BRINCH HANSEN ON PASCAL COMPILERS

PER BRINCH HANSEN

BRINCH HANSEN ON PASCAL COMPILERS

PER BRINCH HANSEN

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

CONTENTS

1

WHAT A COMPILER DOES

A program written in a programming language is a piece of text; for example,

```
program P;
var x: integer;
begin x := 1 end.
```

This Pascal program describes the following sequence of actions:

(1) Allocate storage for a variable x.

(2) Assign the value 1 to x.

(3) Release the storage space of x.

Before a computer can execute this program, it must be *translated* from Pascal into *machine code*. The machine code is a sequence of numbers that instruct the computer to perform the actions. In this example, the machine code might be the following numbers:

```
24 1 2 5 1
37 3
3 1
38
7
```

Each line defines a computer instruction consisting of an operation code possibly followed by some arguments. The code becomes a bit more readable if we replace the operation codes by readable names and enclose the arguments in parentheses:

```
Program(1, 2, 5, 1)
LocalVar(3).
Constant(1)
SimpleAssign
EndProgram
```

A system program, known as a *compiler*, performs the translation from Pascal to machine code. Compilers play a crucial role in software development: They enable you to ignore the complicated instruction sets of computers and write programs in a readable notation. Compilers also detect numerous programming errors even before you begin testing your programs.

However, you can ignore the code generated by a compiler only if you know that the compiler never makes a mistake! If a compiler does not always work, programming becomes extremely complicated. In that case, you will discover that the Pascal report no longer defines exactly what your program does. This is obviously unacceptable.

So the following design rule is essential:

Rule 1.1:
A compiler must be error-free.

This requirement is quite a challenge to the compiler designer when you consider that a compiler is a program of several thousand lines. Compiler writing forces you to apply very systematic methods of program design, testing, and documentation. It is one of the best educational experiences for a software engineer.

The input to a compiler is a program text. The first task of the compiler is to read the program text character by character and recognize the symbols of the language. The compiler will, for example, read the characters

p r o g r a m

and recognize them as the single word program. At this point, the compiler views the previous program example as the following sequence of symbols:

program name P **semicolon**
var name x **colon name** integer **semicolon**
begin name x **becomes numeral** 1 **end period**

This phase of compilation is called *lexical analysis.* (The word *lexical* means "pertaining to the words or vocabulary of a language.")

The next task of the compiler is to check that the symbols occur in the right order. For example, the compiler recognizes the sentence

```
x := 1
```

as an assignment statement. But if you write

```
x = 1
```

instead, the compiler will indicate that this is not a valid statement. This phase of compilation is called *syntax analysis.* (The word *syntax* means "the structure of the word order in a sentence.")

The syntax analysis is concerned only with the sequence of symbols in sentences. As long as a sentence consists of a name followed by the := symbol and an expression, it will be recognized as an assignment statement. But even though the syntax of the statement is correct, it may still be meaningless, as in the following example:

```
y := 1
```

which refers to an undefined variable y. The assignment statement

```
x := true
```

is also meaningless because the variable x and the value *true* are of different types (integer and Boolean). The phase of compilation that detects meaningless sentences such as these is called *semantic analysis.* (The word *semantics* means "the study of meaning.")

As these examples show, the compiler must perform two kinds of semantic checks: First, the compiler must make sure that the names used in a program refer to known objects: either predefined standard objects, such as the type *integer*, or objects that are defined in the program, such as the variable *x*. The problem is to recognize a definition such as

```
var x: integer;
```

and determine in which part of the program the object x can be used. This task is called *scope analysis.* (The word *scope* means "the extent of application.") During this part of the compilation, the compiler will indicate if the program uses undefined names, such as y, or introduces ambiguous names in definitions, such as

```
var x: integer; x: integer;
```

Second, the compiler must check that the operands are of compatible types. This task is called *type analysis.*

When you compile a new program for the first time, the compiler nearly always finds some errors in it. It will often require several cycles of editing and recompilation before the compiler accepts the program as formally correct. So, *in designing a compiler, you must keep in mind that most of the time it will be used to compile incorrect programs!*

If there are many errors in a program, it is convenient to output the error messages in a file which can be inspected or printed after the compilation. However, the compiler will be able to complete this file and close it properly only if the compilation itself terminates properly. If the compilation of an incorrect program causes a run-time failure, such as an arithmetic overflow, the error messages will be lost and you will have to guess what happened.

To avoid this situation, we must impose the following design requirement:

Rule 1.2:

A compilation must always terminate, no matter what the input looks like.

The easiest way to satisfy this requirement is to terminate the compilation when the first error has been detected. The user must then remove this error and recompile the program to find the next error, and so on.

Since a compilation may take several minutes, this method is just too slow. To speed up the program development process, we must add another design requirement:

Rule 1.3:

A compiler should attempt to find as many errors as possible during a single compilation.

As you will see later, this goal is not easy to achieve.

There is one exception to the rule that a compilation must always terminate: If a program is so big that the compiler exceeds the limits of its

tables, the only reasonable thing to do is to report this and stop the compilation. This is called a compilation *failure*.

If the compiler finds no formal errors in a program, it proceeds to the last phase of compilation, *code generation*. In this phase, the compiler determines the amount of storage needed for the code and variables of the program and emits final instructions. The main difficulty is that most computers have very unsystematic instruction sets that are ill suited for automatic code generation.

These, then, are the major tasks of a compiler:

Lexical analysis
Syntax analysis
Scope analysis
Type analysis
Code generation

Each of these tasks will be discussed in a separate chapter.

2

A PASCAL SUBSET

The compiler described in this book accepts a subset of the programming language Pascal known as Pascal− ("Pascal Minus"). This chapter describes Pascal− and defines the syntax of the language. I assume that you already know Pascal.

2.1 PASCAL MINUS

The Boolean and integer types are the only simple types in Pascal−. These types are standard types. The language does not have characters, reals, subrange types, or enumerated types.

The structured types are array types and record types. Packed types and variant records are not supported, nor are sets, pointers, and file types.

Every type has a name: either a standard name (*integer* or *Boolean*) or a name introduced by a type definition: for example,

```
type
  table = array [1 . . 100] of integer;
  stack = record contents: table; size: integer end;
```

A variable definition, such as

```
var A: table;
```

is correct, but the following is not:

```
var A: array [1 . . 100] of integer;
```

since it introduces an array type that has no name.

A type definition cannot rename an already existing type, as in the example:

```
type number = integer;
```

Most operations on a pair of operands are valid only if the operands are of the same type. Since every type has a name (and one name only!), it is tempting to suggest that two types are the same if, and only if, they have the same name. A name can, however, be defined as a type name in one block and as a variable name in another block. When this is taken into account, we end up with the following rule: Two types are the same only if they have the same name and the same scope. In Pascal, the rules for type compatibility are more complicated [IEEE, 1983].

Constant definitions, such as

```
const max = 100; on = true;
```

introduce names for constants.

In Pascal−, all constants have simple types. There are no string constants.

Variable definitions have the same form as in Pascal; for example,

```
var x, y: integer; yes: Boolean; lifo: stack;
```

The relational operators

```
<   =   >   <=   <>   >=
```

can be applied only to operands of simple types.

Pascal− includes assignment statements, procedure statements, if statements, while statements, and compound statements. Following are some examples of these statements.

```
x := x - 1

search(x, yes, i)

if x > 0 then x := x - 1

while index < limit do
  if A[index] = value then limit := index
  else index := index + 1

begin
  lifo.size := lifo.size + 1;
  lifo.contents[lifo.size] := x
end
```

There are no goto statements (or labels), no case or repeat statements, and no for or with statements.

Pascal− supports nested procedures with value parameters and variable parameters. A procedure cannot be used as a parameter of another procedure, and functions cannot be defined.

The only standard procedures are

```
read(x)
```

which inputs an integer value and assigns it to a variable x, and

```
write(e)
```

which outputs an integer value given by an expression e.

Algorithm 2.1 illustrates most of the features of Pascal−.

To summarize, Pascal− includes the following features of Pascal:

Standard types (Boolean, integer)
Standard procedures (read, write)
Constant definitions
Type definitions (arrays, records)
Variable definitions
Expressions
Assignment statements
Procedure statements
If statements
While statements
Compound statements
Procedure definitions

```
program ProgramExample;
const n = 100;
type table = array [1 . . n] of integer;
var A: table; i, x: integer; yes: Boolean;

procedure search(value: integer;
  var found: Boolean; var index: integer);
var limit: integer;
begin index := 1; limit := n;
  while index < limit do
    if A[index] = value then limit := index
    else index := index + 1;
  found := A[index] = value
end;

begin {input table} i := 1;
  while i <= n do
    begin read(A[i]); i := i + 1 end;
  {test search} read(x);
  while x <> 0 do
    begin search(x, yes, i);
      write(x);
      if yes then write(i);
      read(x)
    end
end.
```

Algorithm 2.1

and excludes the following concepts:

Char and real
Subrange types
Enumerated types
Variant records
Set types
Pointer types
File types
Packed types
Nameless types
Renamed types
Function definitions
Procedural parameters

Goto statements (and labels)
Case statements
Repeat statements
For statements
With statements

Pascal– has enough features to illustrate all the problems of compilation. The omitted features add more detail to a compiler, but the added logic is basically "more of the same."

2.2 VOCABULARY

The vocabulary of a natural language like English is *words.* The vocabulary of a programming language like Pascal– is symbols such as

begin sort 13 :=

Pascal– has four kinds of symbols, called word symbols, names, numerals, and special symbols.

The *word symbols* are

and array begin const div do else
end if mod not of or procedure
program record then type var while

In this book, the word symbols are shown in boldface.

The *special symbols* are

\+ – * < = > <= <> >= :=
( ) [ ] , . : ; ..

A *numeral* is a decimal notation for a nonnegative integer; for example,

0 1351

A *name* consists of a letter which may be followed by more letters and digits; for example,

x Edison RC4000

In word symbols and names, capital letters are considered equivalent to

the corresponding small letters. So the following names are equivalent to one another:

```
PASCAL    pascal    Pascal
```

Although a word symbol, such as **then**, is printed in boldface here, it will normally be displayed in roman type on a computer terminal, as in the following example:

```
if x > 0 then x := x - 1
```

You may omit some of the spaces between the symbols; for example,

```
if x>0 then x:=x-1
```

But if you remove the space between the word *then* and the name x, you get an incorrect sentence:

```
if x>0 thenx:=x-1
```

in which the Boolean expression x>0 is followed by an undefined name *thenx* instead of a **then** symbol. The purpose of the *space* is to separate the two symbols **then** and x.

Every line of program text ends with a *newline* character. Two symbols can also be separated by a newline character; for example,

```
if x>0 then
x:=x-1
```

or by a *comment* enclosed in braces:

```
if x>0 then{reserve resource}x:=x-1
```

A comment may extend over several lines and may contain other (nested) comments:

```
{This is a {nested} comment
that extends over two lines}
```

The character { cannot occur within a comment (except as part of a nested comment).

Spaces, newline characters, and comments are called *separators*. Any symbol may be preceded by one or more separators. Two adjacent word symbols, names, and numerals must be separated by at least one separator.

2.3 SYNTACTIC RULES

A sequence of symbols that is formed according according to the rules of a programming language is called a *sentence* in the language. In Pascal−, a variable definition, such as

```
var x: integer;
```

and an assignment statement, such as

```
x := x - 1
```

are sentences. However, as the following example shows, not every sequence of symbols is a sentence:

```
x - x := 1
```

The rules that define all possible sentences of a programming language are called the *grammar* of the language. We will define the grammar of Pascal− in a notation known as the extended *Backus-Naur form* (or BNF).

We will introduce BNF rules by means of examples that define the syntax of very simple arithmetic expressions. In these expressions, the only operands are numerals and the operators are either + or −. Some examples of these expressions are

```
−5    3 + 1066 − 4    118 − (7 + 12)
```

The following grammar defines all possible expressions of this form:

(1) Expression = [Operator] Term { Operator Term } .

(2) Operator = "+" | "−" .

(3) Term = Numeral | "(" Expression ")" .

(4) Numeral = Digit { Digit } .

(5) Digit = "0" | "1" | "2" | "3" | "4" |
"5" | "6" | "7" | "8" | "9" .

The grammar consists of five BNF rules:

Rule 1 says that an expression consists of three parts. The first part is either an operator or nothing. This is expressed by the notation

```
[ Operator ]
```