Contents

1 The Structure of Compilers

Our series of books treats the compilation of higher programming languages into the machine languages of virtual or real computers. Such compilers are large, complex software systems. Realizing large and complex software systems is a difficult task. What is special about compilers such that they can be even implemented as a project accompanying a compiler course? A decomposition of the task into subtasks with clearly defined functionalities and clean interfaces between them makes this, in fact, possible. This is true about compilers; there is a more or less standard conceptual compiler structure composed of components solving a well-defined subtask of the compilation task. The interfaces between the components are representations of the input program.

The compiler structure described in the following is a *conceptual* structure. i.e. it identifies the subtasks of the translation of a *source* language into a *target* language and defines interfaces between the components realizing the subtasks. The concrete architecture of the compiler is then derived from this conceptual structure. Several components might be combined if the realized subtasks allow this. But a component may also be split into several components if the realized subtask is very complex.

A first attempt to structure a compiler decomposes it into three components executing three consecutive phases:

1. The *analysis phase*, realized by the *Frontend*. It determines the syntactic structure of the source program and checks whether the static semantic constraints are satisfied. The latter contain the type constraints in languages with static type systems.
2. The *optimization* and *transformation* phase, performed by what is often called the *Middleend*. The syntactically analysed and semantically checks program is transformed by *semantics-preserving* transformations. These transformations mostly aim at improving the efficiency of the program by reducing the execution time, the memory consumption, or the consumed energy. These transformations are independent of the target architecture and mostly also independent of the source language.
3. The *code generation and the machine-dependent optimization* phase, performed by the *Backend*. The program is being translated into an equivalent program in the target language. Machine-dependent optimizations might be performed, which exploit peculiarities of the target architecture.

This coarse compiler structure splits it into a first phase, which depends on the source language, a third phase, which depends only on the target architecture, and a second phase, which is mostly independent of both. This structure helps to adapt compiler components to new source languages and to new target architectures.

The following sections present these phases in more detail, decompose them further, and show them working on a small running example. This book describes the analysis phase of the compiler. The transformation phase is presented in much detail in the volume *Analysis and Transformation*. The volume *Code Generation and Machine-oriented Optimization* covers code generation for a target machine.

1.1 Subtasks of compilation

Fig. 1.1 shows a conceptual compiler structure. Compilation is decomposed into a sequence of phases. The analysis phase is further split into subtasks as this volume is concerned with the analysis phase. Each component realizing such a subtask receives a representation of the program as input and delivers another representation as output. The format of the output representation may be different, e.g. when translating a symbol sequence into a tree, or it may be the same. In the latter case, the representation will in general be augmented with newly computed information. The subtasks are represented by boxes labeled with the name of the subtask and maybe with the name of the module realizing this subtask.

We now walk through the sequence of subtasks step by step, characterize their job, and describe the change in program representation. As a running example we consider the following program fragment:

$$\begin{aligned}&\mathsf{int}\ a, b;\\ &a = 42;\\ &b = a * a - 7;\end{aligned}$$

where $' ='$ denotes the assignment operator.

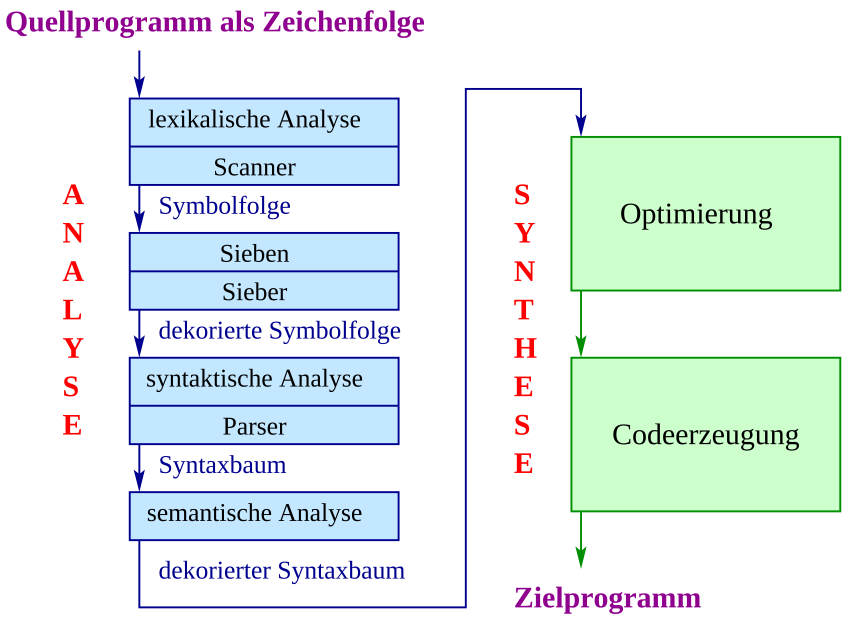


Fig. 1.1. Structure of a compiler together with the program representations during the analysis phase.

1.2 Lexical Analysis

The component performing lexical analysis of source programs is often called the *scanner*. This componen reads the source program represented a sequence of characters mostly from a file. It decomposes this sequence of characters into a sequence of lexical units of the programming language. These lexical units are called *symbols*. Typial lexical units are keywords such as **if**, **else**, **while** or **switch** and special charactes and character combinations such as $=, ==, != , <=, >=, <, >, (,), [,], \{, \}$ or comma and semicolon. These need to be recognized and converted into corresponding internal representations. The same holds for reserved identifiers such as names of basic types int, float, double, char, bool or string, etc. Further symbols are identifier and constants. Examples for identifiers are value42, abc, Myclass, x, while the character sequences 42, 3.14159 and ″HalloWorld!″ represent constants. Something special to note is that there are, in principle, arbitrarily many such symbols. However, they can be categorized into finitely many *classes*. A symbol class consists of symbols that are equivalent as far as the syntactic structure of programs is concerned. Identifiers are an example of such a class. Within this class, there may be subclasses such as type constructors in OCAML or variables in PROLOG, which are written in capital letters. In the class of constants, *int*-constants can be distinguished from floating-point constants and *string*-constants.

The symbols we have considered so far bear semantic interpretations and need, therefore, be considered in code generation. However, there are symbols without semantics. Two symbols need a separator between them if their concatenation would also form a symbol. Such a separator can be a blank, a newline, or an indentation or a sequence of such characters. Such so-called white space can also be inserted into a program to make visible the structure of the program.

Another type of symbols, without meaning for the compiler, but helpful for the human reader, are comments and can be used by software development tools. A similar type of symbols are *compiler directives* (*pragmas*). Such directives may tell the compiler to include particular libraries or influence the memory management for the program to be compiled.

The sequence of symbols for the example program might look as follows:

$$\begin{array}{l}
\mathsf{Int}(''int'')\ \mathsf{Sep}(''\ '')\ \mathsf{Id}(''a'')\ \mathsf{Com}('','')\ \mathsf{Sep}(''\ '')\ \mathsf{Id}(''b'')\ \mathsf{Sem}('';'')\ \mathsf{Sep}(''\backslash n'')\\
\mathsf{Id}(''a'')\ \mathsf{Bec}(''='')\ \mathsf{Intconst}(''42'')\ \mathsf{Sem}('';'')\ \mathsf{Sep}(''\backslash n'')\\
\mathsf{Id}(''b'')\ \mathsf{Bec}(''='')\ \mathsf{Id}(''a'')\ \mathsf{Mop}(''*'')\ \mathsf{Id}(''a'')\ \mathsf{Aop}(''-'')\ \mathsf{Intconst}(''7'')\ \mathsf{Sem}('';'')\ \mathsf{Sep}(''\backslash n'')
\end{array}$$

To increase readability, the sequences was brolen into lines according to the original program structure. Each symbol is represented with its symbol class and the substring representing it in the program. More information may be added such as the position of the string in the input.

1.3 The Screener

The scanner delivers a sequence of symbols to the screener. These are substrings of the program text labeled with their symbol classes. It is the task of the screener to further process this sequence. Some symbols it will eliminate since they have served their prupose as separators. Others it will transform into a different representation. More precisely, it will perform the following actions, specific for different symbol classes:

Reserved symbols: These are typically identifiers, but have a special meaning in the programming language. e.g. begin, end, var, int etc.

Separators and comments: Sequences of blanks and newlines serve as separators between symbols. They are of not needed for further processing of the program and can therefore be removed.. An exception to this rule are some functional languages, e.g. HASKELL, where indentation is used to express program nesting. Comments will also not be needed later and can be removed.

Pragmas: Compiler directives (pragmas) are not part of the program. They will separately passed on to the copmpiler.

Other types of symbols are typically preserved, but their textual representation may be converted into some more efficient internal representation.

Constants: The sequence of digits as representation of number constants is converted to a binary representation. *String*-constants are stored into an allocated object. In JAVA implementations, these objects are stored in a dedicated data structure, the *String Pool*. The String Pool is available to the program at run-time.

Identifier: Compilers usually do not work with identifiers represented as string objects. This representation would be too inefficient. Rather, identifiers are coded as unique numbers. The compiler needs to be able to access the external representation of identifiers, though. For this purpose, the identifiers are kept in a data structure, which can be efficiently addressed by their codes.

The screener will produce the following sequence of annotated symbols for our example program:

$$\begin{array}{l}\mathsf{Int}()\ \mathsf{Id}(1)\ \mathsf{Com}()\ \mathsf{Id}(2)\ \mathsf{Sem}()\\ \mathsf{Id}(1)\ \mathsf{Bec}()\ \mathsf{Intconst}(42)\ \mathsf{Sem}()\\ \mathsf{Id}(2)\ \mathsf{Bec}()\ \mathsf{Id}(1)\ \mathsf{Mop}(Mul)\ \mathsf{Id}(1)\ \mathsf{Aop}(Sub)\ \mathsf{Intconst}(7)\ \mathsf{Sem}()\end{array}$$

All separators are removed from the symbol sequence. Semantical values were computed for some of the substrings. The identifiers a and b were coded by the numbers 1 and 2, resp. The sequences of digits for the *int* constants were replaced by their binary values. The internal representations of the symbols Mop and Aop are elements of an appropriate enumeration type.

Scanner and screener are usually combined into one module, which is also called *scanner*. Conceptually, however, they should be kept separate. The task that the scanner, in the restricted meaning of the word, performs can be realized by a finite-state machine. The screener, however, can be realized by arbitrary pieces of code.

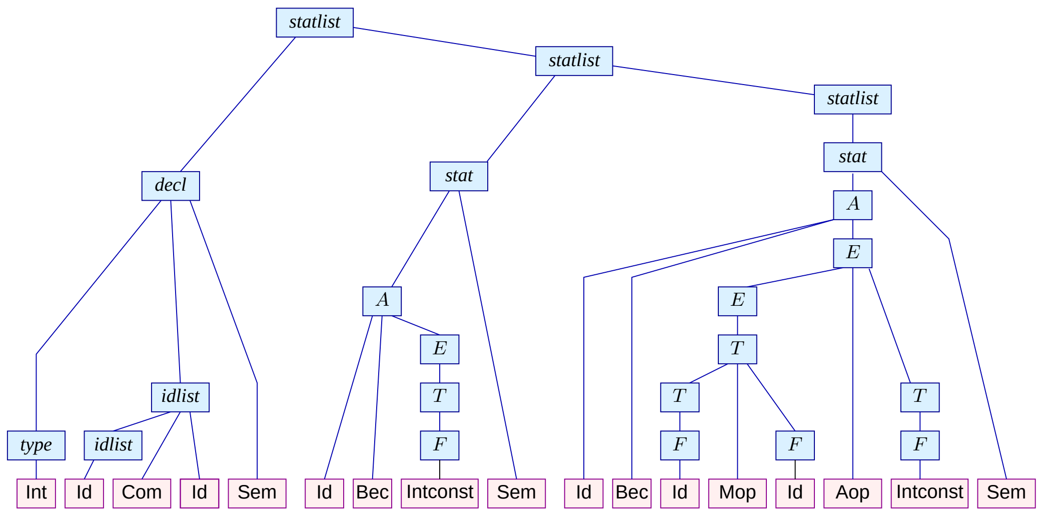


Fig. 1.2. Syntactic analysis of the example program.

1.4 Syntactic Analysis

The lexical and the syntactic analysis together recognize the syntactic structure of the source program. Lexical analysis realized the part of this task that can be realized by a finite-state machine. Syntactic analysis recognizes the hierachical structure of the program, a task a finite-state machine can not do in general. The syntactical structure of the program consists of sequential and hierarchical composition of language constructs. The hierarchical composition corresponds to the *nesting* of language constructs. Programs in an object-oriented programming language like JAVA consist of class declarations, which may be combined into packages. The declaration of a class may contain declarations of attributes, constructors, and methods. A method consists of a method head and a method body. The latter contains the implementation of the method. Some language constructs may be nested arbitrarily deep. This is the case for arithmetic expressions, where an unlimited number of operators can be used to construct an expression of arbitrary size and depth. Finite-state machines are incapable of recognizing such nesting of constructs, and regular expressions are not expressive enough to describe it. We need to resort to more powerful specification mechanisms and recognizers.

Pushdown automata are used as recognizers. The pushdown automaton used to recognize the syntactic structure is called *parser*. This component should not only recognize the syntactic structure of correct programs. It should also be able to properly deal with syntactically incorrect programs. After all, most programs submitted to a program contain mistakes. Typical syntax errors are spelling errors in keyword, missing parentheses or separators. The parser should detect these kind of errors, diagnose them, and maybe even try to correct them.

The syntactic structure of programs can be described by *context-free grammars*. From the theory of formal languages and automata we know that pushdown automata are equivalent to context-free grammars. Parsers are, therefore, deterministic pushdown automata. There exist many different methods for syntactic analysis. The two major ones are described in Chapter 3.

The output of the parser may have several different equivalent formats. In our conceptual compiler structure and in Fig. 1.2 (C), we use as output *parse trees*.

1.5 Semantic Analysis

The job of semantic analysis is to determine properties and check conditions that are relevant for the well-formedness of programs according to the rules of the programming language, but that go beyond what can be described by context-free grammars. These conditions can be completely checked on the basis of the program text are called *static semantic* properties. This phase is, therefore, called semantic analysis. The *dynamic* semantics, in constrast, describes the behavior of programs when they are executed. The atrtributes *static* and *dynamic* are associated with the *compile time* and the *run time* of programs, respectively. We list some static semantic properties of programs:

- type correctness in strongly typed programming languages like C, PASCAL, JAVA or OCAML. Necessary for type correctness is that all identifiers are declared, either explicitily or implicitily and possible the absence of multiple declarations of the same identifier.
- the existence of a *consistent type association* with all expressions in languages with type polymorphism.

Example 1.5.1 For the program of Example 1.2, semantic analysis will collect the declarations of the *decl*-subtree in a map

$$env = \{\mathsf{Id}(1) \mapsto \mathsf{Int}, \mathsf{Id}(2) \mapsto \mathsf{Int}\}$$

. This map associates each identifier with its type. Using this map, semantic analysis can check in the *stat*-subtrees whether variables and expressions are used in a type-correct way. For the first assignment, $a = 42;$, it will check whether the left side of the assignment is a variable identifier, and whether the type of the left side is compatible with the type on the right side. In the second assignement, $b = a * a - 7;$, the type of the right side is less obvious. It needs to be determined from the types of the variable a and the constant 7. One should not forget that the arithmetic operators are *overloaded* in most programming languges. This means that they stand for the designated operations of several types, for instance on *int*- as well as on *float*-operands, possibly even for different precision. The type checker has to resolve overloading. In our example, it determines that the multiplication is an *int*-multiplication and the subtraction an *int*-subtraction, both returning values of type **int**. The result type of the right side of the assignment, therefore, is **int**. □

1.6 Machine-Independent Optimization

Static analyses of the source program might detect potential run-time errors or possibilities for program transformation that will increase the efficiency of the program while preserving the semantics of the program. A *data-flow analysis* or *abstract interpretation* can detect, among others, the following properties of a source program:

- There exists a program path on which a variable would be used without being initialized.
- There exist program parts that cannot be reached or functions that are never called. These superfluous parts don't need to be compiled.

- A program variable x at a statement in an imperative program has always the same value, c. In this case, variable x can be repaced by the value c in this statement.
  This analysis would recognize that at each execution of the second assignment, $b = a * a - 7;$, variable a has the value 42. Replacing both occurrences of a by 42 leads to the expression $42{*}42{-}7$, whose value can be evaluated at compile time. This analysis and transformation is called *constant propagation* with *constant folding.*

A major empasis of this phase is on evaluating subexpressions whose value can be determined at compile time. Besides this, the following optimizations can be performed by the compiler:

- *Loop invariant* computations can be moved out of loops. A computation is *loop invariant* if it only depends on variables that do not change their value during the execution of the loop. Such a computation is executed only once instaed of in each iteration when it has been moved out of a loop.
- A similar transformation can be applied in the compilation of functional programs to reach the *fully lazy*-property. Expressions that only contain variables bound outside of the function can be moved out of the body of the function and passed to the function in calls with an additional parameter.

These kinds of optimizations are performed by many compilers. They make up the *middle end* of the compiler. The volume *Compiler Design - Analysis and Transformation* is dedicated to this subject area.

1.7 Memory Allocation

The allocation of memory and the assignment of addresses starts the *synthesis phase* of compilation. This phase strongly depends on properties of the target architecture, such as word length, the address length, the directly addressable units of the machine, and the existence or non-existence of instructions for efficient direct access ton parts of directly addressable units. These machine parameters determine the allocation of memory units to basic types and the possibility to pack values of “small”types such as Booleans and characters, into bigger memory units. This memory-saving optimization needs to consider the constraints for directly addressable units. For instance, *int* values can only be directly accessed or operated upon on many machines when they are allocated at word borders. These constraints are called *alignment* rules.

Example 1.7.1 (see Fig. 1.3) We assume to have a machine with addressing of full words, that is, consecutive words have addresses that differ by 1. The compiler allocates *int*-variables to full words. Increasing addresses in the order in which the variables are declared are assigned starting with address 0. Variable *a* is assigned address 0, *b* address 1. □

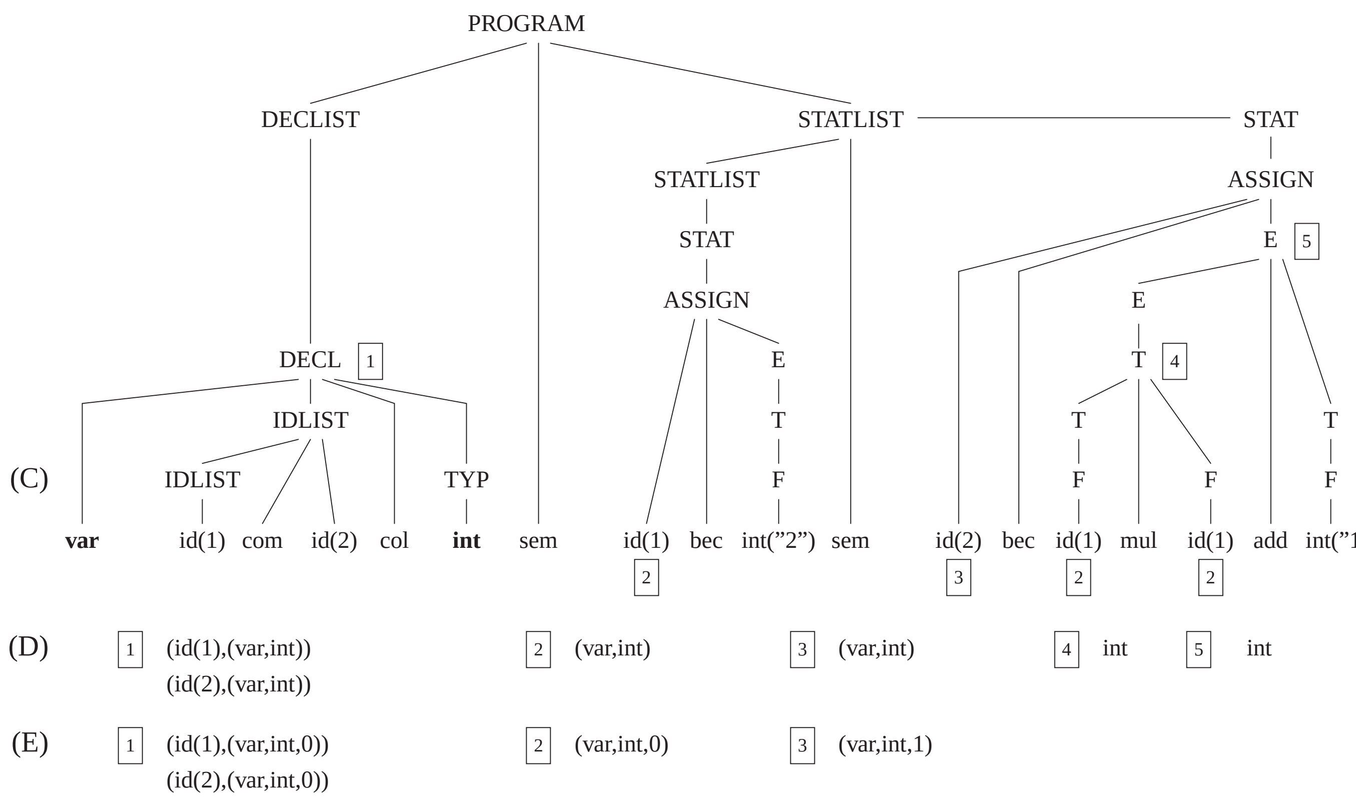
1.8 Generation of the Target Program

The code generator takes the intermediate representation of the program and generates the target program. A systematic way to translate several types of programming languages to adequate virtual machines is presented in the volume, *Compiler Design — Virtual Machines*. Code generation, as described there, works recursively over the structure of the program. It could, therefore, start directly after syntactic and semantic analysis and work on the decorated parse tree.

The code generator uses the addresses assigned to variables as described in the preceding step. However, the access to values is more efficient if the values are stored in the registers of the machine. Target machines have a limited number of such registers. One task of the code generator is to make good use of this restricted resource. The task to assign registers to variables and intermediate values is called *register allocation*.

Example 1.8.1 Let us assume that a virtual or concrete target machine would have registers $r_1, r_2, \ldots, r_N$ for a (mostly small) N and that it would have, among others, the instructions

Fig. 1.3. Analyse eines Programmausschnitts, (D) semantische Analyse, (E) Adre"szuordnung.


PROGRAM
DECLIST
STATLIST
STAT
STATLIST
ASSIGN
STAT
E 5
ASSIGN
E
DECL 1
E
T 4
IDLIST
T
T
T
(C)
IDLIST
TYP
F
F
F
F
var id(1) com id(2) col int sem id(1) bec int("2") sem id(2) bec id(1) mul id(1) add int("1
2 3 2 2
(D) 1 (id(1),(var,int)) (id(2),(var,int)) 2 (var,int) 3 (var,int) 4 int 5 int
(E) 1 (id(1),(var,int,0)) (id(2),(var,int,0)) 2 (var,int,0) 3 (var,int,1)

instruction		meaning
load	r_i, q	$r_i \leftarrow M[q];$
store	q, r_i	$M[q] \leftarrow r_i;$
loadi	r_i, q	$r_i \leftarrow q;$
subi	r_i, r_j, q	$r_i \leftarrow r_j - q;$
mul	r_i, r_j, r_k	$r_i \leftarrow r_j * r_k;$

where q stands for an arbitrary *int*-constant, and $M[\ldots]$ for a memory access. Let us further assume that variables a and b were assigned the global addresses 1 and 2. One potential translation of the example program, which would store the values for a and b in the corresponding memeory cells, could look like follows:

```
loadi   $r_1, 42$
store   $1, r_1$
mul     $r_2, r_1, r_1$
subi    $r_3, r_2, 7$
store   $2, r_3$
```

Registers r_1, r_2 and r_3 serve for to store intermediate values during the evaluation of right sides. Registers r_1 and r_3 hold the values of variables a and b, resp. Closer inspection reveals that the compiler could save registers. For instance, register r_1 can be reused for register r_2 since the value in r_1 is no longer needed after the multiplication. Evan the result of the instruction subi may be stored in the same register. We, thus, obtain the improved instructions sequence:

```
loadi  r1, 42
store  1, r1
mul    r1, r1, r1
subi   r1, r1, 7
store  2, r1
```

□

The code generator needs to observe limitations enforced by the number of registers. It may not store in registers more intermediate results concurrently than the number of registers allows. These and similar constraints are to be found in realistic target architectures. Furthermore, they typically offer many instruction that make special cases very efficient. This makes the generation of efficient code very difficult. The necessary techniques are presented in the volume: *Compiler Design — Code Generation and Machine-Level Optimization*.

1.9 Specification and Generation of Compiler Components

The theory of formal languages and automata tells us that some analysis subtasks of compilation are word problems of certain languages and that certain type of automata are acceptors for these languages and, thus, solve these word problems. One also knows that these automata can be automatically generated from grammars that are used as specification mechanisms.

All the tasks that are to solved during the syntactic analysis can be elegantly specified by different types of grammars. Symbols, the lexical units of the languages, can be described by regular expressions.

A non-deterministic finite-state machine recognizing the language described by a regular expression can be automatically derived from the regular expression. This non-deterministic finite-state machine can be automatically converted into a deterministic finite-state machine.

A similar correspondence is known between context-free grammars and pushdown automata. A non-deterministic pushdown automaton recognizing the language of a context-free grammar can be automatically constructed from the context-free grammar. For practical applications such as compilation, one prefers deterministic pushdown automata. However, unlike in the case of finite-state machines, non-deterministic pushdown automata are more powerful than deterministic pushdown automata. Most designers of programming languages have succeeded to stay within the class of deterministically analyzable context-free languages, so that syntax analysis of their languages is relatively simple and efficient. The example of C++, however, shows that a badly designed syntax requires nondeterministic parsers and considerably more effort, both in building a parser and in actually parsing programs in the language.

The compiler components for lexical and syntactic analysis, thus, need not be programmed by hand, but can be automatically generated from appropriate specifications. These two example suggest to look for more compiler subtasks that could be solved by automatically generated components. As another example for this approach, we meet *attribute grammars* in this volume. These are an extension of context-free grammars in which computations on parse trees can be specified. These computations typically check the conformance of the program to static-semantics conditions like typing rules. Table 1.1 lists compiler subtasks treated in this volume that can be formally specified in such a way that implementations of the corresponding components can be automatically generated. The specification and the implementation mechanisms are listed with the subtask.

Program invariants as they are needed for the semantics-preserving application of optimizing program transformations can be computed using generic approaches based on the theory of *abstract interpretation*. This is the subject of the volume *Compiler Design — Analysis and Transformation*

There also exist methods to automatically produce components of the compiler backend. For instance, instruction scheduling can be solved by ILP (*Integer Linear Programming*). All the subtasks of code generation are treated in depth in the volume *Compiler Design – Code Generation and Machine-Level Optimization*

compilation subtask	specification mechanism	implementation mechanism
lexical analysis	regular expressions	deterministic finite-state machines
syntactic analysis	context-free grammars	deterministic pushdown automata
semantic analysis	attribute grammars	attribute evaluators

Table 1.1. Compiler subtasks, specification mechanisms, and corresponding implementation mechanisms

1.10 Literature

How to structure compilers was well understood rather early. The following articles may be taken as witnesses [MD74], [McK74], and [GW75].

2 Lexical Analysis

We start this chapter with a description of the task of lexical analysis and then present regular expressions as specification mechanism for this task. Regular expressions can be automatically converted into non-deterministic finite state machines, which implement lexical analysis. Non-deterministic finite-state machines can be made deterministic, which is preferred for implementing lexical analyzers, often called *scanners*. Another transformation on the resulting deterministic finite-state machines attempts to reduce the size of the machines. These three steps together make up an automatic process generating lexical analyzers from specifications. Another module working in close cooperation with such a finite-state machine is the *screener*. It filters out keywords, comments etc. and may do some bookkeeping.

2.1 The Task of Lexical Analysis

Let us assume that the source program is stored in a file. It consists of a sequence of characters. Lexical analysis, i.e., the scanner, reads this sequence from left to right and decomposes it into a sequence of lexical units, called *symbols*. Scanner, screnner, and parser may work in an integrated way. In this case, the parser calls the combination scanner-screener to obtain the next symbol. The scanner starts the analysis with the character that follows the end of the last found symbol. It searches for the longest prefix of the remaining input that is a symbol of the language. It passes a representation of this symbol on to the screener, which checks whether this symbol is relevant for the parser. If not it is ignored, and the screener reactivates the scanner. Otherwise, it passes a possibly transformed representation of the symbol on to the parser.

The scanner must, in general, be able to recognize infinitely many or at least very many different symbols. The set of symbols is, therefore, divided into finitely many classes. One *symbol class* will consist of symbols that have a similar syntactic role. We distinguish:

- The alphabet is the set of characters that may occur in program texts. We use the letter Σ to denote alphabets.
- A *symbol* is a word over the alphabet Σ. Examples are $xyz12$, 125, class, “abc''.
- A *symbol class* is a set of symbols. Examples are the set of identifiers, the set of *int*-constants, and the set of character strings. We denote these by Id, $\mathsf{Intconst}$ and String, respectively.
- The *representation of a symbol* comprises all of the mentioned informations about a symbol that may be relevant for later phases of compilation. The scanner might represent the word $xyz12$ as pair $(\mathsf{Id}, “xyz12'')$, consisting of the name of the class and the found symbol, and pass this representation on to the screener. The screener could replace “$xyz12''$ by the internal representation of an identifier, for example, a unique number, and then pass this on to the parser.

2.2 Regular Expressions and Finite-State Machines

2.2.1 Words and Languages

We introduce some basic terminology. We use Σ to denote some *alphabet*, that is a finite, non-empty set of characters. A *word* x over Σ of length n is a sequence of n characters from Σ. The *empty word* ε is the empty sequence of characters, i.e. the sequence of length 0. We consider individual characters from Σ as words of length 1.

Σ^n denotes the set of words of length n for $n \geq 0$. In particular, $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^1 = \Sigma$. The set of all words is denoted as Σ^*. Correspondingly is Σ^+ the set of *non-empty* words, i.e.

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \qquad \text{and} \qquad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$

Several words can be concatenated to a new word. *Concatenation* of the words x and y puts the sequence of characters of y after the sequence of characters of x, i.e.

$$x\,.\,y = x_1 \ldots x_m y_1 \ldots y_n,$$

if $x = x_1 \ldots x_m, y = y_1 \ldots y_n$ for $x_i, y_j \in \Sigma$.

Concatenation of x and y produces a word of length $n + m$ if x and y have length n and m, respectively. Concatenation is a binary operation on the set Σ^*. In contrast to the addition on numbers, concatenation of words is not *commutative*. This means that the word $x\,.\,y$ is , in general, different from the word $y\,.\,x$. Like the addition on numbers, concatenation of words is *associative*, i.e.

$$x\,.\,(y\,.\,z) = (x\,.\,y)\,.\,z \qquad \text{for all } x, y, z \in \Sigma^*$$

The empty word ε is the *neutral* element with respect to concatenation of words, i.e.

$$x\,.\,\varepsilon = \varepsilon\,.\,x = x \qquad \text{for all } x \in \Sigma^*.$$

In the following, we will write xy for $x\,.\,y$.

For a word $w = xy$ with $x, y \in \Sigma^*$ we call x a *prefix* and y a *suffix* of w. Prefixes and suffixes are special *subwords*. In general, word y is a subword of word w, if $w = xyz$ for words $x, y \in \Sigma^*$. Prefixes, suffixes and, in general, subwords of w are called *proper*, if they are different from w.

Subsets of Σ^* are called (formal) *languages*. We need some operations on languages. Assume that $L, L_1, L_2 \subseteq \Sigma^*$ are languages. The *union* $L_1 \cup L_2$ consists of all words from L_1 and L_2:

$$L_1 \cup\ L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ oder } w \in L_2\}.$$

The *concatenation* $L_1.L_2$ (abbreviated L_1L_2) consists of all words resulting from concatenation of a word from L_1 with a word from L_2:

$$L_1\,.\,L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

The *complement* $\overline{L}$ of language L consists of all words in Σ^* that are not contained in L:

$$\overline{L} = \Sigma^* - L.$$

For $L \subseteq \Sigma^*$ we denote L^n as the n-times concatenation of L, L^* as the union of arbitrary concatenations, and L^+ as the union of non-empty concatenations of L, i.e.

$$\begin{aligned} L^n &= \{w_1 \ldots w_n \mid w_1, \ldots, w_n \in L\} \\ L^* &= \{w_1 \ldots w_n \mid \exists n \geq 0.\ w_1, \ldots, w_n \in L\} = \bigcup_{n \geq 0} L^n \\ L^+ &= \{w_1 \ldots w_n \mid \exists n > 0.\ w_1, \ldots, w_n \in L\} = \bigcup_{n \geq 1} L^n \end{aligned}$$

The operation $(\_)^*$ is called *Kleene-star*.

Regular Languages and Regular Expressions

The languages described by symbol classes as they are recognized by the scanner are non-empty *regular languages*.

Each non-empty regular language can be constructed starting with singleton languages and applying the operations union, concatenation, and Kleene-Star.Formally, the set of all *regular languages* over an alphabet Σ is inductively defined by:

- The empty set $\emptyset$ and the set $\{\varepsilon\}$, consisting only of the empty word, are regular .
- The sets $\{a\}$ for all $a \in \Sigma$ are regular over Σ.
- Are R_1 and R_2 regular languages over Σ, so are $R_1 \cup R_2$ and $R_1 R_2$.
- Is R regular over Σ, then also R^*.

According to this definition, each regular language can be specified by a regular expression. *Regular expression* over Σ and the regular languages described by them are also defined inductively:

- $\emptyset$ is a regular expression over Σ, which describes the regular language $\emptyset$.
  ε is a regular expression over Σ, and it describes the regular language $\{\varepsilon\}$.
- For each $a \in \Sigma$ is a a regular expression over Σ that describes the regular language $\{a\}$.
- Are r_1 and r_2 regular expressions over Σ that describe the regular languages R_1 and R_2 respectively, then $(r_1 \mid r_2)$ and $(r_1 r_2)$ are regular expressions over Σ that describe the regular languages $R_1 \cup R_2$ and $R_1 R_2$, respectively.
- Is r a regular expression over Σ, that describes the regular language R, then r^* is a regular expression over Σ that describes the regular language R^*.

In practical applications, $r?$ is often used as abbreviation for $(r \mid \varepsilon)$ and sometimes also r^+ for the expression (rr^*).

In the definition of regular expressions we assumed that the symbols for the empty set and the empty word were not contained in Σ, similarly to the parentheses $($, $)$ and the operators $\mid$ and $*$ and also $?, +$. These characters belong to the description mechanism for regular expressions and not to the regular languages described by the the regular expressions. They are called *meta characters* However, the set of representable characters is limited, so that some meta characters may also appear in the described regular languages. A programming system generating scanners from descriptions given as regular expressions needs to make clear when such a character is a meta character and when it is a character of the language. One way to do this is ti use *escape characters*. In many specification languages for regular languages the $\backslash$ character is used as escape character. For example, to represent the meta character $\mid$ also as a character of the alphabet one would precede it with a $\backslash$. So, in a regular expression, the vertical bar would be represented as $\backslash|$.

We introdce operator precedences to save on parentheses: The ?-operator has the highest precedence, follwoed by the Kleene-star $(\_)^*$, and then possibly the operator $(\_)^+$, then concatenation and finally the alternative operator $\mid$.

Example 2.2.1 The following table lists a number of regular expressions together with the languages described by them, and some ot even all of their elements.

regular expression	described language	elements of the language
$a \mid b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \ldots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \ldots$
$abba$	$\{abba\}$	$abba$ □

Regular expressions that contain the empty set as symbol can be simplified by repeated application of the following equalities:

$$
\begin{aligned}
r \mid \emptyset &= \emptyset \mid r = r \\
r \,.\, \emptyset &= \emptyset \,.\, r = \emptyset \\
\emptyset^* &= \epsilon
\end{aligned}
$$

The equality symbol, ’=’, between two regular expressions means that both describe the same language. We can prove:
Our applications only have regular expressions that describe non-empty languages. No symbol to describe the empty set is, therefore, needed. The empty word is needed to represent empty alternatives. The $?$-operator suffices to represent this. No extra representation of the empty word is needed.

Finite-State Machines

We have seen that regular expressions are used for the specification of symbol classes. The implementation of recognizers uses finite-state machines (FSMs). Finite-state machines are acceptors for regular languages. They maintain one state variable that can only take on finitely many values, the *states* of the finite-state machine. Fig. 2.1 shows that furthermore FSMs have an input tape and an input head, which reads the input on the tape from left to right. The working of the FSM is described by a *transition relation* Δ.

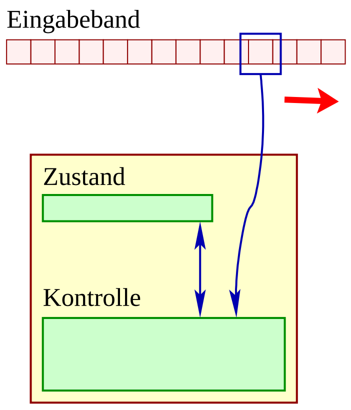


Fig. 2.1. Schematic representation of a finite-state machine.

Formally, we represent a *non-deterministic finite-state machine (with* ε*-transitions)* (NFSM) as a tuple $M = (Q, \Sigma, \Delta, q_0, F)$ where

- Q is a finite set of *states*,
- Σ is a finite alphabet, the *input alphabet*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is the set of *final states*, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the *transition relation*.

A transition $(p, x, q) \in \Delta$ expresses that M can change from its current state p into the state q. Is $x \in \Sigma$ then x must be the next character in the input and after reading x the input head if moved by one character. Is $x = \varepsilon$ then no character of the input is read upon this transition. The input head remains at its actual position. Such a transition is called a ε*-transition*.

Of particular interest for implementations are finite-state machines without ε-transitions, which in addition have in each state exactly one transition under each character. Such a finite-state machine *deterministic finite-state machine* (DFSM). For such a DFSM the transition relation Δ is a *Funktion* $\Delta : Q \times \Sigma \to Q$.

We describe the workings of a DFSM in comparison with a DFSM used as a scanner. The description of the working of a scanner is put into boxes. A deterministic finite-state machine should check whether given input words are contained in a language or not. It accepts the input word if it arrives in a final state after reading the whole word.

A deterministic finite-state machine used as a scanner decomposes the input word into a sequence of subwords corresponding to *symbols* of the language. Each symbol drives the DFSM from its initial state into one of its final states.

The deterministic finite-state machine starts in its initial state. Its input head is positioned at the beginning of the input head.

A scanner's input head is always positioned at the first not yet consumed character.

It then makes a numnber of steps. Depending on the actual state and the next input symbol the DFSM changes its state and moves its input head to the next character. The DFSM accepts the input word when the input is exhausted and the actual state is a final state.

Quite analogously, the scanner performs a number of steps. It reports that it has found a symbol or that it has detected an error when no further step is possible.
If the actual state is not a final state and there is no transition under the next input character the scanner returns to the last input character that brought it into a final state for some symbol class. It delivers as value this class together with the newly consumed prefix of the input. Then the scanner restarts in the initial state with its input head positioned at the first not yet consumed input character. The scanner has detected an error if by rewinding the last transitions it does not find a final state.

Our goal is to derive an implementation of an acceptor of a regular language out of a specification of the language, that is, to construct out of a regular expression r a deterministic finite-state machine that accepts the language described by r. In a first step, a *non-deterministic* finite-state machine for r is constructed that accepts the language described by r. In a second step this is made deterministic.

A finite-state machine $M = (Q, \Sigma, \Delta, q_0, F)$ starts in its initial state q_0 and non-deterministically performs a sequence of steps, a *computation*, under the given input word The input word is accepted if the computation leads to a final state,

The future behavior of a finite-state machine is fully determined by its actual state $q \in Q$ and the remaining input $w \in \Sigma^*$. This pair (q, w) makes up the *configuration* of the finite-state machine. A pair (q_0, w) is an *initial configuration*. Pairs (q, ε) such that $q \in F$ are *final configurations*.

The *step*-relation $\vdash_M$ is a binary relation on configurations. For $q, p \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $w \in \Sigma^*$ holds $(q, aw) \vdash_M (p, w)$ if and only if $(q, a, p) \in \Delta$ and $a \in \Sigma \cup \{\varepsilon\}$. $\overset{*}{\vdash}_M$ denotes the reflexive, transitive hull of the relation $\vdash_M$. The language accepted by the finite-state machine M is defined as

$$L(M) = \{w \in \Sigma^* | (q_0, w) \overset{*}{\vdash}_M (q_f, \varepsilon) \text{ with } q_f \in F\}.$$

Example 2.2.2 Table 2.1 shows the transition relation of a finite-state machine M in the form of a two-dimensional matrix T_M. The states of the FSM are denoted by the numbers $0, \ldots, 7$. The alphabet is the set $\{0, \ldots, 9, ., E, +, -\}$. Each row of the table describes the transitions for one of the states of the FSM. The columns correspond to the characters in $\Sigma \cup \{\varepsilon\}$. The entry $T_M[q, x]$ contains the set of states p such that $(q, x, p) \in \Delta$. The state 0 is the initial state. $\{1, 4, 7\}$ is the set of final states. This FSM recognizes unsigned *int*- and *float*-constants. The accepting (final) state 1 can be reached through computations on *int*-constants. Accepting states 4 and 6 can be reached under *float*-constants. □

A finite-state machine M can be graphically represented as a finite *transition diagram*. A transition diagram is a finite, directed, edge-labeled graph. The vertices of this graph correspond to the states of M, the edges to the transitions of M. An edge from p to q that is labeled with x corresponds to a transition (p, x, q). The start vertex of the transition diagram, corresponding to the initial state, is marked by an arrow pointing to it. The *end vertices*, corresponding to final states, are represented by doubly encircled vertices. A w-*path* in this graph for a word $w \in \Sigma^*$ is a path from a vertex q to a vertex p, such that w is the concatenation of the edge labels. The language accepted by M consists of all words in $w \in \Sigma^*$, for which there exists a w-Weg in the state diagram from q_0 to a vertex $q \in F$.

Example 2.2.3 Fig. 2.2 shows the transition diagram corresponding to the finite-state machine of example 2.2.2. □

T_M	i	.	E	$+,-$	ε
0	{1,2}	{3}	∅	∅	∅
1	{1}	∅	∅	∅	{4}
2	{2}	{4}	∅	∅	∅
3	{4}	∅	∅	∅	∅
4	{4}	∅	{5}	∅	{7}
5	∅	∅	∅	{6}	{6}
6	{7}	∅	∅	∅	∅
7	{7}	∅	∅	∅	∅

Table 2.1. The transition relation of a finite-state machine to recognize unsigned *int*- and *float*-constants. The first column represents the identical columns for the digits $i = 0, \ldots, 9$, the fifth the ones for $+$ and $-$.

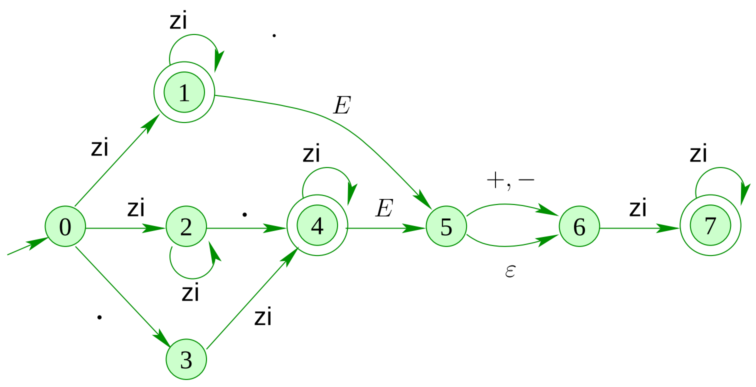


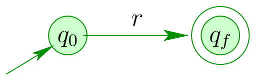
Fig. 2.2. The transition diagram for the finite-state machine of Example 2.2.2. The character zi stands for the set $\{0, 1, \ldots, 9\}$, an edge labeled with zi for edges labeled with $0, 1, \ldots 9$ with the same source and target vertices.

Acceptors

The next theorem guarantees that a non-deterministic finite-state machine can be constructed for a regular expression.

Theorem 2.2.1 For each regular expression r over an alphabet Σ there exists a non-deterministic finite-state machine M_r with input alphabet Σ, such that $L(M_r)$ is the regular language described by r.

We now present a method that constructs the transition diagram of a non-deterministic finite-state machine for a regular expression r over an alphabet Σ. . The construction starts with an edge leading from the initial state to a final state. This edge is labeled with r.



r will be decomposed according to its syntactical structure, and in parallel the transition diagram is built up. This is done by the rules of Fig. 2.3. They are applied until all remaining edges are labeled with ∅, ε or characters from Σ. Thenm, the edges labeled with ∅ are removed.

The application of a rule replaces the edge whose label is matched by the label of the left side by a corresponding copy of the subgraph of the right side. Exactly one rule is applicable for each operator. The application of the rule removes an edge labeled with a regular expression r and inserts new edges that are labeled with the argument expressions of the outermost constructor in r. The rule for the Kleene-star inserts additional ε-edges. This method can be implemented by the following program snippet if we take natural numbers as states of the finite-state machine.

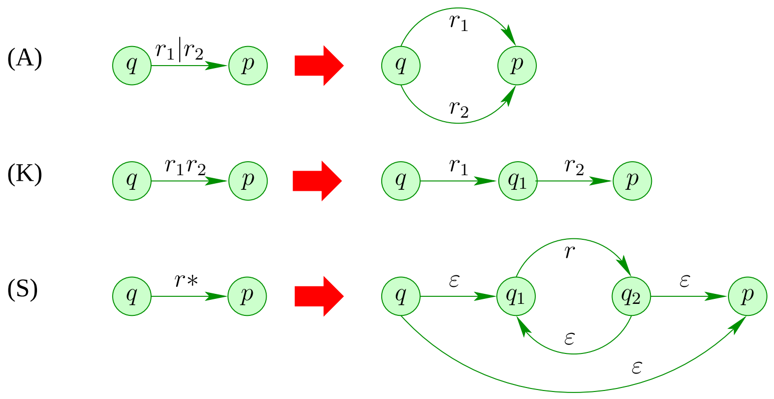


Fig. 2.3. The rules for the construction of a finite-state machine for a regular expression.

```
$trans \leftarrow \emptyset$;
$count \leftarrow 1$;
generate$(0, r, 1)$;
return $(count, trans)$;
```

The set $trans$ globally collects the transitions of the generated FSM, and the global counter $count$ keeps track of the largest natural number used as state. A call to a procedure generate for (p, r', q) inserts all transitions of a finite-state machine for the regular expression r' with initial state p and final state q into the set $trans$. New states are created by incrementing the counter $count$. This procedure is recursively defined over the structure of the regular expression r':

```
void generate (int $p$, Exp $r'$, int $q$) {
    switch $(r')$ {
    case $(r_1 \mid r_2)$ :   generate$(p, r_1, q)$;
                              generate$(p, r_2, q)$; return;
    case $(r_1.r_2)$ :        int $q_1 \leftarrow$ ++$count$;
                              generate$(p, r_1, q_1)$;
                              generate$(q_1, r_2, q)$; return;
    case $r_1^*$ :            int $q_1 \leftarrow$ ++$count$;
                              int $q_2 \leftarrow$ ++$count$;
                              $trans \leftarrow trans \cup \{(p, \varepsilon, q_1), (q_2, \varepsilon, q), (q_2, \varepsilon, q_1)\}$
                              generate$(q_1, r_1, q_2)$; return;
    case $\emptyset$ :        return;
    case $x$ :                $trans \leftarrow trans \cup \{(p, x, q)\}$; return;
    }
}
```

Exp denotes the type ’regular expression’ over the alphabet Σ. We have used a JAVA-like programming language as implementation language. The *switch*-statement was extended by *pattern matching* to elegantly deal with structured data such as regular expressions. this means that patterns are not only used to select between alternatives but also to identify partial structures.

A procedure call generate$(0, r, 1)$ terminates after n rule applications where n is the number of occurrences of operators and symbols in the regular expression r. If l is the value of the counter after the call, the generated FSM has $\{0, \ldots, l\}$ as set of states, where 0 is the initial state and 1 the only final state. The transitions are collected in the set $trans$. The FSM M_r can be computed in linear time.

Example 2.2.4 The regular expression $a(a \mid 0)^*$ over the alphabet $\{a, 0\}$ describes the set of words $\{a, 0\}^*$ beginning with an a. Fig. 2.4 shows the construction of the state diagram of a NFS that accepts this language.
□

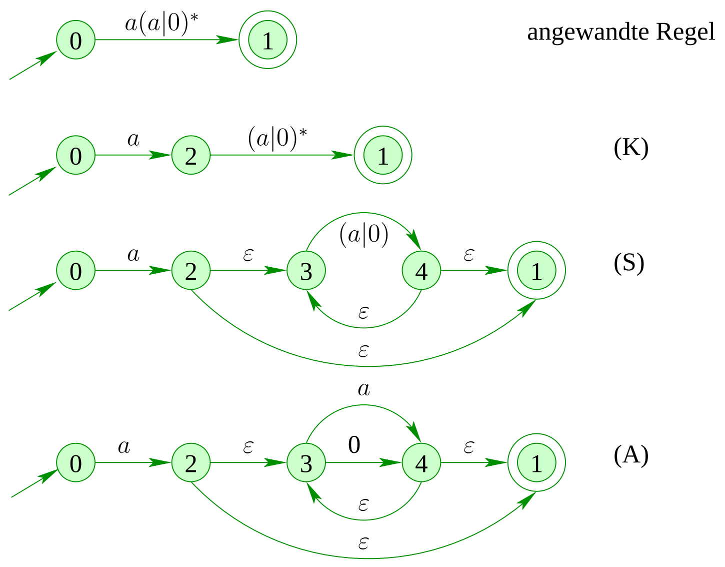


Fig. 2.4. Construction of a state diagram for the regular expression $a(a \mid 0)^*$

The Subset Construction

For implementations, *deterministic* finite-state machines are preferable to non-deterministic finite-state machines. A deterministic finite-state machine M has no transitions under ε and for each pair (q, a) with $q \in Q$ and $a \in \Sigma$, it has exactly one successor state. So, for each state q in M and each word $w \in \Sigma^*$ it has exactly one w-path in the transition diagram of M starting in q. If q is chosen as initial state of M then w is in the language of M if and only if this path leads to a final state of M. Fortunately, we have Theorem 2.2.2.

Theorem 2.2.2 For each non-deterministic finite-state machine one can construct a deterministic finite-state machine that recognizes the same language. □

Proof. The proof is constructive and provides the second step of the generation method for scanners. It uses the *subset construction*. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be an NFSM. Goal of the subset construction is to construct a DFSM $\mathcal{P}(M) = (\mathcal{P}(Q), \Sigma, \mathcal{P}(\Delta), \mathcal{P}(q_0)\mathcal{P}(F))$ that recognizes the same language as M. For a word $w \in \Sigma^*$ let $\mathsf{states}(w) \subseteq Q$ be the set of all states $q \in Q$ for which there exists a w-path leading from the initial state q_0 to q. The DFSM $\mathcal{P}(M)$ is given by:

$$\begin{aligned}
\mathcal{P}(Q) &= \{\mathsf{states}(w) \mid w \in \Sigma^*\} \\
\mathcal{P}(q_0) &= \mathsf{states}(\varepsilon) \\
\mathcal{P}(F) &= \{\mathsf{states}(w) \mid w \in L(M)\} \\
\mathcal{P}(\Delta)(S, a) &= \mathsf{states}(wa) \qquad \text{for } S \in \mathcal{P}(Q) \text{ and } a \in \Sigma \text{ if } S = \mathsf{states}(w)
\end{aligned}$$

We convince ourselves that our definition of the transition function $\mathcal{P}(\Delta)$ is *reasonable*. To do this we show that for words $w, w' \in \Sigma^*$ with $\mathsf{states}(w) = \mathsf{states}(w')$ it holds that $\mathsf{states}(wa) = \mathsf{states}(w'a)$ for all $a \in \Sigma$. It follows in particular that M and $\mathcal{P}(M)$ accept the same language.

We need a systematic way to construct the states and the transitions of $\mathcal{P}(M)$. The set of final states of $\mathcal{P}(M)$ can be easily constructed if the set of states of $\mathcal{P}(M)$ is known because it holds:

$$\mathcal{P}(F) = \{A \in \mathcal{P}(M) \mid A \cap F \neq \emptyset\}$$

For a set $A \subseteq Q$ we define the set of ε-successor states A as

$$\mathsf{FZ}_\epsilon(S) = \{p \in Q \mid \exists\, q \in S.\, (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$$

This set consists of all states that can be reached from states in S by ε-paths in the transition diagram of M. This closure can be computed by the following function:

```
$\mathbf{set}\langle state\rangle$ closure($\mathbf{set}\langle state\rangle$ $S$) {
    $\mathbf{set}\langle state\rangle$ $result \leftarrow \emptyset$;
    $\mathbf{list}\langle state\rangle$ $W \leftarrow$ list_of($S$);
    $state\ q, q'$;
    $\mathbf{while}$ ($W \neq []$) {
        $q \leftarrow$ hd($W$); $W \leftarrow$ tl($W$);
        $\mathbf{if}$ ($q \notin result$) {
            $result \leftarrow result \cup \{q\}$;
            $\mathbf{forall}$ ($q'$ : $(q, \varepsilon, q') \in \Delta$)
                $W \leftarrow q' :: W$;
        }
    }
    $\mathbf{return}$ $result$;
}
```

The states of the non-deterministic finite-state machine reachable from A are collected In the set $result$. The list W contains all elements in $result$ whose ε-transitions are not yet processed. As long as W is not empty, the first state q from W is selected. To do this, functions hd and tl are used that return the first element and the tail of a list, respectively. Is q already contained in $result$ nothing needs to be done. Otherwise, q is inserted into the set $result$. The all transitions (q, ε, q') for q in Δ are considered and the successor states q' are added to W. By applying the closure operator $\mathsf{FZ}_\epsilon(\_)$, the initial state $\mathcal{P}(q_0)$ of the subset automaton can be computed:

$$\mathcal{P}(q_0) = S_\varepsilon = \mathsf{FZ}_\epsilon(\{q_0\})$$

To construct the set of all states $\mathcal{P}(M)$ together with the transition function $\mathcal{P}(\Delta)$ of $\mathcal{P}(M)$, bookkeeping of the set $Q' \subseteq \mathcal{P}(M)$ of already generated states and the set $\Delta' \subseteq \mathcal{P}(\Delta)$ of already created transitions is performed. Initially, $Q' = \{\mathcal{P}(q_0)\}$ and $\Delta' = \emptyset$.

For a state $S \in Q'$ and each $a \in \Sigma$ its *successor state* S' under a and Q' and the transition (S, a, S') are added to Δ. The successor state S' for S under a character $a \in \Sigma$ is obtained by collecting the successor states of all states $q \in S$ under a and adding all ε-successor states:

$$S' = \mathsf{FZ}_\epsilon(\{p \in Q \mid \exists q \in S :\ (q, a, p) \in \Delta\})$$

The function nextState() serves to compute this set:

```
$\mathbf{set}\langle state\rangle$ nextState($\mathbf{set}\langle state\rangle$ $S$, $symbol\ x$) {
    $\mathbf{set}\langle state\rangle$ $S' \leftarrow \emptyset$;
    $state\ q, q'$;
    $\mathbf{forall}$ ($q' : q \in S, (q, x, q') \in \Delta$) $S' \leftarrow S' \cup \{q'\}$;
    $\mathbf{return}$ closure($q'$);
    }
```

The extensions of Q' and Δ' are performed until all successor states of the states in Q' under characters from Σ are already contained in the set Q'. Technically, this means that the set of all states *states* and the set of all transitions *trans* of the subset automaton can be computed iteratively by the following loop:

```
**list**$\langle$**set**$\langle state \rangle\rangle$ $W$;
**set**$\langle state \rangle$ $S_0 \leftarrow \mathsf{closure}(\{q_0\})$;
$states \leftarrow \{S_0\};\ \ W \leftarrow [S_0]$;
$trans \leftarrow \emptyset$;
**set**$\langle state \rangle$ $S, S'$;
**while** $(W \neq [])$ {
    $q \leftarrow \mathsf{hd}(W);\ \ W \leftarrow \mathsf{tl}(W)$;
    **forall** $(x \in \Sigma)$ {
        $S' \leftarrow \mathsf{nextState}(S, x)$;
        $trans \leftarrow trans \cup \{(S, x, S')\}$;
        **if** $(S' \notin states)$ {
            $states \leftarrow states \cup \{S'\}$;
            $W \leftarrow W \cup \{S'\}$;
        }
    }
}
```

□

Example 2.2.5 The subset construction, applied to the finite-state machine of Example 2.2.4 could be executed by the steps described in Fig. 2.5. The states of the DFSM to be constructed are denoted by primed natural numbers $0', 1', \ldots$. The initial state $0'$ is the set $\{0\}$. The states in Q' whose successor states are already computed are underlined. The state $3'$ is the empty set of states, i.e. the *error state*. It can never be left.

It is the successor state of a state q under a if there is no transition under a from q heraus. □

Minimization

The deterministic finite-state machines generated from regular expressions in the first two stepss are in general not the smallest possible that would accept the given language. There might be states that have the same *acceptance behavior*. We say, states p and q of a DFSM have the same acceptance behavior if the DFSM goes from p and q either under all input words into a final state or under all input words into a non-final state. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a deterministic finite-state machine. To formalize the concept, same acceptance behavior, we extend the transition function $\Delta : Q \times \Sigma \rightarrow Q$ of the DFSM M function $\Delta^* : Q \times \Sigma^* \rightarrow Q$ that maps each pair $(q, w) \in Q \times \Sigma^*$ to the unique state in which ends the w-path from q in the transition diagram of M. The function Δ^* is defined inductively over the length of words:

$$\Delta^*(q, \varepsilon) = q \quad \text{und} \quad \Delta^*(q, aw) = \Delta^*(\Delta(q, a), w)$$

for all $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$. States $p, q \in Q$ have the same acceptance behavior if

$$\Delta^*(p, w) \in F \quad \text{if and only if} \quad \Delta^*(q, w) \in F$$

In this case we write $p \sim_M q$. The relation $\sim_M$ is an equivalence relation on Q. The DFSM M is called *minimal* if the equivalence relation $\sim_M$ is trivial, that is, there are no states $p \neq q$ in Q with $p \sim_M q$. For each DFSM a minimal DFSM can be constructed, which is even unique up to isomorphism. This is the claim of the following theorem.

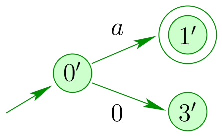
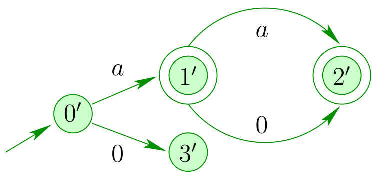
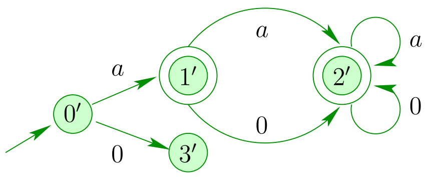
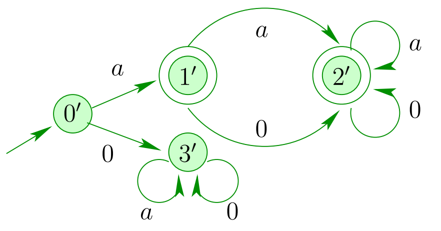
ausge-wählter Zustand	neues Q'	neuer (Teil-) DEA
$0'$	$\{\underline{0'}, 1', 3'\}$ mit $1' = \{1, 2, 3\}$	
$1'$	$\{\underline{0'}, \underline{1'}, 2', 3'\}$ mit $2' = \{1, 3, 4\}$	
$2'$	$\{\underline{0'}, \underline{1'}, \underline{2'}, 3'\}$	
$3'$	$\{\underline{0'}, \underline{1'}, \underline{2'}, \underline{3'}\}$	

Fig. 2.5. The subset construction for the NFSM of Example 2.2.4

Theorem 2.2.3 For each deterministic finite-state machine M, a minimal deterministic finite-state machine M' can be constructed that accepts the same language as M. This minimal deterministic finite-state machine is unique up to renaming of states.

Proof. For a deterministic finite-state machine $M = (Q, \Sigma, \Delta, q_0, F)$ we define a deterministic finite-state machine $M' = (Q', \Sigma, \Delta', q'_0, F')$ that is minimal. As set of states of the deterministic finite-state machine M' we choose the set of equivalence classes of states of the DFSM M under $\sim_M$. For a state $q \in Q$ let $[q]_M$ be the equivalence class of state s q with respect to the relation $\sim_M$, i.e.

$$[q]_M = \{p \in Q \mid q \sim_M p\}$$

The set of states of M' is given by:

$$Q' = \{[q]_M \mid q \in Q\}$$

Correspondingly, the initial state and the set of final states of M' are defined by

$$q'_0 = [q_0]_M \qquad\qquad F' = \{[q]_M \mid q \in F\},$$

and the transition function of M for $q' \in Q'$ and $a \in \Sigma$ is defined by

$$\Delta'(q', a) = [\Delta(q, a)]_M \quad \text{for a } q \in Q \text{ such that } q' = [q]_M.$$

One convinces oneself that the new transition function Δ' is well-defined, i.e. that for $[q_1]_M = [q_2]_M$ it holds $[\Delta(q_1, a)]_M = [\Delta(q_2, a)]_M$ for all $a \in \Sigma$. Furthermore, one shows that

$$\Delta^*(q, w) \in F \quad \text{if and only if} \quad (\Delta')^*([q]_M, a) \in F'$$

holds for all $q \in Q$ and $w \in \Sigma^*$. This implies that $L(M) = L(M')$. We claim that the DFSM M' is minimal. To show this we assume there were still states $[q_1]_M \neq [q_2]_M$ in M' that had the same acceptance behavior in M'. This would mean that $(\Delta')^*([q_1]_M, w) \in F'$ holds if and only if $(\Delta')^*([q_2]_M, w) \in F'$. But then also holds $\Delta^*(q_1, w) \in F$ if and only if $\Delta^*(q_2, w) \in F$. Therefore, q_1 and q_2 would have the same acceptance behavior in M, i.e. $q_1 \sim_M q_2$. But since $\sim_M$ is an equivalence relation this means that $[q_1]_M = [q_2]_M$, which is a contradiction to our assumption. □

We conclude that M' is indeed the desired minimal deterministic finites-state machine. The practical construction of M' requires to compute the equivalence classes $[q]_M$ of the relation $\sim_M$.

Were *each* state a final state, i.e. $Q = F$ then all states were equivalent, and $Q = [q_0]_M$ were the only state of M'.

Let us assume in the following that not every state is a final state, i.e. $Q \neq F$. The algorithm manages a *partition* Π on the set Q of the states of the DFSM M. A partition on the set Q is a set of non-empty subsets of Q, whose union is Q.

A partition Π is called *stable* under the transition relation Δ, if for all $q' \in \Pi$ and all $a \in \Sigma$ there is a $p' \in \Pi$ such that

$$\{\Delta(q, a) \mid q \in q'\} \subseteq p'$$

In a stable partition, all transitions from one set of the partition lead into exactly one set of the partition.

In the partition Π, the sets of states are managed of which we assume that they have the same acceptance behavior. If it turns out that a set $q' \in \Pi$ contains states with different acceptance behavior then the set q' is split up. Different acceptance behavior of two states q_1 and q_2 is recognized when the successor states $\Delta(q_1, a)$ and $\Delta(q_2, a)$ for a $a \in \Sigma$ lie in different sets of Π. The partition is apparently not stable. Such a split of a set in a partition is called *refinement* of Π. The successive refinement of the partition Π terminates if there is no need for further splitting of any set in the obtained partition. Π is stable under the transition relation Δ.

The construction of the minimal deterministic finite-state machine proceeds as follows: The partition Π is initialized with $\Pi = \{F, Q\backslash F\}$. Let us assume that the actual partition Π of the set Q of states of M' is not yet stable under Δ. Then there exists a set $q' \in \Pi$ and a $a \in \Sigma$ such that the set $\{\Delta(q, a) \mid q \in q'\}$ is not completely contained in any of the sets in $p' \in \Pi$. Such a set q' is then split to obtain a new partition Π' that consists of all non-empty elements of the set

$$\{\{q \in q' \mid \Delta(q, a) \in p'\} \mid p' \in \Pi\}$$

The partition Π' of q' consists of all non-empty subsets of states from q' that lead under a into the same sets in $p' \in \Pi$. The set q' in Π is replaced by the partition Π' of q', i.e. the partition Π is refined to the partition $(\Pi\backslash\{q'\}) \cup \Pi'$.

If a sequence of such refinement steps arrives at a stable partition in Π the set of states of M' has been computed.

$$\Pi = \{[q]_M \mid q \in Q\}$$

Each refinement step increases the number of sets in partition Π. A partition of the set Q may only have as many sets as Q has elements. Therefore, the algorithm terminates after finitely many steps. The proof that the minimal DFSM is unique up to renaming of states is the subject of Exercise 9. □

Example 2.2.6 We illustrate the presented method by minimizing the deterministic finite-state machine of Example 2.2.5. At the beginning, partition Π is given by

$$\{\,\{0', 3'\}, \{1', 2'\}\,\}$$

This Partition is not stable. The first set $\{0', 3'\}$ must be split into the partition $\Pi' = \{\{0'\}, \{3'\}\}$. The coreresponding refinement of partition Π produces the partition

$$\{\{0'\}, \{3'\}, \{1', 2'\}\,\}$$

This partition is stable under Δ. It therefore delivers the states of the minimal deterministic finite-state machine. The transition diagram of the so constructed deterministic finite-state machine is shown in Fig. 2.6. □

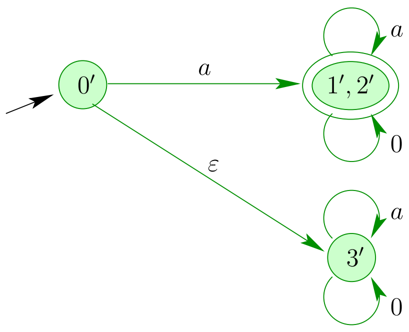


Fig. 2.6. The minimal deterministic finite-state machine of Example 2.2.6.

2.3 A Language for the Specification of Lexical Analyzers

We have met regular expressions as specifcation mechanism for symbol classes in lexical analysis. For practical purposes, one often would like to have something more comfortable.

Example 2.3.1 The following regular expression describes the language of unsigned *int*-constants of Examples 2.2.2 and 2.2.3.

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

A similar specification of *float*-constants would stretch over three lines. □

In the following, we will present some extensions of the specfication mechanism that increase the comfort, but not the expressive power of this mechanism. The class of languages that can be described remains the same.

2.3.1 Character classes

In the specification of a lexical analyzer, one should be able to group sets of characters into *classes* if these characters can be exchanged against each other without changing the symbol class of symbols in which they appear. This is particularly helpful in the case of large classes, for instance the class of all *Unicode*-characters. Examples of frequently occurring character classes are:

$$\begin{aligned} \mathsf{bu} &= a - zA - Z \\ \mathsf{zi} &= 0 - 9 \end{aligned}$$

The first wo definitions of character classes define classes by using intervals in the underlying character code, e.g. the ASCII. Note that we need another meta character, '-', for the specification of intervals. Using this feature, we can nicely specify the symbol class of identifiers:

$$\mathsf{Id} = \mathsf{bu}(\mathsf{bu} \mid \mathsf{zi})^*$$

The specification of character classes only uses three meta character, namely '=', '−', and the blank.

Example 2.3.2 The regular expression for unsigned *int*- and *float*-constants is simplified through the use of the character classes $\mathsf{zi} = 0 - 9$ to:

$$\begin{array}{l} \mathsf{zi}\ \mathsf{zi}^* \\ \mathsf{zi}\ \mathsf{zi}^* E(+ \mid -)?\mathsf{zi}\ \mathsf{zi}^* \mid \mathsf{zi}^*(.\mathsf{zi} \mid \mathsf{zi}.)\mathsf{zi}^*(E(+ \mid -)?\mathsf{zi}\ \mathsf{zi}^*)? \end{array}$$

□

2.3.2 Non-recursive Parentheses

Programming languages have lexical units that are characterized by the enclosing parentheses. Examples are strings and comments. . Parentheses limiting comments can be composed of several characters: $(*$ and $*)$ or $/*$ and $*/$ or $//$ and $\backslash n$ (newline). More or less arbitrary texts can be enclosed in the opening and the closing parentheses. This is not easily described. A comfortable abbreviation for this is:

$$r_1 \;\textsf{until}\; r_2$$

Let L_1, L_2 the languages described by r_1 bzw. r_2 where L_2 does not contain the empty word. The language described by the *until*-expression is:

$$L_1 \;\overline{\Sigma^* L_2 \Sigma^*}\; L_2$$

A comment starting with $//$ and ending at the end of line can be described by:

$$// \;\textsf{until}\; \backslash n$$

2.4 Scanner Generation

Section 2.2 described methods to derive a non-deterministic finite-state machine from a regular expression, from this a deterministic finite-state machine, and finally a minimal deterministic finite-state machine. In what follows we present the necessary extension for the practical generation of scanners and screeners.

2.4.1 Character Classes

Character classes were introduced to simplify regular expressions. They may also lead to smaller finite-state machines. The character-class definition

$$\begin{aligned} \textsf{bu} &= a - z \\ \textsf{zi} &= 0 - 9 \end{aligned}$$

can be used to replace the 26 transitions between states under letters by one transition under bu. This simplifies the FSM for the expression

$$\textsf{Id} = \textsf{bu}(\textsf{bu} \mid \textsf{zi})^*$$

considerably. The implementation uses a map χ that associates each character a with its class or practically with a code for the class. This map is stored in an array indexed by the character codes. The array components contain the code for the character class. In order for χ to be a function each character must be member of exactly one character class. Character classes are implicitly introduced for characters that don't explicitly occur in a class and those that occur directly in a symbol-class definition. The problem of non-disjoint character classes is resolved by refining the classes to become disjoint. Let us assume that the classes $z_1, \ldots, z_k$ were specified. The generator introduces for each intersection $\tilde{z}_1 \cap \ldots \cap \tilde{z}_k$ that is non-empty a new character class. $\tilde{z}_i$ either denotes z_i or the complement of z_i. Let D be the set of these newly introduced character classes. Each character class z_i corresponds to one of the alternatives $d_i = (d_{i1} \mid \ldots \mid d_{ir_i})$ of character classes in D. Each occurrence of the character class z_i in the regular expression is then replaced by d_i.

Example 2.4.1 Let us assume we had introduced the two classes

$$\begin{aligned} \textsf{bu} &= a - z \\ \textsf{buzi} &= a - z0 - 9 \end{aligned}$$

to define the symbol classes $\textsf{Id} = \textsf{bu}\;\textsf{buzi}^*$. The generator would divide one of these character classes into

$$\begin{aligned} \textsf{zi}' &= \textsf{buzi} \backslash \textsf{bu} \\ \textsf{bu}' &= \textsf{bu} \cap \textsf{buzi} = \textsf{bu} \end{aligned}$$

The occurrence of buzi in the regular expression will be replaced by $(\textsf{bu}' \mid \textsf{zi}')$. $\square$

2.4.2 An Implementation of the *until*-Construct

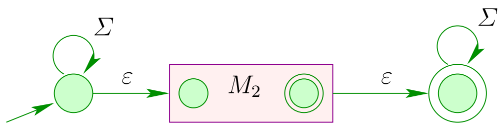
Let us assume the scanner should recognize symbols whose symbol class is specified by the expression $r = r_1$ until r_2. After recognizing a word of the language for "ur r_1 it needs to find a word of the language for r_2 and then halt. This task is a generalization of the *pattern-matching* problem on strings. There exist algorithms for this problem that solve this problem for regular patterns in time, linear in the length of the input. These are, for example, used in the UNIX-program EGREP. They construct a finite-state machine for this task. One could solve the task presented above by starting such an automaton in the final state of an automaton M_1 that recognizes the language for r_1. We will not do this, but present an approach to construct such an automaton.

Let L_1, L_2 be the languages described by the expressions r_1 and r_2. The language L defined by the expression r_1 until r_2 is:

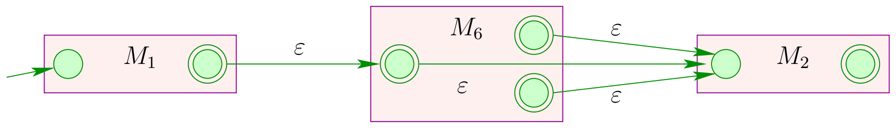
$$L = L_1\ \overline{\Sigma^* L_2 \Sigma^*}\ L_2$$

The process starts with automata for the languages L_1 and L_2, decomposes the regular expression describing the language, and applies standard constructions for automata. The process has the following seven steps: Fig. 2.7 shows all seven steps for an example.

1. The first step constructs FSMs M_1 and M_2 for the regular expressions r_1, r_2 where $L(M_1) = L_1$ and $L(M_2) = L_2$. A copy of the FSM for M_2 is needed for step 2 and one more in step 6.
2. A FSM M_3 is constructed for $\Sigma^* L_2 \Sigma^*$ using the first copy of M_2.



The FSM M_3 non-deterministically accepts all words over Σ that contain a subword from L_2.
3. The FSM M_3 is transformed into a DFSM M_4 by the subset construction.
4. A DFSM M_5 is constructed that recognizes the language for $\overline{\Sigma^* L_2 \Sigma^*}$. To achieve this, the set of final states of M_4 is exchanged with the one of non-final states. Each state that was a final state is now a non-final state and vice versa. In particular, M_5 accepts the empty word since according to our assumption $\varepsilon \notin L_2$. Therefore, the initial state of M_5 also is a final state.
5. The DFSM M_5 is transformed into a minimal DFSM M_6. All final states of M_4 are equivalent and dead since it is not possible to reach a final state of M_5 from any final states of M_4. This error state is removed.
6. Using the FSMs M_1, M_2 for L_1 and L_2 and M_6 a FSM M_7 for the language $L_1\ \overline{\Sigma^* L_2 \Sigma^*}\ L_2$ is constructed.



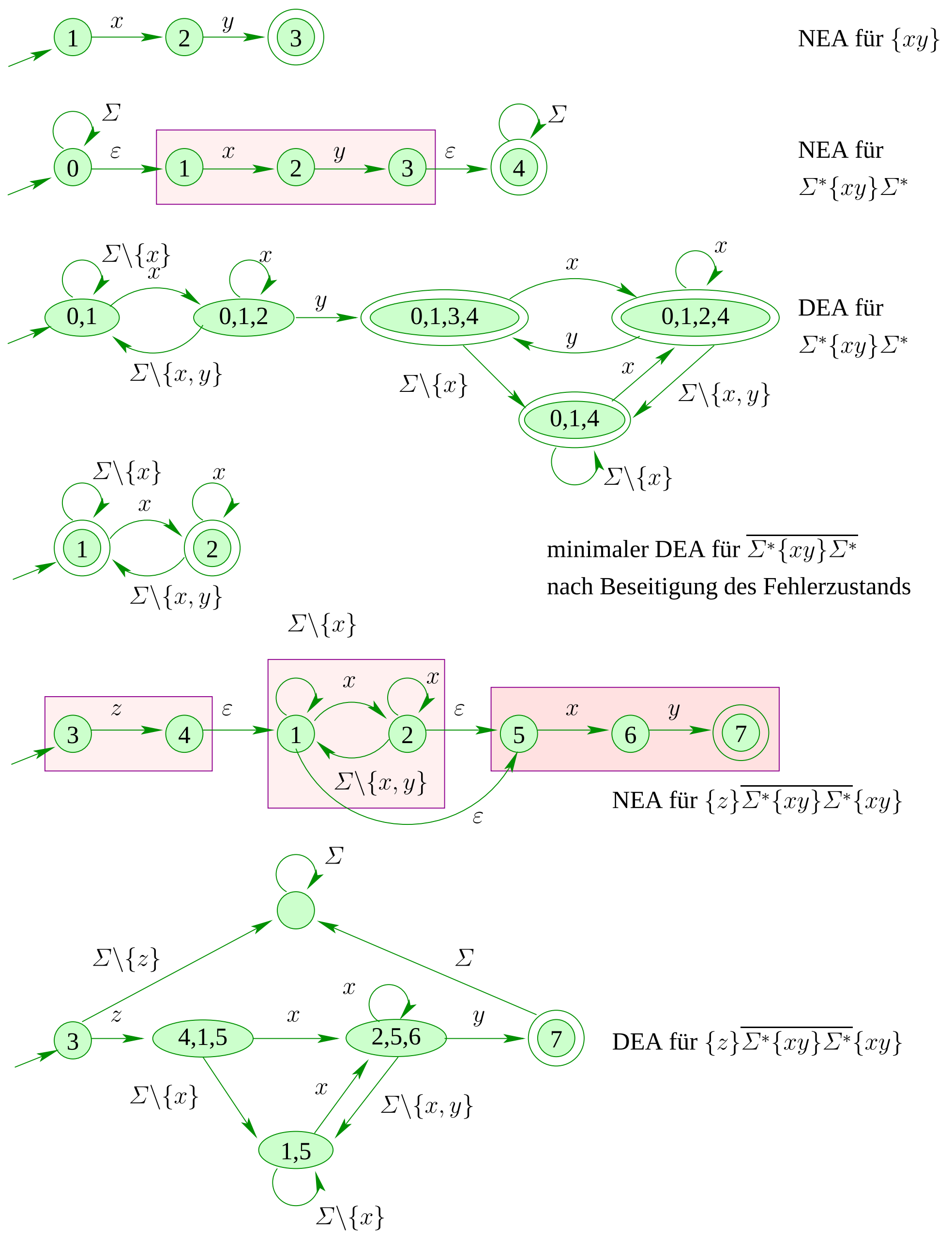
From each final state of M_6 including the initial state of M_6, there is a ε-transition to the initial state of M_2. From there paths under all words $w \in L_2$ lead into the final state of M_2, which is the only final state of M_7.
7. The FSM M_7 is converted into a DFSM M_8 and possible minimized.

2.4.3 Sequences of regular expressions

Let a sequence

$$r_0, \ldots, r_{n-1}$$

of regular expression be given for the symbol classes to be recognized by the scanner. A scanner recognizing the symbols in these classes can be generated in the following steps:



NEA für $\{xy\}$
NEA für $\Sigma^*\{xy\}\Sigma^*$
DEA für $\Sigma^*\{xy\}\Sigma^*$
minimaler DEA für $\overline{\Sigma^*\{xy\}\Sigma^*}$
nach Beseitigung des Fehlerzustands
NEA für $\{z\}\overline{\Sigma^*\{xy\}\Sigma^*}\{xy\}$
DEA für $\{z\}\overline{\Sigma^*\{xy\}\Sigma^*}\{xy\}$
0,1
0,1,2
0,1,3,4
0,1,2,4
0,1,4
4,1,5
2,5,6
1,5

Fig. 2.7. The derivation of a DFSM for z until xy with $x, y, z \in \Sigma$.

1. In a first step, FSMs $M_i = (Q_i, \Sigma, \Delta_i, q_{0,i}, F_i)$ for the regular expressions r_i are generated where the Q_i should be pairwise disjoint.
2. The FSMs M_i are combined into a FSM $M = (\Sigma, Q, \Delta, q_0, F)$ iby adding a new initial state q_0 together with ε-transitions to the initial states $q_{0,i}$ of the FSMs M_i. The FSM M, therefore, looks as follows:

$$\begin{aligned} Q &= \{q_0\} \cup Q_0 \cup \ldots \cup Q_{n-1} \quad \text{f"ur ein} \quad q_0 \notin Q_0 \cup \ldots \cup Q_{n-1} \\ F &= F_0 \cup \ldots \cup F_{n-1} \\ \Delta &= \{(q_0, \varepsilon, q_{0,i}) \mid 0 \leq i \leq n-1\} \cup \Delta_0 \cup \ldots \cup \Delta_{n-1} \ . \end{aligned}$$

The FSM M for the sequence accepts the *union* of the languages that were accepted by the FSMs

M_i. The final state reached by a succesful run of the automaton indicates to which class the found symbol belongs.

3. The subset construction is applied to the FSM M resulting in a determinstic finite-state machine $\mathcal{P}(M)$. A word w is associated with the i-th symbol class if it belongs to the language of r_i, but to no language of the other regular expressions $r_j, j < i$. Expressions with a smaller index are here preferred over expressions with larger indices.
   To which symbol class a word w belongs can be computed by the DFSM $\mathcal{P}(M)$. The word w belongs to the i-th symbol class if and only if it drives the DFSM $\mathcal{P}(M)$ into a state $q' \subseteq Q$ such that
   $$q' \cap F_i \neq \emptyset \quad \text{und} \quad q' \cap F_j = \emptyset \quad \text{f"ur alle } j < i.$$
   The set of all these states q' is denoted by F'_i.
4. After this stp, one may minimized the DFSM $\mathcal{P}(M)$. During minimization, the sets of final states F'_i and F'_j for $i \neq j$ should be kept separate. The minimization algorithm should, therefore, start with the initial partition
   $$\Pi = \{F'_0, F'_1, \ldots, F'_{n-1}, \mathcal{P}(Q) \backslash \bigcup_{i=0}^{n-1} F'_i\}$$
   .

Example 2.4.2 Let the following sequence of character classes be given:

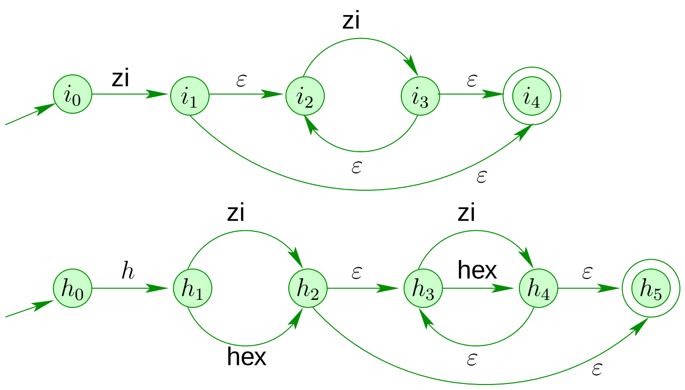
$$\begin{aligned} \mathsf{zi} &= 0-9 \\ \mathsf{hex} &= A-F \end{aligned}$$

The sequence of regular definitions

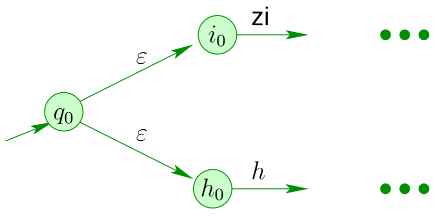
$$\mathsf{zi}\ \mathsf{zi}^*$$
$$h(\mathsf{zi} \mid \mathsf{hex})(\mathsf{zi} \mid \mathsf{hex})^*$$

for the symbol classes Intconst and Hexconst are processed in the following steps:

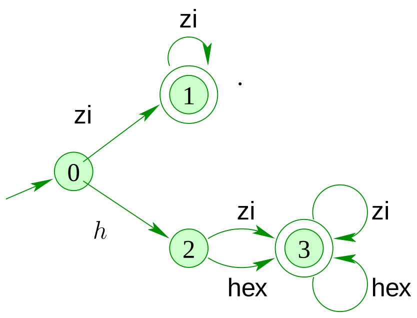
- FSMs are generated for these regular expressions.



  The final state i_4 stands for symbols of the class Intconst, while the final state h_5 stands for symbols of the class Hexconst.
- The two FSMs are combined with a new initial state q_0:



- The resulting FSM is then made deterministic:



  An additional state 4 is needed, the error state corresponding to the empty set of original states. This state and all transitions into it are left out in the transition diagramm in order to keep the readability.
- Minimzation of the DFSM does not change it in this example.

The new final state of the generated DFSM contains the old final state i_4 and, therefore, signals the recoginition of symbols of symbol class Intconst. Final state 3 contains h_5 and, therefore, signals the symbol class Hexconst.

Generated scanners always search for longest prefices of the remaining input that leads into a final state. The scanner will, therefore, make a transition out of state 1 if this is possible, that is, if a digit follows. If the next input character is not a digit, the scanner should return to state 1 and reset its reading head. □

2.4.4 The Implementation of a Scanner

We have seen that the core of a scanner is a deterministic finte-state machine. The transition function of this machine can be represented by a two-dimensional array delta. This array is indexed by the actual state and the character class of the next input character. The selected array component contains the new state into which the DFSM should go when reading this character in the actual state. States and character classes are coded at non-negative integers. The access to delta$[q, a]$ is usually fast. However, the size of the array delta may be large. This DFSM often contains many transitions into the error state *error*. We, therefore, choose this state as the *default value* for the entries in delta. It then suffices to only represent transitions into non-error states. This might lead to a sparsely populated array, whcih can be compressed using well-known methods. These save much space at the cost of slightly increased access time. It should not be forgotten that the now empty entries represent transitions into the error state. These are still relevant for the scanners error-detecting capabilities. Thus, this information must still be available.

Let us consider one such compression method. Instead of using the original array delta to represent the transition function we represent it by an array RowPtr, which is indexed by states and whose components are addresses of the original rows of delta, see Fig. 2.8.

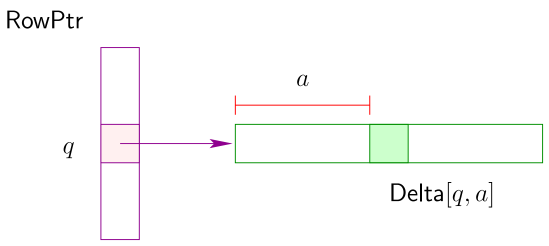


Fig. 2.8. Representation of the transition function of a DFSM.

We haven't won anything, yet, but even lost access efficiency. As said above, the rows of delta to which entries in RowPtr point are often almost empty. The rows will, therefore, be overlaid into a 1-dimensional array Delta in such a way that non-empty entries of delta do not collide. To find the starting position for the next row to be inserted into Delta one can use the *first-fit*-strategy. This row will be shifted over the array Delta starting at its beginning, until no non-empty entries of this row collide with non-empty entries already allocated in Delta.

The index in Delta at which the q-th row of delta is allocated is stored in RowPtr$[q]$. See Fig. 2.9.

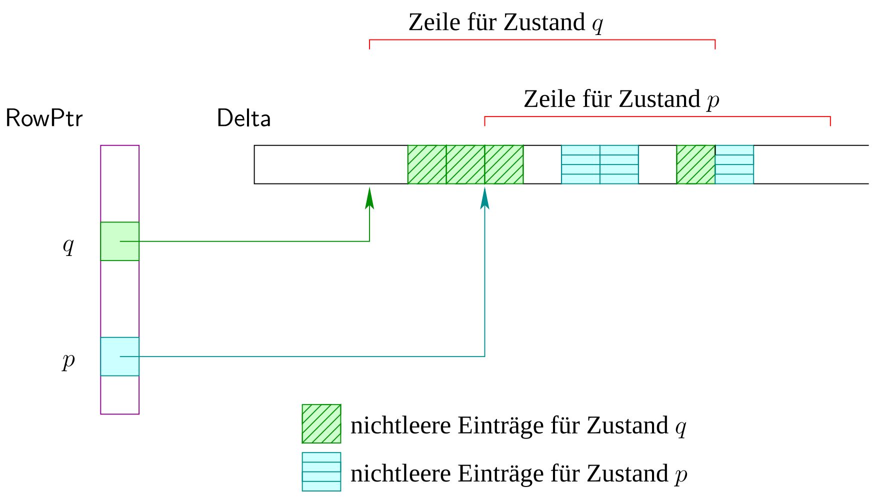

Zeile für Zustand q
Zeile für Zustand p
RowPtr
Delta
q
p
nichtleere Einträge für Zustand q
nichtleere Einträge für Zustand p

Fig. 2.9. Compressed representation of the transition function of a DFSM.

One problem is that the represented DFSM has lost its ability to identify errors, that is, undefined transitions. Let us consider an undefined entry $\Delta(q,a)$, representing a transition into the error state. However, Delta$[\text{RowPtr}[q]+a]$ might contain a non-empty entry stemming from a shifted row of a state $p \neq q$. Another 1-dimensional array Valid is added, whcih has the same length as Delta. It contains the information to which states the entries in Delta belong. This means that Valid$[\text{RowPtr}[q]+a] = q$ if and only if $\Delta(q,a)$ is defined. The transition function of the deterministic finte-state machine can then be implemented by a function next() as follows:

```
$State$ next ($State$ $q$, $CharClass$ $a$) {
    **if** (Valid[RowPtr[$q$] + $a$] $\neq q$) **return** error;
    **return** Delta[RowPtr[$q$] + $a$];
}
```

2.5 The Screener

Scanners can be used in many applications, even beyond the pure splitting of a stream of characters according to a specification by regular expressions. Hence, also scanner generators are useful to automatically implement scanners. Scanner often can do more than splitting character streams, for instance processing the tokens found in the stream.

To specify this extended functionality, each symbol class may have an associated semantic action. A screener can, therefore, be specified as a sequence of pairs of the form

$$\begin{array}{ll} r_0 & \{\text{action}_0\} \\ \dots & \\ r_{n-1} & \{\text{action}_{n-1}\} \end{array}$$

where the r_i are possibly extended regular expressions over character classes specifying the i-th symbol class, and action$_i$ denotes the semantic action to be executed when a symbol of this class is found. The semanic actions are specified as code in a particular progamming language if the screener is to be implemented in this programming language. Different languages offer different adequate ways to return a representation of a found symbol. An implementation in C would, for instance, return an *int*-value as code for a symbol class. All other concerned values are stored into global values. Somewhat more comfort would be offered for an implementation of the screener in a modern object-oriented languages such as JAVA. One could introduce a class Token whose subclasses C_i would correspond to the symbol classes. The last statement in action$_i$ should be a *return*-statement returning an object of class C_i whose attibutes would store all properties of the identified symbol. In a functional language such as OCAML, one could supply a data type $token$ whose constructors C_i correspond to the different symbol classes. The semantic action action$_i$ is written in the form of an expression of type token whose value $C_i(\ldots)$ represents the identified symbol of class C_i.

Semantic actions often need to access the text of the actual symbol. Some generated scanners have access to it in a *global* variable $yytext$. Further global variables contain information such as the position of the actual symbol in the input. These are important for the generation of meaningful error messages. Some symbols should be ignored bu the screener. Instead of returning such a symbol to the parser the scanner would be asked for the next symbol from the input. For example, a comment might have to be skipped or a compiler directive might be realized and the next symbol be asked for. In a generator for C oder JAVA no *return*-statement would terminate the semantic actions.

A function yylex() is generatd from such a specification. It returns the next symbol every time it is called. Let us assume a a function scan() has been generated for the sequence $r_0, \ldots, r_{n-1}$ of regular expression. It would store the next symbol as a string in the global variable $yytext$ and return the number i of the class of ths symbol. The function yylex() might then be

```
Token yylex() {
    while(true)
        switch scan() {
        case 0      :  action_0; break;
                       ...
        case n − 1 :  action_{n−1}; break;
        default     :  return error();
        }
}
```

The function error() handles the case that an error occurs while the scanner attempts to identify the next symbol. If an action action$_i$ does not have a *return*-statement the this action will resume execution at the beginning of the *switch*-statement and reads the next symbol in the remaining input. If if doess possess a *return*-statement, executing it will terminate the *switch*-statement, the *while*-loop and the actual call of the function yylex().

2.5.1 Scanner States

Sometimes it is useful to recognize different symbol classes depending on some context. Many scanner generators produce scanners with *scanner states*. The scanner may pass from one state to another one upon reading a symbol.

Example 2.5.1 Skipping comments can be elegantly implemented using scanner states. For this purpose, a distinction is made between a state normal and a state comment.

Symbols from symbol classes that are relevant for the semantics are processed in state normal. An additional symbol class CommentInit contains the start symbol of a comment, e.g. $/*$. The semantic action triggered by recognizing the symbol $/*$ switches to state comment In state comment, only the end symbol for comments, $*/$, is recognized. All other input characters are skipped. The semantic action triggered upon finding the end-comment symbol switches back to state normal.

The actual scanner state can be kept in a global variable $yystate$. The assignment $yystate \leftarrow$ state changes the state to the new state state. The specification of a scanner possessing scanner states has the form

$$\begin{array}{ll} A_0: & class\_list_0 \\ & \dots \\ A_{r-1}: & class\_list_{r-1} \end{array}$$

where $class\_list_j$ is the sequence of regular expressions and semantic actions for state A_j. For the states normal and comment of Example 2.5.1 we get

```
normal :
    /*   { yystate ← comment; }
         ...      //  further symbol classes
comment :
    */   { yystate ← normal; }
    .    {   }
```

The character . stands for an arbitrary input symbol. Since none of the actions for start, content, or end of comment has a *return*-statement no symbol is returned for the whole comment. □

Scanner states only influence the selection of symbol classes of which symbols are recognized. To classify symbols according to scanner states the generation process of the function yylex() can be applied to the concatenation of the sequence $class\_list_j$. The only function that needs to be modified is the function scan(). To identify the next symbol this function has no longer *one* deterministic finte-state machine but a particular one, M_j, for each subsequence $class\_list_j$. Depending on the actual scanner state A_j first the corresponding DFSM M_j is selected and used for the identification of the next symbol.

2.5.2 Recognizing Reserved Words

Many possibilities exist for the distribution of duties between scanner and screener and for the functionality of the screener. The advantages and disadvantages are not easily determined. One example for two alternatives is the recognition of keywords. According to the distribution of duties given in the last chapter, the screener is in charge of recognizing reserved symbols (keywords). One possibility to do this is to form an extra symbol class for each reserved word. Fig. 2.10 shows a finite-state machine that recognizes several reserved words in its final states. Reserved keywords in C, JAVA and OCAML have the same form as identifiers. An alternative to recognizing them in the final states of a DFSM is to let the screener do it when it processes found identifiers.

The function scan() will signal that an identifier has been found. The semantic action associated with the symbol class identifier will then check whether and if yes which keyword has been found. This distribution of work between scanner and screener keeps the size of the DFSM small. On the other hand, an efficient way to recognize keywords should be used.

Identifiers are often internally represented by unique INT-values. The screener typically uses a hash table to compute this internal code. A hash table supports the efficient comparison of a newly found identifier with identifiers that have already been entered before. The keywords should be entered into the table before lexical analysis starts. The screener can then identify strings with the same effort necessary for other identifiers.

2.6 Exercises

1. **Kleene-Star**
   Let Σ be an alphabet and $L, M \subseteq \Sigma^*$. Show:

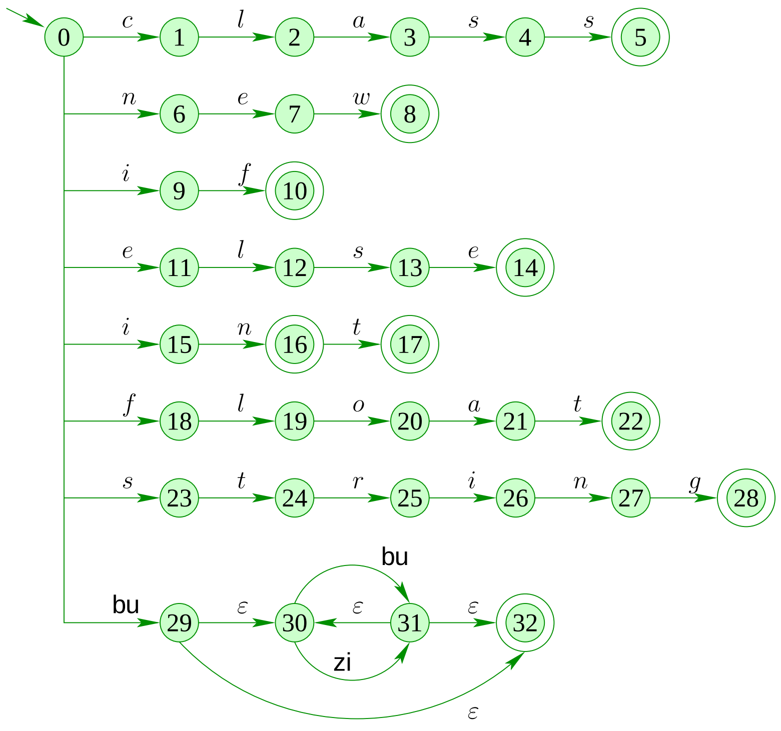


Fig. 2.10. Finite-state machine for the recognition of identifiers and keywords class, new, if, else, in, int, float, string.

a) $L \subseteq L^*$.
b) $\varepsilon \in L^*$.
c) $u, v \in L^*$ implies $uv \in L^*$.
d) L^* is the smallest set with properties (1) - (3), that is, if a set M satisfies:
$L \subseteq M,\ \varepsilon \in M$ and $(u, v \in M \ \Rightarrow\ uv \in M)$ it follows $L^* \subseteq M$.
e) $L \subseteq M$ implies $L^* \subseteq M^*$.
f) $(L^*)^* = L^*$.

2. **Symbol classes**
FORTRAN provides the implicit declaration of identifiers according to their leading character. Identifiers beginning with one of the letters i, j, k, l, m, n are taken as *int*-variables or *int*-function result. All other identifiers denote *float*-variables.
Give a definition of the symbol classes FloatId and IntId.
3. **Extended regular expressions**
Extend the construction of finite-state machines for regular expressions from Fig. 2.3 in a way that it processes regular expressions r^+ and $r?$ directly. r^+ stands for rr^* and $r?$ for $(r \mid \varepsilon)$.
4. **Extended regular expressions**
Extend the construction of finite-state machines for regular expressions by a treatment of *counting iteration*, that is, by regular expressions of the form:

$r\{u - o\}$	at least u and at most o consecutive instances of r
$r\{u-\}$	at least u consecutive instances of r
$r\{-o\}$	at most o consecutive instances of r

5. **Deterministic finite-state machines**
Convert the finite-state machine of Fig. 2.10 into a deterministic finite-state machine.

6. **Sequences of regular definitions**
   Construct a deterministic finites-state machine for the sequence of regular definitions:

$$\begin{array}{l} \mathsf{bu}\ (\mathsf{bu} \mid \mathsf{zi})^* \\ \mathsf{bu\&}\ (\mathsf{bu} \mid \mathsf{zi})^* \\ \mathsf{bu}\ \mathsf{bu\&}\ (\mathsf{bu} \mid \mathsf{zi})^* \end{array}$$

   for symbol classes Id, SysId and ComId.

7. **Character classes and symbol classes**
   Consider the following definitions of character classes:

$$\begin{array}{ll} \mathsf{bu} & = a - z \\ \mathsf{zi} & = 0 - 9 \\ \mathsf{bzi} & = 0 \mid 1 \\ \mathsf{ozi} & = 0 - 7 \\ \mathsf{hzi} & = 0 - 9 \mid A - F \end{array}$$

   and the definitions of symbol classes:

$$\begin{array}{l} b\ \mathsf{bzi}^+ \\ o\ \mathsf{ozi}^+ \\ h\ \mathsf{hzi}^+ \\ \mathsf{zi}^+ \\ \mathsf{bu}\ (\mathsf{bu} \mid \mathsf{zi})^* \end{array}$$

   a) Give the partitioning of the character classes that a scanner generator would compute.
   b) Describe the generated finite-state machine using these character classes.
   c) Convert this finite-state machine into a deterministic one.

8. **Reserved identifiers**
   Construct a deterministic finite-state machine for the finite-state machine of Fig. 2.10.

9. **Uniqueness of minimal automata**
   Let $M = (Q, \Sigma, \Delta, q_0, F)$ a *minimal* deterministic finite-state machine with $L(M) = L$. Let $M' = (Q', \Sigma, \Delta', q'_0, F')$ another minimal deterministic finite-state machine with $L(M') = L$. Prove that M and M' are identical up to the renaming of states.
   Define a relation $\sim \subseteq Q \times Q'$ with

$$q \sim q' \quad \text{falls} \quad (\forall\, w \in \Sigma* \,.\, \Delta(q, w) \in F \Leftrightarrow \Delta'(q', w) \in F')\ .$$

   Show that this relation relates each element of Q to exactly one element in Q'. Show in particular that $q_0 \sim q'_0$ holds. Derive the claim from this.

10. **Table compression**
    Compress the table of the deterministic finite-state machine using the method of Section 2.2.

11. **Processing of Roman numbers**

    a) Give a regular expression for Roman numbers.
    b) Generate a deterministic finite-state machine from this regular expression.
    c) Extend this finite-state machine such that it computes the decimal value of a Roman number. The finite-state machine can perform an assignment to *one* variable w with each state transition. The value is composed of the value of w and of constants. w is initialized with 0. Give an appropriate assignment for each state transition such that w contains the value of the recognized Roman number in each final state.

12. **Generation of a Scanner**
    Generate a OCAML-function yylex from a scanner specification in OCAML.
    Use wherever possible only functional constructs.
    a) Write a function skip that skips the recognized symbol.
    b) Extend the generator by scanner states. Write a function next that receives the successor state as argument.

2.7 Literature

The conceptual separation of scanner and screener was already proposed by F.L. DeRemer [DeR74]. Many so-called compiler generators support the generation of scanners from regular expressions. Johnson u.a. [JPAR68] describes such a system. The corresponding routine under UNIX, LEX, was realized by M. Lesk [Les75]. FLEX was implemented by Vern Paxson. The approach described in this chapter follows the scanner generator JFLEX for JAVA.

Compression methods for sparsely populated matrices as they generated in scanner and parser generators are described and analyzed in [TY79] and [DDH84].