon w \to s \in \Omega$; symbols in $\Omega_{\varepsilon,s}$ ($s \in S$) are called *constant*, and will be denoted as, e.g., c.

A *many-sorted algebra* over Σ (Σ-algebra, for short) $\mathbf{B}$ consists of a family $B = (B_s)_{s \in S}$ of non-empty *carrier sets* for the sorts of S, and of functions $op_B\colon B_{s_1} \times \cdots B_{s_k} \to B_{s_0}$ for every operator $op\colon s_1 \cdots s_k \to s_0$ (where $s_i \in S$ for $0 \leqslant i \leqslant k$). Constant symbols $c\colon \to s$ will be associated with a value $c_B \in B_s$.

Definition 4.2 (Terms). Die *terms* over Σ and over a family $X = (X_s)_{s \in S}$ of sorted variable names (Σ-terms, for short) are the least family of sets $T_\Sigma(X) = (T_\Sigma(X)_s)_{s \in S}$ for that hold:

- $X_s \subseteq T_\Sigma(X)_s$ for every $s \in S$.
- $op(t_1, \ldots, t_k) \in T_\Sigma(X)_s$ for every $op\colon s_1 \cdots s_k \to s \in \Omega$ and for all terms $t_i \in T_\Sigma(X)_{s_i}$, $1 \leqslant i \leqslant k$, $k \geqslant 0$. Constant terms $c()$ für $c \in \Omega_{\varepsilon,s}$ are written without their parentheses.

A family $\alpha = (\alpha_s\colon X_s \to B_s)_{s \in S}$ of functions is called a *value assignment*; it can be lifted to an *evaluation function* on terms, which is denoted by the same symbol, and is defined with

- $\alpha(c) = c_B$ for every constant $c\colon \to s$, $s \in S$.
- $\alpha_s(op(t_1, \ldots, t_k)) = op_B(\alpha_{s_1}(t_1), \ldots, \alpha_{s_k}(t_k))$ for every operator $c\colon s_1 \cdots s_k \to s$, and for every term $t_i \in T_\Sigma(X)_{s_i}$ $s_i, s \in S$, $1 \leqslant i \leqslant k$.

Definition 4.3 (Attribute Grammar). An *attribute grammar* $G = \langle G_{\mathrm{u}}, \mathbf{B}, A, E \rangle$ consists of components as follows:

- an *unterlying context-free grammar* $G_{\mathrm{u}} = \langle V, M, R, z\rangle$, with rules of the form $n_0 \to t\, n_1 \ldots n_k$;[2]
- a Σ-Algebra $\mathbf{B}$ as its *semantic basis*;
- a family $A = (A_s)_{s\in S}$ of *attribute names* for sorts. To every nonterminal $n \in N$ we associate a set $A_s(n) \subseteq A$ of attribute names; we assume that $A_s(z) = \emptyset$ for every sort $s \in S$.
- for every rule $r = n_0 \to t\, n_1 \cdots n_m \in R$ (with $n_i \in N$ and $t \in V \setminus N$ for $0 \leqslant i \leqslant m$), there is a family

  $$E(r) = (E_s(r) \subseteq T_\Sigma(X(r))_s \times T_\Sigma(X(r))_s)_{s\in S}$$

  of *semantic equations* over the family

  $$X(r) = (X_s(r) = \{a.m \mid a \in A_s(n_m), 0 \leqslant i \leqslant k\})_{s\in S}$$

  of *attribut variables.*

a tree T is an *attributied tree* of G if its nodes are labeled with terminal symbols $V \setminus N$, values vor lexemes, and values of the semantiscc basis $\mathbf{B}$, and if its edges are labeled with positive numbers, so that the following properties hold:

1. every inner node k_0 in T iis labeled wit hthe terminal symbol t of a rule $n_0 \to t\, n_1 \ldots n_m$, and is the source of edges labeled with t $1 \ldots m$to nodes $k_1, \ldots, k_m$, which are labeled with symbols $v_1, \ldots, v_m \in V \cup \{\varepsilon\}$ so that $n_0 \to v_1 \cdots v_m \in R$, and if there is an edge labeled with an attribute name $a \in A_s$ from k_0 to a node k_a, which is labeled with a values $b \in B_s$ whenever $a \in A_s(n_0)$.
2. The root ϱ of T is labeled with the $z \in N$.
3. every inner node k_0 with its children $k_1, \ldots, k_m$ *satisfies all attribute equations* $t = u \in E(r)$; i.e., for the value assignment defined with α with $\alpha(a.i) = k_i.a$ for all $a \in A(n_i)$, $1 \leqslant i \leqslant m$, it holds that $t^\alpha = u^\alpha$.

Example 4.3 (First Attribute Grammar for Expressions). We shall define an attribute grammar for evaluating expressions like

(4+9) ∗ (x–3–2)

The concrete syntax is

$$S \triangleq \textbf{eval}\ E$$
$$E \triangleq Num \mid E + E \mid E - E \mid E * E \mid E \div E \mid (E)$$

The corresponding abstract syntax is

[2] These rules define nodes of the abstract syntax trees rather than words of the concrete syntax.

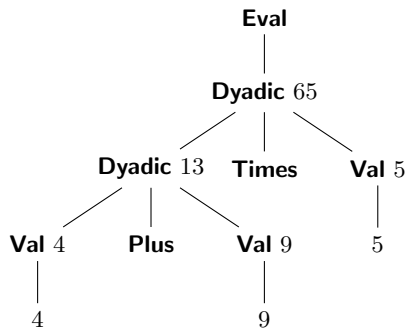


Fig. 4.3. An attributed tree for an expression

$S \triangleq$ **Eval** E

$E \triangleq$ **Val** Int | **Dyadic** $E\ O\ E$

$O \triangleq$ **Plus** | **Minus** | **Times** | **Div**

The semantical basis contains sorts *Int*, with operators: $(+), (-), (*), (\div)$ :: *Int* $\rightarrow$ *Int* $\rightarrow$ *Int*. The carrier sets are the integers $\mathbf{B}_{\mathsf{Int}}$. Attribute names are $A_{\mathsf{Int}}(E) = \{v\}$, and $A_{\mathsf{Int}}(O) = \{\}$.

The semantic rules are depicted in Table 4.1.

Figure 4.3 zeigt einen attributierten Baum für den Ausdruck $(4 + 9) * 5$, der die semantischen Gleichungen erfüllt.

Example 4.4 (Second Attribute Grammar for Expressions). We shall extend the attribute grammar for expressions in Example 4.3 with (nested) variable definitions like

let $x \triangleq 4+9$ **in** **let** $y \triangleq x-3$ **in** $x * (y-2)$

Table 4.1. Semantic rules the evaluation of expressions

syntax	*semantic rules*
$S \triangleq$ **Eval** E	
$E \triangleq$ **Val** n	$E.v = n$
| **Dyadic** $E_1\ O\ E_2$	$E.v =$ **case** O **of**
	Plus $\rightarrow E.v = E_1.v + E_2.v$
	Minus $\rightarrow E.v = E_1.v - E_2.v$
	Times $\rightarrow E.v = E_1.v * E_2.v$
	Div $\rightarrow E.v = E_1.v \div E_2.v$

Table 4.2. Semantic rules the evaluation of exprssions

syntax	*semantic rules*
$S \triangleq$ **Eval** E	$E.\ell = \mathsf{initial}$
$E \triangleq$ **Let** $x\ E_1\ E_2$	$E_1.\ell = E.\ell;\ E_2.\ell = \mathsf{enter}\ (x, E_1.v)(E.\ell);\ E.v = E_2.v$
$\mid$ **Op** $E_1\ O\ E_2$	$E_1.\ell = E.\ell;\ E_2.\ell = E.\ell;$
	$E.v =$ **case** $O.r$ **of**
	Plus $\rightarrow E.v = E_1.v + E_2.v$
	Minus $\rightarrow E.v = E_1.v - E_2.v$
	Times $\rightarrow E.v = E_1.v * E_2.v$
	Div $\rightarrow E.v = E_1.v \div E_2.v$
$\mid$ **Val** n	$E.v = n$
$\mid$ **Use** x	$E.v = \mathsf{def}\ x\ (E.\ell)$

The concrete syntax is

$$S \triangleq \textbf{eval}\ E$$
$$E \triangleq \textbf{let}\ Id = E\ \textbf{in}\ E \mid Id$$
$$\mid\ E + E \mid E - E \mid E * E \mid E \div E \mid (E) \mid Num$$

The corresponding abstract syntax of E is extended to

$$E \triangleq \textbf{Let}\ Id\ E\ E \mid \textbf{Use}\ Id \mid \textbf{Dyadic}\ E\ O\ E \mid \textbf{Val}\ Int$$

The semantical basis is extended by sorts *Id* of identifiers, and by *TAB* (of tables associating identifiers with values) with the following operators:

$initial :: Tab$
$enter :: (Id, Int) \rightarrow Tab \rightarrow Tab$
$def :: Id \rightarrow Tab \rightarrow Maybe\ Int$

The new carrier set are association lists of identifiers with values $\mathbf{B}_{\mathsf{Tab}}$, with corresponding operations, where $initial_{\mathbf{B}}$ is the empty list of pairs, $enter_{\mathbf{B}}$ adds a pair to a list, and $def_{\mathbf{B}}$ yields the value defined for an identifier. Since this definition may not exist, the result of the function is *Maybe Int*.

$A_{\mathsf{Tab}}(E) = \{\ell\}$ is added as a new attribute name.

The semantic rules are depicted in Table 4.2.

Figure 4.4 shows an attributed tree for the expression $\underline{\text{let}}\ x = 4 + 9\ \underline{\text{in}}\ x * 5$ that satisfies the semantic equations.

In order to determine an attributed tree for some given syntax tree, one had to try to find solutions for the attribute equations of the tree. In general, this is dificult and not efficient. In the following, we therefore assume that the

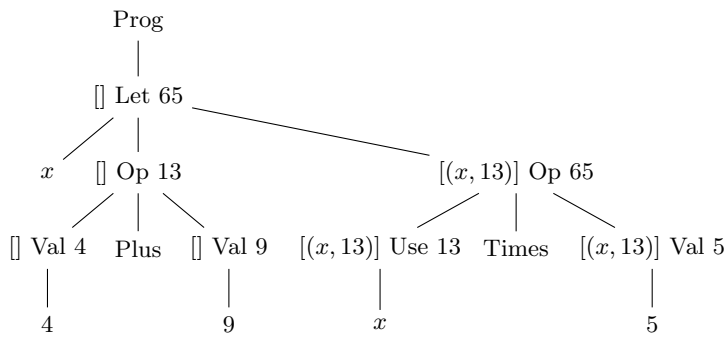


Fig. 4.4. An attributierted tree fpr an expression

attributes come with a direction that specifies the order in which they have to be determined, and that the equations are solved so that computations following that order are simple.

Definition 4.4 (Directed Attribute Grammars). In a *directed attribute grammar* A, the attribute names A are devided into *downward attributes* $\check{A}$ and *upward attributes* $\hat{A}$; for all nonterminals $n \in N$, it therefore holds that $A(n) = \check{A}(n) \uplus \hat{A}(n)$.[3] The values of downward attributes are computed in the context of their parent nodes (using the attribute values of the parent, or of its other childen), whereas upward attributes are computed in the context of the children of a node (using attributes of the children, or downward attributes of the node). The occurrenceds $X(r)$ of variables in a rule $r = n_0 \to t\, n_1 \ldots n_k \in R$ can then be divided into

defining occurrences: $D(r) = \{a.m \in X(r) \mid m = 0 \wedge a \in \check{A}(n_m) \vee m > 0 \wedge a \in \hat{A}(n_m)\}$ and
using occurrences: $U(r) = X(r) \setminus D(r)$.

The semantic equations $E(r)$ of a rule $r \in R$ are *solved* and *directed*, if there is, for every using attribute occurrence $a.m \in U(r)$, exactly one equation of the form $a.m = t$, where all variables in the term t are defining, i.e., from $D(r)$.

Directed equations induce a *dependency relation* on attribute occurrences: For every equation of the form $a.m = t$ and every atttribute occurrence $b.n$ in t holds that $b.n \rightsquigarrow_E a.m$.

This relation can be extended to the attribute instances of an attributed tree by gluing together the dependencies for the semantic rules occurring in the tree.

A directed attribute grammar G is *cycle-free* when none of its attributed trees has a cyclic dependency graph. Altogether, G *well defined* if its attributes

[3] In the literature, downward attributes are called *inherited*, and upward attributes are called *synthesized* or *derived*.

are directed, its equations are solved and directed, and if its attributes trees are cycle-free.

A syntax tree can be evaluated according to a directed attribute grammar by decorating it with semantic equations, and computing their values in an order that is compatible with the dependency relation. It holds:

Theorem 4.1 ([16]). *For a well defined (directed) attribute grammar, every syntax tree has a uniquely determined attributed tree.*

For the context analysis of a practical compiler, the general evaluation method is rather costly, in particular when considering the size of main storage in the seventies when attribute grammars were invented. For, the entire syntaxt tree and its dependencies has to be kept in main memory. Therefore, a lot of simpler evaluation methods have been developed. These are based on the idea to traverse the syntax tree, evaluating a previously selected subset of attribute values, until all have been determined. However, these methods do not work for well defined attribute grammars in general, but only for grammars satisfying stronger conditions wrt. the dependency relations.

The most important methoda are the following ones:

Passes: The tree is traversed completely, visiting the children of a node either from left to right, or from right to left.

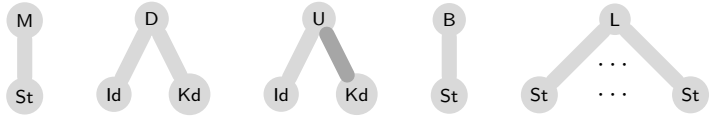
Visit sequences: For every node in the tree, the attributes are assigned to be computed during particular visits to this node. A visit to a node may visit some children several times, while leaving out others, before the node is visited another time.

Evaluation in multiple passes is particularly interesting if the tree is read from a file and written onto a file. Given today's storage sizes, this is not important any more. If the tree is kept in main mamory anyway, visit sequences are more general, and practical.

4.3 Declaration Analysis

We will be using a strongly simplified abstract syntax for discussing declaration analysis. In this syntax, a statement (St) is either the use (U) of an identifier, or the declaration (D) of an identifier of some kind, a block (B), or a list of statements (L).

The abstract syntax consists of the branches



In the syntax, Id is (the key of) an identifier, and Kd the kind of declaration that is not considered at depth here (imagine types, variables, and procedures

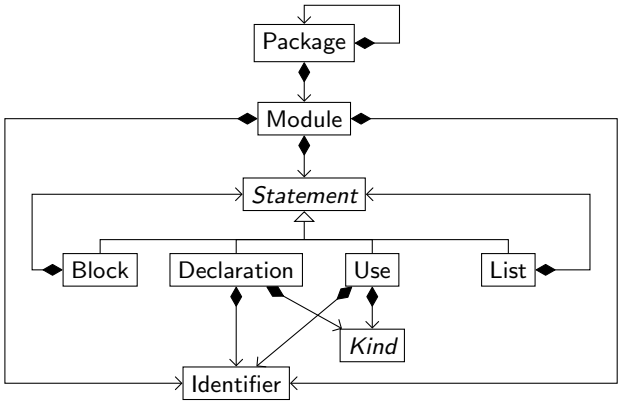


Fig. 4.5. Classes of an abstract syntax for declaration analysis

as examples). The second child of the *U* node is *Nothing* in the syntax tree, and shall be set to *Just kd*, where *kd* is the kind of the declaration that is valid for the identifier.

In a functional language like Haskell, such trees can easily be defined as algebraic data types:

```
data M ≜ M St \\
data St ≜ D Id Kd | U Id (Maybe Kd) | B St | L [St]
```

In an object oriented language, these trees would rather be defined as an abctract class *St* (without attributes), with subclasses *U*, *D*, *B*, and *L*, which have attributes corresponding to every child in the tree.

Then we specify the semantic basis for the computations on the syntax trees. This is an abstract data type providing the types and operations that are needed for declaration analysis. In particular, an abstract type *TAB* of tables is defined, with following operations:

- *initial* yields the start state of the table, with all definitions for predefined names..
- *enter* adds a declataion of some kind to the table.
- *def* looks up the kind that is associated with the identifier in the table.
- *nest* opens a new block and scope.
- *unnest* closes a scope and removes its local declarations from the table.

In Haskell, the interface of this data type would look as follows:

```
module DeclarationTable
where type TAB ≜ [[(Id,Kd)]]
      type Id ≜ String
      data Kd ≜ Unbound | Typ | Const | Var | Proc deriving (Eq, Show)
      initial :: TAB
```

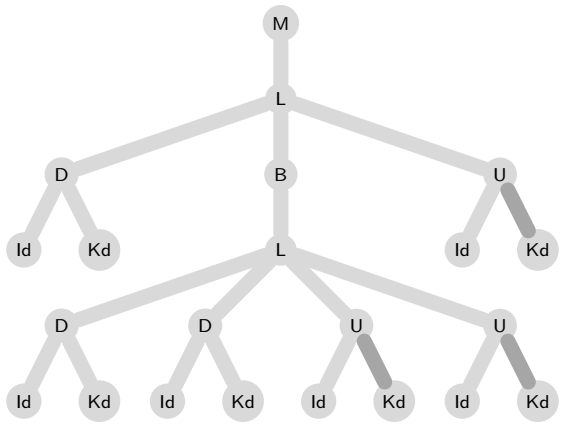


Fig. 4.6. An abstract syntax tree of blocks

$$\begin{aligned}
&\mathit{nest}\,,\ \mathit{unnest} :: \mathit{TAB} \to \mathit{TAB}\\
&\mathit{enter} :: \mathit{TAB} \to (\mathit{Id}\,,\mathit{Kd}) \to \mathit{TAB}\\
&\mathit{def} :: \mathit{TAB} \to \mathit{Id} \to \mathit{Kd}
\end{aligned}$$

We shall discuss efficient implementations of such tables in Section 4.4; so far, we just assume that the table is a list of lists of identifier-kind pairs.

Now we can define how the declaration table is constructed while traversing the tree, in order to determine the kind of used identifiers. First we deal with the simpler case when a language adopts *linear visibility* of declarations: then only declarations appearing left of a use are taken into account. Then we extend this solution to cope with languages that adopt *simultaneous vivibility*, where alle declarations in a block are visible in the entire block.

4.3.1 Declaration Analysis for Linear Visibility

For the attribute grammar of linear visibility, we associate, with every node, states of the declaration table. Every *St*-node gets two attributes of type *TAB*:: the first one, named *v*, contains all declarations occurring before the statement; the second one, named *n*, contains all declarations tha are visible after visiting the node. [4]

The attribute equations can then be defined rather easily:

- For the module node *M*, the attribute *v* is set by *initial* . The attribute *n* of the statement is not needed.
- For an *U*-node, attribute *v* and the identifier is used to determine the kind. Dem Attribut n wird das unveränderte Attribut v zugewiesen. If the kind is not defined (*Nothing*), a contextual error occurs.

[4] The letters refer to the German words *vor* and *nach*.

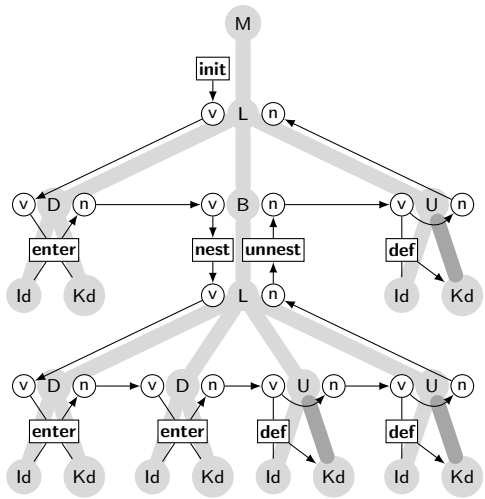
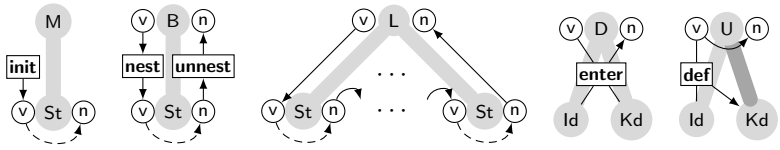


Fig. 4.7. A linearly atributed tree for blocks

- For a *D*-node, the identifier and its kind is entered to the attribute *v* and defines attribute *n*. If another declaration of the identifier appears locally, again, a context error is signalled.
- In an *L*-node, the attributes are propagated from left to right.
- For a *B*-node, *nest* opens a new scope before analyzing the statement, and *unnest* closes it before leaving the block.

The attribute equations can be drawn as follows:



The attributed tree of a sample block is shown in Fig. 4.7.

In these rules, dashed arrows show implicit dependencies of attributes when evaluating subtrees of a branch of the abstract syntax, namely from v to n), resp.

This attribute grammar can easily be turned into recursive procedures idfy' and idfy that perform the attribute computations specified in the rules.

```
idfy' :: Pr → Pr
idfy' (P s) ≜ P s' where (s',_) ≜ idfy (initial) s

idfy:: TAB → St → (St,TAB)
```

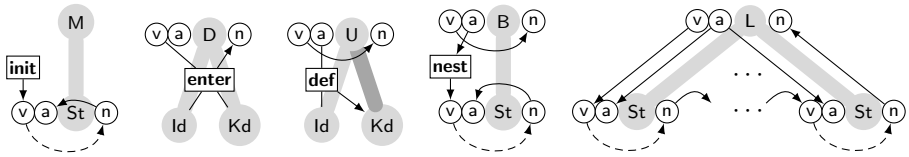
```
idfy v (U x _  ) ≜ (U x k',n) where k' ≜ def v x;  n ≜ v
idfy v (B s    ) ≜ (B s',unnest n) where (s',n) ≜ idfy (nest v) s
idfy v (D x k  ) ≜ (D x k, n) where n ≜ enter v (x,k)
idfy v (L ss) ≜ (L ss',n') where (ss',n)  ≜ idfy' v ss
                                 idfy' []       v ≜ ([],v)
                                 idfy' (s:ss)   v ≜ (s':ss',n)
                                            where (s',v') ≜ idfy s v
                                                  (ss',n) ≜ idfy' ss v'
```

4.3.2 Declaration Analyse for Simultaneous Visibility

If all declarations in a block have be to considered for identifiing uses – even if they appear right of their first use – the attributes an semantic equations must be modified as follows:

- An attribute (named a) must be added, which shall contain all declarations of a block.
- This attribute is used in the rule for **U**.
- In the rule for blocks (**B**), the attribute a can be defined by assigning to it the table (n) of all collected declarations. For collecting the locally valid declarations (in the attribute v), the operation nest now must applied to the attribute a of the enclosing block.
- In the list node, the attribute a is distributed to all childeren.

The attribute equations can be visualized as follows:



The attributed tree of a sample block is shown in Fig. 4.7.

This attribute grammar cannot be handled with a single traversal of the syntax tree: every node has to be visited twice:

1. First, all declarations are collected by computing the attributes v and n, leaving the tree unchanged.
2. Then the attribute a is determined, and all uses of identifiers are checked.

More precisely, the traversals have to be nested: On the first visit of a block node, the attribute n is determined; at the second visit, the a attribute is used to collect the declarations of the block first, and then the a-attribute is used for identification, before the block is left. has to be visited anew. We use two functions for this purpose, $collect :: TAB \rightarrow TAB$, and $idfy :: TAB \rightarrow St \rightarrow St$.

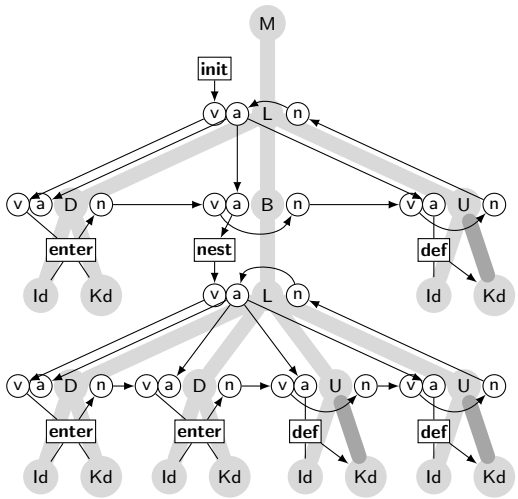


Fig. 4.8. A simultaneously atributed tree for blocks

```
data Prg ≜ P [Mod] deriving (Eq,Show)

data Mod ≜ M Id [Id] St [Id] deriving (Eq,Show)

data St ≜ B St | L [St] | D Id Kd | U Id Kd deriving (Eq,Show)
                                    -- Kd in U is empty, originally

idfy' :: Pr → Pr
idfy' (P s) ≜ P s' where s' ≜ idfy (collect (initial) s)

collect:: TAB → St → TAB
collect v (U _ _ ) ≜ n where n ≜ v
collect v (B s   ) ≜ unnest n where n ≜ collect (nest v) s
collect v (D x k ) ≜ n where n ≜ enter v (x,k))
collect v (L ss)   ≜ collect' v ss
          where collect' v []     ≜ v
                collect' v (s:ss) ≜ collect' (collect s v) ss

idfy:: TAB → St → St
idfy a (U x _ ) ≜ U x k' where k' ≜ def a x
idfy a (B s   ) ≜ B s' where s' ≜ idfy a' s; a' ≜ collect (nest a) s
idfy a (D x k ) ≜ D x k
```

```
idfy a (L ss) ≜ L (map (idfy a) ss)
```

4.3.3 Abstract Syntax, More Concrete

The abstract syntax used so far is minimal for the purpose of discussing declatration analysis. The following question remains:

How does the ceclaration analyis look like for a realistic programming language?

It shows that the more concrte abstract syntax trees of a full programming language can be represented as compouns syntax trees of the abstract syntax we have been using. Let us consider a variable declaration:

```
x : T := expression
```

From an abstract point of view, this declaration is a list with one declaration (of x), and several uses of declarations, namely of the identifiers in the type and the expression. Thus x will be entered to the declaration table (if there is no other local declaration for x), and it is checked whether the other identifiers have been declared properly.

Still missing:

- More examples:
  - Type definition
  - Commands and command sequences
  - Labels and jumps.
  - Procedure declarations
- Einige Beispiele als Aufgaben

4.3.4 Modules and Compilation Units

We now discuss declaration analysis on the level of modules and compilation units, with export and imports.

Again, languages may choose between linear visibility and simultaneaous visibility. Her we just consider simultaneous visibility:

```
module ModSimVis where
import DeclarationTable
import SimVis

-- abstract syntax

data Prg ≜ P [Mod] deriving (Eq,Show)

data Mod ≜ M Id [Id] St [Id] deriving (Eq,Show)
```

```
-- attributes

idfy'' :: Prg → Prg
idfy'  :: TABS → Mod → (Mod,(Id,LOC))

-- attribute equations

idfy'' (P ms) ≜ P (map fst msIdLocs)
                  where msIdLocs ≜ map (idfy' lib) ms
                        lib ≜ map snd msIdLocs

idfy' t (Mm ixs s exs) ≜ (Mm ixs s' exs,(m,ex))
      where init ≜ [importT ixs t]
            glob ≜ collect (nest init) s
            s'   ≜ idfy glob s
            ex   ≜ export exs (unnest glob)
```

4.4 Declaration Tables

The abstract data type *TAB* defining the semantic basis can also easily be defined in a functional setting.

```
module DeclarationTable
where
type TAB ≜ [[(Id,Kd)]]
type Id ≜ String
data Kd ≜ Unbound | ...

initial :: TAB
initial ≜ [[("read",Proc), ("write",Proc)]]

nest, unnest :: TAB → TAB
nest t ≜ []:t
unnest (l:t) ≜ t

enter :: TAB → (Id,Kd) → TAB
enter (l:t) (x,k)
  ≜ if (is_local l x) then error "double␣def." else ((x,k):l):t
    where is_local l x ≜ any (λy→x = (fst y)) l

def :: TAB → Id → Kd
def t x ≜ if t ≠ [] then (snd (head d')) else Unbound
```

$$\textbf{where } d' \triangleq (\mathit{filter}\ (\lambda y \to x = (\mathit{fst}\ y))\ (\mathit{concat}\ t))$$

However, this straight-forward implementation, using lists of association lists, is far to inefficient for real compilers. Even if the implementation of *enter*, *nest*, and *unnest* is "optimal", the complexity for the most frequently used function *def* is linear, giving a quadratic complexity for declaration, which is inacceptable.

TAB is a stack for the declaration sets of nested blocks. In a purely functional language, this could be represented with balanced trees, reducing the complexity to $\mathcal{O}(\log n)$, at least.

In an imperative language, it can be done more efficient. We can take thze hash table for identifiers as a start. The declarations will be stored in a gridg-like table, as follows:

- For every identifier x, its declarationc can be found in a row of the table.
- For every block in the program, there is a column in which all declarations of the block can befound. The leftmost column represnts the block that is actually inspected, and the others represent surrounding blocks.
- The declarations of an identifier are organized as a stack, with the actual declaration on top, and the other, hidden ones further down on the stack.
- The table is sparse, and therefore represented by crosswise start-connected lists. Entries of the table are extended by organisational fields:
  1. A pointer *glob* refers to the next hidden declaration of an identifier.
  2. The pointer *next_local* refers toi the next declaration within the same block.
  3. The key of the identifier refers to the start of all declarations for x. Thus the rows are doubly linked to the start. This is needed when *unnest* removes the local declaration of an identifier, since it then has to re-insert the global declaration at the top of the list.
  4. The *level* stores the nesting depth of a declaration, and thus gives access to the start of the declaration list of a block.

This implementation is optimal, since all operations work without searching. In particuler, *def* is of constant complexity, making declaration analysis linear.

Example 4.5 (Declaration Table). Figure 4.9 shows a declaration table with four declarations, one of t and y at level 3 and 2, resp., and two of x at level 3 and 0. The block on level 1 does not contain declarations.

4.5 Type Analysis

Type analysis assumes declaration analysis to be done, so that every use of an identifier has a cross reference to its declaration. Expressions are the most important places where types must be analysed. So we concentrate on them, even if other places may also require some typte analysis.

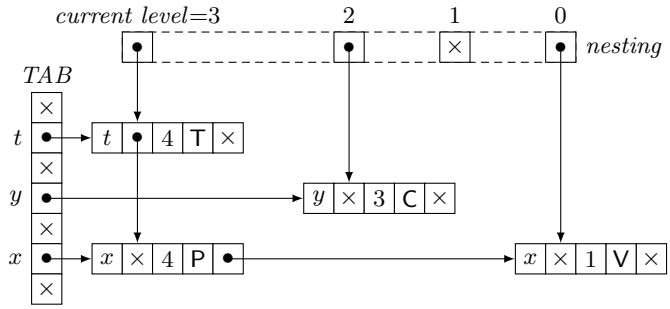


Fig. 4.9. A declaration table

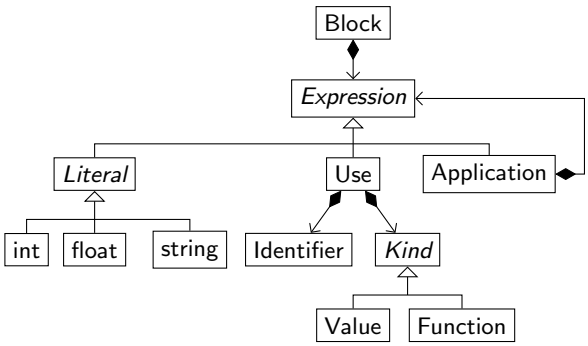


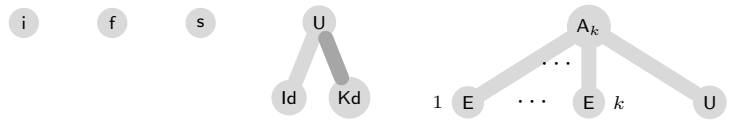
Fig. 4.10. Classes of an abstract syntax for type analysis

We assume an extremely simplified abstract syntax. Each expression is either an operand (an identifier), or the application of a function (a function identifier) to a sequence of arguments.

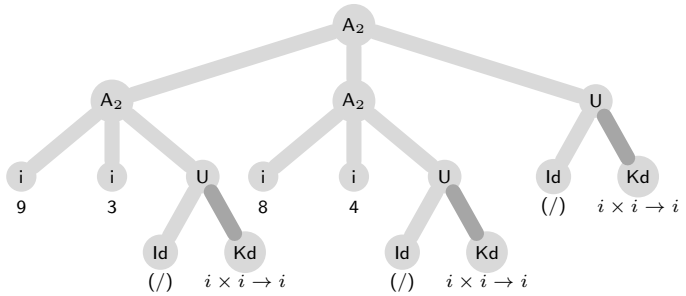
We start with a simple situation where each function has at most one valid declaration (monomorphism). Then we turn to functions that may have several declarations that can be distoinguished by the number ans types of their paraneters (context-free overloading), before we turn to tha case that the parameters of overloaded functions may be equal if their result types are different (context-sensitive overloading.

4.5.1 Type Analysis without Overloading

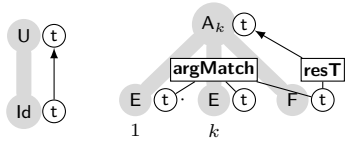
The abstract syntax just distinguishes application nodes *Ap* and operand nodes *U*, which correspond to the constructor introduced for in declaration analysis. Thus the abstract syntax has the following kind of branches:



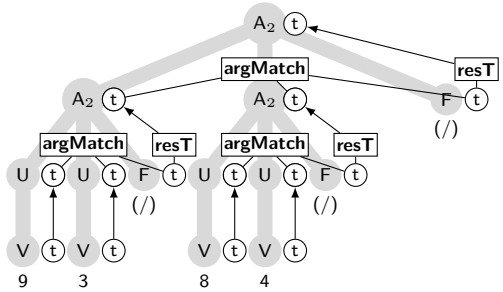
The sample expression "(9/3)/(8/4)" has the following abstract syntax tree:



The semantic equations are as shon below:



The attributed tree for the sample expression is as follows:



Hier stellt $\perp$ den *Fehlertyp* und **N** x t den benannten Typ x dar, dessen Vereinbarung t lautet.

Zur Typüberprüfung gibt es zwei Funktionen:

- type: Kd $\to$ T extrahiert die Typinformation aus einer Deklaration.
- check: T $\times$ T $\to$ T überprüft die Typen einer Applikation, also ob der erste Typ ein Funktionstyp ist, der auf den zweiten anwendbar ist, und liefert den resulttattyp dieser Funktion.

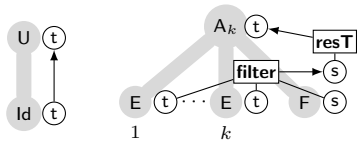
Die Typanalyse arbeitet *bottom-up* im Baum.

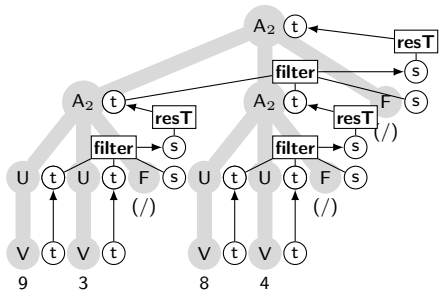
4.5.2 Type Analysis with Context-free Overloading

Almost no programming language is entirely monomorphis, as we have assumed in the previous section. At least some of the predefined operators used in expressions actually do denote several operations at the same time: The operator "+" usually denotes addition of integer and floating point numbers. They have different implementations, by different hardware instructions, and their parameters have different representations, sometimes of different size.

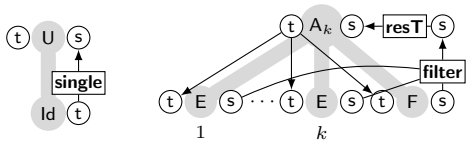
If only predefined operations are overloaded, type analysis has to determine which of the operations overloading the operator is actually applied in an application. This can be decided by considering the types of the arguments.

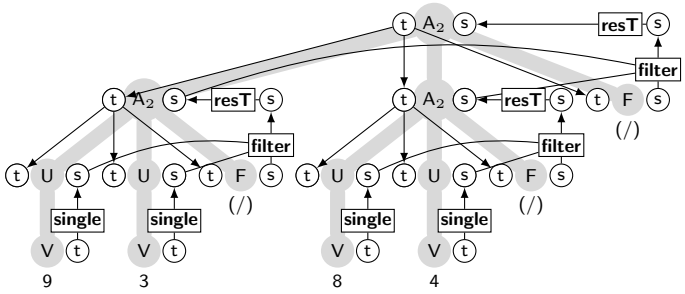
If a programming language allows that the programmer introduces overloaded operations, type analysis (and, as a matter of fact, declaration analysis as well) undergo further changes. Then variables, constants and typers stay monomorph, but functions and procedures may have several declarations at the same place. This has implication for the declaration table, where every identifier may now have a *set* of valid declarations. Thus the function *def* will have the signature *def* :: *Tab* → *Id*→ [*Kd*], returninf a list of declarations in general.





4.5.3 Type Analysis with Context-sensitive Overloading





4.5.4 Type Coercions

In many programming languages, some types acn be cast to others. E.g., integers can be cast to floating point numbers. In Ada, this is done by writing

```
x : Float := Float'1;
```

At some places in a program, many languages insert type casts "silently". An implicit type cast is also called by the nasty name of a tyüe coercion. In many languages, but not in Ada, we can write:

```
x : Float := 1;
```

This is equivalent to the assignemtn above. The reverse coercion is more problematic, as the floats could be rounded or truncated.

Contextual analysis has to turn type coercions into explicit type cast in order to make code generation easier. Thus type analyisis may have to change the abstract syntax tree, in general.

It should be noted that coercions do not fit very well with overloading, as it may increase the danger of ambiguous uses of overloaded operations,
Change: Coercion can be best motivated when discussing monomorphic functions.

4.5.5 Fehlende Inhalte bei Kontextanalyse

- Formale Semantik der Attributgrammars
- konkrete abstrakte Syntax für Deklarationsanalyse
- konkrete abstrakte Syntax für Typanalyse
- Textuelle Definition der semantischen Regeln
- Betrachtungen zu Typverträglichkeit, Anpassungen versus OO-Untertypen
- Haskell-Code für die Attributgrammars (abstrakte abstrakte Syntax)
- Graphersetzungsregeln für Deklarationstabelle

4.6 Attribute Evaluation in Haskell

Evaluators for attribute grammars can easily be implemented in Haskell:

- Nonterminals of the underlying syntax are the names of data types.
- Each rule of the underlying syntax is a constructor definition for a data type.
- The semantic basis is defined as a module, containing semantis types and operations.
- Evaluators are families of functions, one for every nonterminal (data type) of the underlying syntax. They take the inherited attributes and the data type of the nonterminal as a parameter, and the tuple of its derived attribzutes as results. They are defined by structural recursion on the underlying syntax, computing attribute values according to the semantic equations. For simple (left-to-right) attribute dependencies, e.g., in the declaration analysis of linear visibility in Section 4.3.1, and the type analysis for context-free overloading in Section 4.5.2, one evaluator function suffices for every nonterminal. More complex dependencies, e.g., in the declaration analysis of simultaneous visibility in Section 4.3.2, and the type analysis for context-sensitive overloading in Section 4.5.3, may require several evaluators for certain nonterminals which call one another, also in a nested fashion.
- If certain attribute values, e.g., the declaration of a Use node, or the type of an expression, shall be made "persistent", the evaluators have the (updated) tree as another component of their result. The data types have then to be extended by additional components that are initially *Nothing*, and are filled with *Just* a value during evaluation.

For the examples in this chaper, some Haskell evaluators exist:

- `Expr.hs` is the evaluator for expression in Example 4.4.
- The evaluators
  - `LinearVisibilty.hs` und
  - `SimultaneousVisibility.hs`

  implement declaration analysis for linear ans simultaneous visibility, as discussed in Section 4.3.1 and Section 4.3.2, resp.; module `DeclarationTable.hs` defines their semantic basis.
- The evaluatorsr
  - `MonoTypes.hs`,
  - `CFoverloadableTypes.hs`, and
  - `CSoverloadableTypes.hs`

  define type analysis without, with context-free, and with context-sensitive overloading, as discussed in Section 4.5.1, Section 4.5.2, and Section 4.5.3, resp.; the polymorphic version of the declaration table needed for these examples is :`PolymorphicDeclarationTable.hs`.

Bibliographical Notes

Attribute grammars where devised by Donald E. Knuth, for defining the *semantics of context-free languages* [16]. They have been studied intensively in the seventies and eighties (see [7]), and have been used in many compiler writing tools.

Excercises

Solutions to some of these exercises can be found on page 167 *ff*.

Exercise 4.1. Consider an abstract fragment of a language like C or Java that contains *declarations* $x : k$ of some *kind*, say $k = \{\mathbf{var}, \mathbf{type}, \mathbf{proc}\}$, *uses* x, *blocks* $\{\ldots\}$, and sequences $s_1; \ldots; s_n$ of *statements*, being either declarations, uses, or blocks. Consider a sample "program"

$$\{x : \mathbf{proc}; \{x : \mathbf{var}; y; x\}; y : \mathbf{var}; x\}$$

in this abstract language. Is this program *wellformed* according to the contextual rules for declarations in (1) C? (2) Java? (3) Haskell?

Exercise 4.2. Define an evaluator for the attribute grammar in Section 4.3.1 in Java. Use the class diagram in Figure 4.5 as a guideline, and represent the declaration table as a class *Tab* with a suitibly adapted method signatures. (You need not implement this class.)

Exercise 4.3. Analyze the evaluator for the attribute grammar in Exercise 4.2. Note that the table is single-threaded, i.e., passed around so that it could be kept in a global variable.

Represent *Tab* as a **static class**, modify its signatures, and the definition of the evaluator accordingly.

Exercise 4.4. Consider the following program:

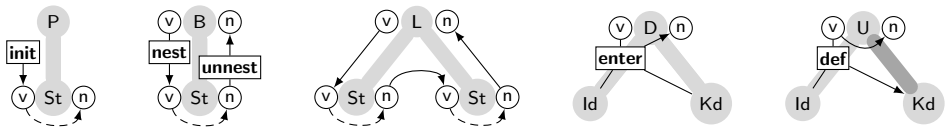
```
declare x:k1;
begin
  declare x:k2;
  declare y:k3;
  use y:
  use x
end;
use x
```

Use the attribute grammar presented in the course in the following steps:

- Construct an abstract syntax tree.
- Determine the attribution rules for the program.

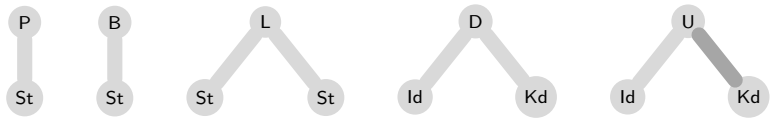
• Which declarations are associated with the identifiers `x` and `y`?

Recall the attribute rules:



Exercise 4.5. Define, for the known abstract syntax and semantical basis, an attribute grammar that defines identification according to *simultaneous visibility*.

The rules of the underlying grammar define the following syntax trees:



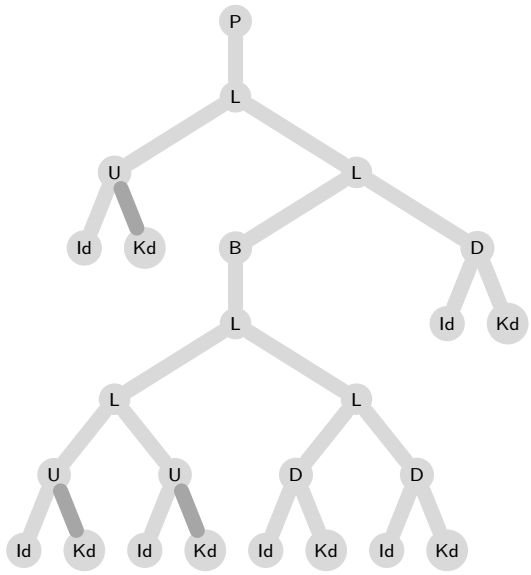
The semantical basis should be extended by one operation:

• The predicate `isLocal` can be useful to determine whether an identifier has already be declared within the current block.

In Haskell, the extended interface would be:

```
module DeclarationTable
where type TAB ≜ [[(Id,Kd)]]
      type Id ≜ String
      data Kd ≜ Unbound | Typ | Const | Var | Proc deriving (Eq, Text)
      initial :: TAB
      nest, unnest :: TAB → TAB
      enter :: TAB → (Id,Kd) → TAB
      def :: TAB → Id → Kd
      isLocal :: TAB → Id → Bool
```

Exercise 4.6. Determine the attribution of the following syntax tree in a language with simultaneous visibility.



Exercise 4.7. For the type analysis of expressions, we have introduced a relation $t \trianglelefteq u$ expressing that a type u is compatible with a type t.

Tasks Consider a programming language you know (Java, Haskell, C, C# ...).

1. At which program places, type compatibility is required?
2. How is type compatibility defined in the language?
3. Does a single notion of type compatibility hold throughout the language, or are there different notions, depending on the context?

Exercise 4.8. Implement the semantic basis for identification, the declaration table, in Java. The signature is:

```
module DeclarationTable
where type TAB ≜ [[(Id,Kd)]]
      type Id ≜ String
      data Kd ≜ Unbound | Typ | Const | Var | Proc deriving (Eq, Text)
      initial :: TAB
      nest, unnest :: TAB → TAB
      enter :: TAB → (Id,Kd) → TAB
      def :: TAB → Id → Kd
      isLocal :: TAB → Id → Bool
```

Exercise 4.9. Until now, the grid-like declaration table copes with declarations that are *monomorphic* so that an identifier is associated with at most declaration. In a *polymorphic* language, a set of declarations can be associated with certain identifiers (of operations and functions). Then the function `def`

yields a list (`Kinds`) of `Kd`. The function `enter`, when adding a declaration `(x,k)` to a table, has to decide whether

1. the entry can just be added, or
2. whether a similar entry is hidden by the new one, or
3. whether the new declaration is illegal, since the entry to be hidden appeared in the same block.

The predicate former called `isLocal` is not helpful any more. A predicate `distinct` should be provided, which determines whether two declarations (their `Kd`, that is) are distinguishable, and can hence be overloaded. [5] This predicate can be used in the new implementation of `enter`.

In Haskell the changed interface is::

```
module DeclarationTable
where type TAB ≜ ...
      type Kinds ≜ [Kd]
      type Kd  ≜ ...
      type Id  ≜ ...
      initial :: TAB
      nest, unnest :: TAB → TAB
      enter :: TAB → (Id,Kd) → TAB
      def :: TAB → Id → Kinds
      distinct :: Kd → Kd → Bool
```

Tasks

1. Extend the efficient implementation of declaration tables so that it copes with overloaded declarations.
2. Does the complexity of the operations change?

[5] Languages differ wrt. the conditions under which functions or operations are considered to be distinguishable. This will be discussed in the course.

5

Transformation of Imperative Languages

In this chapter, we discuss how imperative programming languages like Pascal, C, and Ada can be transformed into code for conventional hardware. Since concrete hardware may be quite diverse, we use an abstract machine as a reference model for hardware. More specifically, we use the (simplified) P-machine that has been developed for the portable Zürich Pascal implementation [5]. The P-machine abstracts from many details of concrete hardware, and has a reduced instruction set tailored towards the easy transformation of Pascal and similar languages. The focus is *not* on gaining a particularly efficient execution of target code, but to allow easy portation to concrete hardware.

The overall aim of this chapter is to explain how the concepts of a "high-level" imperative language – expressions, data structures, commands, procedures – can be translated to "low-level machine architecture with the following concepts:

- random access storage for representing a stack for variables, pointers, and the results of computations, and a heap for data structures,
- registers for special purposes, not for computation,
- instructions for arithmetic operations, comparisons, and Boolean operations, and jump instructions.

Thereby the reader shall get a precise understanding of how concepts of imperative languages are transformed to machine-level, without going into details of "production quality" code generation that would include efficient register allocation, instruction selection, possibly also code improvement, globally or locally.

5.1 The P-Machine

The P-machine is a simple interpreter for P-code, the abstract machine code of the portable Pascal compiler [5]. Its storage consists of three parts:

1. The *data storage* S is an array, of fixed length $StoreMax$, representing the data of a program in a stack and a heap.

2. The *program storage* *C* is an array of fixed length *StoreMax*, representing the P-code, its instructions. The first instruction is at address 0.
3. Registers, like the program counter *PC* and the stack pointer *SP* point to addresses in the code and storage, respectively; Later, we shall introduce three further (storage) registers.

The control of the interpreter is a simple loop:

```
type Value = ...;
type Instruction = ...;
var S : array [0..StoreMax–1] of Value;
var C : array [0..CodeMax–1] of Instruction;
var SP: Integer range –1..StoreMax–1 := –1;
var PC: Integer range –1..CodeMax –1 := –1;
begin
PC := 0;
loop PC := PC + 1;
      execute(Code[PC–1]);
end loop;
<<stop>>
end;
```

The P-code instructions consists of a name and up to two arguments, which can be values, or code and storage addresses. E.g., termination of the interpreter is achieved by executing the instruction *hlt* (without arguments) that executes a jump to the label <<*stop*>>.

5.2 Transforming Simple Expressions

Simple expressions have one of the basic types integer, float, and Boolean, and consists of variables, values of literals, and applications of monadic and dyadic operations:

$$
\begin{aligned}
E \triangleq\ & x_1 \mid \cdots \mid x_n \\
\mid\ & c_1 \mid \cdots \mid c_n \\
\mid\ & \otimes_t E \\
\mid\ & E \oplus_t E
\end{aligned}
$$

The operators are indexed with one of the types $t \in \{\mathrm{b}, \mathrm{i}, \mathrm{r}\}$ in order to resolve overloading of the basic operations. Table 5.1 shows the instructions needed for simple expressions, and Table 5.2 defines the concrete instructions for the generic operations $\mathbf{mop}_{\oplus,t}$ and $\mathbf{dop}_{\otimes,t}$. [1]

[1] Some of these operators are overloaded for *power sets* as well, a type not discussed here because it is rarely used in other languages.

Table 5.1. Instructions for expressions and simple assignments

instruction	semantics
mop$_{\oplus,t}$	$S[SP] := \oplus_t S[SP]$
dop$_{\otimes,t}$	$S[SP-1] := S[SP-1] \otimes_t S[SP]; SP := SP - 1$
ldc c	$SP := SP + 1; S[SP] := c$
lda a	$SP := SP + 1; S[SP] := a$
ind	$S[SP] := S[S[SP]]$

Table 5.2. Instructions for simple monadic and dyadic operations

op.	arg. type		
$\otimes$	b	i	f
or	or		
and	and		
=	eq	eq	eqf
<>	neq	neq	neqf
>	gt	gt	gtf
<	lt	lt	ltf
>=	gte	gte	gtef
<=	lte	lte	ltef

op.	arg. type		
$\otimes$	b	i	f
+		addi	addf
−		subi	subf
*		muli	mulf
/			divf
div		divi	
mod		modi	
**			pow

op.	arg. type		
$\oplus$	b	i	f
xor	or		
−		negi	negr
not	not		

Simple expressions are transformed in the context of an *address environment* $\alpha\colon X \to [0..\text{storemax} - 1]$ that assigns storage addresses (near the bottom of the stack) to every variable. Then the semantic function $[\![\cdot]\!]_V$ producing code that yields values can be defined as follows:

$$
\begin{aligned}
[\![c]\!]V\alpha &\triangleq \textbf{ldc}\ c \\
[\![x]\!]V\alpha &\triangleq [\![x]\!]A\alpha; \textbf{ind} \quad {}^{2} \\
[\![\oplus_t E]\!]V\alpha &\triangleq [\![E]\!]V\alpha; \textbf{mop}_{\oplus,t} \\
[\![E_1 \otimes_t E_2]\!]V\alpha &\triangleq [\![E_1]\!]V\alpha; [\![E_2]\!]V\alpha; \textbf{dop}_{\otimes,t}
\end{aligned}
$$

The semantic function $[\![\cdot]\!]_A$ produces code that yields addresses of identifiers:

$$[\![x]\!]A\alpha \triangleq \textbf{lda}\ \alpha(x)$$

In Section 5.6, this semantic function will be extended to accesses of components of compound variables.

Example 5.1 (Transforming and Evaluating an Expression). The expression $x * (x * 4)$ is transformed as follows:

[2] In Pascal, only basic variables may be operands in expressions. If x is a compound variable, its value may occupy $g > 1$ cells; then it must be moved with **mvs** g. I

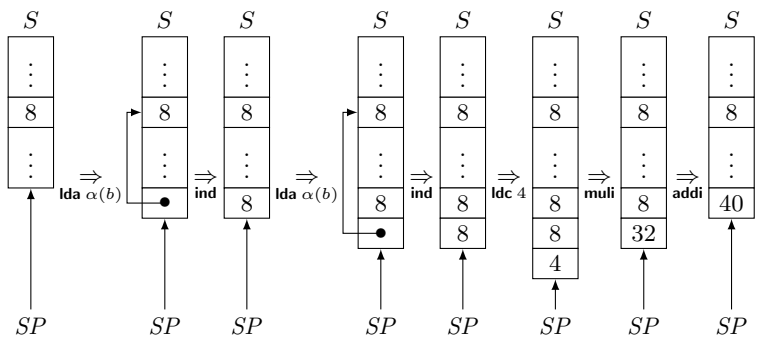


Fig. 5.1. Evaluating the expression $x * (x * 4)$

$$
\begin{aligned}
[\![x + (x * 4)]\!]_{\mathrm{V}}\alpha &= [\![x]\!]_{\mathrm{V}}\alpha; [\![x * 4]\!]_{\mathrm{V}}\alpha; \textbf{addi} \\
&= [\![x]\!]_{\mathrm{A}}\alpha; \textbf{ind}; [\![x * 4]\!]_{\mathrm{V}}\alpha; \textbf{addi} \\
&= \textbf{lda}\ \alpha(x); \textbf{ind}; [\![x * 4]\!]_{\mathrm{V}}\alpha; \textbf{addi} \\
&= \textbf{lda}\ \alpha(x); \textbf{ind}; [\![x]\!]_{\mathrm{V}}\alpha; [\![4]\!]_{\mathrm{V}}\alpha; \textbf{muli}; \textbf{addi} \\
&= \textbf{lda}\ \alpha(x); \textbf{ind}; \textbf{lda}\ \alpha(x); \textbf{ind}; [\![4]\!]_{\mathrm{V}}\alpha; \textbf{muli}; \textbf{addi} \\
&= \textbf{lda}\ \alpha(x); \textbf{ind}; \textbf{lda}\ \alpha(x); \textbf{ind}; \textbf{ldc}\ 4; \textbf{muli}; \textbf{addi}
\end{aligned}
$$

Assuming that $S[\alpha(x)] = 8$, the code is executed as shown in Figure 5.1.

Concepts missing in simple expressions, such as variable accesses ("lvalues" in C), and calls of programmer-defined functions are deferred to Section 5.8. Some others, like operands occupying more than one storage cell, and calls to predefined standard functions, are discussed in the exercises Exercise 5.2.

5.3 Commands and Control Structures

Commands are the center of imperative languages. Here we concentrate on assignments, sequential composition, and the "structured" control structures as they were first included in Pascal: Boolean and (simplified) numerical choice (**if** and **case**) "pre-checked" and "post-checked" loops (**while** and **repeat**) and iterative loops (**for**). Procedure calls are deferred to Section 5.8, and sequencers like **break**, **continue**, and **return** will be explained informally.

The syntax of these commands should be self-explanatory:

$$\begin{pmatrix} \textbf{case } E \textbf{ of} \\ \quad v_1 : C_1; \\ \quad \ldots; \\ \quad v_n : C_n \\ \quad \textbf{else } C_0 \end{pmatrix} \equiv \begin{pmatrix} t := E; \\ \textbf{if} \quad t = v_1 \textbf{ then } C_1 \\ \textbf{elseif } \ldots \\ \textbf{elseif } t = v_n \textbf{ then } C_n \\ \textbf{else} \quad C_0 \end{pmatrix}$$

$$\begin{pmatrix} \textbf{for } i := E_1 \\ \quad \textbf{to } E_2 \\ \textbf{do } C \end{pmatrix} \equiv \begin{pmatrix} i := E_1; \\ t := E_2; \\ \textbf{while } i \leqslant t \\ \textbf{do begin} \\ \quad C; \\ \quad i := i + 1 \\ \quad \textbf{end} \end{pmatrix} \qquad \begin{pmatrix} \textbf{for } i := E_1 \\ \quad \textbf{downto } E_2 \\ \textbf{do } C \end{pmatrix} \equiv \begin{pmatrix} i := E_1; \\ t := E_2; \\ \textbf{while } i \geqslant t \\ \textbf{do begi} \\ \quad C; \\ \quad i := i - 1 \\ \quad \textbf{end} \end{pmatrix}$$

Fig. 5.2. Meaning of **case** and **for**

$$\begin{aligned} C \triangleq\ & x := E \\ \mid\ & C_1; \ldots; C_n \\ \mid\ & \textbf{if } E \textbf{ then } C_1 \,[\, \textbf{else } C_2] \\ \mid\ & \textbf{case } E \textbf{ of } v_1 : C_1; \ldots; v_n : C_n \textbf{ else } C_0 \\ \mid\ & \textbf{while } E \textbf{ do } C \\ \mid\ & \textbf{repeat } C \textbf{ until } E \\ \mid\ & \textbf{for } i := E \ [\textbf{down}] \textbf{ to } E \textbf{ do } C \end{aligned}$$

The transformation of commands relies on jumps, unconditional and conditional ones. More specifically, the conditional one *jumps on false*, i.e., jumps to a label if the top of stack contains the truth value *false*, which is represented by the number 0. Labels are just prepended to an instruction to declare the target of a jump; they must be replaced by the concrete address of this instruction. This is done as for every assembly language, in a two pass procedure that first stores all labels and their addresses in a table, and then substitutes the labels by their addresses in jumps.

Numerical choice and iteration are considered to be "syntactic sugar" that can be transformed into the kernel of the language. namely into nested Boolean choices and pre-checked loops. See Figure 5.2. Note that the fresh temporary variable t is used to store the values of the expression used as conditions. This is not only more efficient, but may also avoid counter-intuitive situations: If the evaluation of these expressions has a side effect, the conditions would check different values in different branches translating the **case**, or in different iterations of the loop. In C, C++, and Java, the syntax of iterations is more general, and the semantics says that the terminating condition E_2 is evaluated anew after each iteration. This may lead to non-termination.

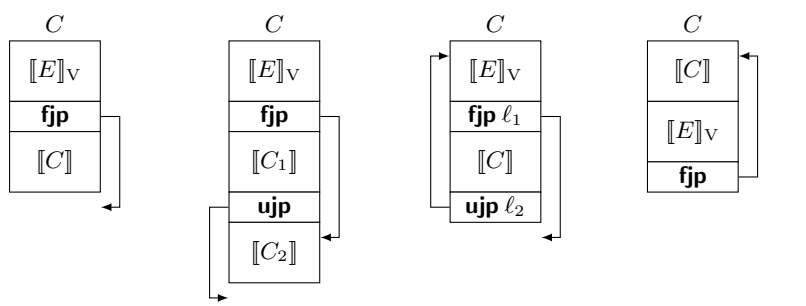
The remaining commands are transformed into P-code, using the instructions shown in Table 5.3.

Table 5.3. Instructions for control structures

instruction	semantics
sto	$S[SP-1] := SS[[SP]]; SP := SP - 2$
ujp ℓ	$PC := \ell;$
fjp ℓ	**if** $S[SP] = 0$ **then** $PC := \ell;\ SP := SP - 1;$

$$
\begin{aligned}
[\![x := E]\!] &\triangleq [\![x]\!]_{\mathrm{A}};\ [\![E]\!]_{\mathrm{V}};\ \textbf{sto} \\
[\![C_1; \ldots; C_n]\!] &\triangleq [\![C_1]\!];\ \ldots;\ [\![C_n]\!] \\
[\![\textbf{if } E \textbf{ then } C]\!] &\triangleq [\![E]\!]_{\mathrm{V}};\ \textbf{fjp}\ \ell;\ [\![C]\!];\ \ell : \ldots \\
[\![\textbf{if } E \textbf{ then } C_1]\!] \textbf{ else } C_2 &\triangleq [\![E]\!]_{\mathrm{V}};\ \textbf{fjp}\ \ell_1;\ [\![C_1]\!];\ \textbf{ujp}\ \ell_2;\ \ell_1 : [\![C_1]\!]; \ell_2 : \ldots \\
[\![\textbf{while } E \textbf{ do } C]\!] &\triangleq \ell_1 : [\![E]\!]_{\mathrm{V}};\ \textbf{fjp}\ \ell_2;\ [\![C]\!];\ \textbf{ujp}\ \ell_1;\ \ell_2 : \ldots \\
[\![\textbf{repeat } C \textbf{ until } E]\!] &\triangleq \ell : [\![C]\!];\ [\![E]\!]_{\mathrm{V}};\ \textbf{fjp}\ \ell;
\end{aligned}
$$

See Figure 5.3 for the coding schemata of control structures.



if E **then** C **if** E **then** C **else** C_2 **while** E **do** C **repeat** C **until** E

Fig. 5.3. Coding schema for transforming control structures

5.4 Data Types

Imperative languages do not only support basic values, like truth values, integral and floating point numbers (maybe of different precision and size), and characters, but also provide several ways to construct compound values from simple ones: arrays, products (records, structures), sums (unions, disjoint unions), where products and sums are termed *algebraic*. And, they may support references (pointers): variables containing variables. Strings are often

predefined; however, usually not as basic values, but as (references to) arrays or list of characters.

As many imperative languages are statically typed, the structure of values can be defined as (data) types. This could look as follows, where the t_i are type names introduced by type definitions, which are accessible as their *definition* $def(t)$:

$$
\begin{aligned}
T \triangleq\ & Void \\
\mid\ & Bool \mid Int \mid Float \mid Char \\
\mid\ & t_1 \mid \cdots \mid t_k \\
\mid\ & (s_1 : T_1 * \cdots * s_k : T_k) \\
\mid\ & (v_1 : T_1 + \cdots + v_k : T_k) \\
\mid\ & [l..h]\, T \\
\mid\ & \uparrow T
\end{aligned}
$$

For most of the types, it is assumed that their values have a fixed *size*, the number of cells needed to store them on the stack of local variables. Exceptions are dynamic arrays, whose length is determined by evaluating an expression at run-time, and functions (if they are used as parameters). These will be treated in Section 5.6.1 and Section 5.9.1. Dynamic data structures like lists and trees have to be represented as references to sums of products; every component of these structures has again static size, but is stored on the heap, another part of the store that can grow and shrink dynamically, by allocating and de-allocating values.

We assume a simplified definition of the size of types where *Void* requires no space at all, all basic types occupy a single cell (although in practice, a bytes suffices for truth values, and floating point numbers may require more than one word):

$$
\begin{aligned}
size(Void) &= 0 \\
size(Bool) = size(Int) &= size(Float) = size(Char) = 1 \\
size(t) &= size(def(t)) \\
size(s_1 : T_1 * \cdots * s_k : T_k) &= \sum_{i=1}^{k} size(T_i) \\
size(v_1 : T_1 + \cdots + v_k : T_k) &= 1 + max(size(T_1), \ldots, size(T_k)) \\
size([\ell]T) &= (h - l + 1) \cdot size(T) \\
size(\uparrow t) &= 1
\end{aligned}
$$

Recursive types require some care: a type like

$$t \triangleq \mathbf{type}(head : Int * tail : t)$$

is illegal as it occupies infinite amount of storage. A recursive definition of a product, sum, and array must always contain a reference constructor "↑", like the type definition

$$t \triangleq \uparrow \textbf{type}(head : Int * tail : t)$$

This definition yield finite sizes for t and its components:

$$size(t) = 1 \qquad size(\uparrow t) = 2 \qquad size(\uparrow t.head) = 1 \qquad size(\uparrow t.tail) = 1$$

Because, values of type $\uparrow t$ are allocated dynamically, on the heap.

Example 5.2 (The Size of Types). Consider the following type definitions:

$$\begin{aligned}
\textbf{type}\ Complex &= \textbf{product}\ re : Float; im : Float)\ \textbf{end} \\
\textbf{type}\ Row &= \textbf{row}\ [-2.. + 2\ Compl \\
\textbf{type}\ List &= \textbf{ref}\ Node; \\
\textbf{type}\ Node &= \textbf{product}\ hd : Float; next : List\ \textbf{end} \\
\textbf{type}\ Shape &= \textbf{sum}\ point : Point; circle : Circle;\ \textbf{end} \\
\textbf{type}\ Point &= \textbf{product}\ x : Float; y : Float;\ \textbf{end} \\
\textbf{type}\ Circle &= \textbf{product}\ x : Float; y : Float; r : Float;\ \textbf{end}
\end{aligned}$$

These have the sizes

$$\begin{aligned}
size(Complex) &= size(Float) + size(Float) = 2 \\
size(Row) &= (+2 - (-2) + 1) \cdot size(Complex) = 7 * 2 = 10 \\
size(List) &= 1 \\
size(Node) &= size(float) + size(float) = 2 \\
size(Circle) &= size(float) + size(float) + size(float) = 3 \\
size(Shape) &= 1 + \max(size(Point), size(Circle)) = 4 \\
size(Point) &= size(float) + size(float) = 2
\end{aligned}$$

5.5 Allocation of Variables

Imperative languages distinguish four categories of variables by their lifetime:

- *Global* variables exist for the whole execution time of a program.
- *Local* variables exist for the lifetime of a block in a program, for instance within the body of a function, procedure, or loop. They are allocated on the stack.
- *Dynamic* variables are created by a constructor, and deleted if they are no longer used. Dynamic variables are allocated on the heap, and typically accessed via a pointer that resides on the stack, or is the component of another dynamic variable on the heap. In older languages, dynamic variables my be destructed by the programmer; since this is error-prone, modern languages rely on automatic storage management with a garbage collector.

- *Persistent* variables have a lifetime that exceeds that of the program. Examples are files or data bases. In most languages, persistent variables are distinct to the three other categories; typically, they are constructed by different means, and have a different structure.

As a global variable is local to the main program, we just discuss local and dynamic variables in the following.

5.5.1 Local Variables

A block may contain declarations of types and variables. (Constants are not discussed separately as they are treated – almost – like variables; procedures and functions, which can also be declared locally, are discussed later, in Section 5.8.)

The declarations of a block are *elaborated* to extend the address environment by addresses for local variables. This is done in three stages:

- All types occurring in the local declarations are elaborated in order to determine their size, and the offsets of the selectors of all product types.
- All local variables and values are elaborated in order to determine their address and size in the local storage.
- The commands are transformed with the so obtained address environment.

Let the local declarations take the following form (where local functions and procedures are omitted):

$$\begin{array}{l}\textbf{type}\ t_1 = T_1;\ \ldots;\textbf{type}\ t_k = T_k;\\ \textbf{var}\ x_1 : T_{k+1};\ \ldots;\textbf{var}\ x_m = T_{k+m}\\ \textbf{do}\ C_1;\qquad\qquad \ldots;C_n\end{array}$$

The local variables get the relative following addresses:

$$\alpha(x_i) = \sum_{j=1}^{i-1} size(T_{k+i}),\ \text{for } 1 \leqslant i \leqslant m$$

Likewise, for all products **product** $s_1 : S_1; \ldots s_k : S_k$ **end** appearing in the types $T_1, \ldots, T_{k+m}$, the offsets of the selectors is defined as

$$\alpha(s_i) = \sum_{j=1}^{i-1} size(S_i),\ \text{for } 1 \leqslant i \leqslant k$$

Example 5.3 (An Address Environment). Assume that a block with type definitions as in Example 5.2 declares the following variables:

var i : *int*;
var n : *Complex*;
var a : *Row*;
var ℓ_1 : *List*;
var ℓ_2 : *List*;
var c : *Circle*;
var s : *Shape*;

The addresses and sizes of these variables are shown in Figure 5.5, and the offsets and sizes of the selectors occurring in their product types are shown in Figure 5.5. These variables are allocated as shown in Figure 5.6.

Usually, a selector may be used in different product types at the same time. So x and y are used both in *Point* and *Circle*. In general, the base type is needed to determine which of the selectors is meant.

variable	α	*size*
i	0	1
n	1	2
a	3	10
ℓ_1	13	1
ℓ_2	14	1
c	15	3
s	18	4

Fig. 5.4. Addresses of variables

product	selector	α	*size*
Complex	*re*	0	1
	im	1	2
Node	*hd*	0	1
	tl	1	1
Point	x	0	1
	y	1	1
Circle	x	0	1
	y	1	1
	r	2	1

Fig. 5.5. Offsets of selectors

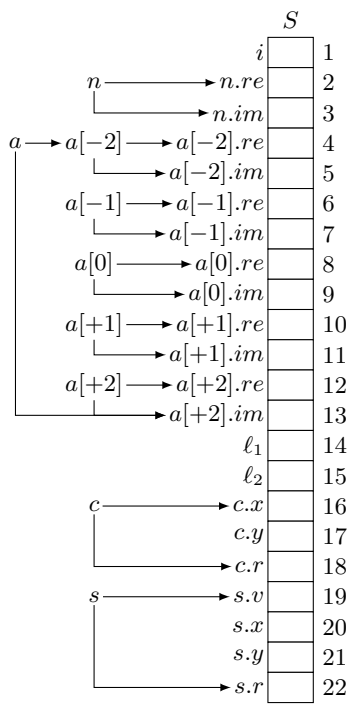


Fig. 5.6. Allocation of variables

Table 5.4. The instruction for heap allocation

instruction	semantics
new s	$HP := HP - c;$ **if** $EP >= HP$ **then** $error(\text{"}heap\ overflow\text{"});$ $S[S[SP]] := HP; SP := SP - 1$

5.5.2 Dynamic Variables

Dynamic data structures, such as lists and trees, are of unbounded size. They are represented with *pointers* that contain addresses (variables) of values that are stored outside the stack: On the heap, values can be allocated by constructor operations. The size of a particular list or tree node will be still static even if the size of the tree and list is unbounded.

The P-machine uses a very primitive kind of heap organization: The heap is in the store S, under the higher addresses (usually appearing in the lower cells in figures). It is not foreseen to increase the heap size at run-time, or to collect unused variables automatically, by garbage collection.

Another register, HP, the *heap pointer*, contains the address of the last used cell of the heap – the lowest index of any used heap cell. Allocation of space is done by an operation **new** c that gathers c cells of free heap space, decreases the heap pointer accordingly, and saves the value of the heap pointer in the stack address lying on top of the stack.

The primitive organization may cause a problem if space is running out: stack and heap may collide, either when the stack pointer is increased – as done by many of the instructions – or when the heap pointer is decreased by **new**. This must be recognized in order to avoid serious storage inconsistencies that are hard to detect.

One could check collision whenever the heap or stack grows. However, as many operations let the stack grow, this would cause considerable overhead, just for error prevention. So the P-machine takes another approach, based on a typical "compiler constructor's observation": The maximal size of the stack within a block can be determined statically, by inspecting the code for the block, and summing up the modifications of the stack pointer performed by them. (Where the control flow branches, the maximal possible increase has to be assumed.) This value, the *extreme stack extension*, can be stored in another register, the *extreme stack pointer* EP. EP and HP have to be compared less frequently:

- Whenever the extreme stack pointer is set (upon entrance and return to a block), it must be checked that the maximal stack will not collide with the heap.
- The same has to be done during every call of **new**.

Now, the instruction **new** can be defined completely, as in Table 5.4.

We extend the commands introduced Section 5.3 by an allocator command for a variable that has a reference type **ref** T :

$$C \triangleq \ldots$$
$$\mid \textbf{new}\ V$$

This command can be transformed easily:

$$[\![\textbf{new}\ V]\!] \triangleq [\![V]\!]_{\mathrm{A}};\ \textbf{new}\ \mathit{size}(T)$$

Constructor functions, as known from object-oriented programming, can be derived from this rather primitive operation.

Example 5.4 (Allocation of Dynamic Variables). Consider the block with the variable allocations in Example 5.3, for the type definitions as in Example 5.2. Let the commands of the block contain the following instructions:

new ℓ_1;
new ℓ_2;

Then heap and stack are updated as shown in Figure 5.7.

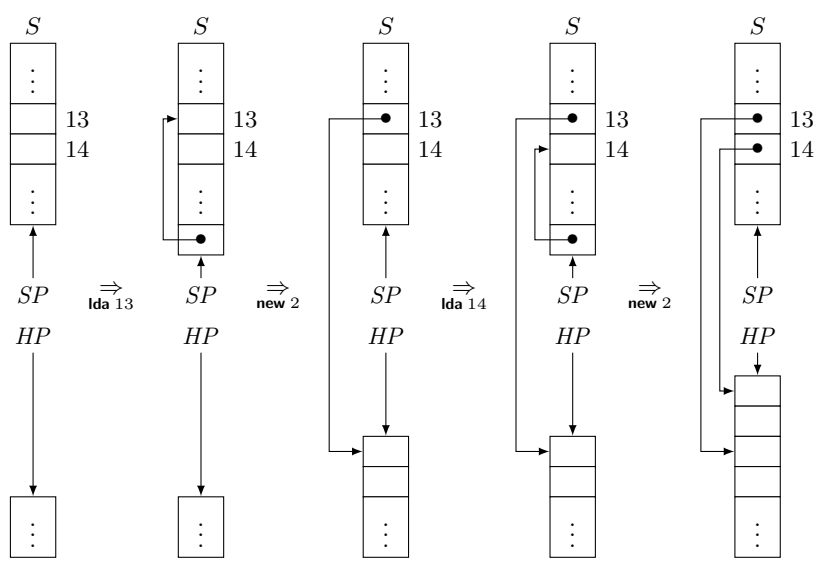


Fig. 5.7. Allocation of dynamic variables

5.6 Accessing Components of Compound Values

Accesses to components of compound variables or values, which are called *lvalues* in C, are expressions yielding the address of a value in store instead of the values itself. They appear on the left-hand side of assignments and in similar contexts, and may be one of the following:

- An *identifier* x of a variable.
- The *selection* of a component s of some product variable X.
- The *projection* (type cast) of a sum variable X to one of its variants v.
- The subscription of an array variable X with an expression E.
- The referenced value of a pointer X.
- The call of a function variable X (or value) with a sequence of arguments $E_1, \ldots, E_k$. This will be discussed later, in Section 5.8.

Syntactically, this may take the form:

$$
\begin{aligned}
V \triangleq\ & x_1 \mid \cdots \mid x_n \\
\mid\ & V.s \\
\mid\ & V : v \\
\mid\ & V[E] \\
\mid\ & V \uparrow \\
\mid\ & V(E_1, \ldots, E_k)
\end{aligned}
$$

The transformation $[\![\cdot]\!]_A$ generates code that determines the address of a variable. For the simple case of identifiers (of basic variables), we have discussed this in Section 5.2 above. The transformation is *type-safe*:

- A *variant error* is raised if a sum variable V is projected to a variant v but contains holds a value of another variant $v' \neq v$.
- An *indexing error* is raised if an array variable is subscribed with a value that lies outside its bound.
- A *dereferencing error* is raised if a variable containing **nil** shall be dereferenced.

The transformation issues instructions that raise the corresponding errors while they generate code computing the address of a variable:

- For an identifier x, its address is popped onto the stack.
- For a product selection, the address of the the variable V is incremented by the offset of the selector s.
- For a sum projection, the address is incremented by one, provided there does not occur a variant error.
- For a subscription, the value of the index is added to the address of the array variable (multiplied with its element size), provided there does not occur an indexing error.

Table 5.5. Instructions for variable access

instruction	meaning explanation
lda a	$SP := SP + 1; S[SP] := a$
inc g	$S[SP] := S[SP] + g$
chv s	**if** $S[SP] \neq s$ **then** error "variant error"
chk ℓ	**if** $S[SP] < 0$ **or** $S[SP] \geqslant \ell$
	then error "index error"
ixa g	$S[SP-1] := S[SP-1] + S[SP] \cdot g;\ SP := SP - 1;$
chn	**if** $S[SP] =$ **nil then** error "variant error"
ind	$S[SP] := S[S[SP]]$

- For a de-referenciation, the address of the variable is replaced by its contents, provided there does not occur a de-referenciation error.

Precisely, this is defined as follows, where g denotes the element size of the (array) type of V:

$$
\begin{aligned}
[\![x]\!]_{\mathrm{A}} &= \textsf{lda}\ \alpha(x) \\
[\![V.s]\!]_{\mathrm{A}} &= [\![V]\!]_{\mathrm{A}};\ \textsf{inc}\ \alpha(s) \\
[\![V : v]\!]_{\mathrm{A}} &= [\![V]\!]_{\mathrm{A}};\ \textsf{chv}\ v;\ \textsf{inc}\ 1 \\
[\![V[E]]\!]_{\mathrm{A}} &= [\![V]\!]_{\mathrm{A}};\ [\![E]\!]_{\mathrm{V}};\ \textsf{chk}\ \ell;\ \textsf{ixa}\ g;\ \textsf{inc}\ (-g * lo); \\
[\![V \uparrow]\!]_{\mathrm{A}} &= [\![V]\!]_{\mathrm{A}};\ \textsf{chn}; \textsf{ind}
\end{aligned}
$$

Example 5.5 (Transforming a Variable Access). Consider the type and variable declarations in Example 5.2. The access to the variable $a[1].im$ is transformed as follows:

$$
\begin{aligned}
[\![a[-1].im]\!]_{\mathrm{A}}\alpha &= [\![a[1]]\!]_{\mathrm{A}}\alpha\ \textsf{inc}\ \alpha(im); \\
&= [\![a]\!]_{\mathrm{A}}\alpha\ [\![1]\!]_{\mathrm{A}}\alpha;\ \textsf{ixa}\ g;\ \textsf{inc}\ (-g * lo);\ \textsf{inc}\ 1; \\
&= \textsf{lda}\ \alpha(a); \textsf{ldc}\ 1\ \textsf{chk}\ \ell;\ \textsf{ixa}\ 2;\ \textsf{inc}\ (4);\ \textsf{inc}\ 1;
\end{aligned}
$$

See Figure 5.8 for the storage states during the execution of this code. Note that the last two instructions could be combined into one, **inc** 5.

Example 5.6 (Access to Pointer Components). The pointers ℓ_1, ℓ_2 allocated in Example 5.4 can be initialized by the following commands:

$$
\begin{aligned}
&\ell_1 \uparrow .hd := 1.0; \\
&\ell_2 \uparrow .hd := 2.0; \\
&\ell_2 \uparrow .tl := \textbf{nil} \\
&\ell_1 \uparrow .tl := \ell_2
\end{aligned}
$$

These commands are translated as follows:

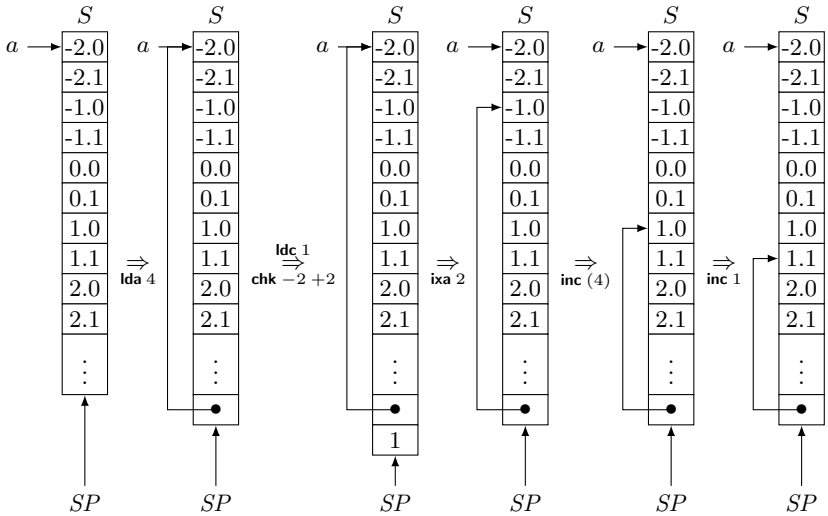


Fig. 5.8. Accessing the component $a[1].im$

$$
\begin{aligned}
[\![\ell_1 \uparrow .hd := 1.0]\!] &= [\![\ell_1 \uparrow .hd]\!]_{\mathrm{A}};\ [\![1.0]\!]_{\mathrm{V}};\ \textbf{sto} \\
&= [\![\ell_1 \uparrow]\!]_{\mathrm{A}};\ \textbf{inc}\ 0;\ \textbf{ldc}\ 1.0;\ \textbf{sto} \\
&= [\![\ell_1]\!]_{\mathrm{A}};\ \textbf{chn};\ \textbf{ind};\ \textbf{inc}\ 0;\ \textbf{ldc}\ 1.0;\ \textbf{sto} \\
&= \textbf{lda}\ \alpha(\ell_1);\ \textbf{chn};\ \textbf{ind};\ \textbf{inc}\ 0;\ \textbf{ldc}\ 1.0;\ \textbf{sto}
\end{aligned}
$$

The remaining commands are translated as analogously:

$$
\begin{aligned}
[\![\ell_1 \uparrow .tl := \ell_2]\!] &= \textbf{lda}\ \alpha(\ell_1);\ \textbf{chn};\ \textbf{ind};\ \textbf{inc}\ 1;\ \textbf{lda}\ \alpha(\ell_2);\ \textbf{ind};\ \textbf{sto} \\
[\![\ell_2 \uparrow .hd := 2.0]\!] &= \textbf{lda}\ \alpha(\ell_2);\ \textbf{chn};\ \textbf{ind};\ \textbf{inc}\ 0;\ \textbf{ldc}\ 2.0;\ \textbf{sto} \\
[\![\ell_2 \uparrow .tl := \textbf{nil}]\!] &= \textbf{lda}\ \alpha(\ell_2);\ \textbf{chn};\ \textbf{ind};\ \textbf{inc}\ 1;\ \textbf{ldc}\ 0;\ \textbf{sto}
\end{aligned}
$$

Note that the instruction **inc** 0 could be omitted. Then heap and stack get the contents shown in Figure 5.9.

5.6.1 Dynamic Arrays

Our assumption that every value of a data type has a static size turns out to be to restrictive for arrays. Three kinds of arrays are used in imperative languages:

- *Static* arrays have lower and upper bounds that are given by a constant (a number literal, a named number, it could also be a constant expression). Their size is static, i.e., known at compile time.

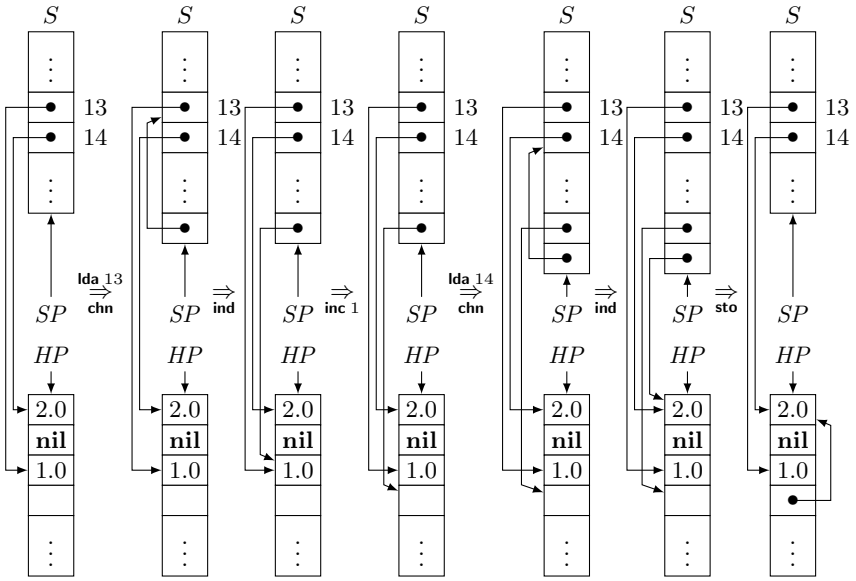


Fig. 5.9. Executing the assignment $\ell_1 \uparrow .tl := \ell_2$

- *Dynamic* arrays are declared with lower and upper bounds that are given by expression. Their size is not static. It may change any time the declaration is elaborated. When an instance of a dynamic array is allocated, its size remains unchanged during its lifetime.
- *Flexible* arrays are declared without any specific bounds. Their size may change with every operation applied to them, e.g., with every assignment of an array value to the variable.

Static arrays are insufficient, for at least one reason: It should be possible to define procedures (and functions) on arrays so that they can work on arrays of *any* size. Whenever such a procedure is invoked, an array of arbitrary size can be passed as an actual parameter. Such a procedure may also need to declare a local array of matching size. Such arrays are dynamic.

Flexible arrays can be stored as pointers to a heap value, which is a static array. Operations on flexible arrays, e.g., assignments, just replace the pointer to one array value by another one. Since languages make heavy use of the heap, some of them just provide static and flexible arrays. Of course, this is sufficient, as every dynamic array can as well be represented as a flexible one. However, this has a disadvantage: flexible arrays have to be removed by the garbage collector, whereas dynamic arrays can be allocated on the stack (as we shall see) so that their values are automatically freed when the procedure terminates that allocated them.

The organization of local variables does not suffice for dynamic arrays: If a procedure declares more than one dynamic array, all but the first could not be stored at a static address. The solution is to split the representation of dynamic arrays into two parts:

1. The values of dynamic arrays will be stored in an area between that of static local variables and the stack for temporary computations.
2. The *descriptor* provides access to the values. Since it is of static size, it can be stored in the area for static local variables. Usually, the descriptor does not just contain a simple pointer to the address of the array value, but provides some additional components that makes the typical operations on arrays easy to code:
   - Accessing an array element.
   - Checking its bounds.
   - Copying the entire array (e.g., when passing its value as a parameter).

Consider a dynamic array **var** a : **row** $[E_{\mathrm{lo}}..E_{\mathrm{up}}]T$, where lo and up shall be the values of E_{lo} and E_{lup}. The descriptor of a has the following components:

1. The *ficticious start address*, of the element $a[0]$ – whether this belongs to the array or not.
2. The *subtrahend*, the difference between $a[0]$ and $a[lo]$.
3. The *lower bound lo*.
4. The *upper bound up*.
5. The *size* $(up - lo + 1) \cdot size(T)$.

For a dynamic array like a, the function *size* yields just the size of its descriptor:

$$size(\mathbf{row}\ [E_{\mathrm{lo}}..E_{\mathrm{up}}]T) = 5$$

When the declaration of a dynamic array is elaborated, the address environment α can thus be determined as before. In addition, code has to be generated in order to define the components of the descriptor, and to reserve space in the area for dynamic array values.

We assume that after elaborating the local declarations of types, variables and values - including descriptors of dynamic arrays – and updating the address environment α, the instruction **isp** g is used to increase the stack pointer SP by the size of the local declarations.

Then declaration of a dynamic arrow is transformed as follows:

$$[\![\mathbf{var}\ a : \mathbf{row}\ [E_{\mathrm{lo}}..E_{\mathrm{up}}]T]\!]\alpha = [\![a]\!]_{\mathrm{A}}\alpha; [\![E_{\mathrm{u}}]\!]_{\mathrm{W}}\ \alpha; [\![E_{\mathrm{o}}]\!]_{\mathrm{W}}\ \alpha; \mathbf{sad}\ \alpha(x)\ size(t);$$

The subscription of a dynamic array is transformed as follows:

$$[\![V[E]]\!]\alpha = [\![V]\!]_{\mathrm{A}}\ \alpha; \mathbf{dpl}; \mathbf{ind}; [\![E]\!]_{\mathrm{W}}\ \alpha; \mathbf{chd}; \mathbf{ixa}\ size(type(x[E]));$$

The instructions are shown in Table 5.6.

Table 5.6. Instructions allocating and accessing dynamic arrays

instruction	meaning	explanation
sad $a\ g$	$S[a+4] := S[SP];$	(upper bound)
	$S[a+3] := S[SP-1];$	(lower bound)
	$S[a+2] := S[a+3] * g;$	(subtrahend)
	$S[a+1] := S[a+4] - S[a+3] + 1;$	(size)
	$S[a] := SP - S[a-2] - 1;$	(ficticious start)
	$SP := SP + S[a+1] - 2$	(reserve space)
dpl	$SP := SP + 1;$	
	$S[SP] := S[SP-1];$	
ind	$S[SP] := S[S[SP]]$	
chd	**if** $S[SP] < S[S[SP-2]+3]$ **or** $S[SP] > S[S[SP-2]+4]$	
	then error "index error");	
ixa g	$S[SP-1] := S[SP-1] + S[SP] \cdot g;\ SP := SP - 1;$	
sli	$S[SP-1] := (S[SP]; SP := SP - 1;$	

For a dynamic array x, the code $[\![V]\!]_{\mathrm{A}}\,\alpha$ yields the address of its *descriptor*, not of the array values. The descriptor is duplicated with the instruction **dpl**, and then the instruction **ind** yields the (ficticious) address of its values; then code for the selection of elements is generated. The original descriptor is needed to check the array bounds with the instruction **chd**. Whenever the variable V is not just an identifier x, but, e.g., the product component $y.s$, the address of the descriptor cannot be passed to the instruction via a modificator. The original descriptor is found two cells below the top of stack (below the address of the array values and the value of the index). After executing the code tor array subscription, the address of the subscribed variable is slided down the stack, and overwrites the original descriptor.

Example 5.7 (Allocation and Access of a Dynamic Array). Consider the Pascal-Program

```
procedure Feld;
  var a: array [−n..+n] of Complex;
  var i: Integer;
  begin
    i:= 1;
    a[ i ] := a[ i+1]
  end;
```

Let the global variable `n` have the value 2. Assume that $size(\texttt{Complex}) = 2$ and $\alpha = \{\texttt{n} \mapsto 5, \texttt{a} \mapsto 10, \texttt{i} \mapsto 15\}$.

After elaborating the declarations, the stack pointer is set to the end of the region for local variables (to 15 in this case).

The code for allocating the array looks as follows:

$$
\begin{aligned}
&[\![\texttt{x: array [-n..+n] of Complex}]\!]\alpha \\
&= [\![\texttt{x}]\!]_{\mathrm{A}}\ \alpha; [\![-\texttt{n}]\!]_{\mathrm{W}}\ \alpha; [\![\texttt{n}]\!]_{\mathrm{W}}\ \alpha; \textbf{sad}\ \alpha(x)\ \mathit{size}(\texttt{Complex}); \\
&= \textbf{lda}\ \alpha(x); \textbf{lda}\ \alpha(\texttt{n}); \textbf{ind}; \textbf{neg}; \\
&\quad \textbf{lda}\ \alpha(\texttt{n}); \textbf{ind}; \textbf{sad}\ \alpha(x)\ \mathit{size}(\texttt{Complex}); \\
&= \textbf{lda}\ 10; \textbf{lda}\ 5; \textbf{ind}; \textbf{neg}; \textbf{lda}\ 5; \textbf{ind}; \textbf{sad} 102;
\end{aligned}
$$

The storage is modified by these descriptions as described in Figure 5.10.

The code for the subscription of `x` is as follows:

$$
\begin{aligned}
[\![\texttt{x[i]}]\!]\alpha &= [\![\texttt{x}]\!]_{\mathrm{A}}\ \alpha; \textbf{dpl}; \textbf{ind}; [\![\texttt{i}]\!]_{\mathrm{W}}\ \alpha; \textbf{chd}; \textbf{ixa}\ \mathit{size}(\mathit{type}(\texttt{x[i]})); \\
&= \textbf{lda}\ \alpha(x); \textbf{dpl}; \textbf{ind}; \textbf{lda}\ \alpha(\texttt{i}); \textbf{ind}; \textbf{chd}; \textbf{ixa}\ \mathit{size}(\mathit{type}(\texttt{x[i]})); \\
&= \textbf{lda}\ 10; \textbf{dpl}; \textbf{ind}; \textbf{lda}\ 15; \textbf{ind}; \textbf{chd}; \textbf{ixa}\ 2;
\end{aligned}
$$

The storage during array subscription is as shown in Figure 5.11.

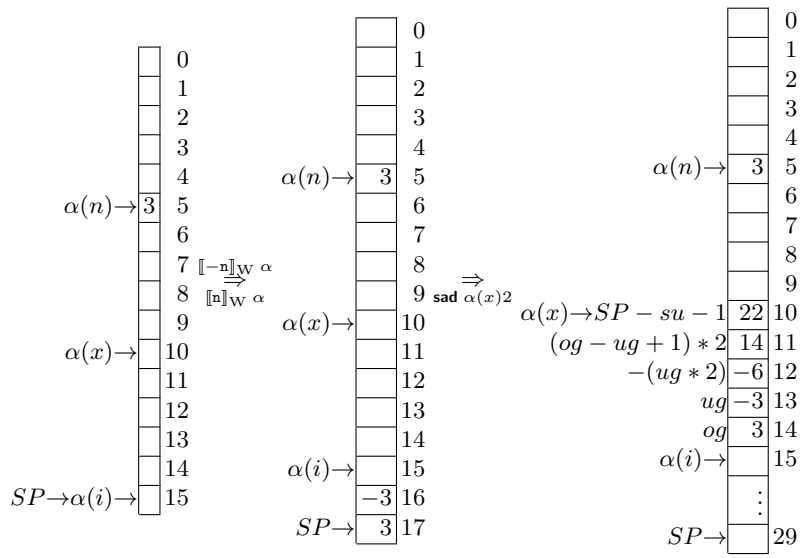


Fig. 5.10. Allocation of an array descriptor

5.7 Blocks

...

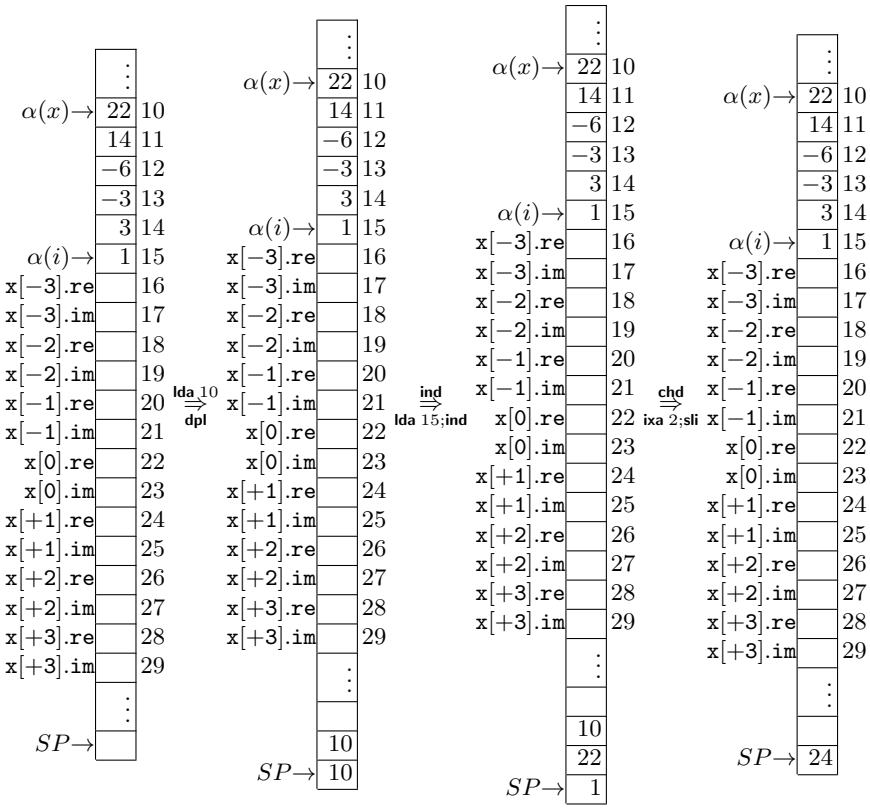


Fig. 5.11. Subscibing a dynamic array

5.8 Procedures and Functions

For transforming a procedures or function, the following actions must be taken at the place of their declaration:

1. With a function, reserve space for storing its result value.
2. Generate code for evaluating the actual parameters.
3. Save the state of the calling procedure q.
4. Jump to the code of the called procedure p.
5. Initialize the state for the called procedure p. This includes allocation of space for its local variables (their static portion, to be precise).
6. Allocate space for dynamic arrays that are declared locally, or passed as value parameters.
7. Generate code for the passing of dynamic arrays as values.
8. Generate code for the body of p.
9. Generate code jumping back to the procedure q. Restore the state of q.

Table 5.7. Instructions for transforming procedure calls

instruction	meaning explanation
isp n	$SP := SP + n$
mst ℓ	$S[\ell] := FP;$
	$S[SP+1] := FP;$
	$S[SP+2] := EP;$
cup a	$S[SP+3] := PC;$
	$FP := SP;$
	$PC := a;$
sep n	$EP := SP + n;$
	if $EP >= HP$ **then** $error(\text{"}stack\ overflow\text{"});$
ret s	$SP := FP - s - 1;$
	$PC := S[FP+2];$
	$EP := S[FP+1];$
	if $EP >= HP$ **then** $error(\text{"}stack\ overflow\text{"});$
	$FP := S[FP];$

10. For instructions "**return** $[E]$", a jump must be generated to the end of the procedure code. In a function, the value of the expression E must be stored at the location of the function's result. (Context analysis has to make sure that a return is called before terminating a function.)

Tasks 1-5 are done at the place of the call, and the rest is done at the place of the declaration of the procedure p.

$$
\begin{aligned}
[\![p(E_1,\ldots,E_k]\!]\alpha\,\ell &= [\textbf{isp}\ size(T_0)] \qquad \text{(only with functions)}\\
&= [\![E_1]\!]_{M_1}\alpha\,\ell\\
&\quad \ldots\\
&\quad [\![E_k]\!]_{M_k}\alpha\,\ell\\
&\quad \textbf{mst}\ (\ell - \lambda(p))\\
&\quad \textbf{cup}\ \alpha(p)
\end{aligned}
$$

Here, $m_1,\ldots,m_k$ are the *modes of passing* specified for the parameters.

A procedure declaration with formal parameters $P = m_1x_1 : t_1,\ldots,m_kx_k : t_k$ has to be transformed as follows:

$$
\begin{aligned}
&[\![\textbf{proc}\ p(P)\,[\textbf{return}\ T_0]\,;\textbf{let}\ D\ \textbf{in}\ C]\!]\ \alpha\,\ell\\
&= [\![P]\!]_{\mathrm{DA}}\ \alpha\,\ell\\
&\quad [\![D]\!]_{\mathrm{DA}}\ \alpha\,\ell\\
&\quad \alpha(p) : [\![C]\!]\ \alpha\,(\ell - \lambda(p));\\
&\quad \textbf{ret}\ size(t_0)\\
&\ldots
\end{aligned}
$$

Table 5.8. Instructions for passing dynamic arrays as parameters

instruction	semantics	explanation
ssb ℓ	$S[SP+4] := \ell - 1;$	(upper bound)
sss g	$S[SP+2] := S[SP+4] * g;$	
	$S[SP+1] := (S[SP+4] - S[SP+4] + 1) * g;$	
	$SP := SP + *wieviel??*$	
sss g	$S[SP-4] := SP - S[SP-2];$	
mda a	**for** $i := 1$ **to** $S[FP+a+1]$ **do**	
	$S[SP+i] := S[S[FP+a] + S[FP+a+2]];$	
	$S[FP+a] := SP + 1 - S[FP+a+2];$	
	$SP := S[FP+a+1]$	

5.9 Parameter Passing

Let us now consider different modes of parameter passing, and how these are transformed.

The *mode* of a formal parameter p specifies what shall be achieved by a parameter

- If p shall pass data from the call into its body, the mode is "*in*".
- If p shall pass data from its body to where it is called, the mode is "*out*".
- If both shall be achieved, the mode is "*in-out*".

...

5.9.1 Procedure and Function Parameters

Excercises

Solutions to some of these exercises can be found on page 175 *ff*.

Exercise 5.1. Determine the transformation $c = \text{code}_W(3+(x*4))\alpha$. Assume that $\alpha(x) = 3$, and that $\mathsf{dop}_* = \mathsf{mul}$ has the meaning:

$$S[SP] := S[SP-1] * S[SP]; SP := SP - 1$$

Execute c with the P-Machine. Assume the code c starts at location 10; $S[3]$ shall contain the value 10 , and SP shall have the value 3.

The equations for transformation are on slide 169/171. The sketch of the P-machine is on page 166. [There is a minor error on this page. Do you find it?]

Exercise 5.2. 1. *Predefined Functions.* Allow calls to standard functions of the form $f(e_1, \ldots e_k)$ in expressions, and extend the transformations to handle these calls. Assume that that $\alpha(f)$ contains the address of code(f).

Table 5.9. Instructions transforming numerical choices

instruction	semantics
ijp	$PC := PC + S[SP];\ SP := SP - 1$
ljp $n\ \ell$	**if** $S[SP] < n$ **then** $PC := \ell$
gjp $n\ \ell$	**if** $S[SP] > n$ **then** $PC := \ell$
ind	$S[SP] := S[S[SP]]$

Transform function calls into machine code. You may wish to introduce an instruction that jumps to the code of standard functions. This instruction could use a register, say OPC, to save the program counter. An instruction for returning from a standard function concludes the code of standard functions. Assume that this instruction restores the program counter with "$PC := OPC$".

2. *Compound Operands.* Allow that the infix operations and standard functions in an expression may have operands and results that occupy several storage cells, not just one.
   Extend the transformation scheme accordingly.

Exercise 5.3. A simple form of numerical choice takes the following form:

$$\mathsf{C} \triangleq \underline{\text{case}}\mathsf{E}\ \underline{\text{of}}\,\underline{u}\ \underline{=>}\ \mathsf{C}_{\,u}\ \underline{;}\ \ldots\ \underline{;}\ \underline{o}\ \underline{=>}\ \mathsf{C}_{\,o}\ \underline{\text{default}}\ C_{\text{d}}$$

The value v of the expression E is used to select the command C_v labeled with the corresponding value, from a contiguous integer range $u \ldots o$. If v is not in the range $u \ldots o$, the *default* command C_{d} is selected. (The pre-historic **switch** of Fortran and Algol-60, inherited by C, C++, and Java does not necessarily perform a unique selection (only if all commands are terminated with **break**).

Numerical choice can be transformed, on the source level, into an equivalent cascade of Boolean choices (<u>if</u> – <u>then</u> – <u>else</u>). It can also be implemented with a goto table: n unconditional jumps <u>ujp</u> l_i are inserted in the code – not in the data – and a *computed goto* jumps to the right place in that table. This instruction **ijp** is defined in Table 5.3. The instruction expects the goto table to be placed in the directly succeeding code locations. The instructions **ljp** $n\ \ell$ and **gjp** $n\ \ell$ check whether the expression yields a value that is less then the smallest value, and greater than the greates value in the range of the cases; it occurs in Table 5.9 as well.

Tasks

1. Translate numeric choice, either to equivalent <u>if</u>-cascades, or with a goto table.
2. Which of the implementations do you consider to be better? More precisely, under which circumstances could one be better than the other one?
3. If the same command shall be selected for different numerical values, the simple form of <u>case</u> forces to duplicate code, which should be avoided for several reasons.

How would you change the transformation, if a list of literals can be placed in front of the commands?

Exercise 5.4. Consider the following program with the type definitions known from Example 5.2 and the variable declarations of Example 5.3:

```
a[0].re := 1.0;
a[0].im := 2.0;
a[2].re := −1.0;
a[2].im := −4.0;
i:= 1;
a[i] := a[i+1]+a[a−1];
c.x := 1.0;
c.y := 0.0;
c.r := 2.0;
s := c;
c.y := s:circle.r
```

Translate the second and the last line of the program. Execute the translated P-code with the values suggested by the other commands of the program.

Does the transformation of assignments in Section 5.3 cover all relevant aspects for the assignment "$s := c$"?

Exercise 5.5. The transformation in the course cover arrays wit index ranges of integers.

Some imperative languages, like Pascal and Ada, allow character ranges and ranges of enumerands as well. How can this be covered by the transformations as well?

Could products over ranges of integers, characters, and enumerands be allowed as indexes as well? What has to be done in this case? As an example, consider the hypothetical type

$$\textbf{type}\ \mathit{Index} = \textbf{product}\ i : 4..7; \ell :' A'..'Z'; c : (\mathit{Red}, \mathit{Green}, \mathit{Blue});\ \textbf{end}$$
$$\textbf{type}\ \mathit{Obscure} = \textbf{row}\ [\mathit{Index}]\ \mathit{int}$$

Exercise 5.6. Exercise 5.7. In the course, we have discussed how static predecessors can be organized as a linked list. This leads, with deeply nested blocks, to a certain overhead in the access to non-local identifiers.

This can be avoided if static predecessors are organized in a *display vector*. Here the static predecessors are held in a (small) global stack at the start of the storage area. As the depth of block nesting is a static value, the maximal extension of the display vector can be determined in advance.

Tasks: Organize static predecessors in a display vector.

1. What has to be done when entering and leaving a procedure?
2. Redefine the instruction **lda**. You may change the meaning of the modifier ℓ for that purpose.

3. If a variable is global, i.e., declared on level $\ell = 0$, or local, i.e., declared on $\ell =$ *current level*, the display vector is not needed at all. Define the instructions **ldg** and **ldl** for access to global and local variables, respectively.

Exercise 5.8. Translate the following PASCAL function:

```
function fac (n: Integer): Integer;
begin
   if n <= 0 then fac := 1 else fac := fac(n–1)
end
```

Task: The statement "`fac` := E" defines the result of the function `fac` (instead of "**return** E" in other languages). Execute the code.

Exercise 5.9. This is about endangered ways of parameter passing. Let **proc** $p(x : t); B$ be a procedure that is called as $p(E)$.

Constant Parameters

If x is handled as a *constant parameter*, x acts as a constant name for the value of E while p is executed, as if a declaration **const** $x = E$; would be elaborated before entering p.

Questions

1. In what differs this passing mode to *value parameters* ("*call by value*")?
2. Consider the advantages and drawbacks of constant wrt. value parameters.
3. What code has to be generated for passing a dynamic array as a constant parameter?

Name Parameters This way of parameter passing resembles textual substitution, and has been used in the lambda calculus. It is defined as follows: If x is passed as a name parameter, it denotes the expression E while the body B of p is executed. Whenever x is used, the expression E will be evaluated; this is done in the context of the block B, considering all declarations for B.

Questions

1. Consider advantages and drawbacks of name parameters.
2. Sketch how name parameters could be transformed into code.
3. What is the difference of name parameters to pure textual substitution?

Exercise 5.10. Translate the following program:

```
program fakultaet;
  var x : Integer;
  function fac (n: Integer): Integer;
  begin
   if n <= 0 then fac := 1 else fac := n * fac(n–1)
```

```
  end
begin
  x := fac(2)
end.
```

6

Transforming Object-oriented Languages

This chapter is about the transformation of object-oriented languages into machine code. We will use an abstract machine resembling that one used in the previous chapter, and also re-use many transformations defined in the previous chapter. Because, even if object-oriented languages do extend imperative languages, several concepts are the same, or similar.

We first consider the new concepts of object-oriented languages, before we study the organization of storage for single inheritance, and define transformations for class definitions, accesses to variables, and method calls.

6.1 Concepts of Object-Orientation

The novel concepts of object-oriented concepts are the following:

- *Modularity*: Classes have components, called features. A *feature* is an attribute (a variable or constant), or a method (a procedure or a function). Methods have a separate parameter, called its *receiver object*.
- *Information hiding*: Certain features of a class can be hidden from other classes; this should be done for variables in particular.
- *Instantiation*: It is possible to create any number of objects for a class.
- *Inheritance*: A class S can extend another class B by additional features. Then S is called the *subclass* of B, and B is called the *base class* of S.
- *Type extension*: The methods of a class can also be applied to objects of any of its subclasses.
- *Overriding*: In a subclass, the implementation of a method in a class can be replaced by another implementation.
- *Late binding* (*dynamic dispatch*): If a message is sent to an object, the actual type of that object determines which method is being applied.
- *Purity*: Every class defines a compound type, and every compound type is defines as a class.
- *Garbage collection*: The storage of objects is reclaimed if it is no longer used.

To be done: Look for definitions in [1] and in Bertrand Meyer's new book on OO.

6.1.1 Relation to Imperative Concepts

- A class defines a cartesian product (known as structure, record or tuple in progrsaming languages), consisting of values, pointer to objects, and pointers to procedures.
- The collection of all subtypes of a class defines a sum (union). In general, the sum is not disjoint, sinces all summands share the features of their base class, and is "open", as subclasses can be introduced at any time; they may even be loaded dynamically.
- Objects are pointers to instances of classes that are allocated in the heap.

6.1.2 The Syntax of a Ficticious Object-orineted Language

program ::= {{ **class** }}

class ::= [[*abstract*]] *CLASS Id EXTENDS Id WITH* {{ *Kind Feature* ; }}
END

Kind ::= [[*COMMON* [] *INDIVIDUAL*]] [[*PUBLIC* [] *HERITARY*]]

Feature ::= ((*VAR* [] *CONST*)) *Id*: *Id*
[] *METHOD Id Signature*: ((*block* [] *ABSTRACT*))
[] *OVERRIDE Id Signature*: *Block*

Signature ::= ...

Block ::= ...

Command ::= ...

Expression ::= ...

Variable ::= *Id*
[] *THIS*
[] *SUPER*
[] *Variable* . *id*
[] *Variable* (*Expression* {{ , *Expression* }})

6.2 Storage Organization for Object-oriented Languages

The storage for executing an object-oriented language is organized as follows:

- The global program store contains:
  - *Code* for the method bodies of classes.
  - *Global storage* for the class attributes and pointers to the bodies of object methods of classes (*class tables*.
  - *Code* for initializing class tables ("class loader).
- The program stack contains:
  - method frames with the parameters, results, and local variables of method instances.
- The program heap holds:
  - Attributes of objects, containing values, pointers to their object attributes, and a pointer to the class table of the object.

If the code of method bodies, the global data store, and the stack are kept independent of each other, classes can be loaded dynamically.

6.3 Transformation of Object-Orientied Programs

- Class declarations
- Instantiation of an object
- Method call
- Assignment (because of type checking)
- Garbage collection

Transforming a Class Declaration

1. The size of a class table is determined by
   - the size of its new class attributes,
   - the size of its inherites class attributes
   - one cell für the pointer to the superclass,
   - The number of inherited object methods, and
   - the number of new object methods

   This size is static.
2. The address of the class table is that of the superclass pointer. The class attributes lie above that place, so their relative addresses of are negative. The (pointers to) object methods lie below that place, so their relative addresses of are positive.
3. The class table is initialized by copying the inherited class attributes and object methods from the class table of the super class, and setting the super pointer to that table. Then the new class attributes and object methods are initialized.
4. The method bodies of object methods are translated; their code address is stored in the address in the class table. This address overwrites the address of an inherited method if it is overwritten. The adresses of abstract methods stay undefined.

Table 6.1. Instructions for classes

instruction	meaning explanation
smp $a\,\ell$ **ssp** $a\,\bar{a}$ **ind**	$S[a] := \ell$ $S[a] := \bar{a}$ $S[SP] := S[S[SP]]$
mct $a\,n$	**for** $i := n-1$ **downto** 0 **do** $\quad S[a+i] := S[S[SP+i]];$ $SP := SP + n$
new $a\,\bar{a}$	$SP := SP + 1;$ $S[SP] := HP;$ $H[HP] := a$ $HP := HP + \bar{a}$
cup ℓ **mdm** $a\,n$	$\ldots$ $PC := \ell;$ $\ldots;$ $PC := S[H[S[SP - \pi(m)]] + \alpha(m)];$

$$
\begin{aligned}
[\![C_1; \ldots; C_n]\!]\alpha &= [\![C_1]\!]\alpha; \ldots; [\![C_n]\!]\alpha \\
[\![\textbf{cls}\ c\ \textbf{ext}\ \bar{c}\ \textbf{with}\ f_1 \ldots f_n]\!]\alpha &= \textbf{ssp}\ \alpha(c)\ \alpha(\bar{c}); [\![f_1]\!]\alpha; \ldots [\![f_2]\!]\alpha; \\
\forall f \in \bar{c} : \alpha(c.m) &:= \alpha(\bar{c}.m); \\
[\![\textbf{common}\ x : c]\!]\alpha &= \varepsilon \\
[\![\textbf{common}\ m : s\ b]\!]\alpha &= \ell : [\![b]\!]\alpha;\ \alpha(c.m) := \ell \\
[\![\textbf{individual}\ x : c]\!]\alpha &= \varepsilon \\
[\![\textbf{individual}\ m : s\ \ \textbf{abstract}]\!]\alpha &= \ \alpha(c.m) := \textbf{null} \\
[\![\textbf{individual}\ m : s\ b]\!]\alpha &= \ell : [\![b]\!]\alpha;\ \alpha(c.m) := \ell
\end{aligned}
$$

Instantiating Objects

1. The basic operation **new** reserves space for the object attributes on the heap, and sets the pointer to the class table.
2. User-defined constructors may take arguments in order to initialize onject attributes.

Accessing Features

1. Like the class table as a whole, the class attributes $c.a$ have a static address $\alpha(c.a)$ in the global storage.
2. Class methods $c.m$ have a static address $\alpha(c.m)$ in the code.
3. Object attributes $o.a$ have an indirect dynamic address $S[SP] + \alpha(a)$ on the heap.

4. Objects x have a static address $\alpha(x)$ on the stack, containing an address of the object structure on the heap.
5. The receiver object, **this**, has an address that lies before the parameters of the methof being called, also vor dem Rahmen der Prozeduri.e., before the frame of the procedure.
6. Statically bound object methodhave a static address $\alpha(c) + \alpha(c.m)$ in the code (like class methods).
7. Dynamically bound bject methods have a double indirect dynamic address $S[H[S[SP]] + \alpha(am)]$ in the code.

...In the following equations we assume c to be a class, , x to be an object, a to be an attribute, $m, \bar{m}$ to be n-ary methods, where $\bar{m}$ shall be a classmethod, and m shall be an object method. V denotes an object access with base type $\bar{c}$ (of a class).

$$
\begin{aligned}
[\![c.a]\!]_{\mathrm{A}}\ \alpha &= \textsf{lda}\ \alpha(c.a);\\
[\![c.\bar{m}(E_1,\ldots,E_k)]\!]_{\mathrm{A}}\ \alpha &= [\![E_1]\!]_m\ \alpha; [\![E_k]\!]_m\ \alpha; \textsf{cup}\ \alpha(c.m)\\
[\![x]\!]_{\mathrm{A}}\ \alpha &= \textsf{lda}\ \alpha(x); \textsf{chn}\\
[\![\mathbf{this}]\!]_{\mathrm{A}}\ \alpha &= \textsf{lda}\ (-(\pi(m)+1));\\
[\![\mathbf{super}]\!]_{\mathrm{A}}\ \alpha &= [\![\mathbf{this}]\!]_{\mathrm{A}}\ \alpha; \textsf{ind};\\
[\![V.a]\!]_{\mathrm{A}}\ \alpha &= [\![V]\!]_{\mathrm{A}}\ \alpha; \textsf{chv}\ \mathit{type}(a); \textsf{inc}\ \alpha(a);\\
[\![V.m(E_1,\ldots,E_k)]\!]_{\mathrm{A}}\ \alpha &= [\![V]\!]_{\mathrm{A}}\ \alpha; [\![E_1]\!]_m\ \alpha; \ldots [\![E_k]\!]_m\ \alpha; \textsf{cup}\ \alpha(\bar{c}.m)\\
[\![V.m(E_1,\ldots,E_k)]\!]_{\mathrm{A}}\ \alpha &= [\![V]\!]_{\mathrm{A}}\ \alpha; [\![E_1]\!]_m\ \alpha; \ldots [\![E_k]\!]_m\ \alpha; \textsf{cdm}\ \alpha(m)\ \pi(m)
\end{aligned}
$$

Assignments

1. The actual type of the object hold by the right-hand side expression must be of a subtype of the base type of the left-hand side variable.
   Der dynamische Typ der linken Seite muss ein Untertyp des Basistyps der linken Seite sein.
2. Reference semantics is the default: The reference of the object, not its components are assigned.

$$[\![V := E]\!]_{\mathrm{A}}\ \alpha = [\![V]\!]_{\mathrm{A}}\ \alpha; [\![E]\!]_{\mathrm{W}}\ \alpha; \textsf{sto}\ ;$$

Excercises

Solutions to some of these exercises can be found on page 182.

Exercise 6.1. Consider how the organization of object-oriented code must be changed in the presence of "proper" multiple inheritance (not only of interfaces, as in Java).

1. How can method tables be organized?
2. How can dynamic binding of methods be done?

7

Conclusions

The previous chapters describe what every computer scientists should know about the implementation of programming languages with interpreters and compilers, including:

- Specification of lexemes, systematic implementation of scanners recognizing them, and organization of the screener selecting the information relevant for later phases.
- Specification of syntax by context-free grammars, systematic transformation of grammars for easier parsing, systematic construction of recursive descent parsers, principles of the generation of bottom-up parsers with the $LR(k)$ algorithm, and construction of abstract syntax trees.
- Specification of contextual constraints, concerning declaration analysis (or identification) and type checking, with attribute grammars, and systematic generation of attribute evaluators.
- Principles of transforming imperative languages into abstract machine code.
- Extension of this scheme for the transformation of object-oriented languages.

The readers of this text – and the participants of the course that is accompanied by it – should be able to use the analysis methods – scanning, parsing, context checking – for implementing all kinds of languages that may appear in their everyday work. And, they should understand how the low-level code for imperative and object-oriented languages is organized, in order to know which overhead incurs for which type of concept in high-level programming languages.

This text does not describe methods for the construction of compilers that generate code in production quality. This would require additional chapters on tasks like the following:

- *Code generation for concrete machines* has to consider the peculiarities and irregularities of memory organization and instruction sets of real hardware, and to to come up with advanced algorithms for resource allocation – in particular with register allocation – and for instruction selection – in particular if the instruction set is complex, with redundant operations.

- *Code improvement* – often euphemistically called *code optimization* – deals with
  - *global code improvement*, advanced methods to analyze data and control flow in the source program, and the use of its result for producing better code, and with
  - *local code improvement* by analysis of the intermediate or target program, e.g., by peephole optimization.

These are advanced topics of compiler construction, useful only for few software developers.

Bibliographical Notes

[3, Chapters 8–12] describe code generation and improvement. [14] and [18] are modern books on the design and implementation of optimizing compilers.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, New York, 1996. 132
2. Alfred V. Aho, S. C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. *CACM*, 18(8):441–452, 1975. 75
3. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison-Wesley, Boston, Massachusetts, 2006. 11, 138, 220
4. Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling – I: Parsing, II: Compiling*. Prentice Hall, 1972. 141
5. Urs Ammann. On code generation in a Pascal compiler. *Software: Practice and Experience*, 7(3):391–423, 1977. 4, 105
6. Andrew W. Appel. *Modern Compiler Construction in ML*. Cambridge University Press, 1998. 220
7. Jean-Pierre Déransart, Michel Jourdan, and Bernard Lorho. *Attribute Grammars*, volume 323 of *Lecture Notes in Computer Science*. Springer, 1988. 101
8. Frank DeRemer. Lexical analysis. In Friedrich L. Bauer and Jürgen Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 1976. 40
9. Franklin L. DeRemer. *Practical Translators for LR(k) languages*. PhD thesis, Massachusetts Institute of Technology, Cambridge, 1969. /Elib/Compilers/Syntax. 75
10. Franklin L. DeRemer. Simple $lr(k)$ grammars. *CACM*, 14(7):453–460, 1971. /Elib/Compilers/Syntax. 75
11. Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, 2nd edition, 1998. 144
12. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley (Pearson Education), 2nd edition, 2001. 75
13. Stephen C. Johnson. Yacc: Yet anaother compiler-compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, 1975. 75
14. Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 138

15. D.E. Knuth. On the translation of languages from left to right. *Information & Control*, 8:607–639, 1965. /Elib/Compilers/Syntax. 66, 75
16. Donald E. Knuth. Semantics of context-free languages. *Math. Sys. Theory*, 2(2):127–145, 1968. Correction: *Math. Sys. Theory* 5(1): 95-96 (1971). 87, 101
17. Michael E. Lesk and Erik Schmidt. Lex – a lexical analyzer generator. In Bell Labs, editor, *Unix Programmers manual*, volume 2B, pages 38–50, Murray Hill, NJ, USA, 1979. 38, 40
18. Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. 138
19. Tom Niemann. Lex and yacc tutorial, 2012. Url: http://epaperpress.com. 40
20. David A. Watt and Derryck F. Brown. *Programming Language Processors in Java – Compilers and Interpreters*. Prentice Hall, Harlow, Essex, 2000. 11, 75, 220
21. Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Springer, 2. edition, 1997. 21, 22, 25, 220

B

Answers to Selected Exercises

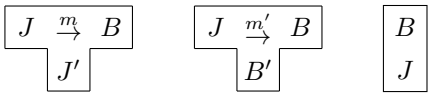
Answers to Exercises in Chapter 1

Exercise 1.1. For C, JAVA,and HASKELL holds:

1. *Binding* of names to types, classes, variables, or procedures is *static*, with one notable exception: in Java, the binding of method names to method bodies is, in general, dynamic.
2. In C the type of a variable is *static*, even if it is not always checked.
   In Haskell, variables are "single-assignment", and *static* as well. In Java, variables are declared with a static base type b; at runtime, they may contain values of any subtype of b.
3. In C or Java, an *array* of the form "$[n]\langle type\rangle a$" has a *static* length. In C, arrays can be declared as pointers, and be allocated (with `malloc`) a storage of dynamic size.
   The predefined arrays of Haskell are of dynamic size.
4. Values of a variable are dynamic, in general. Only if the variable is "single assignment" (like "**final**" in Java), and the expression defining the variable is constant, the value itself is constant as well.

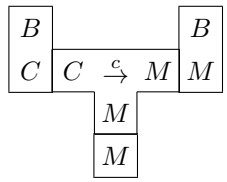
Exercise 1.2. The differences between source languages are just too big. A universal language representing them would be either very big as well (coming close to the union of these languages) so that the compiler from U would be very hard to implement. Or, the languages would lead to rather large representations of particular languages. Then the compiler from U would probably be inefficient.

Exercise 1.3. The starter kit of portable Java implementation consists of the following components:

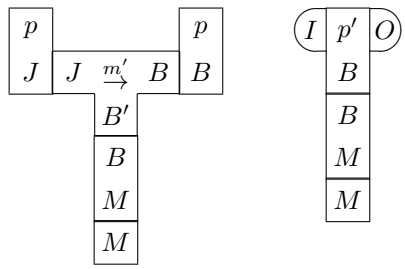


The compilers m and m' use only subsets $J' \subseteq J$ and $B' \subseteq B$ of Java and JBC, respectively.

1: Porting to M. The JVM could be rewritten in C, and be translated with the native C compiler:

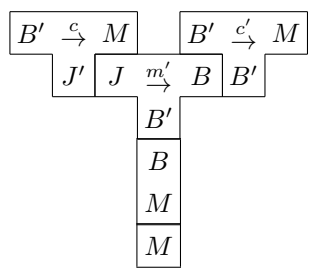


Now the compiler m' can be interpreted on M, as well as every program p translated with it:

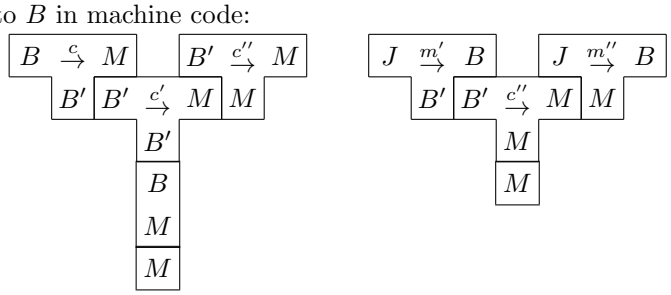


However, the compiler will be slow on M as it is interpreted.

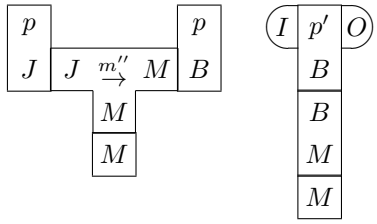
2: Bootstrap on M. Instead of an interpreter, we write a compiler c of B' into M code; c will just be used to translate m' so that not all of the JVM needs to be translated; c could be done in C.



The result c' can be used to compile c "with itself", in order to translate, with the resulting c'', the master compiler m' to a "semi-native" translator for Java to B in machine code:



Now we have the required native compiler. The B programs produced by it can be interpreted:



Answers to Exercises in Chapter 2

Exercise 2.1. A regular definition for *Identifiers* is

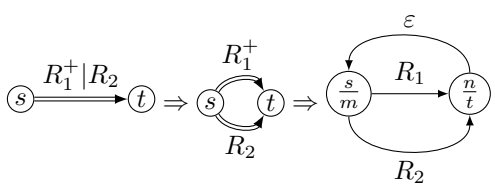
$$\langle\text{identifier}\rangle = \langle\text{letter}\rangle\ \left((\_)^?\ (\langle\text{letter}\rangle \mid \langle\text{digit}\rangle)\right)^*$$
$$\langle\text{letter}\rangle = \texttt{a} \mid \cdots \texttt{z} \mid \texttt{A} \mid \cdots \texttt{Z}$$
$$\langle\text{digit}\rangle = \texttt{0} \mid \cdots \texttt{9}$$

Exercise 2.2. A regular definition for *String Literals* is

$$\langle\text{stringchar}\rangle = \text{"}\bar{,}\,n$$
$$\langle\text{string}\rangle = \text{"}\ (\text{""} \mid \langle\text{stringchar}\rangle)^*\ \text{"}$$

Exercise 2.3.

1. The transformation rule for R^+ in Figure 2.5 may appear too complicated at first glance, with all the empty transitions around (which – very inconvenient – make the automata non-deterministic.
   Identifying the pairs $(s, m), (n, t)$ of nodes in the transformation rule for R^+ allows a transformation step:



   Now the target graph represents the expression $(R \mid R')^+$, which is not equivalent to the expression $R^+ \mid R'$ represented by the source graph.
2. Identifying the pair (m, n) of nodes in the transformation rule for R^+ preserves equivalence. This can be easily shown by considering the forms allowed for R.

Exercise 2.4. The NFA for *Identifiers* as defined for Exercise 2.1 is derived as shown in Figure B.1.

The *DFA* is constructed as follows. The quotient set construction gives the following states and transitions:

1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, \langle\text{letter}\rangle) = \{1, 2, 3, 5\}$ with transition under $\langle\text{letter}\rangle$.
3. $P_2 = F(P_1, \_) = \{5\}$ with transition under $\_$.
4. $P_3 = F(P_1, \langle\text{letgit}\rangle) = \{1, 3, 4, 5\}$ with transition under $\langle\text{letgit}\rangle$.
5. $F(P_3, \_) = \{5\} = P_2$ with transition under $\_$.
6. $F(P_3, \langle\text{letter}\rangle|\langle\text{digit}\rangle) = \{1, 3, 4, 5\} = P_3$ with transition under $\langle\text{letgit}\rangle$.

In the DFA, states P_1 and P_3 turn out to be equivalent, and are joined to $P_{1|3}$. There are no dead or unreachable states. The DFA and its *Minimal DFA* are:

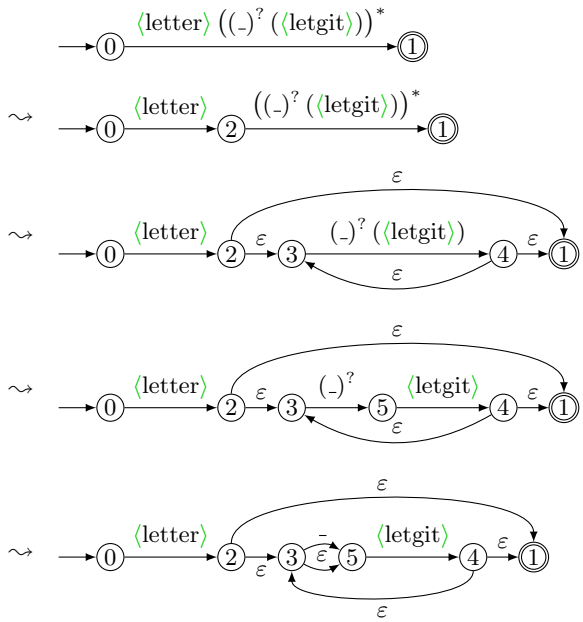
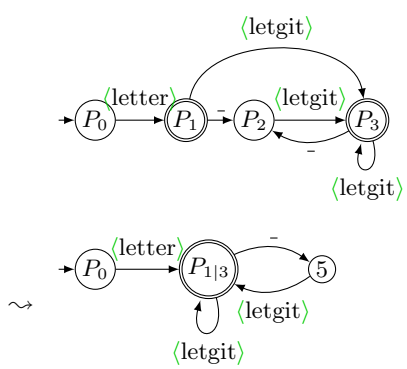


Fig. B.1. Deriving the NFA for identifiers (Answer to Exercise 2.4)



The NFA for *string literals* is derived as shown in Figure B.2.

The *DFA* is constructed as follows. The quotient set construction gives the following states and transitions:

1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, ") = \{2, 3, 4\}$ with transition under ".
3. $P_2 = F(P_1, ") = \{1, 6\}$ with transition under ".
4. $P_3 = F(P_1, \langle\text{StringChar}\rangle) = \{3, 4, 5\}$ with transition under $\langle\text{StringChar}\rangle$.

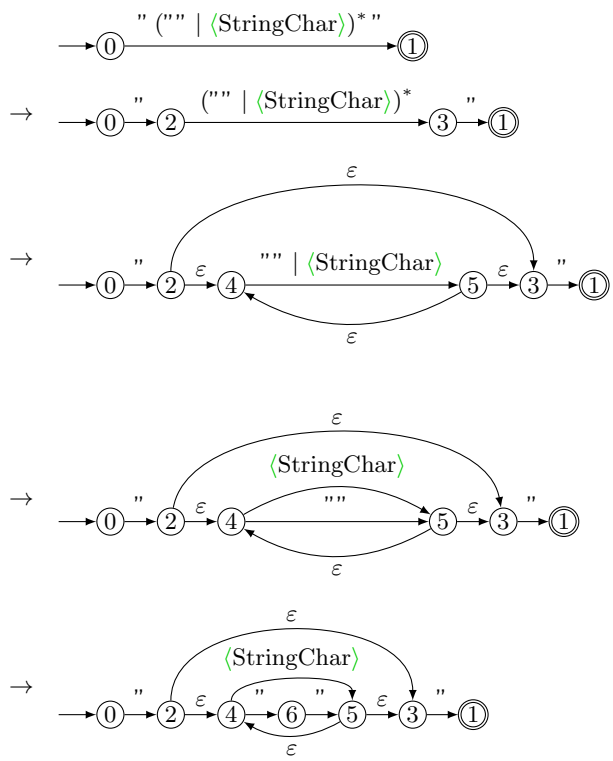
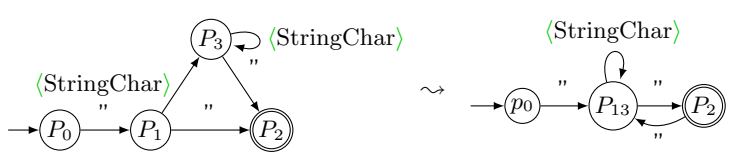


Fig. B.2. Deriving the NFA for string literals (Answer to Exercise 2.4)

5. $F(P_3, ") = \{1, 6\} = P_2$ with transition under ".
6. $F(P_3, \langle\text{StringChar}\rangle) = \{3, 4, 5\} = P_3$ with transition under $\langle\text{StringChar}\rangle$.

In the automaton, states P_1 and P_3 turn out to be equivalent, and can be joined to a state P_{13}. No state is dead or unreachable. So the DFA and its *Minimal DFA* are:



Exercise 2.5. We start from the regular definition of string literals in the answer to Exercise 2.2 above.

We treat the character class ⟨stringchar⟩ like a single character.

$$
\begin{aligned}
\langle\text{stringliteral}\rangle &= \underline{\text{"}}\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} = q_0 \\
R_1 &= (\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \\
&\equiv \varepsilon\,\underline{\text{"}} \\
&\mid (\langle\text{stringchar}\rangle \mid \underline{\text{""}})\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \\
&\equiv \underline{\text{"}} \\
&\mid \langle\text{stringchar}\rangle\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \\
&\mid \underline{\text{""}}\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \\
&\equiv \underline{\text{"}}\,\left(\underline{\text{"}}\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}}\right)^? \\
&\mid \langle\text{stringchar}\rangle\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \\
R_3 &= \left(\underline{\text{"}}\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}}\right)^? \\
&\equiv \varepsilon \\
&\mid \underline{\text{"}}\,(\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} \in F_0 \\
R_4 &= (\langle\text{stringchar}\rangle \mid \underline{\text{""}})^*\,\underline{\text{"}} = R_2
\end{aligned}
$$

The automaton for string literals equals the automaton constructed in the answer to Exercise 2.2.

Exercise 2.6. The following regular definition describes comments.

$$\langle\text{comment}\rangle = \underline{/}*\left(\overline{*} \mid (*)^+\,\overline{*/}\right)^*\,(*)^+\,/$$

Here, $\overline{t_1 \ldots t_n}$ denotes the *complement* of the set $\{t_1, \ldots, t_n\}$, that is, all characters except $t_1, \ldots, t_n$.

The NFA is derived as shown in Figure B.3.

For the *DFA*, the quotient set construction gives the following states and transitions:

1. $P_0 = E(0) = \{0\}$.
2. $P_0 \rightarrow_{/} P_1 = F(P_0, /) = \{2\}$.
3. $P_1 \rightarrow_{*} P_2 = F(P_1, *) = \{3, 4, 6, 8, 11\}$.
4. $P_2 \rightarrow_{*} P_3 = F(P_2, *) = \{5, 8, 9, 10, 11, 12\}$.
5. $P_2 \rightarrow_{\overline{*}} P_4 = F(P_2, \overline{*}) = \{4, 6, 7, 8, 11\}$.
6. $P_3 \rightarrow_{/} P_5 = F(P_3, /) = \{1\}$.
7. $P_3 \rightarrow_{*} F(P_3, *) = \{5, 9, 8, 10, 11, 12\} = P_3$.
8. $P_3 \rightarrow_{\overline{*/}} F(P_3, \overline{*/}) = \{4, 6, 7, 8, 11\} = P_4$.
9. $P_4 \rightarrow_{*} F(P_4, *) = \{5, 8, 9, 10, 11, 12\} = P_3$.
10. $P_4 \rightarrow_{\overline{*}} F(P_4, \overline{*}) = \{4, 6, 7, 8, 11\} = P_4$.

Then states $P_2 = \{3, 4, 6, 8, 11\}$ and $P_4 = \{4, 6, 7, 8, 11\}$ turn out to be equivalent, since both are not final, and their successor states (under $*$ and $\overline{*}$) are the equivalent. So the DFA and minimal DFA are:

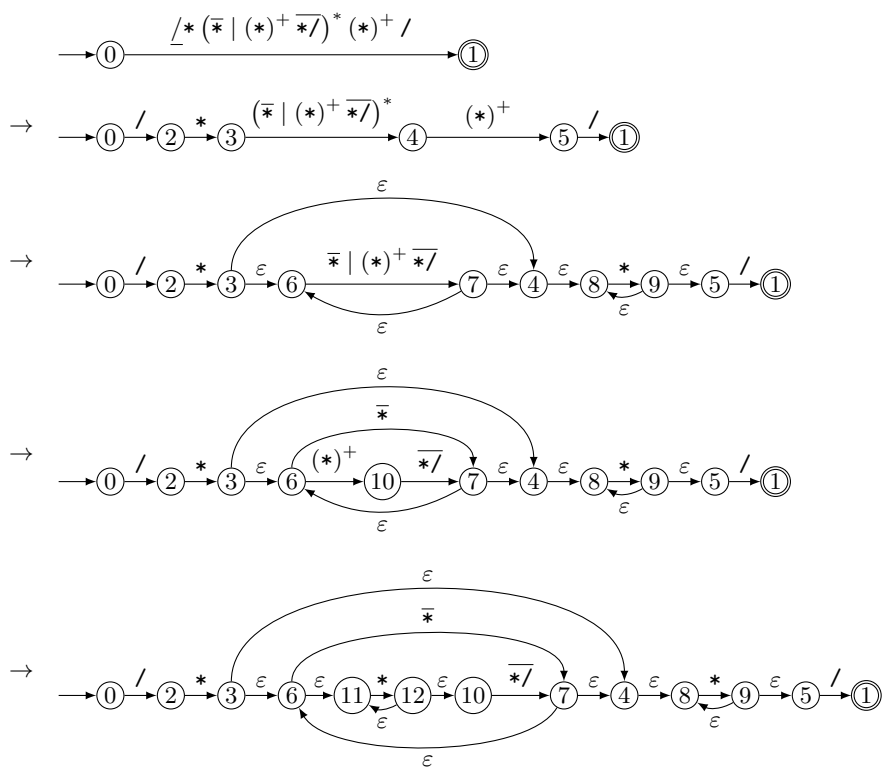
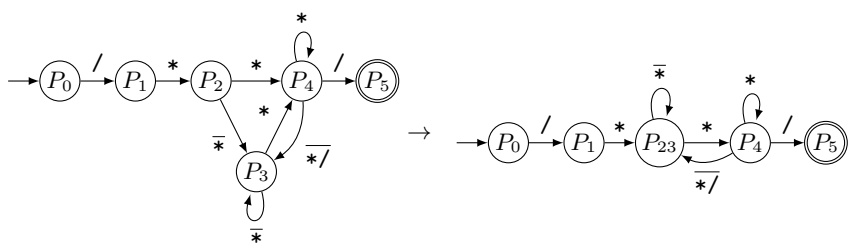


Fig. B.3. Deriving the NFA for string literals (Answer to Exercise 2.6)



Exercise 2.7. We use Algorithm 2.2 on the combined regular expression in Example 2.13, which defines the start state q_0; the other states are shown in Table B.1, with their quasi-regular forms, and their successor states. This defines the automaton shown in Figure B.4. Its three final states accept the different kinds of numbers: integers (2), fractionals (3), and floats (5).

Exercise 2.8. The scanner definition for LOOP *without keywords* is:

Table B.1. States, their quasi-regular form, and their successors in the DFA for number literals

No.	State	Quasi-Linear Form	Successors
q_0	$(\langle\text{digit}\rangle^*\underline{.})^?\ \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$	$\underline{.}\ \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$ $\mid \langle\text{digit}\ \rangle \left(\begin{array}{l}\langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\\ \mid \langle\text{exp}\rangle^?\\ \mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\end{array}\right)$	q_1 q_2
q_1	$\langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$	$\langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{exp}\rangle^?\\ \mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\end{array}\right)$	q_3
q_2	$\langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$ $\mid \langle\text{exp}\rangle^?$ $\mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$	ε $\mid \underline{.}\langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$ $\mid \underline{\text{E}}\ (+\mid -)^?\langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\ \rangle \left(\begin{array}{l}\langle\text{digit}\rangle^*\underline{.}\langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\\ \mid \langle\text{exp}\rangle^?\\ \mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\end{array}\right)$	q_1 q_4 q_2
q_3	$\langle\text{exp}\rangle^?$ $\mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?$	ε $\mid \underline{\text{E}}\ (+\mid -)^?\langle\text{digit}\rangle^+$ $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\langle\text{exp}\rangle^?\\ \mid \langle\text{digit}\rangle^+\langle\text{exp}\rangle^?\end{array}\right)$	q_4 q_3
q_4	$(+\mid -)^?\langle\text{digit}\rangle^+$	$\langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon\\ \mid \langle\text{digit}\rangle^+\end{array}\right)$ $\mid +\langle\text{digit}\rangle^+$ $\mid -\langle\text{digit}\rangle^+$	q_5 q_6 q_6
q_5	ε $\mid \langle\text{digit}\rangle^+$	ε $\mid \langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon\\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_5
q_6	$\langle\text{digit}\rangle^+$	$\langle\text{digit}\rangle \left(\begin{array}{l}\varepsilon\\ \mid \langle\text{digit}\rangle^+\end{array}\right)$	q_5

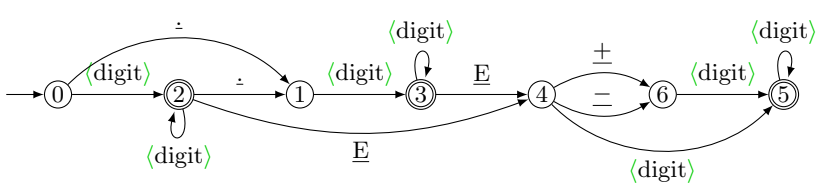


Fig. B.4. A combined DFA for integer, fraction, and float literals

```
%option noyywrap
/* Lexemes of LOOP, without keyword

/* needed for output of numbers */
%{
#include <math.h>
%}

/* character classes */
layout      [ \t\n\f]
letter      [a-zA-Z]
digit       [0-9]
noast       [^\*]
noclose     [^\)]
noastclose  [^\*\)]

/* lexeme definitions by regular expressions */
whitespace  {layout}+
comment     \(\*({noast}|\*+{noastclose})*\*+\)
prchar      [^\\\']|\\n|\\t|\\\'
string     \'({prchar})*\'
identifier {letter}({letter}|{digit})*
number     {digit}+

%%
/* screener transformation - rather trivial  */
{whitespace}  {printf("Whitespace: []%s[]\n", yytext);}
{comment}     {printf("A comment: %s\n", yytext);}
{identifier}  {printf("An identifier: %s\n", yytext);}
{string}      {printf("A string literal: %s\n", yytext);}
{number}      {printf("An integer literal: %s (%d)\n", yytext, atoi(yytext));}
"<="|">="|":="|"*"|"/"|"+"|"-"|"="|"#"|"<"|">"|"?"|"!"           {
              printf("An operator: %s\n", yytext);
              }
","|":"|";"|"("|")"           {
              printf("A delimiter: %s\n", yytext);
              }
(.)           {printf("an erroneous character: %s\n", yytext);}

%%
main(argc, argv)
int argc;
char **argv;
    {
        ++argv; --argc; /* skip over the program name */
```

```
␣␣␣␣␣␣␣␣if␣(argc␣>␣0␣)
␣␣␣␣␣␣␣␣␣␣␣␣yyin␣=␣fopen(argv[0],␣"r");
␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣yyin␣=␣stdin;
␣␣␣␣␣␣␣␣yylex();
␣␣␣␣}
```

The definition *with keywords* can be defined with the following regular expression:

keyword (*AND*|*ATTRIBUTE*|*BEGIN*|*CLASS*|*ELSE*|*ELSEIF*|*END*|*EXTENDS*|*IF*|*IS*|*METHOD*|*MOD*|*NEW*|*NOT*|*OR*|*OVERRIDE*|*READ*|*RETURN*)

The line

{*keyword*} { *printf*("A␣keyword:␣%s\n", *yytext*);}

has to appear before the line for identifiers.

Called with option `-v`, `flex` reports (without keywords):

```
...
116/2000 NFA states
30/1000 DFA states (165 words)
...
71 epsilon states, 40 double-epsilon states
...
532 table entries needed altogether
```

With keywords, it reports:

```
...
216/2000 NFA states
79/1000 DFA states (470 words)
...
90 epsilon states, 58 double-epsilon states
...
828 table entries needed altogether
```

Thus, keywords enlarge the scanner by 60%.

Answers to Exercises in Chapter 3

Exercise 3.1. The rules

$$R \to R\,\underline{|}\,R \mid R\,R \mid R\,\underline{+} \mid R\,\underline{*} \mid R\,\underline{?} \mid \underline{(}\,R\,\underline{)} \mid \underline{\mathrm{a}} \mid \underline{\mathrm{b}} \mid \underline{\mathrm{c}}$$

define regular expressions over three terminal symbols.

The grammar is ambiguous as the $\underline{\mathrm{a|ba}}$ und $\underline{\mathrm{a|b|c}}$ und $\underline{\mathrm{abc}}$ have two leftmost derivations each, because it fails to define the associativity of alternatives and concatenations, as well as their precedence.

As with expressions, two subcategories T and F of R have to be introduced, where the former does not generate alternatives, and the latter does generate neither alternatives, nor sequences.

$$\begin{aligned}
&R \to R\,\underline{|}\,T \mid R\\
&T \to T\,F \mid F\\
&F \to F\,\underline{+} \mid F\,\underline{*} \mid F\,\underline{?} \mid \underline{(}\,R\,\underline{)} \mid \underline{\mathrm{a}} \mid \underline{\mathrm{b}} \mid \underline{\mathrm{c}}
\end{aligned}$$

Then the example words above have only one leftmost derivation.

Exercise 3.2. The (unrevised) definition of Algol-60 for conditional commands allows two leftmost derivations of a nested if command with one (dangling) else:

$$\begin{aligned}
C &\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then } C\\
&\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then if } E \textbf{ then } C \textbf{ else } C\\
C &\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then } C \textbf{ else } C\\
&\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then if } E \textbf{ then } C \textbf{ else } C
\end{aligned}$$

We disambiguate the grammar by distinguishing *balanced commands* B (where every then part has an else part) from general commands:

$$\begin{aligned}
C &\triangleq \textbf{if } E \textbf{ then } C\\
&\mid B\\
B &\triangleq \textbf{if } E \textbf{ then } B \textbf{ else } C\\
&\mid R
\end{aligned}$$

The placement of B in the then part of a B enforces the desired grouping of a dangling else to the closest preceding then part, i.e., the grouping of the first leftmost derivation:

$$\begin{aligned}
C &\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then } C\\
&\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then } B\\
&\overset{\mathrm{lm}}{\Rightarrow} \textbf{if } E \textbf{ then if } E \textbf{ then } C \textbf{ else } C
\end{aligned}$$

Exercise 3.3. For the rules of statements, and the first and last rule rules of statement, the *First* sets are not disjoint, and thus violate the $SLL(1)$ condition.

We factorize these three rule pairs, where the conditional rules are factorized in the middle, in order to get an intuitive rule.

$$\begin{aligned}
\text{program} &\triangleq \text{statement } \underline{\square} \\
\text{statements} &\triangleq \text{statement statement_rest} \\
\text{statement_rest} &\triangleq \varepsilon \mid \underline{;}\ \text{statement statement_rest} \\
\text{statement} &\triangleq \textbf{if}\ \text{expression}\ \textbf{then}\ \text{statements else_part}\ \textbf{end if} \\
&\quad \mid \textbf{while}\ \text{expression}\ \textbf{do}\ \text{statements}\ \textbf{end while} \\
&\quad \mid \text{id assignment_rest} \\
\text{else_part} &\triangleq \varepsilon \mid \textbf{else}\ \text{statements} \\
\text{assignment_rest} &\triangleq \varepsilon \mid \textbf{:=}\ \text{expression}
\end{aligned}$$

Then the *First* and *Follower* sets are as follows:

Nonterminal	$Pref^1$	$Succ^1$
program	–	$\perp$
statements	$Pref^1$(statement)	$\underline{\text{elsif}}, \underline{\text{else}}, \underline{\text{end}}$
statement_rest	$\varepsilon, \underline{;}$	$Succ^1$(statements)
statement	$\underline{\text{id}}, \underline{\text{if}}, \underline{\text{while}}$	$\square, \underline{;}, \underline{\text{elsif}}, \underline{\text{else}}, \underline{\text{end}}$
else_part	$\varepsilon, \underline{\text{else}}$	$\underline{\text{end}}$
assignment_rest	$\varepsilon, \underline{:=}$	$Succ^1$(statement)

The grammar now satisfies the $SLL(1)$ condition.

Exercise 3.4. We use a version of the syntax with EBNF operators:

$$\begin{aligned}
\text{program} &\triangleq \text{statement } \underline{\square} \\
\text{statements} &\triangleq \text{statement } \{\underline{;}\ \text{statement}\} \\
\text{statement} &\triangleq \text{id } [\ \textbf{:=}\ \text{expression}\ ] \\
&\quad \mid \textbf{while}\ \text{expression}\ \textbf{do}\ \text{statements}\ \textbf{end} \\
&\quad \mid \textbf{if}\ \text{expression}\ \textbf{then}\ \text{statements} \\
&\qquad \{\textbf{elseif}\ \text{expression}\ \textbf{then}\ \text{statements}\ \} \\
&\qquad [\textbf{else}\ \text{statements}]\ \textbf{end}
\end{aligned}$$

Transformation into Recursive Parse Functions.

```
⟦statements ≜ ···⟧
= procedure statements;
  begin
    ⟦statement {statement }⟧
  end ;

= procedure statements;
  begin
    ⟦statement ⟧;
    while ℓ = ;
    loop scan;
      ⟦statement ⟧
    end loop;
  end ;

= procedure statements;
  begin
    statement;
    while ℓ = ;
    loop scan;
      statement
    end loop;
  end ;
```

```
⟦statement ≜ ···⟧
= procedure statement;
  begin
    ⟦id ... end⟧
  end ;

= procedure statement;
  begin
    if ℓ = id then ⟦id[:=expression ]⟧
    elsif ℓ = while then ⟦while ... end⟧
    elsif ℓ = if then ⟦if ... end⟧
    else ... end if ;
  end ;
```

```
= procedure statement;
  begin
    if ℓ = id then scan; if ℓ = := then scan; expression end if
    elsif ℓ = while then ⟦while ... end⟧
    elsif ℓ = if then ⟦if ... end⟧
    else ... end if ;
  end ;
= procedure statement;
  begin
    if ℓ = id then ... (wie oben)
    elsif ℓ = while
      then scan; expression; match(do); statements; match(end);
    elsif ℓ = if then ⟦if ... end⟧
    else ... end if ;
  end ;
```

```
= procedure statement;
  begin
    if ℓ = id then ... (wie oben)
    elsif ℓ = while then ... (wie oben)
    elsif ℓ = if
    then scan; expression; match(then); statements;
      ⟦{elsifexpression thenstatements }⟧ ⟦[elsestatements ]⟧
      match(end);match(if)
    else ... end if ;
  end ;
= procedure statement is
  begin
    if ℓ = id then ... (wie oben)
    elsif ℓ = while then ... (wie oben)
    elsif ℓ = if
    then scan; expression; match(then); statements;
      while ℓ = elsif
      loop scan; expression; match(then); statements;
      end loop;
      if ℓ = else then scan; statements; end if;
      match(end);match(if)
    else ... end if ;
  end ;
```

Construction of Abstract Syntax Trees. We just show how the parser for conditionals can be extended, which is the most interesting case: We assume that new(t1,If) allocates a record with components cond, thenpart, and elsepart. Missing else parts are represented by a node for the empty statement sequence (Skip); **elsif** parts are represented by nested If nodes.

Parsing with Tree Construction Every parsing procedure is turned a function returning its abstract syntax tree. The components of the If node are filled while parsing its components.

```
function statement return Stmt is
  var t1, t2: Stmt;
begin
  if ...
  elseif ...
  elsif ℓ = if
  then new(t1,If);
    scan; t1.cond := expression; match(then); t1.thenpart := statements; t2 := t1
    while ℓ = elsif
    do t2:= t1.elsepart; new(t1,If);
      scan; t2.cond := expression; match(then); t2.thenpart := statements; t2 := t2.elepart
    if ℓ = else then scan; t2.elsepart := statements; else t2.elsepart:= Skip;
    match(end);match(if);
    return t1
  else ...
  end if ;
kwreturn t1 end ;
```

Error Recovery.

1. *Insert a symbol.* If procedure match does not find the expected symbol, it acts as if this symbol has been present. This procedure is called if some part of an alternative has already been parsed, e.g., for the keywords **do**, **end**, **then**, **elseif**, and **else**.
2. *Skip symbols.* If no alternative of a rule can be parsed, error handling should skip all symbols until a possible successor of the rule is found, e.g., symbols □, ;, **elseif**, **else**, and **end** in the rule for statement.
3. *Replace a symbol.* This should be done only in situations where
   a) a specific symbol is expected,
   b) the parser will not know how to continue if it is missing,
   c) the input contains a symbol that is *similar* to the expected symbol, and
   d) the overnext symbol is something that may follow the expected symbol.
   In our example, procedure *match*(then) could replace words them, than, ... by **then** if the overnext input symbol ia a possible starter of statement. With **elsif** or **else**, this is not advisable, ab both symbols are optional, and resemble the next keyword **end**.

Exercise 3.5. The item set of the start set is

$$q_0 = \{S' \to \,.\, S\Box, S \to \,.\, L = R, S \to \,.\, R, L \to \,.\, * R, L \to \,.\, \underline{\mathrm{x}} R \to \,.\, L\}$$

The goto states of q_0 are

$$
\begin{aligned}
G(q_0, S) &= q_1 = \{S' \to S\,\textbf{.}\,\Box\} \\
G(q_0, L) &= q_2 = \{S \to L\,\textbf{.}\,= R, R \to L\,\textbf{.}\,\} \\
G(q_0, R) &= q_3 = \{S \to R\,\textbf{.}\,\} \\
G(q_0, *) &= q_4 = \{L \to *\,\textbf{.}\,R, R \to\,\textbf{.}\,L, L \to\,\textbf{.}\,*R, L \to\,\textbf{.}\,\underline{\mathrm{x}}\} \\
G(q_0, \underline{\mathrm{x}}) &= q_5 = \{L \to \underline{\mathrm{x}}\,\textbf{.}\,\}
\end{aligned}
$$

States q_2 and q_4 have further goto states, the last two are already known:

$$
\begin{aligned}
G(q_2, =) &= q_6 = \{S \to L =\,\textbf{.}\,R, R \to\,\textbf{.}\,L, L \to\,\textbf{.}\,*R, L \to\,\textbf{.}\,\underline{\mathrm{x}}\} \\
G(q_4, R) &= q_7 = \{L \to *R\,\textbf{.}\,\} \\
G(q_4, L) &= q_8 = \{R \to L\,\textbf{.}\,\} \\
G(q_4, *) &= \quad q_4 \\
G(q_4, \underline{\mathrm{x}}) &= \quad q_5
\end{aligned}
$$

Now q_6 has further goto states, but only one of the is new:

$$
\begin{aligned}
G(q_6, R) &= q_9 = \{S \to L = R\,\textbf{.}\,\} \\
G(q_6, L) &= \quad q_8 \\
G(q_6, *) &= \quad q_4 \\
G(q_6, x) &= \quad q_5
\end{aligned}
$$

Since q_9 does not have goto states, the state set is now complete. The simple CFA is shown in Figure B.5. The automaton has a shift-reduce conflict in state q_2: for the input "$\underline{=}$", we may shift, but reduce rule $R \to L$ as well, since $\underline{=} \in Fo^1(R)$. G_7 is not $SLR(1)$ although it is deterministic. The problem is that the SLR parser construction attempts to insert a reduction whenever a symbol is to be read that may, in some context, appear after the left-hand side of the rule, even if such a symbol may not occur in the particular context. In this case, $R \to L$ should only be reduced if the next symbol is $\Box$, because this reduction will only appear after the equal symbol has already been read. To do this in general, the lookaheads of items in the states of the CFA have to be augmented with lookahead symbols – this is the $LALR$ parser construction.

Exercise 3.6.

$$
\begin{array}{rrrr}
\vdash & 0\textbf{.}a - b * c\Box & \vdash & 0\underline{\mathrm{i}}3\textbf{.} - b * c\Box \\
\vdash & 0\mathsf{E}\ 1\textbf{.} - b * c\Box & \vdash & 0\mathsf{E}\ 1 - 4\textbf{.}b * c\Box \\
\vdash & 0\mathsf{E}\ 1 - 4\underline{\mathrm{i}}3\textbf{.} * c\Box & \vdash & 0\mathsf{E}\ 1 - 4\mathsf{E}\ 7\textbf{.} * c\Box \\
\vdash & 0\mathsf{E}\ 1 - 4\mathsf{E}\ 1 * 5\textbf{.}c\Box & \vdash & 0\mathsf{E}\ 1 - 4\mathsf{E}\ 1 * 5\underline{\mathrm{i}}3\textbf{.}\Box \\
\vdash & 0\mathsf{E}\ 1 - 4\mathsf{E}\ 1 * 5\mathsf{E}\ 8\textbf{.}\Box & \vdash & 0\mathsf{E}\ 1 - 4\mathsf{E}\ 7\textbf{.}\Box \\
\vdash & 0\mathsf{E}\ 1\textbf{.}\Box & \vdash & \text{yes!}
\end{array}
$$

$$
\begin{array}{rrrr}
\vdash & 0\textbf{.}(a - b) * c\Box & \vdash & 0(2\textbf{.}a - b) * c\Box \\
\vdash & 0(2\underline{\mathrm{i}}3\textbf{.} - b) * c\Box & \vdash & 0(2\mathsf{E}\ 6\textbf{.} - b) * c\Box \\
\vdash & 0(2\mathsf{E}\ 6 - 4\textbf{.}b) * c\Box & \vdash & 0(2\mathsf{E}\ 6 - 4\underline{\mathrm{i}}3\textbf{.}) * c\Box \\
\vdash & 0(2\mathsf{E}\ 6 - 4\mathsf{E}\ 7\textbf{.}) * c\Box & \vdash & 0(2\mathsf{E}\ 6\textbf{.}) * c\Box \\
\vdash & 0(2\mathsf{E}\ 6)9\textbf{.} * c\Box & \vdash & 0\mathsf{E}\ 1\textbf{.} * c\Box \\
\vdash & 0\mathsf{E}\ 1 * 5\textbf{.}c\Box & \vdash & 0\mathsf{E}\ 1 * 5\underline{\mathrm{i}}3\textbf{.}\Box \\
\vdash & 0\mathsf{E}\ 1 * 5\mathsf{E}\ 8\textbf{.}\Box & \vdash & 0\mathsf{E}\ 1\textbf{.}\Box \\
\vdash & \text{yes!} & \vdash &
\end{array}
$$

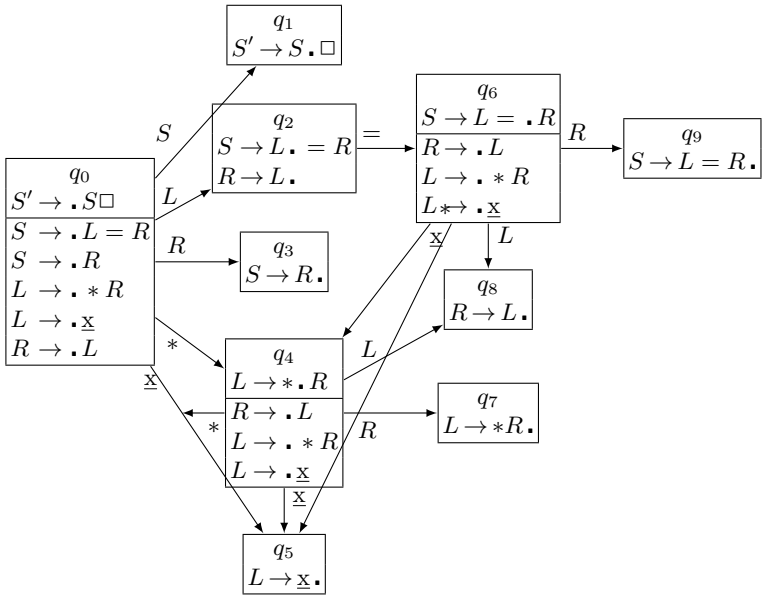


Fig. B.5. Diagram of the simple CFA for G_x (Exercise 3.5)

$\vdash$	$0.(a-b)c\square$	$\vdash$	$0(2.a-b)c\square$
$\vdash$	$0(2\underline{i}3.-b)c\square$	$\vdash$	$0(2\mathsf{E}\ 6.-b)c\square$
$\vdash$	$0(2\mathsf{E}\ 6-4.b)c\square$	$\vdash$	$0(2\mathsf{E}\ 6-4\underline{i}3.)c\square$
$\vdash$	$0(2\mathsf{E}\ 6-4\mathsf{E}\ 7.)c\square$	$\vdash$	$0(2\mathsf{E}\ 6.)c\square$
$\vdash$	$0(2\mathsf{E}\ 6)9.c\square$	$\vdash$	no!

$\vdash$	$0.(a-b)*-c\square$	$\vdash$	$0(2.a-b)*-c\square$
$\vdash$	$0(2\underline{i}3.-b)*-c\square$	$\vdash$	$0(2\mathsf{E}\ 6.-b)*-c\square$
$\vdash$	$0(2\mathsf{E}\ 6-4.b)*-c\square$	$\vdash$	$0(2\mathsf{E}\ 6-4\underline{i}3.)*-c\square$
$\vdash$	$0(2\mathsf{E}\ 6-4\mathsf{E}\ 7.)*-c\square$	$\vdash$	$0(2\mathsf{E}\ 6.)*-c\square$
$\vdash$	$0(2\mathsf{E}\ 6)9.*-c\square$	$\vdash$	$0\mathsf{E}\ 1.*-c\square$
$\vdash$	$0\mathsf{E}\ 1*5.-c\square$	$\vdash$	no!

Exercise 3.7.

1. The grammar is left-recursive.
2. The rules can be transformed into the following EBNF rules, where M is short for "modifier":

$$
\begin{array}{rcll}
S & \triangleq & X & (1\ \ ) \\
X & \triangleq & \underline{\mathrm{x}}\,\{M\} & (2'\ ) \\
M & \triangleq & \underline{\uparrow} & (3'\ ) \\
 & | & \underline{.}\,\underline{\mathrm{X}} & (4'\ ) \\
 & | & \underline{[}\,E\,\underline{]} & (5'\ ) \\
 & | & \underline{(}\,A\,\underline{)} & (6'\ ) \\
A & \triangleq & E\{\underline{,}\,E\} & (7|8) \\
E & \triangleq & \underline{\mathrm{e}} & (9\ \ )
\end{array}
$$

   This grammar is $SLL(k)$ because the rules for M start with mutually distinct symbols.
3. *Recursive Descent parsers still missing*

4. *Abstract top-down parses still missing*
5. *Abstract syntax still missing*
6. *abstract bottom-up parses still missing*

Exercise 3.8. Variables could be defined in `yacc` as follows:

```
%token id E

%% /* Syntax      { semantic actions        } */

X : id              { $$ = key(yyvalue);      }
  | X "*"           { $$ = Deref($2);         }
  | X "." id        { $$ = Select($1,$3);     }
  | X "[" A "]"     { $$ = Subscribe($1,$3);  }
  | X "("   ")"     { $$ = Call($1,Empty);    }
  | X "(" A ")"     { $$ = Call($1,$3);       }
  ;

A : E               { $$ = Cons($1,Empty);    }
  | E "," A         { $$ = Cons($1,$3);       }
  ;
```

The grammar is $LALR(1)$ as `yacc` succeeds without error messages (and without specifying priorities). The CFA can be produced by calling `yacc` with the option `--report=state`.

Exercise 3.9. The following `yacc` definition specifies **else** to bind stronger than **then**:

```
%token O E

/* operator associativity and precedences                    */

%left "then"
%left "else"

%% /* Syntax                  { semantic actions              } */

C : O                          { $$ = $1;                        }
  | "if" E "then" C            { $$ = IfThenElse($2, $4, Skip); }
  | "if" E "then" C "else" C   { $$ = IfThenElse($2, $4, $6);   }
  ;
```

Inspection of the output[1] reveals that in the state with conflicting items

$C \to$ **if** E **then** C**.** and $C \to$ **if** E **then** C**.else** C

[1] File `y.output` after calling `yacc --report=all danglingelse.y`.

the conflict is resolved in favor of shift of **else** so that the **else** is bound to the latest **then**. As to be expected, the grammar *without* priority specification has a shift-reduce conflict.

Answers to Exercises in Chapter 4

Exercise 4.1. Is the following sample "program" *wellformed*?

$$\{x : \mathbf{proc}; \{x : \mathbf{var}; y; x\}; y : \mathbf{var}; x\}$$

(1) In C, it is not well-formed as the declaration of y should occur before its use. This reshuffled program is wellformed:

$$\{x : \mathbf{proc}; y : \mathbf{var}; \{x : \mathbf{var}; y; x\}; x\}$$

(2) In Java or Haskell, the program is well-formed, as the order of declarations and uses in a block does not matter (as long as declarations do not come with initializations).

Exercise 4.2. The interface of the declaration table may look as follows:

```
class Tab {
  static Tab initial() {...};
  static Tab enter (Identifier id, Kind kd, Tab t) {...};
  static bool isLocal (Identifier id, Tab t) {...};
  static Kind def (Identifier id, Tab t) {...};
  static Tab nest (Tab t) {...};
  static Tab unnest (Tab t) {...};
}
```

The module class is omitted. The abstract class *Statement* with the abstract method *idfy* and its subclasses can be defined as follows:

```
abstract class Statement{
  abstract Tab idfy (Tab v);
}

class Declaration extends Statement {
  Identifier id;
  Kind kd,

  Tab idfy(Tab v) {
    if (Tab.isLocal(this.id,v))
       {error ("doule␣declaration!");
        return v;
       }
    else { return Tab.enter(this.id, this.kd, v); }
  }
}

class Use extends Statement {
  Identifier id;
```

```
  Kind kd;

  Tab idfy(Tab v) {
    this.kd = Tab.def(this.kd, v);
    if (this.kd = null)
       {error ("missing declaration!"); }
    return v
  }
}

class Block extends Statement {
  Statement body;

  Tab idfy(Tab v) {
    return Tab.unnest(this.body.idfy(Tab.nest(v)));
  }
}

class List extends Statement {
  LIST<Statement> sequence;

  Tab idfy(Tab v) {
    foreach st in this.sequence
      { v = st.idfy(v); }
    return v;
  }
}

class Identifiewr {
  ...
}
class Kind {
  ...
}
```

Exercise 4.3. The interface of the declaration table can be simplified as follows:

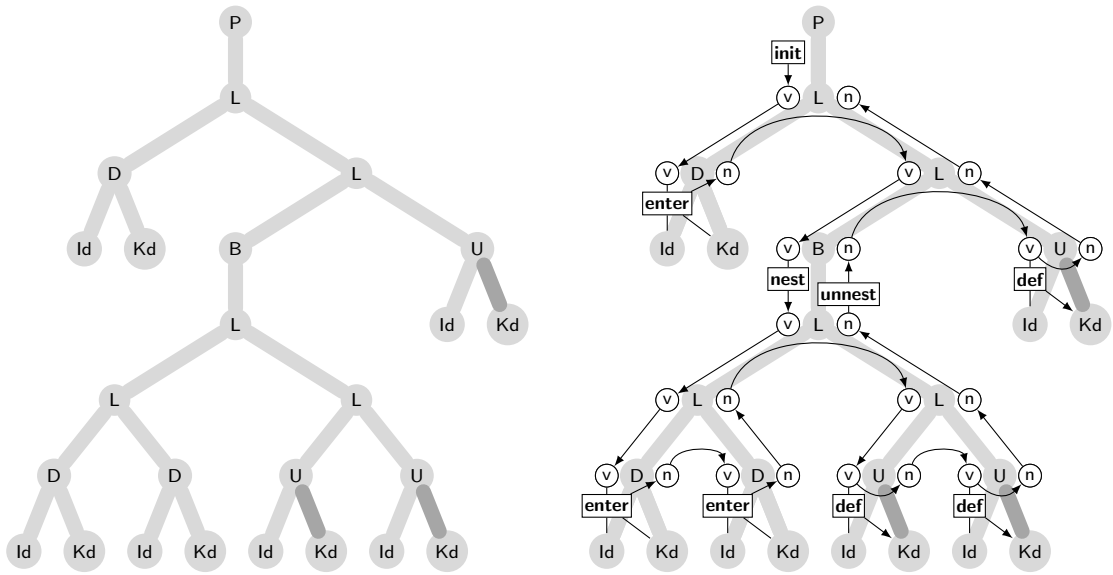
```
static class Tab {
  ... hidden_tab = initial();
  static Tab initial() {...};
  static void enter (Identifier id, Kind kd) {...};
  static bool isLocal (Identifier id) {...};
  static Kind def (Identifier id) {...};
  static void nest () {...};
  static void unnest () {...};
}
```

The signature of the method *idfy* can be simplified, so that the evaluator *idfy* gest a more imperative flavor:

```
abstract class Statement{
  abstract void idfy ();
}

class Declaration extends Statement {
  Identifier id;
  Kind kd,

  void idfy() {
    if (Tab.isLocal(this.id))
       {error ("doule declaration!");
       }
    else { Tab.enter(this.id, this.kd); }
  }
}

class Use extends Statement {
  Identifier id;
  Kind kd;

  void idfy() {
    this.kd = Tab.def(this.kd);
    if (this.kd = null)
       {error ("missing declaration!"); }
  }
}

class Block extends Statement {
  Statement body;

  void idfy() {
    Tab.nest();
    this..body.idfy();
    Tab.unnest();
  }
}

class List extends Statement {
  LIST<Statement> sequence;

  void idfy(Tab v) {
    foreach st in this.sequence
      { st.idfy();}
```

```
  }
}
```

Exercise 4.4. 1. The program has the abstract syntaxt tree below left.
2. Applying the attribute rules yields the attrbuted tree below right.



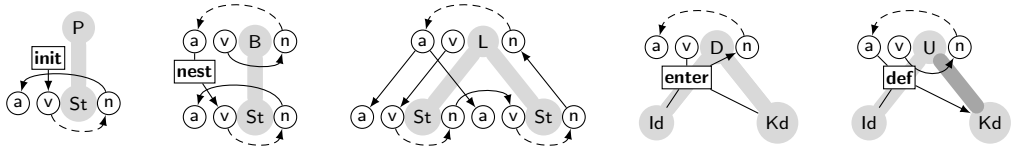
3. The first uses of `y` and `x` refer to `k3` and `k2`, resp.; the last use of `x` refers to `k1`.

Exercise 4.5. We use the abstract syntax known from Exercise 4.4. The semantical basis should be extended by one operation:

- The predicate `isLocal` can be useful to determine whether an identifier has already be declared within the current block.

Identification with Global Visibility. If declarations shall be visible if they appaer after (right of) their use, the attribute rulkes have to be extended as follows:

- An attribute (a) is added to **St**. It shall contain *all* declarations visible in a block.
- This is the attribute used in **U**.
- Rules **L**, **D**, and **U** collect attributes v and n as before.
- In rule (**B**), the attribute a for the contained statement is defined to be its n attribute. Th v attribute of the statement is obtained by applying nest to the a attribute of the block.
- In **L**, the attribute a is distributed to all children.



The evaluator for this attribute grammar needs more than one traversal of every node:

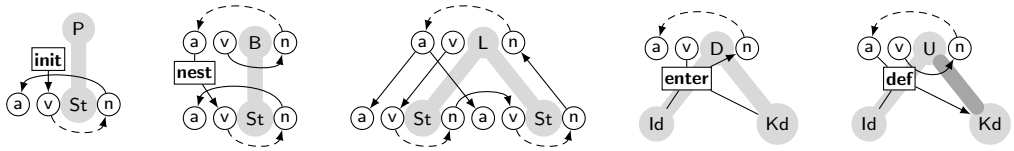
1. For each block, the declarations are collected by computing the attributes v and n.
2. Then all uses in that block are evaluated, and all blocks, recursively.

The traversal is nested: The program node and every block has to be traversed twice before the nested blocks are evaluated. In the Haskell function, this is done in two functions, collect: TAB → TAB and idfy: TAB → St → St.

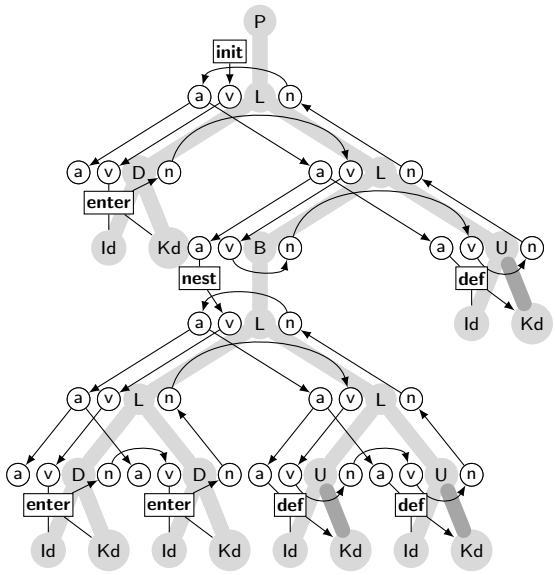
```
idfy' :: Pr → Pr
idfy' (P s) ≜ P s' where s' ≜ idfy (collect (initial) s)

collect:: TAB → St → TAB
collect v (U _ _  ) ≜ n          where n ≜ v
collect v (B s    ) ≜ unnest n where n ≜ collect (nest v) s
collect v (D x k  ) ≜ n          where n ≜ enter v (x,k))
collect v (L s1 s2) ≜ n'         where n  ≜ collect v s1; n' ≜ collect n s2

idfy:: TAB → St → St
idfy a (U x _  ) ≜ U x k'     where k' ≜ def a x
idfy a (B s    ) ≜ B s'       where s' ≜ idfy a' s; a' ≜ collect (nest a) s
idfy a (D x k  ) ≜ D x k
idfy a (L s1 s2) ≜ L s1' s2' where s1' ≜ idfy a s1; s2' ≜ idfy a s2
```

Error handling is not yet included in these definitions.

Exercise 4.6. Graphically, the attribute rules look as follows:



The attribution of a tree is as follows:



Exercise 4.7. Just for fun, I have studied the rather elaborate type coercions in Algol-68.

Type Coercions Types can be coerced in the following ways::

1. *Widening* extends the precision of numbers: **short short int** → **short int** → **int** → **long int** → **long long int**, or extends the kind of numbers: **int** → **real** → **compl**.
2. *Dereferencing* yields the base type of a reference (pointer): **ref** $\tau \to \tau$.
3. *Voidening* coerces types to the unit type: $\tau \to$ **void**.
4. *Deproceduring*) calls a parameterless procedure: **proc** $\tau \to \tau$.
5. *Uniting* coerceds a type to a union type containing it: $\tau \to$ **union** $\tau\ \sigma$.
6. *Rowing* coerces values to one-element arrays: $\tau \to [\,i : i\,]\ \tau$.

Contexts for type coercions The following program parts are subject to coercions:

1. *strong*: Actual to formal parameter of a procedure: all coercions.
2. *firm*: Operands to argument types of operations: no widening, no voiding, no rowing.
3. *meek*: Conditions of if statements: only dereferencing and deproceduring.
4. *weak*: Operands of structure selections and array subscriptions: only dereferencing and deproceduring.
5. *soft*: right-hand side to left-hand side of an assignment: onlx deproceduring.

Further rules define how single coercions can be combined in the different contexts.

Not bad, isn't it?

Exercise 4.8. The signatures of the semantic basis is turned into method signatures by considering the table as their receiver object.

We assume that identifiers are represented as numbers in the range $\{0, \ldots, n\}$.

The contents of a table consists of objects of type `Entry`.

```
class Entry {
   int Id;
   Decl decl;
   int level;
   Entry next, global;
   }

class TAB {
   int currentLevel;
   LinkedList<Entry> nesting;
   Entry[] entries = new Entry[n];

   public void init() {
     nest();
     enter(x_1, d_1);
—    ...
     enter(x_n, d_n);
     };

   public void nest() {
     currentLevel++;
     nesting.addFirst()
     };

   public void unnest() {
     currentLevel—;
     Entry e = nesting.getfirst();
     while e != null {
        entries[nesting.Id] = entries[nesting.Id].global;
        e = e.next
        };
      nesting.removeFirst();
     };

   public void enter(Id x, Decl d) {
      Entry e = new Entry;
      e.Id = x;
      e.decl = d;
      e.level = currentLevel;
      e.next = nesting.getFirst();
```

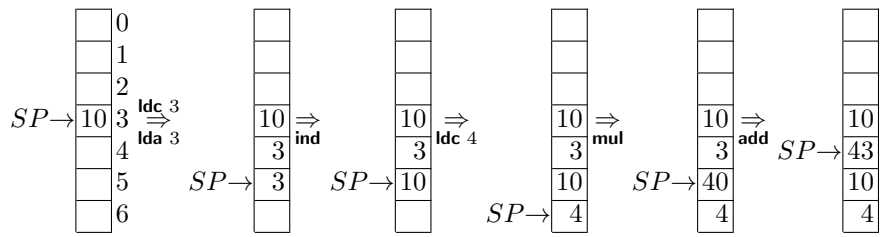
```
      e.global = entries[x];
      entries[x] = e
    };

  public Decl def(Id x) {
      return entries[x].decl
    };

  public bool isLocal(Id x) {
      return (entries[x] != null)
          && (entries[x].level == currentLevel)
    };
  }
```

Answers to Exercises in Chapter 5

Exercise 5.1. The expression$3+(x*4)$ mit $\alpha(x) = 3$ is transformed as follows:

$$
\begin{aligned}
&\mathrm{code}_W(3 + (x * 4))\alpha \\
&= \mathrm{code}_W(3)\alpha; \mathrm{code}_W(x * 4)\alpha; \textbf{add}; \\
&= \textbf{ldc}\ 3; \mathrm{code}_W(x)\alpha; \mathrm{code}_W(4)\alpha; \textbf{mul}; \textbf{add}; \\
&= \textbf{ldc}\ 3; \mathrm{code}_A(x)\alpha; \textbf{ind}; \textbf{ldc}\ 4; \textbf{mul}; \textbf{add}; \\
&= \textbf{ldc}\ 3; \textbf{lda}\ 3; \textbf{ind}; \textbf{ldc}\ 4; \textbf{mul}; \textbf{add};
\end{aligned}
$$

The store of the P machine takes the following states:



Exercise 5.2. 1. *Predefined Functions.*

$$
\begin{aligned}
\mathrm{code}_W(sf(e_1, \ldots e_k))\,\alpha = {} & \mathrm{code}_W(e_1)\,\alpha; \\
& \vdots \\
& \mathrm{code}_W(e_k)\,\alpha; \\
& \textbf{jpsf}\ \alpha(sf)
\end{aligned}
$$

Instructions **jpsf** and **retsf** could be defined as follows:

jpsf a	$OPC := PC;\ PC := a$
retsf	$PC := OPC$

For expressions over values that occupy $k > 1$ storage cells, the transformation scheme can stay as it was.

The invariant for code_W must be generalized so that $\mathrm{code}_W(e)$ sets k top cells of the stack for every expression yielding a value of that size. The P-Code instructions $\textbf{monop}_\oplus$ and $\textbf{dyop}_\otimes$ have to know about the size of their arguments, and how to access their components. The same would hold for predefined functions.

2. *Compound Operands.* The transformation of identifiers and literals would have to be extended as follows:

$$
\begin{aligned}
\mathrm{code}_W(x)\,\alpha &= \mathrm{code}_A(x)\,\alpha; \textbf{ind} && \text{wenn } gr(x) = 1 \\
&= \mathrm{code}_A(x)\,\alpha; \textbf{mvs}\ gr(x) && \text{sonst} \\
\mathrm{code}_W(l)\,\alpha &= \textbf{ldc}\ r_1;\ \ldots;\ \textbf{ldc}\ r_k;
\end{aligned}
$$

Here $(r_1, \ldots, r_k)$ represent the literal l in $k > 1$ storage cells, and the instruction **mvs** (“*move static block*”) is defined as follows:

mvs n	**for** $i := n-1$ **downto** 0
	do $S[SP+i] := S[S[SP]+i]$
	end ;
	$SP := SP + n - 1$

(Copying is done from top to bottom so that the address of the block is overwritten only in the last step.)

Exercise 5.3. Computed gotos for numeric choices (**case**) are generated as follows:

$$
\begin{array}{l}
\text{code}(\textbf{case}\ e\ \textbf{of}\ u : C_u; \ldots;\ o : C_o\ \textbf{default}\ C_\mathrm{d})\alpha \\
\ = \ \text{code}_W(e)\alpha; \\
\quad \textbf{ljp}\ ; u\ l_\mathrm{d} \\
\quad \textbf{gjp}\ ; o\ l_\mathrm{d} \\
\quad \textbf{ldc}\ u \\
\quad \textbf{sub}; \\
\quad \textbf{ijp}; \\
\quad \textbf{ujp}\ l_u; \cdots ; \textbf{ujp}\ l_o; \\
l_u : \text{code}(C_u)\alpha; \textbf{ujp}\ l_\mathrm{e}; \\
\quad \vdots \\
l_o : \text{code}(C_o\alpha; \textbf{ujp}\ l_\mathrm{e} \\
l_\mathrm{d} : \text{code}(C_n)\alpha; \\
l_\mathrm{e} : \ldots
\end{array}
$$

This kind of transformation is good as long as the range $0..n$ is rather small, since the goto table occupies n code cells. Two instructions have to be executed for every choice, a typical trade-off between storage and time requirements.

Transformation into nested ifs is good for big n, and if many of the selected commands are actually equal.

Numeric choice should be extended to join cases with equal commands. Then the transformation would look as follows:

$$
\begin{array}{lll|lll}
\textbf{case} & e & & \textbf{var} & t : \mathsf{Integer} := e; & \\
\textbf{of} & c_{0,1}, \ldots, c_{0,k_0} : C_0; & & \textbf{if} & t \in \{c_{0,1}, \ldots, c_{0,k_0}\}\ \textbf{then}\ C_0 & \\
| & c_{1,1}, \ldots, c_{1,k_1} : C_1; & \Rightarrow & \textbf{elsif} & t \in \{c_{1,1}, \ldots, c_{1,k_1}\}\ \textbf{then}\ C_1 & \\
| & \vdots & & & \vdots & \\
| & c_{n,1}, \ldots, c_{n,k_n} : C_n & & \textbf{elsif} & t \in \{c_{n,1}, \ldots, c_{n,k_n}\}\ \textbf{then}\ C_n & \\
\textbf{default}\ C_e & & & \textbf{else} & C_e &
\end{array}
$$

The condition "$t \in \{c_{n,1}, \ldots, c_{n,k_n}\}$" must be expresses as a disjunction "$t = c_{n,1} \vee \cdots \vee c_{n,k_n}\}$" if the language does not provide power sets over the type of e.

Exercise 5.4. It holds that $gr(\mathsf{Complex}) = 2$ and

$$
\begin{aligned}
gr(\textbf{array}\ [-2..+2\textbf{of}\ \mathsf{Complex}) &= (lo - hi + 1) \times gr(\mathsf{Complex}) \\
&= (2 - (-2) + 1) \times 2 = 5 \cdot 2 = 10
\end{aligned}
$$

$a[-3].re$		0
$a[-3].im$		1
$a[-2].re$		2
$a[-2].im$		3
$a[-1].re$		4
$a[-1].im$		5
$a[0].re$		6
$a[0].im$		7
$a[+1].re$		8
$a[+1].im$		9
$a[+2].re$		10
$a[+2].im$		11
$a[+3].re$		12
$a[+3].im$		13
i	1	14

(a) Storing a and i

$a[-3].re$		0
$a[-3].im$		1
$a[-2].re$		2
$a[-2].im$		3
$a[-1].re$		4
$a[-1].im$		5
$a[0].re$		6
$a[0].im$		7
$a[+1].re$		8
$a[+1].im$		9
$a[+2].re$		10
$a[+2].im$		11
$a[+3].re$		12
$a[+3].im$		13
i	1	14
$\alpha(a)$	0	15
$SP \rightarrow \quad i$	1	16

(b) Stack before **ixa** 2; **inc** 6 ...

$a[-3].re$		0
$a[-3].im$		1
$a[-2].re$		2
$a[-2].im$		3
$a[-1].re$		4
$a[-1].im$		5
$a[0].re$		6
$a[0].im$		7
$a[+1].re$		8
$a[+1].im$		9
$a[+2].re$		10
$a[+2].im$		11
$a[+3].re$		12
$a[+3].im$		13
i	1	14
$SP \rightarrow \quad \alpha(a[i])$	$0+2+6$	15

(c) ... and afterwords

Fig. B.6. Storage and subscription of static arrays

The address environment defines $\alpha(a) = 0$ and $\alpha(i) = 14$. The transformation is as follows; we assume staic adresses, and omit the bounds checks "**chk** $u\,o$":

$\text{code}(i := 1)\alpha$
$= \text{code}_A(i)\alpha;\ \text{code}_W(1)\alpha;$ **sto**
$=$ **loa** $\alpha(i)$; **ldc** 1; **sto**
$\text{code}(a[i] := a[i+1])\alpha$
$= \text{code}_A(a[i])\alpha;\ \text{code}_W(a[i+1])\alpha;$ **sto** 2;
$= \text{code}_A(a)\alpha;\ \text{code}_W(i)\alpha;$ **ixa** 2; **inc** 6; $\text{code}_A(a[i+1])\alpha;$ **ind**; **sto** 2;
$=$ **loa** $\alpha(a)$; $\text{code}_A(i)\alpha;$ **ind**; **ixa** 2; **inc** 6; $\text{code}_A(a)\alpha;\ \text{code}_W(i+1)\alpha;$ **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** $\alpha(i)$; **ind**; **ixa** 2; **inc** 6; **loa** $\alpha(a)$; $\text{code}_W(i)\alpha;\ \text{code}_W(1)\alpha;$ **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** 14; **ind**; **ixa** 2; **inc** 6; **loa** 0; $\text{code}_A(i)\alpha;$ **ind**; **ldc** 1; **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;
$=$ **loa** 0; **loa** 14; **ind**; **ixa** 2; **inc** 6; **loa** 0; **loa** 0; **ind**; **ldc** 1; **add**; **ixa** 2; **inc** 6; **mvs** 2; **sto** 2;

Executing the first command "$i := i + 1$" sets the first storage cell to 1. Then the storage for laocal variables looks as in Figure B.6(a). The store operation is as follows, where the modifier is the element size:

sto n	**for** $i := 0$ **to** $n-1$ **do** $S[S[SP-n]+i] := S[SP-n+1+i]$ $SP := SP - n - 1$

Yes, this code is far from optimal.

Exercise 5.7. The block nesting is fixed, say at $\ell_{\max}$. So static predecessors can be stored in the cells $S[0]$ to $S[\ell_{\max} - 1]$.

Actions upon Procedure Entry

Let us assume that procedure p calls q which is declared on level ℓ_q.

1. $S[FP + SVV] := S[\ell_q]$; The old static predecessor is stored in the frame of q.
2. $S[\ell_q] := FP$; the current FP of q is the base address for q.

Actions upon Procedure Exit

1. $S[\ell_q] := S[FP + SVV]$; The previous static predecessor is restored.

Load Instruction for Non-Local Variables

The access operation uses a cell of the displax vector as a basis.

$$\begin{aligned}\textbf{lda}\ a\ \ell \mathrel{\hat{=}}\ & SP := SP + 1;\\ & S[SP] := S[S[\ell]] + a;\end{aligned}$$

This is more efficient if the variable is declared more than one level above its acccess.

Load Instruction for Local Variables

The register FP can be used instead of the static predecessor.

$$\begin{aligned}\textbf{ldl}\ a \mathrel{\hat{=}}\ & SP := SP + 1;\\ & S[SP] := S[FP] + a;\end{aligned}$$

(Access to a register is – a little – more efficient than acces to the storage celle $S[currentLevel]$.

Load Instruction for Global Variables

Global variables have completely static addresses.

$$\begin{aligned}\textbf{ldg}\ a \mathrel{\hat{=}}\ & SP := SP + 1;\\ & S[SP] := a;\end{aligned}$$

Exercise 5.8. The PASCAL function

```
function fac (n: Integer): Integer;
begin
 if n <= 0 then fac := 1 else fac := n * fac(n-1)
end
```

is translated as follows:

$$
\begin{array}{l}
\text{code}(\textbf{function}\ \ \text{fac}\ldots)\alpha \\
= f:\ \textbf{ssp}\ l';\ \textbf{sep}\ k';\ \textbf{ujp}\ m'; \\
\quad m':\ \text{code}(\textbf{if}\ \mathsf{n} \leqslant 0\ \textbf{then}\ \ \text{fac} := 1\ \textbf{else}\ \ \text{fac} := \mathsf{n} * \text{fac}(\mathsf{n}-1))\alpha;\ \textbf{retp}
\end{array}
$$

The variable fac for the result of the function is on address -2, and its value parameter n on address -1. The size l' of local storage is 4 (for the organization cells) – there are no local variables. The maximal stack growth, needed for register EP, equals 4. The jump to m' can be omitted. The code continues as follows:

$$
\begin{array}{rl}
= & \textbf{ssp}\ 4;\ \textbf{sep}\ 3; \\
 & \text{code}_W(\mathsf{n} \leqslant 0)\alpha;\ \textbf{jpf}\ l''; \\
 & \text{code}(\text{fac} := 1)\alpha;\ \textbf{ujp}\ l'''; \\
l'': & \text{code}(\text{fac} := \mathsf{n} * \text{fac}(\mathsf{n}-1))\alpha; \\
l''': & \textbf{retp}
\end{array}
$$

$$
\begin{array}{rl}
= & \textbf{ssp}\ 4;\ \textbf{sep}\ 3; \\
 & \textbf{loa}\ 0\,(-1);\ \textbf{ind};\ \textbf{ldc}\ 0;\ \textbf{leq}\ ;\ \textbf{jpf}\ l''; \\
 & \textbf{loa}\ 0\,(-2);\ \textbf{ldc}\ 1;\ \textbf{sto}\ 1;\ \textbf{ujp}\ l'''; \\
l'': & \textbf{loa}\ 0\,(-2);\ \textbf{loa}\ 0\,(-1);\ \textbf{ind}\ ;\ \text{code}_W(\text{fac}(\mathsf{n}-1))\alpha;\ \textbf{mul}\ ;\ \textbf{sto}\ 1; \\
l''': & \textbf{retp}
\end{array}
$$

$$
\begin{array}{rl}
= & \textbf{ssp}\ 4;\ \textbf{sep}\ 3; \\
 & \textbf{loa}\ 0\,(-1);\ \textbf{ind};\ \textbf{ldc}\ 0;\ \textbf{leq}\ ;\ \textbf{jpf}\ l''; \\
 & \textbf{loa}\ 0\,(-2);\ \textbf{ldc}\ 1;\ \textbf{sto}\ 1;\ \textbf{ujp}\ l'''; \\
l'': & \textbf{loa}\ 0\,(-2);\ \textbf{loa}\ 0\,(-1);\ \textbf{ind}\ ;\ \textbf{mst}\ 0;\ \text{code}_W(\mathsf{n}-1)\alpha;\ \textbf{cup}\ 2\,f;\ \textbf{mul}\ ;\ \textbf{sto}\ 1; \\
l''': & \textbf{retp}
\end{array}
$$

$$
\begin{array}{rl}
= & \textbf{ssp}\ 4;\ \textbf{sep}\ 3; \\
 & \textbf{loa}\ 0\,(-1);\ \textbf{ind};\ \textbf{ldc}\ 0;\ \textbf{leq}\ ;\ \textbf{jpf}\ l''; \\
 & \textbf{loa}\ 0\,(-2);\ \textbf{ldc}\ 1;\ \textbf{sto}\ 1;\ \textbf{ujp}\ l'''; \\
l'': & \textbf{loa}\ 0\,(-2);\ \textbf{loa}\ 0\,(-1);\ \textbf{ind}\ ;\ \textbf{mst}\ 0;\ \textbf{loa}\ 0\,(-2);\ \textbf{ind}\ ;\ \textbf{ldc}\ 1;\ \textbf{sub};\ \textbf{cup}\ 2\,f;\ \textbf{mul}\ ;\ \textbf{sto}\ 1; \\
l''': & \textbf{retp}
\end{array}
$$

In Figure B.7 we show the complete code on the left, and the state of the storage in several instances (calls) of fac. On the right, we see the state immediatly before returning from recursion (n equals 0).

Exercise 5.9.

Constant Parameters

1. Constant parameters are names for immutible values. In contrast to value parameters, no value may be assigned to them.
2. If procedure calls are sequential, the value of the actual parameter need not be copied, as it cannot be changed during the call.
3. Value parameters may save to allocate one local varibale.

0	**ssp** l	
1	**sep** k	
2	**ujp** m	
$f =$ 3	**ssp** l'	
4	**sep** k'	
5	**loa** $0\,(-1)$	
6	**ind**	
7	**ldc** 0	
8	**leq**	
9	**jpf** l''	
10	**loa** $0\,(-2)$	
11	**ldc** 1	
12	**sto** 1	
13	**ujp** l'''	
$l'' =$ 14	**loa** $0\,(-2)$	
15	**loa** $0\,1$	
16	**ind**	
17	**mst** 0	
18	**loa** $0\,(-2)$	
19	**ind**	
20	**ldc** 1	
21	**sub**	
22	**cup** $2\,f$	
23	**mul**	
24	**sto** 1	
$l''' =$ 25	**retp**	
$m =$ 26	**loa** $0\,0$	
27	**mst** 0	
28	**ldc** 1	
29	**cup** $2\,f$	
30	**sto** 1	
31	**stp**	

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\mathrm{fac}_0)$
9	2	$val(n_0)$
10		fac_1
11	1	n_1
12	0	SV
13	4	DV
14	4	EP
15	23	RSA
16	10	$\alpha(\mathrm{fac}_1)$
17	1	$val(n_1)$
18		fac_2
19	0	n_2
20	0	SV
21	12	DV
22	4	EP
23	23	RSA
24	18	$\alpha(\mathrm{fac}_2)$
25	1	

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\mathrm{fac}_0)$
9	2	$val(n_0)$
10		fac_1
11	1	n_1
12	0	SV
13	4	DV
14	4	EP
15	23	RSA
16	10	$\alpha(\mathrm{fac}_1)$
17	1	$n_1 * \mathrm{fac}_2$

0		x
1	0	$\alpha(x)$
2		fac_0
3	2	n_0
4	0	SV
5	0	DV
6	3	EP
7	30	RSA
8	2	$\alpha(\mathrm{fac}_0)$
9	2	$n_0 * \mathrm{fac}_1$

Fig. B.7. Code and procedure frame for `fac`

Name Parameters

1. We must generate code for name parameters. Every use of the formal parameter jumps to that code in order to evaluate it.
2. This resembles procedure parameters.
   Example: *Jensen's device*explains a rather unexpected feature of name parameters:

```
var a: array [1..n] of Integer;
var i: Integer;
function sum (name a:Integer) : Integer;
  begin
  sum := 0;
```

```
for i:= 1 to n do sum := sum + a;
end;

begin
write (sum(a[i]));
end.
```

Function `sum` sums up the elements of the array `a` – not `n*a[i]` for the value of `i` when the procedure is entered. So one avoids the "expensive" passing of arrays.
This obfuscates the definition of `sum`.

Exercise 5.10. The Pascal Program is translated as follows:

code(**program** fakultaet ...)α
= **ssp** l; **sep** k; **ujp** m; code(**function** fac ...)α; m : code(x := fac(2))α; **stp**
= **ssp** l; **sep** k; **ujp** m; code(...)α; m : **loa** 0 0; **mst** 0; **ldc** 1; **cup** 2 f; **sto** 1; **stp**

Here x has the address 0, the extension l of local variables is 1, and the maximal stack growth k is 3. (For translating"x := fac(2)", the address of x, the space for the result of fac, and the value 2 of the actual parameter has to be pushed on the stack.)

The translation of fac can be found in Solution 23.

Answers to Exercises in Chapter 6