PLDI3 notes

Paul Cockshott

October 3, 2001

Contents

1 Grammars and machines **4**
1.1 Class 3 5
1.2 Class 2 6
1.3 Class 1 6
1.4 Hierarchy of grammars 7
1.5 How should state bits be organised? 9
1.6 Random Addressed store 10

2 Lexical analysis **13**
2.1 Tabular FSM representation 14
2.2 Character Classes 15
2.2.1 The primitive state machine 18
2.2.2 Class table 18
2.2.3 McCtab program 21
2.2.4 Output Classes and Actions 21
2.2.5 Assign Characters to Classes 21
2.2.5.1 Make comment 23
2.2.5.2 Output class 23
2.2.5.3 Define Charactersets 24
2.2.5.4 Opsys 24
2.2.5.5 Brackets 25
2.2.5.6 Alphanumerics 25
2.2.6 Transition table 25
2.2.7 Recognise and print 25
2.2.8 Recognise 25
2.3 Interface to FSM 28
2.3.0.1 Compute new state 29
2.3.0.2 Handle end of buffer 29
2.3.0.3 Get next character 29
2.3.1 TEXTBUFFER 29
2.3.2 Rewind 29
2.3.3 Push buffer 30
2.3.4 Pop buffer 30
2.4 Identifier management 31

2.5 Tries 33
2.5.1 Trie data structure 34
2.5.2 Trie insertion 35
2.5.2.1 Create a new node 35
2.5.3 Recursive insert 37
2.5.3.1 Add another letter 37
2.5.4 Last letter of a word 38
2.5.4.1 Char less than prefix 38
2.6 Lexeme Definition 38
2.7 Interface to syntax analyser 39
2.7.1 Mustbe 41
2.7.2 Syntax errors 42
2.7.3 Current Id 42
2.7.4 Converting lexemes to strings 43
2.7.5 NextSymbol 43
2.7.5.1 Convert string to number 46
2.7.5.2 Convert string to internal form 46
2.7.5.3 Convert prefixed characters 47
2.7.5.4 Type coercion operation 47
2.7.6 Detect run time error location 48
2.8 Exercises 48

3 Syntax analysis 50
3.1 Specification of a machine readable syntax 52
3.2 Control Structures 53
3.3 Expressions 55

4 Types and identifiers 60
4.1 What types are. 60
4.2 Cartesian composition 61
4.3 Unions 61
4.4 Sets 61
4.5 Subranges 61
4.6 Maps 61
4.7 Constancy 62
4.8 Representing types in a compiler 62
4.9 Representing Identifiers 64

5 Code generation 68
5.1 Modification of syntax analyser 68
5.2 Notion of an abstract machine 70
5.3 Expressions and reverse polish notation 70
5.4 Handling of conditionals 72
5.4.1 If clauses 73
5.4.2 For loops 73
5.5 Variable access 78

5.5.1 Stack variables 78
5.6 Procedure calls 80
5.6.1 Simple procedures 80
5.6.2 Nested procedures 82
5.7 Structure of the code generator 84
5.7.1 Register use 85
5.7.2 Keeping track of the stack 85

6 The Assembler **87**
6.1 Converting the opcodes 89
6.2 Address handling 90
6.3 The interface 91

List of Algorithms

1 tabular FSM representation 15
2 15
3 Finite state transducer module FSM 19
4 mckctab 22
5 22
6 assign characters to classes 23
7 make comment 23
8 Output class 24
9 define charactersets 24
10 Opsys 24
11 Brackets 25
12 Alphanumerics 25
13 recognise and print 27
14 recognise 27
15 interface to FSM 28
16 compute new state 29
17 Handle end of buffer 29
18 get next character 29
19 textbuffer 30
20 rewind 30
21 push buffer 31
22 pop buffer 31
23 program SIMTRANS 33
24 35
25 Insert token 36
26 53
27 The recognizer for case clauses 55
28 parsing exp4s 58
29 Basic type checking functions in a compiler 63
30 If clauses 74
31 for loop code generation 76
32 code generated for a for loop 77
33 example of nested scopes 79
34 Code generated for the nested scopes in algorigthm **??** 79
35 82

36 88
37 90
38 92

Introduction

The notes in this collection are designed to suplement the material presented in lectures. They give more details on several topics covered in the course. In doing this they use illustrations drawn from the compiler writers toolbox, a library of Pascal modules for the implementation of interactive compilers under MS-DOS. The toolbox is available from Paul Cockshotts web pages.

It should be noted that whilst the source code is given in Pascal, the essential principles of the algorithms are not altered by shifting to Java, C or some other language.

Some detailed algorithms are provided with close documentation, in particular the algorithms for implementing lexical analysers. These are provided as background material only. It should not be necessary for you to implement a lexical analyser in your exercises, as a ready written lexical analyser class is provided. However, the interface between the lexical analyser described in these notes and the Basic lexical analyser that you will be using for your course are very similar.

Chapter 1

Grammars and machines

What makes a language a language rather than an arbitrary sequence of symbols is its grammar.

A grammar specifies the order in which the symbols of a language may be combined to make up legitimate statements in the language. Human languages have rather relaxed informal grammars that we pick up as children. Computer languages are sometimes called formal languages because they obey an explicitly specified grammar.

When people came to design computer languages around the end of the 1950's they had to devise methods of formally specifying what the grammar of these new language was to be. By coincidence the linguist Chomsky had been investigating the possibility of formally specifying natural languages, and had published an influential paper in which he had classified all possible grammars into 4 classes. These classes of grammars are now refered to as Chomsky class 0, class 1, class 2 and class 3 grammars. It turns out that Chomsky class 2 and class 3 grammars are most suitable to describe programming languages. To understand what these different classes of grammars are we need to go into a little formal notation.

The syntax or grammar of a language can be thought of as being made up of a 4 tuple $(\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{P})$ where:

$\mathcal{T}$ stands for what are called the terminal symbols of the language. In a human language these terminal symbols are the words or lexicon of the language. In a computer language they are things like identifiers, reserved words and punctuation symbols.

$\mathcal{N}$ stands for what are called the non-terminal symbols of the language. In a human language a non-terminal would be grammatical constructs like a sentence, a noun clause or a phrase. A computer language is likely to have a large number of non-terminals with names like **clause, statement, expression**.

$\mathcal{S}$ is the start symbol of the grammar. It is one of the non terminals. Its meaning will become clear shortly.

$\mathcal{P}$ is a set of productions or rewrite rules. These tell you how to expand a non-terminal in terms of other terminals and non-terminals.

This sounds a bit dry, but it will be clearer if we give an example. Suppose we wish to define a grammar that describes the 'speech' of a traffic light. A traffic light has a very limited vocabulary. It can say **red** or **amber** or **green** or **red-and-amber**. These are the terminal symbols of its language.

$\mathcal{T}$ = { **red, green, amber, red-and-amber** }

At any moment in time the traffic light is in a current state and after some interval it goes into a new state that becomes its current state. Each state is described by one of the colours of $\mathcal{T}$. This can be expressed as a set of non-terminal symbols which we will call:

$\mathcal{N}$ = { *going-red, going-green, going-amber, going-red-and-amber* }

We will assume that when the power is applied for the first time the light enters state going-red. Thus

$\mathcal{S}$ = *going-red*

A traffic light has to go through a fixed sequence of colours. These are the syntax of the traffic light language. Which sequence it goes through is defined by the productions of the traffic light language. If the light is in going-red then it must output a red and go into going-red-and-amber. We can write this down as:

going-red $\rightarrow$ ***red*** *going-red-and-amber*

This is an individual production in the traffic light language. The whole set of productions is given by:

$\mathcal{P}$ = { *going-red* $\rightarrow$ ***red*** *going-red-and-amber*
going-green $\rightarrow$ ***green*** *going-amber*
going-red-and-amber $\rightarrow$ *red-and-amber going-green*
going-amber $\rightarrow$ *amber going-red*
}

This combination of $(\mathcal{T},\mathcal{N},\mathcal{S},\mathcal{P})$ ensures that the only sequence of colours allowed by the traffic light are thing like :

red red-and-amber green amber red *going-red-and-amber*

It turns out that traffic lights speak the simplest of the Chomsky classes of language, which perversely enough is class 3.

To distinguish between these classes of grammars the following notation will be used:

bold letters : **a b c** ... represent non-terminals
italic letters : *a b c* ... represent terminals

1.1 Class 3

Class 3 languages like that of the traffic light have all of their productions of the form:

$\mathbf{a} \rightarrow a\mathbf{b}$
or
$\mathbf{a} \rightarrow c$

The traffic light obviously only has the first type of production or it would stop at some point. These simple languages occur widely in nature. Look at

Figure 1.1: Plant generated by a regular grammar

$\mathcal{T}$ = { **flower, left, right** }
$\mathcal{N}$ = { *lstem, rstem* }
$\mathcal{S}$ = *lstem*
$\mathcal{P}$ = { *rstem*→ ***flower***
lstem→ ***left*** *rstem*
rstem→ ***right*** *lstem*
}

the patterns of leaves round the stem of a plant. They will often alternate left or right, or form a spiral that can be described by a class 3 grammar. In the example in figure 1.1 we can describe the plant shape by the grammar:

Class 3 grammars are also sometimes described at regular grammars and the patterns they describe as regular expressions. It turns out that the reserved words of most computer languages can be described by class 3 grammars. We will go into this in more detail in chapters **??** and **??**.

1.2 Class 2

Class 2 grammars, also called context free grammars have productions of the form

$a \rightarrow b$

where **a** is a non-terminal symbol and **b** is some combination of terminals and non terminals. We could describe the 'if' expression in an algol like language as:

if-expression→ ***if*** *expression* ***then*** *expression* ***else*** *expression*

where italics are non-terminals and bold letters are terminals. Most of the syntax of algol like programming languages can be captured using class 2 grammars. How these are used makes up the main topic of chapter **??**.

1.3 Class 1

Class 1 grammars, also called context sensitive grammars have production of the form

$abc \rightarrow axc$

where **a** and **c** are strings of terminals and non-terminals,

b

is a single non-terminal and **x** is a non-empty string of terminals and non-terminals.

The reason why these are called context sensitive is that the production of **x** from **b** can only occur in the context of **abc**. In the context free languages a non terminal can be expanded out irrespective of the context. Although the

Program Module	Grammar
type checking	class 1
syntax analysis	class 2
lexical analysis	class 3

Figure 1.2: The hierarchy of grammars is reflected in compiler structure

bulk of a programming language's syntax can be described in a context free fashion, some parts are context sensitive. Consider the line:

```
x:=9
```

This will only be valid if at some point previously there has been a line declaring `x`. The name of the variable must have been introduced earlier and it must have been specified that it was an integer or real variable. The context sensitive part of the language is dealt with by the type checking system. In untyped languages like Basic context sensitive parts are minimal. In more advanced languages they are crucial. Class 0 grammars, the most powerful class are not needed for translating programming languages and we will ignore them.

1.4 Hierarchy of grammars

A programming language can be translated by using a hierarchy of grammars.

At the lowest level we use class 3 grammars to recognise the identifiers and reserved words of the language. Above that we use class 2 grammars to analyze the context free parts of the language. Finally we use type checkers to verify that the context sensitive rules of the language are being obeyed. The structure of the compiler reflects this structure of the language. to each of the layers of grammar there is a module of the compiler. It also turns out that in our strategy for writing the compiler we can take advantage of a relationship which exists between classes of grammars and types of computing machines.

The idea of store and stored state will be familiar to all programmers, but a stored program computer need not in principle be anything like the Von Neumann machines that we normally call computers. There are in principle much more general purpose designs. At the most general level digital computer capable of performing computation over time must contain a set of storage cells each capable of holding a bit. The computer is capable of existing in a number of states characterised by the values in its storage cells. If we consider these we can see that the number of states that the computer can occupy will be 2^s where s is the number of storage cells in the machine.

Computation proceeds by the computer going from one state to the next as shown by figure 1.3.

Clearly the number of state that a computer can go through in the course of a computation will be 2^s. The larger the number of storage cells in the machine

Figure 1.3: Computation as an evolution of numbered states

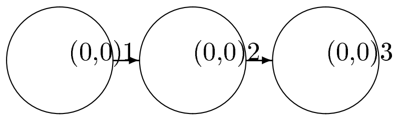

Figure 1.4: A digital door lock

the longer or more complex the sequence of state that it can go through. This relationship is familiar to us all in the way more complex programs demand more store.

To actually perform computation it is necessary to be able to modify the sequence of states that the computer goes through on the basis of input signals. To produce any useful effect the computer must generate one or more output signals, to indicate the result of the computation. Reduced to its most simple a computer must be capable of responding to a sequence of inputs and generating appropriate outputs.

Consider a machine that has to recognise a 3 digit sequence and then respond with a yes or no according to whether or not the sequence was correct. An example might be digital door lock as shown in the diagram below. This requires the sequence 469 to be keyed in to open the lock. This sequence of numbers can be described by a class 3 grammar: $(\mathcal{T},\mathcal{N},\mathcal{S},\mathcal{P})$ where

$\mathcal{T} = \{$ **1, 2, 3, 4, 5, 6, 7, 8, 9, 0** $\}$
$\mathcal{N} = \{ s, t, u \}$
$\mathcal{S} = s$
$\mathcal{P} = \{$
$\boldsymbol{s} \rightarrow 4\boldsymbol{t}$
$\boldsymbol{t} \rightarrow 6\boldsymbol{u}$
$\boldsymbol{u} \rightarrow 9$
$\}$

We define a set of numbered states corresponding to the non terminals of the grammar such that $s = \mathbf{1}$, $t = \mathbf{2}$ etc. The machine starts in state 1 and undergoes transitions to successive states on the basis of the keys that are pressed. If any incorrect key is pressed the state reverts to the start state. This is shown in figure 1.4. The collection of numbered circles with arrows connecting them is called a state transition diagram.

Figure 1.5: A finite state machine

1.5 How should state bits be organised?

Given that we wish to be able to represent state, it still does not follow that we will end up with a random access store. There are other possibilities, which have been tried in the past, and which still have certain limited applications. If each bit represents a state we could easily construct the door lock by stringing 4 cells together in sequence and having them activated sequentially by the and of the signal from a button and the previous state.

This is simple to implement and some digital logic used to be built this way, but it makes poor use of the state bits as we only get s rather than 2^s states from an s bit store.

An improved arrangement is to gather all of the bits in the computer into one word s bits long. This is then treated as a binary number and the computer program can be thought of as a mapping of the form:

program:(int x input)$\rightarrow$ (int x output)

Succesive application of the program function to the state word and the input generates a new state and an output. This is in theoretical terms the ideal way to construct a computer. For large s the number of states possible becomes astronomical. A computer with a 64 bit status word could have a state to represent every centimeter of the distance between here and the nearest star. This sort of computer is a generalised finite state automaton . If the computer is organised as shown in figure 1.5.

This architecture can go from any state to any other in a single step - from here to Alpha Centauri without stopping on the way. Each bit in the current state can be taken into account in determining the next state. All values of the input bits can be taken into account likewise. A class 3 grammar can be handled by a finite state machine. Finite state machines are widely used in computer hardware in the form of PLA's or programmable logic arrays. These are basic components of microprocessors that are used to decode machine instructions. The instruction decode unit of a microprocessor has to parse machine code. Machine code has a class 3 grammar so a finite state machine is enough.

Although this sort of machine is very fast, we have practical difficulties in scaling it up. The problem is the rate at which hardware complexity increases with the size of the computer. The number of interconnection wires required to allow each state bit to affect the next state of every other bit goes up as the square of the number of bits, and the number of logic cells (ANDs , ORs) to do this goes up quadratically in the number of state bits. This is illustrated in figure 1.6.

Figure 1.6: The number of wires used to connect state cells in finite automata goes up as $s(s-1)$

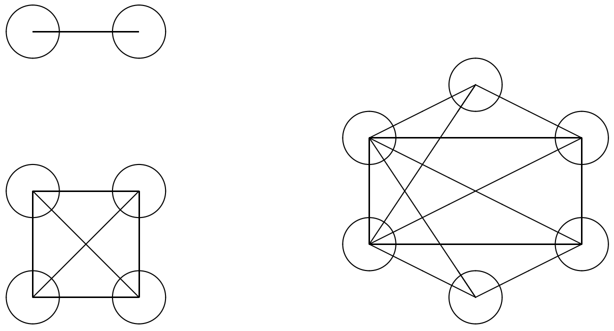

Figure 1.7: A random addressed store computer

1.6 Random Addressed store

If instead of connecting all of the state cells to one another, we organise the state cells into subgroups of bits termed words and lead these into a common logic block then we can diminish the number of wires considerably

If we divide our s state bits into $w = s/b$ words each of b bits, and wire them up in a grid, we need only $(w+b)$ wires to join them to the common logic block. What we then have is the random access memory computer. Like a generalised finite state automaton it runs in a cycle reading the present state and modifying the state vector as a result of what it has read but it is less powerful than an FSA in that at most w bits of the state can be taken into account each cycle and at most w bits of the state altered in each cycle.

The paradox is that although the FSA is the fastest type of computing machine, used in CPU's where speed matters, it is linguistically the least competent. Suppose that I have a class 2 grammar $(\mathcal{T},\mathcal{N},\mathcal{S},\mathcal{P})$

where
$\mathcal{T}$ = { **), (, 1, 2, 3** }
$\mathcal{N}$= {*s*, *t*, *u* }
$\mathcal{S}$= *s*
$\mathcal{P}$= {*s*→*(t)*
t→***1u2u***→***t***
u→***s***
u→***3***
}
This can generate sequences like
(132) or **(11322)** or **(1(132)2)**

Figure 1.8: Stack machine parsing a class 2 grammar

You will find that you can not draw a state transition diagram that is capable of handling this syntax. In fact it can not be handled by a finite state machine. The machine would have to remember how many left brackets and how many 1s it had encountered and in what order they had come so that it could match them up with right brackets and 2s. Since the sequence defined by the grammar can be of arbitrary length, no finite memory could hold the information.

To handle a class 2 grammar like this you need to have an infinite stack memory. As each left bracket or one is encountered, a token representing it is pushed onto the stack. When parsing, the computer looks at the top of the stack and at the next character to decide what state to go into.

Of course in practice any stack that we build will be of finite depth. This means that looked at another way a stack machine is still a finite state automaton.

There will be sequences of symbols that are just too long to parse. For practical purposes we are willing to accept that some programs are too big to compile. But we can write our compiler as if it was going to run on a computer with an infinite stack. This technique allows us to write a program that only needs to have a small number of rules in it. The complexity of the parser is then limited by the size of the grammar itself rather than by the size of the programs it will have to compile.

When we take into account context sensitive information, we will need the full facilities of a random access memory in which we can built up information about what identifiers and types have been declared. Broadly speaking, the lexical analysis part of compiling will be handled by algorithms that mimic a finite state machine. The syntax analysis will be handled using a stack, and the type checking will use a random access heap.

Exercises

1. Consider the following grammar.

   expression→ (expression)

   expression→ expression + expression

   expression→ number

   number→ digit number

   number→ digit

   digit→ 0

   digit→ 1

2. give 4 examples of valid expressions derived from this.

   (a) give a description in words of the language produced by the grammar.
   (b) what class of grammar is this.

3. Give a grammar that describes decimal integers in place notation.
4. Give a grammar for Roman numerals.
5. Give a grammar that describes the behaviour of an automatic coffee machine.

Chapter 2

Lexical analysis

Lexical analysis is the process of recognizing the elementary words or lexemes of a programming language. Suppose we wish to recognise the reserved words **begin** or **while** or **end**.

A file which contained one or more of these words could be produced by a grammar of the form:

```
.*(begin| while|end).*
where
.→ any character
* means an arbitrary number of repetitions
| means alternation
```

This type of grammar which contains no recursive definitions corresponds to the class 3 grammars described in the previous chapter, is also termed a *regular* grammar. It is known that to each regular grammar there corresponds a finite state machine that can act as a recognizer for that grammar. The grammar above would produce the set of strings recognised by the finite state machine shown in Figure 2.1. In this diagram each state of the machine has a number and is drawn as a circle. There are lines (or arcs as they are called in graph theory) going between the states. The arcs are labeled with letters. If we start out in state 0 and get a **b** then we go to state 1.

The letters **e g i** will then take us through states 2 to 4. The final n puts us into the hit state indicating that we have found one of the words. If at any state we get the wrong letter the machine goes back to state 0.

This class of finite state machines can be represented in computer memory in a number of different ways. One way to do it would be with the pascal record types shown in the listing below. To use this datastructure you would need a pointer to the present state, then for each character in the file you were searching you would run down the list of transitions to see if any applied. If

Figure 2.1: A state machine recognizer

they did you would take the transition, otherwise you would jump back tot he start state.

```
type
   transition = record
      letter :char ;
      nextstate : ^ state;
      other_transitions: ^transition;
   end;


   state= record
      hit: boolean;
      transitions:^ transition;
   end;
```

2.1 Tabular FSM representation

A simpler datastructure to use would be a two dimensional array indexed by the current state and the incoming character. The recognition can then be performed by a very simple fast algorithm. Given:

- a state transition table $T[S, C]$ indexed on a set of states
- $S = \{0, P, h\}$
  where
- 0 is the start state,
  - h the hit state
  - P the
    parse states
- a character set C
- a file of characters F with the operator
  $next(F) \rightarrow C$ that returns the next character in a file
- a current state s

then the table T may be interpreted as a finite state recognizer by the algorithm shown in figure 2.2.

This algorithm provides the bare bones of a lexical analyzer. We can envisage coding it up in pascal and getting something like listing **??**.

```
1. $s \leftarrow 0$
2. $s \leftarrow T[s, next(F)]$
3. if $s \in \{0, P\}$ goto 2
4. stop, the pattern has been recognised
```

Figure 2.2: Table driven finite state recognition algorithm

We assume that the finite state automaton algorithm will be applied to a program held in a buffer. A function *scan* can be applied to this buffer and will return whether it found a lexeme. In a var parameter *found* the compiler returns the second last state. In this simple algorithm we will assume that this state is sufficient to distinguish the lexemes that are being looked for.

Algorithm 2

```
type buffer = array[0..maxbuf] of char;
var buf:buffer;
function scan(var found:state;var hitpos:integer):boolean;

 this returns true if a hit is found in the buffer

var i:integer;S:state;
label 1,2;
begin
  for i:=hitpos to maxbuf do begin
    S:=table[s,buffer[i]];
    if S = hitstate then goto 1;
    found:=S;
  end;
  scan:=false;
  goto 2;
  1:
  scan:=true; hitpos:=i;
  2:
end;
```

2.2 Character Classes

In practice the task of a lexical analyzer is more complicated than this.

The lexemes of the language are not just made up of reserved words. There are variables and strings, comments and numbers to contend with. Although

Figure 2.3: Two stage Lexical analysis

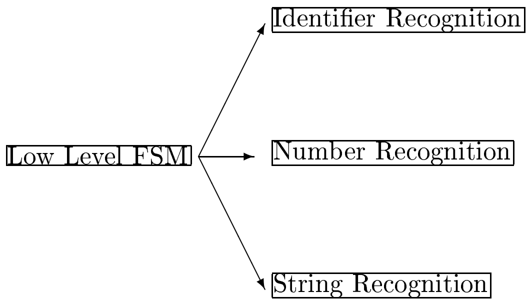

Figure 2.4: Distinguishing identifiers from numbers

the reserved words make up a small finite set, all possible variables of a language make up a very large set indeed. For languages which allow arbitrary length variables, the set is infinite. There can be no question of setting up a transition table that would be capable of recognizing all the possible identifiers, all the possible distinct numbers and all the possible distinct strings.

We are better to handle the process in two stages. First a simple finite automaton which tells us: 'we have a number' or 'we have an identifier'. Second, come other automatons that are invoked by the first to distinguish between different numbers or different identifiers, strings etc. This is the strategy used in the Compiler Writer's Toolbox. Lexical analysis is split into two levels. First level lexical analysis splits the source program into broad categories of lexemes like identifiers, numbers and strings. Second level lexical analysis then distinguishes between different identifiers etc.

Consider the problem of recognizing Pascal identifiers and numbers. We might define them as follows:

number → *digit digit**
identifier → *letter alphanumeric**
alphanumeric → *letter*
alphanumeric → *digit*
digit → [**0-9**]
letter →[**a-zA-Z**]

By [**0-9**] we mean any one of the set of characters from **0** to **9**. This is termed a character class. Using these character classes we can define state machines that will recognise either an identifier or an number.

The graph in figure 2.4 has far fewer states than the one in figure 2.1, although that one could only recognise 4 reserved words and this one will recognise all numbers or identifiers. It is an indication of the simplification that can arise from just dealing with classes of characters and classes of lexemes. Note that the reserved words recognised by the first state machine begin, while, end will all be categorized as identifiers by the state machine in figure 5.3. This does not matter. We can distinguish between reserved words and other identifiers with second level lexical analysis.

The Compiler Writer's Toolbox is supposed to be easily configurable to handle different languages. An attempt has been made to isolate language dependent parts and make them readily alterable. The first level lexical analyzer has been made very general. It splits the input into numbers, strings, comments and lexemes. Lexemes are interpreted very liberally.

Anything which is not a string, number or comment is a lexeme. This means that things like the brackets [ ] { } ( ) or the operators + - = are treated as lexemes, as well as the more obvious alphanumeric strings. Although this may seem strange, you should realise that some languages allow sequences of operator symbols to be strung together as in: <= , ++, +=, % =. The first level analyzer sees all of these as lexemes whose meaning will be sorted out later. All it is concerned with is deciding where a lexeme starts and finishes. Consider this example:

```
x1:=ab<=c;
```

Where do the individual lexemes start and finish here?

If we write it down with spaces between the lexemes it is obvious:

```
x1 := ab <= c ;
```

but the lexical analyzer can not rely upon the programmer putting it down in this way. Without knowing what all the operators in the language are it needs to be able break the string up. This is where character classes come in handy. Using a recognizer like that in figure 5.3 you can pick out the identifiers, but that still leaves a variety of partitionings that are possible.

```
x1 : = ab < = c ;

x1 := ab < = c ;

x1 : = ab <= c ;

x1 := ab <= c ;
```

We can pick out the last one as the right one, because our experience of programming languages leads us to treat := and >= as single lexemes. We know that a string of operator symbols going together should be interpreted as a single lexeme.

We can add to our grammar the rules:
operator →opsym opsym *
opsym → [+ * - % : = /]

Exactly which symbols go to make up the class of operator symbols will vary from language to language, but most languages do have a recognizable class of characters used in operators.

There is another class of symbols which have to be treated distinctly: brackets. We want ((to be two lexemes not one composite lexeme. We have certain broad classes of characters that are likely to be used in many languages : digits, letters, operators, brackets, spaces. The exact definition of these will vary from language to language. In some languages the underbar symbol ' _ ' can be part of an identifier, in others it is an operator.

2.2.1 The primitive state machine

The first level lexical analysis is performed by the unit FSM.PAS. The finite state machine is accessed via the function *newlexeme* which returns an $fsmstate$. Each time this function is called the finite state machine processes a new lexeme and terminates in a state which indicates what class of lexeme has been found. The finite state machine program is encoded as a transition table. This is a two dimensional array indexed on the current state and the current character. In association with the transition table is another array: the action table. Yielding one of (*last*, *skip*, *include*) the action table is accessed in step with the transition table. It provides instructions as to how to build up the lexeme. The finite state machine controls the actions of two pointers into a text buffer. One points at the start of a lexeme, the other, called $finish$ points at the current character. As each character is processed $finish$ pointer is advanced.

- If the action specified is *skip*, the start pointer is advanced to the current character.
- If the action specified is *include*, the start pointer remains where it is.
- If the action specified is *emit*, the machine stops and returns the pointers plus its current state to the second level lexical analyser.

As a subsidiary function, the finite state machine keeps a count of the current line that has been reached in the program. To configure the analyser for a different language, you would alter the mckctab program to produce a new set of character class definitions, and make any necessary changes to the transition and action tables.

2.2.2 Class table

The FSM uses two enumerated types for its operation. $Fsmstate$ lists the states that the machine can be in, whilst *charclass* specifies the classes into which the

Algorithm 3 Finite state transducer module FSM

```
{ -----------------------------------------------------------------
Module      : FSM.cmp
Used in     : Compiler toolbox
Author      : W P Cockshott
Date        : 3 Oct 1986
Version     : 1
Function    : Finite State Transducer to perform first level lexical
              analysis.
              Splits up text in a buffer into lexemes
Copyright (C) WP Cockshott & P Balch
----------------------------------------------------------------
}
UNIT fsm;
INTERFACE  USES
  editdecl;
‘sec 2.3‘Interface to FSM‘
IMPLEMENTATION
var
       include_sp :integer ;
          NEWSTATE:fsmstate;
       buffstack : array[1..includedepth] of textbuffer;

procedure rewind;
‘sec 2.3.2‘Rewind‘ ;
function the_line:integer;
begin
  the_line:=the_buffer.linenumber;
end;
function push_buffer:boolean;
 ‘sec 2.3.3‘Push buffer‘;
function pop_buffer:boolean;
 ‘sec 2.3.4‘Pop buffer‘;
function newlexeme(var B:textbuffer):fsmstate;
 ‘sec 2.2.6‘Transition Table‘ ;
         label 1,99;
         var
            S:fsmstate;
            C:charclass;
            A:action;
            T:textbuffer absolute the_buffer;
            I:integer;
         begin
             1:
               t.start:=t.finish;
              if listprog then
              { put the condition outside the loop to prevent things
                being slowed down too much }
                ‘sec2.2.7‘Recognise and print‘
              else
                ‘sec2.2.8‘Recognise‘ ;
              99:
              newlexeme:=S;
         end;
  var i:integer;
```

Table 2.1: char class table

^@	whitespace,	^ A	whitespace,	^ B	whitespace,	^C	whitespace,
^ D	whitespace,	^ E	whitespace,	^ F	whitespace,	^ G	whitespace,
^ H	whitespace,	^ I	whitespace,	^ J	whitespace,	^ K	whitespace,
^ L	whitespace,	^ M	separator,	^N	whitespace,	^ O	whitespace,
^ P	whitespace,	^} Q	whitespace,	^ R	whitespace,	^ S	whitespace,
^ T	whitespace,	^ U	whitespace,	^ V	whitespace,	^ W	whitespace,
^ X	whitespace,	^ Y	whitespace,	^ Z	whitespace,	^ [	whitespace,
^\	whitespace,	^]	whitespace,	^^	whitespace,	^ _	whitespace,
	whitespace,	!	shriek,	"	dquote,	#	whitespace,
$	operator,	%	operator,	&	operator,	’	quote,
(	bracket,	)	bracket,	*	bracket,	+	operator,
,	bracket,	-	operator,	.	digits,	/	operator,
0	digits,	1	digits,	2	digits,	3	digits,
4	digits,	5	digits,	6	digits,	7	digits,
8	digits,	9	digits,	:	operator,	;	separator,
<	operator,	=	operator,	>	operator,	?	operator,
@	operator,	A	alpha,	B	alpha,	C	alpha,
D	alpha,	E	exponent,	F	alpha,	G	alpha,
H	alpha,	I	alpha,	J	alpha,	K	alpha,
L	alpha,	M	alpha,	N	alpha,	O	alpha,
P	alpha,	Q	alpha,	R	alpha,	S	alpha,
T	alpha,	U	alpha,	V	alpha,	W	alpha,
X	alpha,	Y	alpha,	Z	alpha,	[	bracket,
\	whitespace,	]	bracket,	^	operator,	_	operator,
‘	operator,	a	alpha,	b	alpha,	c	alpha,
d	alpha,	e	exponent,	f	alpha,	g	alpha,
h	alpha,	i	alpha,	j	alpha,	k	alpha,
l	alpha,	m	alpha,	n	alpha,	o	alpha,
p	alpha,	q	alpha,	r	alpha,	s	alpha,
t	alpha,	u	alpha,	v	alpha,	w	alpha,
x	alpha,	y	alpha,	z	alpha,	{	bracket,
|	operator,	}	bracket,	~	operator,	del	whitespace

Figure 2.5: Use of the class table to economise on the finite state machine

```
type
  fsmstate =(startstate,opstate,idstate,numstate,
     expstate,commentstate,stringstate,escapestate,
     lastquotestate,sepstate,brackstate
    );

charclass=(operator,bracket,alpha,digits,exponent,dquote,
     quote,shriek,separator,whitespace
    )
Const classtab:array[0..127] of charclass = (  see table 2.1 );
```

character set can be mapped. Between them, these types will act as indices into the state transition table for the low level Finite State Machine.

2.2.3 McCtab program

The Compiler Writer's Toolbox encodes the classes to which individualcharacters belong in the file { CLASSTAB.CMP}. This includes a definition of the type *charclass*, and an array *classtab* which maps from characters to this type. The definitions of this type and the array are given in table 2.1.

The file CLASSTAB.CMP is produced automatically by a program called *MckCtab*.

This is an example of the use of a program generator program: a program which produces another program. these are very useful tools when writing a compiler. In *MckCtab* the classes can be specified as sets of char, the program then generates the source declaration for an appropriate character class table.

2.2.4 Output Classes and Actions

The program outputs the type definitions that are to be used in the finite state machine module of the compiler. There are two types representing the state of the finite state machine and the character classes that are jointly used to index the state transition table.

An example of the output generated from listing 5 can be seen in section **??**.

2.2.5 Assign Characters to Classes

The program writes out the contents of a constant array that maps characters to their classes. These are written 4 entries to a line separated by commas. A somewhat prettified version of the output from this section was shown in section **??**.

Each entry is made up of a comment followed by the value of the entry.

Algorithm 4 mckctab

```
{ ----------------------------------------------------------------

Program      :Mkctab
Used in     :Compiler~Writer's Toolbox
Author      :W~P~Cockshott
Date        :3~Oct~1986
Version     :1
Function    :to build a class table include file for the
lexical analyser
Copyright (C)~WP~Cockshott & P Balch

}

‘sec 2.2.5.3‘Define Character Sets‘
var~c:char;
begin

‘sec 2.2.4 on the preceding page ‘Output Classes and Actions‘
   writeln('const classtab:array{[}0..127{]} of charclass=(');
for~c:=chr(0) to chr(127) do begin
‘sec2.2.5 ‘Assign Characters to Classes‘
end;
writeln('); {~end~of~classtab~}');~
end.
```

Algorithm 5

```
     writeln('type fsmstate =(startstate,opstate,idstate,numstate,');
       writeln('          expstate,commentstate,stringstate,escapestate,');
       writeln('          lastquotestate,sepstate,brackstate);');
       writeln(' charclass=(operator,bracket,alpha,digits,exponent,dquote,');
       writeln('           quote,shriek,separator,whitespace);')
```

Algorithm 6 assign characters to classes

```
     if (ord(c) mod 4) = 0 then writeln;
          ‘section 2.2.5.1‘Make Comment‘
          ‘section 2.2.5.2‘Output Class‘
```

2.2.5.1 Make comment

The comments for printable characters are the characters themselves. If they are non printable (DEL or a control character) they are output in mnemonic form.

For a control code the menmonic is of the form ˆfollowed by the printable character that is 64 greater than the control code.

Thus the bell character (07) will appear in the comment as Ĝ .

The character ’}’ has to be handled as a special case so as not to prematurely end the comment.

Algorithm 7 make comment

```
    write(’ { ’);
                if c < chr(32) then write(’^’,chr(64+ord(c)),’} ’)
                else if c=’}’ then write(’closing bracket}’)
                else if c=chr(127) then write(’del}’)
                else write(c,’} ’);
```

2.2.5.2 Output class

Certain characters have to be treated as special cases.

The letter E can occur either as part of an identifier or as the label for the exponent part of a floating point number. For this reason it is convenient to assign it to a special class EXPONENT. The newline character (ASCII 13) is assigned to the class of separators, this is done explicityly because it can not be given as a Pascal character literal in the definition of the set SEPSY.

Quotes and double quotes are single member character classes and so are more efficiently dealt with by simple equality tests here.

Algorithm 8 Output class

```
    if c in opsy then write(’operator,’) else
    if c in [’E’,’e’] then write(’exponent,’) else
    if c in alpha then write(’alpha,’) else
    if c in digits then write(’digits,’) else
    if c in bracket then write(’bracket,’) else
    if c = ’!’ then write (’shriek,’) else
    if (c=chr(13)) or (c in sepsy) then write(’separator,’) else
    if c = ’”’ then write(’quote,’) else
    if c = ’"’ then write(’dquote,’) else
    begin
    write (’whitespace’);
    if ord(c)<>127 then write(’,’) else writeln;
    end;
```

Exercises 1. Define the character classes that would be appropriate for first level

lexical analysis of Pascal.

2. Draw a state transition diagram based upon the listing of file fsm.pas that describes the behavior of the first level lexical analyser.
3. Devise a regular grammar to describe Pascal floating point numbers.
4. Modify the files Mckctab.pas and fsm.pas to construct a first level

2.2.5.3 Define Charactersets

The characters are divided into several subsets depending upon where they occur in the source language. All are declared using the structured constant facility of Turbo Pascal.

Algorithm 9 define charactersets

```
const
sepsy:set of char = [';'];
‘sec 2.2.5.4‘Opsys‘ ;
‘sec 2.2.5.5‘Brackets‘;
‘sec2.2.5.6‘Alphanumerics‘
```

2.2.5.4 Opsys

These characters can be put together to form an operator. That is to say if we have the characters '+' and '=' occuring next to one another they are to be treated as the composite operator '+='. An operator in the source language must be made up entirely of these operator characters.

Algorithm 10 Opsys

```
const opsys:set of char=[':',  '-', '=', '+', '@','&',
                         '^','%','‘',  '_','$',
                         '?', '/', '>', '<','~','|'
                        ]
```

2.2.5.5 Brackets

These characters include the obvious bracket characters like '[' or ']' and also, less obviously ',' and '*'. These are thrown in because all of the characters in the bracket set stand as single lexemes. They do not form part of a larger lexeme.

A '*' put next to another '*' must not be read as '**' nor must a pair of '('s be read as '(('.

Algorithm 11 Brackets

```
const  brakets:set of char = [ ')', '(', '{', '}', ']', '[','*',', ]
```

2.2.5.6 Alphanumerics

Alpha numeric characters can occur in two contexts: in numbers, and in identifiers. They are split into subsets according to whether they are staring characters of identifiers and of numbers. Note that the character '.' can occur in either an identifier or a (real) number.

Algorithm 12 Alphanumerics

```
const
alpha:set of char = ['a'..'z','A'..'Z'];
digits:set of char = ['0'..'9','.'];
alphanum:set of char = ['a'..'z','A'..'Z','.','0'..'9','#'];
spacechars:set of char =[' ']
```

2.2.6 Transition table

This table which is stored as a two dimensional array encodes the behaviour of the low level finite state machine for PS-algol lexical analysis. The behaviour of the machine is indicated in the state transition diagram shown in fig **??**.

2.2.7 Recognise and print

This is the basic loop that recognises a lexeme. Between invocations the state of the FSM is stored in NEWSTATE.

2.2.8 Recognise

This is the recognition loop when printing is disabled. Two variants of the loop are provided to prevent a test for printing being enabled within the inner loop.

2.3 Interface to FSM

These are the functions and variables exported from the finite state machine module to the second level lexical analyser. If Turbo Pascal allowed a more modular structure one would like the declaration of the type of the Textbuffer to be hidden.

This can not be done under Turbo Pascal V4.0.

const transtable:array [fsmstate,charclass] of fsmstate =

state	operator	bracket	alpha	digits	exponen
	”	’	!	;	whitespace
startstate	opstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
opstate	opstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
idstate	opstate	brackstate	idstate	idstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
numstate	opstate	brackstate	idstate	numstate	expstate
	stringstate	startstate	commentstate	sepstate	startstate,
expstate	numstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
commentstate	commentstate	commentstate	commentstate	commentstate	commentstate
	commentstate	commentstate	commentstate	sepstate	commentstate,
stringstate	stringstate	stringstate	stringstate	stringstate	stringstate
	lastquotestate	escapestate	stringstate	stringstate	stringstate,
escapestate	stringstate	stringstate	stringstate	stringstate	stringstate
	stringstate	stringstate	stringstate	stringstate	stringstate,
lastquotestate	opstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
sepstate	opstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate,
brackstate	opstate	brackstate	idstate	numstate	idstate
	stringstate	startstate	commentstate	sepstate	startstate)

);

type action =(last,skip,include);

const emit:array [fsmstate,charclass] of action = (

state	operator	bracket	alpha	digits	exponen
	”	’	!	;	whitespace
startstate	skip	skip	skip	skip	skip
	skip	skip	skip	skip	skip,
opstate	include	last	last	last	last
	last	last	last	last	last,
idstate	last	last	include	include	include
	last	last	last	last	last,
numstate	last	last	last	include	include
	last	last	last	last	last,
expstate	include	last	last	include	last
	last	last	last	last	last,
commentstate	skip	skip	skip	skip	skip
	skip	skip	skip	last	skip,
stringstate	include	include	include	include	include
	include	include	include	include	include,
escapestate	include	include	include	include	include
	include	include	include	include	include,
lastquotestate	last	last	last	last	last
	last	last	last	last	last,
sepstate	last	last	last	last	last
	last	last	last	last	last,
brackstate	last	last	last	last	last

Algorithm 13 recognise and print

```
repeat
  S:=NEWSTATE;
  ‘sec 2.3.0.3‘Get the next character‘
  write(listfile,chr(i));
  if I= 10 then begin
   t.linenumber:=t.linenumber+1;
  end ;
  ‘sec 2.3.0.1‘Compute new state‘
  if A= skip then t.start:=t.finish ;  mark start of lexeme
  ‘sec 2.3.0.2‘Handle end of buffer‘
until( A=last)
```

Algorithm 14 recognise

```
repeat
 S:=NEWSTATE;
 ‘sec2.3.0.3‘Get the next character‘
 if I= 10 then begin
  t.linenumber:=t.linenumber+1;
 end ;
 ‘sec 2.3.0.1‘Compute new state‘
 if A= skip then t.start:=t.finish ;  mark start of lexeme
 ‘sec 2.3.0.2‘Handle end of buffer‘
until( A=last)
```

Algorithm 15 interface to FSM

```
const
      includedepth=2;
 type
         'sec 2.3.1 'textbuffer'
         string80= string[80];
 var
       the_buffer:textbuffer;
       stopline  :integer;
 {------------------------------------------------------------ }
 {   THE_LINE                                                   }
 {   return the current line number                            }
 {------------------------------------------------------------ }
 function the_line:integer;
 {------------------------------------------------------------ }
 {    REWIND                                                    }
 {    moves the finite state recogniser back to the start      }
 {    of the text buffer                                        }
 {------------------------------------------------------------ }
 procedure rewind;
 {------------------------------------------------------------ }
 {      Push and Pop buffers                                    }
 {      this is used to implement include files                }
 {      push_buffer saves old text buffer                       }
 {      pop_buffer restores old text buffer                    }
 {-------------------------------------------------------------}
 function push_buffer:boolean;
 function pop_buffer:boolean;
 {------------------------------------------------------------ }
 {  NEWLEXEME                                                  }
 {  skips start and finish to point at new lexeme              }
 {  returns type of lexeme                                      }
 { ----------------------------------------------------------- }
 function newlexeme(var B:textbuffer):fsmstate;
```

2.3.0.1 Compute new state

This uses the class table to determine the class of the input character and then, using 2d array indexing determines the newstate and the action to perform given the current state.

Algorithm 16 compute new state

```
                 C:=classtab[I and 127];
                 NEWSTATE:= transtable[S,C];
                 A:=emit[S,C];
```

2.3.0.2 Handle end of buffer

Check of the end of the lexeme goes pas the end of the buffer. If it does then terminate the lexeme and pop the file buffer.

Algorithm 17 Handle end of buffer

```
     if t.finish >= t.textlen then begin
                    { HANDLE RUNNING OUT OF THE BUFFER }
                        if pop_buffer then goto 1 ;
                        a:=last;
                     end;
```

2.3.0.3 Get next character

Increment the finish pointer for the lexeme and then fetch the character at this position in the text buffer as an integer.

Algorithm 18 get next character

```
     t.finish := t.finish+1;
     I:=ord(T.thetext^[t.finish] );
```

2.3.1 TEXTBUFFER

Program text is represented to the lexical analyser by the type TEXTBUFFER. This is a record which has the form shown in figure 2.3.1, and is declared in listing 19.

2.3.2 Rewind

When a program is to be recompiled for whatever reason there needs to be the facility for the compiler to move its compilation point back to the start of

textbuffer		
thetext	pCharSeq	Pointer to an array of char
start	word	first character of lexeme
finish	word	character after the lexeme
textlen	word	number of chars in the buffer
linenumber	integer	the line number we are at

Algorithm 19 textbuffer

```
type textbuffer = record
              thetext: pCharSeq;
              start,finish,textlen: word;
              linenumber :integer;
      {point at the start and finish of lexeme and give length of buffer }
             end;
```

the buffer. This sequence reintialises the low level finite state machine. The constant *textbuflo* is declared in the editor package.

The variable *include_sp* is used to keep track of the level of nesting of 'include' files in the compilation process.

Algorithm 20 rewind

```
begin
     NEWSTATE:=startstate;
     the_buffer.start:=textbuflo;
     the_buffer.finish:=textbuflo;
     include_sp:=0;
end
```

2.3.3 Push buffer

This operation pushes the current text buffer onto a stack and advances the *include_sp*. This will be done whenever an include file is encountered and our position in a previous buffer has to be saved.

2.3.4 Pop buffer

This is the obverse operation to pushing a buffer. It is performed whenever an include file ends. The space occupied by the buffer can then be freed and the finite state machine switched back to obtaining its input from the old buffer.

Algorithm 21 push buffer

```
begin
 if include_sp<includedepth then begin
  push_buffer:=true;
  include_sp:=include_sp+1;
  buffstack[include_sp]:=the_buffer;
 end else push_buffer:=false;
end
```

Algorithm 22 pop buffer

```
begin
     if include_sp>0 then begin
        pop_buffer:=true;
        with the_buffer do freemem(thetext,textlen);
        the_buffer:=buffstack[include_sp];
        include_sp:=include_sp-1;
     end else pop_buffer:=false;
 end
```

2.4 Identifier management

The first level of lexical analysis picks out the start and finish of lexemes. At this point the lexemes are sequences of characters. This is not a convenient form to deal with computationally. A common operation that has to be performed in the higher levels of a compiler is to compare two lexemes to see if they match. String comparison is slow. A representation more suited to fast manipulation is needed. Computers carry out atomic operations on words of binary data. The fastest type of comparison is between two binary words. This can generally be done in one or two instructions. For speed, the representation of lexemes should be changed from strings into words. Can this be done?

Can strings of perhaps a dozen characters be represented in a 32 bit or 16 bit word?

It all depends upon how many of them there are. An 8 character string contains 64 bits. It is only possible to represent this in a 16 bit word if the string actually contains less than 64 bits of information. With a 64 bit binary number 2^{64} different values can be represented. There are 2^{64} possible 8 character strings. If longer strings are considered the number of possible strings will rise exponentially in the length of the string. A 16 bit word can only encode 2^{16} values. It seems impossible to cram into this all the possible strings.

In principle a program could be written that would use the billions of billions of different identifiers that are theoretically permitted in programming languages. In practice, in our finite world such a program will never occur. All we have to worry about is how many distinct identifiers will actually occur in a program. This number is unlikely to rise above a few hundreds, so a 16 bit word can easily range over them. One simply has to assign ascending integers to the lexemes in the order in which they are encountered. The second level lexical analyzer can be seen as a black box that takes in strings, numbers them and then outputs the numbers.

strings → 2nd level lexical analysis → numbers

In functional notation we can characterize it as a function

$identify(string \rightarrow number)$

such that

$identify(A) = identify(B)$ iff $A = B$

In words this means that the mapping from strings into numbers preserves string equality. If we knew beforehand what strings were to be encountered this would be trivial. We need only construct a fixed table with two columns, strings in one numbers in the other. When we needed to perform a conversion we would scan the first column for a match and output the corresponding number. For the reserved words of a language this is feasible and it is indeed the technique used by some compilers. A fixed table is set up containing the reserved words, which is searched for a match each time a lexeme is produced. If you do this however, you end up with the reserved words of the language embedded in your compiler. This may be a disadvantage when you have to convert the compiler to handle a new language. Furthermore, a compiler will always have to handle additional symbols over and above the reserved words. It has to handle the user defined names of variables. These can not be known before hand, so a compiler is forced to have an extendible data structure - often called the symbol table - to deal with these. An alternative approach is to deal with all lexemes other than literal values (integers, reals etc) in a homogeneous way. The lexical analyzer of the Compiler Toolbox contains no built in knowledge of what identifiers are used in the language. It works entirely with dynamic data structures. The simplest way to build a mapping from strings to numbers would be something like the algorithm shown in program $SIMTRANS$.

This algorithm is simple and reliable but suffers from two serious disadvantages. The first is excessive space consumption. The second is slowness.

In order to be able to handle the longest identifier that you are going to encounter the constant *idmaxlen* is going to have to be set to something like 40 or 50. The type *idrange* may have to go up to 1000 to handle the number of identifiers that might occur in a big program. Already you have reserved 40 to 50 kilobytes of store. A compiler will need several other tables. Is it wise to use up so much space that you may not need? Most identifiers will be closer to 8 characters than 40 characters in length. Perhaps 80 per cent of the table will be wasted. This kind of pressure tempts compiler writers to restrict the length of identifiers that the compiler will accept. When computers had less memory, the temptation to only allow short identifiers was severe. Fortran had only 6 characters in identifiers. Early versions of C ignored all but the first 8 characters of a name. This kind of thing is no longer acceptable. Some sort of dynamic data structure should be chosen for an identifier table that does not restrict how long the identifiers can be.

The simple table used in *SIMTRANS* will be slow to search. On average you will have to scan through half the table to find an identifier. This can mean hundreds of string comparisons for each symbol processed. In order to prevent identifier translation becoming a bottleneck it should be made as fast as possible.

Algorithm 23 program SIMTRANS

```
program simtrans;
 type idrange=1..idmax;
     idstring=string[idmaxlen];
 var idpntr:idrange;
      idtab:array [idrange] of idstring;
 procedure pushid(var id:idstring);
 begin
  idtab[idpntr]:=id;
  idpntr:=succ(idpntr);
 end;
 function identify(var id:idstring):idrange;
 var i:idrange;
 label found;
 begin
  for i:=1 to pred(idpntr) do if id= idtab[i] then
  begin
   identify:=i; goto found
  end;
  pushid(id);
  identify:=identify(id);
  found:
 end;
 begin
  idpntr:=1;
 { rest of program goes here }
 end.
```

2.5 Tries

There are a variety of dynamic data structures that could be used based upon trees or hashing. What the Compiler Toolbox uses is a Trie. This is a special sort of tree that makes use of the observation that many of the names used in a program will be similar to one another. Consider the user defined identifiers in what follows.

```
idrange
```

Figure 2.6: Identifiers organised as a trie

```
founD
IDentifY
  maXleN
  pntR
  rangE
  strinG
  taB
pushiD
```

Figure 2.7: the trie data structure

```
idmax
idstring
idmaxlen
idpntr
idtab
pushid
id
identify
i
found
```

There is a great deal of commonality here. A trie takes advantage of the fact that many identifiers share common prefixes with other identifiers. We can reorganise the identifiers to show this in figure 2.6.

In this arrangement the prefixes are only written down once. The last letter of an identifier is written in capitals. Although this arrangement is more compact, it contains all of the names. A trie achieves this arrangement in computer memory using a linked list.

2.5.1 Trie data structure

A possible record format for a trie is shown in listing24 .

The type lexnode points at a delabrandis record. This has a character in it and two further lexnodes. The follower pointer corresponds to moving horizontally in figure 2.6, the alternates pointer corresponds to moving vertically. The number final will be non zero if this is the last letter in a word, corresponding

Figure 2.8: Use of index to speed up trie access

to the use of bold characters in figure 2.6. The identifiers are further indexed on their first character using the type lexarray.

Algorithm 24

```
type
    lexnode = ^delabrandis;
    lexarray = array[minchar..maxchar] of lexnode;
    delabrandis = record
                     final:integer;
                     pref:char;
                     follower,alternates:lexnode;
              end;
```

2.5.2 Trie insertion

The dynamic trie structure places no limits on the the number of characters that can be in an identifier. Despite this, no unnecessary space is allocated for short identifiers. Very short, one or two letter identifiers are likely to fall entirely within other longer ones, and effectively use no space. Search is fast. Using the index the algorithm restricts its search space to just those identifiers that start with the right letter. With each further letter matched, the search space is restricted.

The trie is manipulated by the function *insert_token* which updates the trie as a side effect of searching it for an identifier. Because the trie is a recursive datatype the actual insertion is done by a recursive procedure *ins*. The function maintains the alternates in alphabetical order. The presence of an order reduces the number of unnecessary comparisons that have to be made if you are inserting a new string.

Note how access to the trie is accelerated by using an array n indexed on characters. This means that there is essentially a sub trie for each possible first letter of the identifiers. This produces a cheap speedup in the search performance. If this were not the case the first letter would have to be matched using a linear search.

2.5.2.1 Create a new node

A new node is created on the heap. Its character field is initialised to the current character. Its final field is set to 0 to indicate that this is not the last character on the list. It has no follower or alternates as yet.

```
    begin
```

Algorithm 25 Insert token

```
function insert_token(var B:textbuffer; var n:lexarray):lextoken;
{  inserts the string into the tree }
{$S-}
var p      : lexnode;
    charno : integer;
    c      : char;
    hit ,inserted   :boolean;
    procedure newnode(var next:lexnode;c:char);
```

‘sec 2.5.2.1‘Create a new node‘

```
    procedure ins(var n:lexnode;charno: word );
    var t:lexnode;
        c:char;
```

‘sec 2.5.3‘Recursive Insert‘

```
begin
 {$r-}
 ins(  n[B.thetext^[B.start]], B.start);
end;
```

```
        new(next);
        with next^ do begin
          pref:=c; final:=0;
          follower:=nil;
          alternates:=nil;
        end;
    end;
```

2.5.3 Recursive insert

The recursive insert procedure has to deal with 3 alternatives.

- We are at a leaf node of the trie with a nil pointer.
- We are at a node that matches the current character.
- The current character is lexx than the character in the node.

```
begin
       c:=b.thetext^[charno];
       if charno <B.finish then with B do
          if n=nil then
```

‘sec 2.5.3.1‘Add another letter‘

```
          else with  n ^ do
           if c = pref then begin
                 if charno=finish -1 then
```

‘sec 2.5.4‘Last letter of word‘

```
                 else ins(follower,charno+1);
            end
            else if c< pref then ins(alternates,charno)
            else
```

‘sec 2.5.4.1‘Char less than prefix‘ ;

```
end;
```

2.5.3.1 Add another letter

A new node is attached to the currenly nil node pointer. The insert procedure is invoked recursively to append any further characters in the word to the trie.

```
begin
             newnode(t,c);
             n:=t;
             ins(n,charno)
end
```

2.5.4 Last letter of a word

We are on the last character of the word. Either the word has been encountered before or it has not.

If it has been met previously then a token will have been stored for the word in the *final* field of the node. Alternatively, the word is new and the *final* field indicates this by holding 0. It is then replaced by the value of *lasttoken*, which is itself then incremented. After this the value in *final* must be the appropriate code for the word and can be returned from *insert_token*.

```
begin
                    { a  hit }
                    hit:=true;
                    if final = 0 then
                    { first time we have encountered this word}
                    begin
                       final:=lasttoken;
                       lasttoken:=succ(lasttoken);
                    end;
                    insert_token:=final;
end
```

2.5.4.1 Char less than prefix

A copy of the pointer to the current node is made in *t*. A new node is then attached to the current node pointer. The old node is then made the first alternative for the new node. Then the insert operation is invoked recursively.

```
begin
           t:=n;
           newnode(n,c);
           n^.alternates:=t;
           ins(n,charno);
end
```

2.6 Lexeme Definition

The identifier table is initially loaded with a list of all of the reserved symbols of the language. This includes not only the reserved words, but also the operators and brackets. At startup the procedure *init_lexanal* gets invoked to read these in from a file: `lexemes.def`. Because the insertion procedure assigns ascending integers to the identifiers read in, the lexemes in the file will be given internal integer representations in the order in which they occur in the file. Within this file they are organised as list of symbols one per line.

The entries in the lexeme definition file are put in one to one correspondence with an enumerated data type: lexeme. This type provides the interface between

Figure 2.9: A section of a lexeme definition file.

```
then
to
TRACEON
TRACEOFF
true
upb
vector
while
write
{
}
~
~=
\
..INT.LIT
..REAL.LIT
..STRING.LIT
..IDENTIFIER
```

the lexical analyzer and the syntax analyzer. The context free grammar of the language will be defined in terms of these lexeme values.

2.7 Interface to syntax analyser

FORDOCUMENTATION

The lexical analyzer communicates with the syntax analyzer via a small group of procedures and variables. The most important of these is the procedure *next_symbol*. Each time it is called it processes one symbol. The finite state machine in the level one syntax analyzer is invoked to delimit the symbol. The terminating state of the machine determines whether the symbol was a name, an operator, a number or a string.

If a number is found then a numeric conversion function is invoked to convert the decimal representation of the number into a binary format. In Turbo pascal this is easily achieved using the val function. This takes a string and returns an integer or real equivalent. If your compiler is in another dialect of pascal, it may become necessary to write your own numeric conversion functions.

Next_symbol leaves the results of lexical analysis in a set of global variables:

```
symbol     :lextoken;
lexsymbol  :lexeme;
the_string :stringv;
```

Figure 2.10: Corresponding section of the type lexemes

```
THEN_SY,
TO_SY,
TRACEON_SY,
TRACEOFF_SY,
TRUE_SY,
UPB_SY,
VECTOR_SY,
WHILE_SY,
WRITE_SY,
CUR_SY,
LEY_SY,
TILDE_SY,
NEQ_SY,
SLASH_SY,
INT_LIT,
REAL_LIT,
STRING_LIT,
IDENTIFIER,
        the_real   :real;
        the_integer:integer;
```

Lexsymbol will hold the lexeme that has been matched. If it was an integer, real or string then it will take on the values $INT\_LIT, REAL\_LIT$ or $STRING\_LIT$ and the corresponding value of the literal will be stored in $the\_integer, the\_real$ or the_string. If the lexeme is a reserved word then the corresponding enumerated type value is stored in lexsymbol. If the lexeme is a user defined identifier, then the value of lexsymbol will be $IDENTIFIER$ and the number associated with that identifier will be returned in symbol as a lextoken.

The writer of a syntax analyzer can choose to call *next_symbol* and perform tests on these global variables. In the process of analysis certain actions have to be carried out repeatedly. A common sequence is to test to see if the current lexeme has a particular value, and, if it has to call *next_symbol* to see what follows. This combination of actions is bundled up in :

```
  function have( t: lexeme) : boolean;
```

If the lexical analyzer has t then it says yes and grabs the next lexeme.

```
{ ----------------------------------------------------------------- }
{           HAVE                                                     }
{           conditionally matches a token                           }
{ ----------------------------------------------------------------- }
function have( t: lexeme ) : boolean;
begin
     if t = lexsymbol then
            begin next_symbol(the_buffer);
                 have:=true
            end
     else  have:=false;
end;
```

2.7.1 Mustbe

The stricter version of have is mustbe. This is called when the syntax stipulates the presence of a particular symbol. If the symbol is not found then an error is reported. The error message specifies what symbol was found and what was expected. This can generate such messages as ‘begin found instead of then’.

Note that as presently configured, mustbe will skip over newlines until it comes to a symbol. This is appropriate for most modern languages, but would not be appropriate for parsing assembly language for instance.

```
{ ----------------------------------------------------------------- }
{           MUSTBE                                                   }
{           compulsorily matches a token                             }
{ ----------------------------------------------------------------- }
procedure mustbe(t : lexeme );
begin
     if not have(t) then  begin
     if have(newline_sy) then mustbe(t) else syntax(t);
     end;
end;
```

2.7.2 Syntax errors

```
{ ----------------------------------------------------------------- }
{          SYNTAX                                                    }
{          report error and stop                                     }
{ ----------------------------------------------------------------- }
procedure syntax( t : lexeme);
var m :stringv;
begin
     m:= currentid +' found instead of '+ psym(ord(t));
     ReportError(m);
end;
```

2.7.3 Current Id

The code generator may need to know the current identifier's printable form in order to be able to plant information for a linker. It is thus convenient to have a function that will return this. This avoids the code generator having to know anything about the data format of the text buffer.

```
{ ------------------------------------------------------------ }
{       CURRENTID                                               }
{       returns the identifier as a string                      }
{ ------------------------------------------------------------ }

function currentid :stringv;
    var n:stringv;
        i,p:integer;
    begin
         with the_buffer do begin
              n:='';
             for i:=start+1 to finish do begin
                  n:=n+thetext^[i];
              end;
              currentid:=n;
        end;
    end;
```

TOPICS

2.7.4 Converting lexemes to strings

For error reporting purposes it is often convenient to be able to convert from lexemes back into printable strings.

The technique that you chose for this will depend upon how frequently you want to make the conversions each way. If conversions back into strings are very rare: just when the compiler stops with an error, then it is not necessary for the conversion process to be very fast. In this case one could perform a depth first traversal of the trie looking for a node marked with the current lexeme. When one is found, one knows that this node represents the last character of the identifier. By tracing ones path back through the trie the original identifier can be extracted.

If speed is more important then it may be worth explicitly storing each identifier in ram. This of course brings with it all the problems of space overheads mentioned earlier. If you use a 2 dimensional character array, you have to agree upon the maximum length of character that you can store.

An alternative storage mechanism shown in figure 2.11 uses an array *starts* indexed on the lexemes to find the starting positions of identifiers in a one dimensional character array: *pool*. The lexeme l will then occupy positions

2.7.5 NextSymbol

The lowest level interface to a lexical analyser is provided by a procedure that gets a symbol. We call this procedure $NEXTSYMBOL$. It reads in a lexeme and stores the token in the variable *symbol*.

If the symbol turns out to have been a literal value, for instance a number or a string then the actual value of the literal must be stored for the subsequent stages of the compiler. The value can be stored in one of several global variables:

$the.integer, the.real, the.string$.

In order to do this some processing will be necessary to convert between source character strings and numbers or strings.

```
{ ----------------------------------------------------------------- }
{      NEXTSYMBOL                                                    }
{ ----------------------------------------------------------------- }

procedure next_symbol;
var S:fsmstate;
    function numconv:lexeme;
    var n:stringv;
```

Figure 2.11: Use of a character pool

$pool[starts[l]]..pool[starts[l+1]-1]$

```
        i,p:integer;
        isint:boolean;
    begin
```

‘sec2.7.5.1‘Convert string to number‘

```
    end;

    procedure printsymbol;
    var i:integer;
        c: char;
    begin
         with the_buffer  do
         if lexsymbol <> newline_sy then begin
            for i:=start to finish -1 do write(thetext^[i]);
            write(' ');
         end else   writeln;
    end;

    function stringconv:lexeme;
    var n:stringv;
        i,p:integer;
        escape:boolean;
        c:char;
       procedure append(c:char);
       begin    if length(n)<MAXSTRING then n:=n+c; end;
    begin
```

‘sec 2.7.5.2‘Convert string to internal form‘

```
    end;
```

‘sec **??**‘Type coercion operation‘

```
begin
  coerce.dummy:=0;
  compilercursor:=the_buffer.start;
  S:=newlexeme(the_buffer);
  with coerce do
  if s in [opstate,brackstate,idstate] then
     l1:=insert_token(the_buffer,predefined)
  else if s in[numstate,expstate]  then l2:=numconv
  else if s in [stringstate,lastquotestate] then l2:=stringconv
  else  symbol:=coerce.l1;
  if symbol >maxpredefined then lexsymbol:=identifier
  else lexsymbol:=coerce.l2;
```

‘sec 2.7.6‘Detect run time error location‘

```
end;
```

2.7.5.1 Convert string to number

The string is searched to see if it contains any non-digit characters.

If it does then the flag *isint* will be set. On the basis of this flag the turbo pascal function *val* is called to convert the string either into an integer or a real. Then the appropriate lexeme is returned from numconv.

```
        isint:=true;
        with the_buffer do begin
           n:='';
           for i:=start to finish-1 do begin
              n:=n+thetext^[i];
              isint:=isint and (thetext^[i] in ['0'..'9']);
           end;
           if isint then begin val(n,the_integer,p); numconv:=INT_LITend
                    else begin val(n,the_real,p);  numconv:=REAL_LITend;
       end;
```

2.7.5.2 Convert string to internal form

Some characters that one may want to place in a string have no printable representation. Most of these are format characters like carriage return, tab, or backspace. Some computer languages provide special means of including these into a string by prefixing a printable character with an escape character. When so prefixed, the printable character means something else.

In C the escape prefix is \ and in S-algol it is ' .

The conversion from the printable to the internal form of a string can be performed by a simple finite state machine that has two states:

- Processing normal characters.
- Processing the character after an escape.

In the first case, the characters in the source string are just copied over to the internal form. In the second case, the translation rule appropriate to the language must be applied to get the internal non-printable character.

```
begin
        escape:=false;
        with the_buffer do begin
           n:='';
           for i:=start+1 to finish -2 do begin
              c:=thetext^[i];
              if not escape then begin
```

```
                 escape:=classtab[ord(c) and 127]=quote;
                 if not escape then    append(c);
              end
              else begin
```

‘sec 2.7.5.3‘Convert prefixed characters‘

```
                  escape:=false;
              end;
           end;
        the_string:=n;
        stringconv:=string_lit;
       end;
end;
```

2.7.5.3 Convert prefixed characters

Most languages provide a mechanism for embedding control characters in strings, typically by providing a prefix character and then some code that follows. This example obviously applies to the language S-algol, but other languages would require similar code. S-algol has the following mapping from escape sequnces to non-printable characters:

$'n \rightarrow LineFeed = 10$
$'t \rightarrow Tab = 9$
$'o \rightarrow CarriageReturn = 13$
$'b \rightarrow BackSpace = 8$
$'p \rightarrow VerticalTab = 11$

Anything else preceded by a ’ stands for itself. Hence ” stands for ’ and ’" stands for " .

FORCOMPILER

```
                  case c of
                  'n' : append(chr(NEWLINE));
                  't' : append(chr(TAB));
                  'o' : append(chr(CR));
                  'b' : append(chr(BS));
                  'p' : append(chr(VTAB));
                  else append(c);
                  end ;
```

2.7.5.4 Type coercion operation

We wish to have an enumerated type lexeme for the reserved words of the language. Subsequent identifiers are assigned ordinal values as their lexical tokens that start up where the predefined lexemes finish. What we obtain from our identifier conversion routine is a *lextoken* which is an ordinal type. We have to convert this to a lexeme. There is no built in way of converting an ordinal to an enumerated type in standard pascal, so we cheat by using a variant record. We assign the *lextoken* to field *l*1 and then read it back as a *lexeme* from field *l*2.

```
var coerce:record
        case boolean of
        true:(l1:lextoken);
        false:(l2:lexeme;dummy:byte);
        end
```

2.7.6 Detect run time error location

If an error occurs during the execution of an S-algol program, the line number on which this occured is passed back to the compiler. The program is then searched using the lexical analyser to find the appropriate position in the text where the error happened. The lexical analyser knows it has been called to find an error if the variable *stopline* is non zero.

```
  if stopline >0 then
      if coerce.l2=newline_sy then
             begin
                  if the_line > stopline then
                  begin
                       if  errorfree then
                       error('Run time Error');
                  end
             end;
```

2.8 Exercises

1. Write numeric conversion functions to convert from the decimal representation of integers and reals to binary.

2. Modify these to handle numbers in any base from 2 to 16. Use the notation defined by the grammar:

anybase $\rightarrow$ base # number

base $\rightarrow$ decimalnumber

number $\rightarrow$ anydigit anydigit*

decimalnumber →decimaldigit decimaldigit*

anydigit → [0-9A-F]

decimaldigit → [0-9]

An example in this notation might be
16#01A

3. Alter the string conversion procedure to handle pascal strings. These are deliminated by single quotes.

4. Modify the data type lexeme to correspond to the lexemes needed for pascal. Create a new version of the file lexemes.def called lexemes.pas that contains the pascal lexicon. Test the ability of your reconfigured lexical analyzer to accept pascal input. This may require you to write a driver program that will print the output of the lexical analyzer.

5. How would you make the lexical analyzer insensitive to the differences between upper and lower case keywords and identifiers whilst retaining the cases of letters in strings?

Chapter 3

Syntax analysis

The syntax analysis method used in this course is called recursive descent. It is the standard method of syntax analysis used in block structured languages.

The idea of recursion or self reference is central to this compiling technique. Consider the class 2 grammar described in chapter 1.

G = T, N, S, P
where
T =), (, 1, 2, 3
N = s, t, u
S = s
P =
s ->(t)
t ->1u 2
u ->t
u ->s
u ->3

The non-terminals of this grammar refer to one another: s refers to t which refers to u which in turn refers back to both of them. This grammar is recursive or self-referential. Recursion is a property of those grammars that allow terms to be nested within one another to an arbitrary depth. Because the conventions governing the way that mathematical formulae are written down allow the use of nested brackets, most computer languages too allow this sort of nesting: if only when handling mathematical expressions. In consequence the grammars used in computer languages tend to be recursive. Most high level languages allow recursion in several parts of their grammar. Recursion in a grammar is handled by recursion in the syntax analyzer. The analyzer is a collection of mutually recursive procedures. Associated with each non-terminal of the grammar is a procedure whose job it is to recognise that non-terminal. This is best understood by example.

To parse G we would set up 3 procedures, one to handle each of s, t, u . We know from the definition of G that an s is a t with brackets round it.

A procedure that would accept an s would be:

```
Procedure to accept s
accept a '('
accept a t
accept a ')'
```

Likewise a procedure that would accept a t would be:

```
Procedure to accept t
accept a '1'
accept a u
accept a '2'
```

These procedures are simple because they have no choices in them. The procedure to accept a u has to chose between three alternatives: 3 s t . This can only be done by looking ahead to see what the next symbol is. If the next symbol is a (then we know we have an s , if it is a 1 then we must have a t.

```
Procedure to accept u
If the nextsymbol = '(' then accept an s
else
if the nextsymbol = '1' then accept a t
else
accept a '3'
```

These pseudo english procedures can be implemented in pascal using the procedures provided by the lexical analyzer described in the last chapter. For each non-terminal in the language a pascal procedure is defined that will recognise an instance of that non terminal. The tests on variable symbol and the procedures have and mustbe can be used to determine the flow of control.

```
procedure U;
begin
if symbol=bra then S else
if symbol=one then T else
mustbe(three)
end;
procedure S;
begin
mustbe(bra);T;mustbe(ket)
end;
procedure T;
begin
mustbe(one);U;mustbe(two)
end;
```

The process of going from a context free grammar to a collection of procedures to recognise it is simple and almost mechanical. Given a specification of a the context free part of a grammar, a collection of recursive procedures can usually be quickly written to parse it. For this technique to work though, the grammar must be such that the decision about which procedure to call next can be made by looking ahead a single symbol as is done in U. Those grammars for which this is possible are termed LL(1) grammars. Designers of programming languages tend to try to make their languages LL(1) to simplify parsing. If you are interested in the technicalities of this is done consult a book on formal language theory like Principles of Compiler Design, by Aho and Ullman. Unless you are designing a programming language ab initio this is unlikely to be a problem you will encounter.

3.1 Specification of a machine readable syntax

It is clear that you will not get very far with producing your parser unless you have a clearly specified grammar for your language. There are a number of formalisms for putting a grammar down on paper. Some of these, like the one we have used up to now depend upon the use of special type faces. We have been showing non-terminals in italics and the terminals in bold. Although this reads well enough on the printed page, and can be handled by most word processors, variations in typeface are difficult to express in the standard ASCII codes used on Western computers.

It is convenient at times to allow parsers to be built automatically by other programs which have been given a specification of a grammar. The automatic construction of a compiler in this way is termed compiler compilation, and the programs which do it compiler compilers. These compiler compilers need a clear machine readable specification of the grammar, without italics or boldening. A formalism that meets these criterion is BNF or Backus Naur Format, that was developed to describe algol 60 by Backus and Naur in 1960.

In BNF non-terminals are shown between angle brackets '<' '>' thus: < expression> means the non terminal whose name is expression. Symbols written down outside the brackets stand for terminals. The way that a non terminal expands out is shown by ::= which plays the same role as -> in out earlier notation. Thus in the production

```
<clause>::=if <clause> do <clause>
```

the symbols 'if' and 'do' are terminals. The exception to this are a few punctuation symbols. If a group of symbols is enclosed in square brackets '[' ']', it means that they are optional. For instance in the production

```
<clause>::=repeat <clause> while <clause> [do <clause>]
```

the final pair of symbols

```
do <clause>
```

is optional. The vertical bar symbol '|' is used to designate alternatives as in:

```
<addop>::= +|-
```

The Kleene star * is used to designate repetition, as in:

```
<exp3>::=<exp4>[<addop><exp4>]*
```

which allows zero or more repetitions of an <addop> followed by an <exp4>. Where any of these punctuation symbols occur in the final language, as they are likely to do they are designated by non-terminals:

```
<exp4>::=<exp5>[<multop><exp5>]*
<multop>::=<star>|div|/|rem
<star>::=*
```

3.2 Control Structures

In S-algol the if clause has two variants depending upon whether else is to be used. We have examples like:

```
if b>0 do write "b positive"
```

and we also have ones like

```
write "b is ",if b>0 then "positive" else "negative"
```

The problem of how to parse these two variants will can give us a concrete example of how to use the primitives supplied by a lexical analyzer to perform a context free parse. A cut down version of the parsing procedure for if clauses is reproduced in 26.

Algorithm 26

```
procedure if_clause ;
 label 1,2,3,4;
 begin
     next_symbol;
 1:   clause;
     if have(do_sy) then begin clause end else
 2:   begin
          mustbe(then_sy);
 3:        clause;
          mustbe(else_sy);
 4:        clause;
     end;
 end;
```

The syntax handled by listing 26 is:

```
     if <clause> do <clause> |
     if <clause> then <clause> else <clause>
```

Assume that the procedure has to parse the clause:

```
if b>0 then "positive" else "negative"

^
```

The up arrow indicates where the current symbol is. The procedure if_clause will only be called if some other procedure has performed a look ahead and decided that the next symbol is an 'if'. Since the 'if' has already been recognised, it is discarded by if_clause when it calls next_symbol. This gets us to label 1. The state of the parse is now:

```
if b>0 then "positive" else "negative"

   ^
```

The syntax stipulates that an 'if' must be followed by a clause so the procedure that parses clauses is called. Next a choice has to be made. If the program that is being read in is valid then the next symbol must either be 'do' or it must be 'then'. Which of these it is can be tested using the have call. This tests the current symbol against its parameter. If they match, then the current symbol is 'eaten' and a true result returned. Otherwise false is returned. Since we have a 'then' not a 'do' false will be returned and we go to label 2. The state of the parse is still.

```
if b>0 then "positive" else "negative"

       ^
```

The grammar now offers no alternative. We must have a 'then'. Mustbe is called to make sure that we do and eats up the 'then'. The program has now moved to label 3. The state of the parse is now:

```
if b>0 then "positive" else "negative"

            ^
```

There are no further alternatives. We must have a clause followed by a an 'else'. This brings us to label 4: with the parse in state:

```
if b>0 then "positive" else "negative"

                            ^
```

One more call on clause and the parse is finished. The other form of conditional clause: the case statement, provides some new problems. Its syntax in S-algol is:

```
<caseclause>::=case<clause>of<caselist>default:<clause> <clauselist>::=<clause>[,<clause
```

In pascal it also has a very similar syntax. The case statement involves recursion on the case list and iteration (the Kleene star) on the clauselist. When we transform this syntax into a parsing procedure it is more convenient to use double iteration.

Algorithm 27 The recognizer for case clauses

```
procedure case_clause ;
 begin
   next_symbol;
   clause(t1);
   mustbe(of_sy);
   while errorfree and not have(default_sy) do begin
         repeat clause (t)
         until not have(comma_sy);
         mustbe(colon_sy);
         clause(t);
   end;
   mustbe(colon_sy); clause(t);
 end;
```

The procedure case_clause uses two loops. The while loop recognizes each of the actions of the case clause and can be terminated either by an error in the parsing of lower level code or by the occurrence of the symbol ' default'. Within this we have a repeat loop that recognizes the list of guards in front of each of the actions.

3.3 Expressions

The other types of clauses in the language: for_clause, while_clause etc, can be parsed by applying the same principles exhibited in the case_clause and the if_clause. A more complicated problem is raised in the parsing of expressions. In most programming languages there is the notion of operator priority. If we consider the expression:

2+3*3

the answer should be 11 rather than 8 since the multiplication operator is taken to be of higher priority than addition. What 'of higher priority' means is that we can rewrite the expression as:

2+(3*3)

without changing its meaning. High priority operators implicitly cause their arguments to be bracketed. They allow the programmer to leave out the brackets.

```
* / div rem ++
+ -
is isnt < > >= <=  = =
and
or
Low
```

Table 7.1 Operator priorities

In S-algol the priorities of the operators are shown in table 7.1. High :=

The operator priorities are encoded in the syntax for expressions. Syntactically expressions are defined to be made up of 8 classes of sub expressions generated by 8 non-terminals called: expression, exp1,exp2,..., exp7. The production rule for an expression is

```
<expression>::= <exp1>[or <exp1>]*
```

This defines an expression to be an exp1 followed by an optional list of exp1s separated by 'or'. An exp1 is defined as:

```
<exp1> ::= <exp2>[and <exp2>]*
```

It can be seen that the definition of an exp1 has the effect of bracketing any 'and' operations together before the 'or' operations are recognised. The sequence

```
a and b or c and d
```

is a valid example produced by the grammar. There are in principle various ways of bracketing this:

```
((a and b) or c )and d)
(a and( b or (c and d)))
((a and (b or c)) and d)
(a and ((b or c) and d))
((a and b) or (c and d))
```

Only the last form matches the operator priorities we want, but this is also the only form that can be generated by the grammar. Its derivation can be seen below with the implicit brackets shown underlined.

```
<expression>
(<exp1> or <exp1>)
((<exp2> and <exp2>) or ( <exp2> and <exp2>))
(( a and b) or ( c and d))
```

The rest of the syntax for expressions is shown in table 3.3.

The parsing of expressions provides a particularly direct example of how to translate a BNF into a set of recursive procedures. We take as an example the procedure that recognizes an exp4.

Table 7.2 Expression syntax

```
<exp7>          ::= <name >|
                <stdproc>|
                <literal>|
                        (<clause>)|
                        <cur> <sequence> <ley>|
                         begin <sequence> end |
                        @<clause> of <type1><bra><clauselist<ket>|
                        vector<bounds> of <clause>
 <clauselist>            ::=<clause>[,<clause>]*
 <subscription>  ::=(<clauselist>)[<subscription>]*|
                (<clause><bar><clause>)
 <exp6>          ::=<exp7>[<subscription>]
 <exp5>          ::=<exp6>[:=<exp6>]
 <exp4>          ::=[<addop>]<exp5>[<multop><exp5>]*
 <multop>                ::=<star>|div|rem|/|++
 <star>                  ::=*
 <bar>                  ::=|
 <addop>        ::= +|-
 <exp3>         ::= <exp4>[<addop><exp4>]*
 <exp2>          ::= [~] <exp3> [<relop> <exp3>]
 <relop>         ::= is|isnt|<|>|>=|<=|~=|=
```

Algorithm 28 parsing exp4s

```
{ ------------------------------------------------------------- }

{     EXP4
          this parses the syntax rules
               <exp4>::=[<addop>]<exp5>[<multop><exp5>]*
               <multop>::=<star>|div|rem|/|++
               <addop>::= +|-
 }

{ ------------------------------------------------------------- }
 procedure exp4 (var t:typerec);
 var continue:boolean;   sym:lexeme;
    adop:(plus,minus,noadop);


begin
     debug(' exp4 ');
     adop:=noadop;
     if have(plus_sy) then adop:=plus else
                if have(minus_sy) then adop:=minus;
     exp5(t);
     sym:=lexsymbol;     continue:=true;


     repeat
           sym:=lexsymbol;
           case lexsymbol of
           star_sy        : begin
                                 next_symbol;exp5(t1);
                                 end  ;
           slash_sy       : begin
                                 next_symbol;
                                 exp5(t1);
                                 end   ;
           div_sy,rem_sy  : begin
                                 next_symbol;
                                 exp5(t1);
                                 end    ;
           dplus_sy       : begin
                                 next_symbol;
                                 exp5(t1);
                                 end     ;
           else continue:=false;
           end;
     until not continue ;
     end;
 end;
```

The important technique to notice in Listing 28 is the way a case statement is used to select between which of the operators has been found. This is used for clarity and speed. We could quite readily have used a list of if .. then... else... statements directed by calls to have but this would have been slower and a little less clear. As a general rule, if you are writing a parsing procedure for a rule that has only a couple of alternatives, then you should use calls on have along with an if .. then... else... statement. When you have many alternatives use a case statement. The techniques described in this chapter are sufficient to parse the context free portion of the language. There are some portions of the language for which they are inadequate. Consider for example:

x(1)

This can either mean subscript a vector called 'x' and extract its first element, or it could mean call a procedure called 'x' with parameter one.What it means can only be determined once we know what type of identifier 'x' is. This is context sensitive. Before we can deal with it we must look athow a compiler stores information about types.

Chapter 4

Types and identifiers

4.1 What types are.

A type is a set of values.

Reals are a set of numbers. Integers are a subset of the reals. Booleans are the set true,false. Characters are the set A, B, C,.., a, b, c ..+, -, *, ...

These sets are what are often termed base types in a computer language, they correspond to sets of values that can be represented in machine words of variaous sorts. Most modern computers provide support for floating point numbers, integer numbers, binary values and characters. For this reason, these types are given priority in computer langauges. They are taken to be predefined. For the machine supported base types there is a one to one mapping between the type and a data format

Type	Format
boolean	bit
character	byte
integer	machine word
real	IEEE 64bit floating point number

Some compilers may not use this particular representation of types. It may be convenient to represent reals, booleans and integers all in 32 bits. Whatever the particular convention followed, the idea remains the same. For each type there is a determinate data format.

In addition to the base types, it is usually possible to define new types ina program. These types are constructed using type composition operations.Such operations have a variety of syntactic forms in different languages.When building a general purpose set of compilation tools it is advisable to look at type composition in a language independent fashion. If we do that wecan capture the generality of types rather than just the particular typesthat one finds in a given language.

4.2 Cartesian composition

We are all familiar with the Cartesian co-ordinate system in geometry. This uses ordered pairs of real numbers, conventionally designated [x, y], to represent points on a two dimensional surface. Triples [x, y, z] represent points in 3 space. Quadruples would represent points in 4 space etc. In computer science the notion of Cartesian composition is generalised to then notion of tuple as in: 4tuple, 5tuple. An ntuple is an ordered list of n components. The components may all have the same type, or they may be different. In the case of Cartesian co-ordinates all components are real numbers. In polar co-ordinates, one is an angle the other a real number. Examples of tuples in computer languages are records, structures and procedure parameter lists.

4.3 Unions

A union type is one that may take on several forms. Let A and B be types representing the sets of values a1, a2, a3,.... and b1, b2, b3,....

Then the type C=union(A,B) is the type formed by the union sets of values a1, a2, a3,.... b1, b2, b3,....

4.4 Sets

Given a type T which represents the set of values t1, t2, t3,... then the type powerset of T is the set of all subsets of T. Powerset types are rather misleadingly termed set types in programming languages like Pascal. Powersets are difficult to implement on a computer, so some compilers restrict themselves to implementing powersets of the integers.

4.5 Subranges

Given an ordered type then a subrange type can be defined on it. Given the integers, 1..2 is a subrange of them. Given the characters, 'A'..'C' is a subrange of them.

4.6 Maps

Given the types T and U then the type map(T->U) is the powerset of the ordered pairs [t,u] where t is an element of T and u is an element of U. An individual value of the type map(T->U) will be a set of ordered pairs [t,u] where t is an element of T and u is an element of U.

Maps in the general sense are relations. These are implemented in database programming languages.

Where the maping is such that for each value of t there is only one pair [t, u] and thus a unique value of u, we have a function from T to U. Functions are widely implemented in programming languages. The two main sorts of functions are algorithmic functions and storage functions. Algorithmic functions are implemented by passing parameters to subroutines that return a result. Storage functions are things like arrays where the result is computed by indexing an area of memory.

4.7 Constancy

Some languages distinguish between the type of a value and the type of a store location holding that value. A location that can be updated is a variable. The type of a variable that can hold an integer is distinguished from the type of an integer itself.

4.8 Representing types in a compiler

These concepts of maps, sets, cartesian composition, constancy and subranging are sufficient to represent most of the concrete types that you will come accross in compilation. We need some way to represent them in a compiler. In the compiler toolbox they are represented using pascal records whose types are declared in idtypes.pas. A functional interface is provided to the data structure with functions implementing the type construction operations described in this chapter. Operations are provided on types to determine whether they are equal or not.

We can distinguish 2 types of equality on types. In the first case the types have the same name or have been formed by the same type operations from base types. We call this name equivalence. In the second case types have the same store format, we call this representational equivalence. From the pointof view of semantics we are interested in name equivalence. When it comes to storage allocation and code generation, we will be concerned with representational equivalence. Procedures shown in listing 29 allow these types of equivalence to be checked.

The equivalence of types can be tested using eq and eq2 which return boolean results. In contexts where only a certain type is permited: for instance with procedure parameters, then the operations match,coerce, and balance can be used. These force type equivalence and generate error messages if the types are not equivalent. Match is used for name equivalence and coerce for representational equivalence. Balance is a special form of coerceion used with numbers. In the arithmetic expression

$1 + 2.9$

Algorithm 29 Basic type checking functions in a compiler

```
EQ : compare two types
EQ - name equality
EQ2 - representation equality
-----------------------------------------------------------------
function eq(var t1:typerec; var t2:typerec):boolean;
function eq2(var t1:typerec; var t2:typerec):boolean;
-----------------------------------------------------------------
MATCH
enforce name equality of types
-----------------------------------------------------------------
procedure match(var t1:typerec;var t2:typerec);
var em:textline;
begin
if not eq(t1,t2) then begin
em:= ptype(t1)+' not compatible with ' +ptype(t2);
error(em );
end;
end;
-----------------------------------------------------------------
COERCE - verify representaional equality
-----------------------------------------------------------------
procedure coerce(var t1:typerec;var t2:typerec);
-----------------------------------------------------------------
BALANCE
-----------------------------------------------------------------
procedure balance(var t1:typerec; var t2:typerec);
```

we have a combination of an integer and a real number. Balance is called in arithmetic expressions to ensure that the two operands are of the same type. If they are not, it tries to convert them to the same type by planting instructions to convert numbers between representations.

The type checking procedures can be called by the syntax analyser for two purposes:
a) to disambiguate a construct,
b) to validate a construct.

Diambiguation is needed if the same syntax can imply several different sequences of machine code instructions have to be generated. Consider:

```
x+y
```

This can either be the addition of two integers or the addition of an integer to a real or the addition of two reals. In each case different machine instructions are needed. It the types of all the identifiers have been recorded then the compiler can decide which instructions to use. Validation is required where only a particular type is allowed in a context. A simple example would be an if then else construct where we need a boolean to switch on.

```
if x then .. else ...
```

In this context the variable x must be a boolean. A similar situation arises with procedure parameters. In a strongly typed language, each of the parameters supplied to a procedure call must be chaecked off against the parameters with which the procedure was declared.

4.9 Representing Identifiers

The lexical analyser will have converted all identifiers into an internal representation in the compiler as numbers. The identifier management software must associate type information with these numbers. In the toolbox each identifier has an idrec associated with it. These are declared in the file idtypes.pas. The record gathers together information about the identifier as shown in table 4.1.

A number of different things can be given names in a program: types, variables, fields of records. The variables may be classified according to where in the program they are declared. Global variables are those that are declared at the outermost level of the program and are accessible to all procedures. Local variables are accessible to the procedures in which they are declared. Parameters are an intermediate category, accessible from two places. Suppose I call a procedure:

```
p(1, 4, z)
```

then its parameters must be accessible at the point of call since the compiler must arrange for them to be assigned the values given. From within the procedure the parameters are also available:

Field	Type	Use
name_type	nametype	Says if the identifer is global, local, a parameter a static or a typename
identifier	lextoken	This holds the numeric form of the identifier output by the lexical analyser
offset	integer	This specifies the address of the to some base
block_level	byte	For block structured languages it specifies how far in the variable is in terms of blocks
lex_level	byte	This specifies how far in the variable is in terms of procedures
typeinfo	typeref	A reference to the type of the identifier
zone	(variable,field)	Says if the identifier is a field of a record

Table 4.1: Attributes of an identifier

```
procedure p (int a,b,c)
begin
write (a+c)/b
end
```

Parameters share many of the properties of local variables, in that their names are only accessible within a procedure but they also have the special property of being initialisable from outside the procedure. Both parameters and local variables vanish when the point of control moves outside the procedure in which they were declared. In vanishing, they loose any information they originally contained. Some languages allow an additional category of variables, termed 'static' in C that can be declared within a procedure, but which retain their values between invocations of the procedure.

In a block structured language procedures and blocks may be nested within one another.

```
let j= 3
procedure p (int a,b,c)
begin
  procedure m
  begin
   let a = b+j
   write (a+c)/b ! point 1
   begin
    let b = j
    write (a+c)/b ! point 2
   end
   write (a+c)/b ! point 3
  end
  write (a+c)/b ! point 0
  m()
end
p(1,2,3)
```

In the above there is a set of 8 identifiers j, p, a, b, c, m, a, b. Of these, j, p are globals. The identifieres in the set a, b, c areparameters of p. The locals of p are made up by the singleton set m andthe locals of m by a, b. Within the example there are 4 write statements,each of which writes out the same lexical expression (a+b)/c. Thisexpression will not always give the same value. The program will write out the numbers:

2.0 4.0 2.66666 4.0

These differences arise from the fact that there are 2 versions of both a and b in the program. At point 0 all that can be seen are the parameters a, b, c. At point 1 the parameter a is hidden by the local variable a, and similarly with b at point 2. Finally at point 3 the local b has vanished. The identifier manager handles this behaviour by using a compile time stack of pointers to identifier records. This is shown in figure 4.1.

Figure 4.1: handling identifiers

Whenever a new variable is declared, the function newname is called to allocate an identifier record for the new id. This record is pushed onto the identifier stack. When an identifier is encountered subsequently the function lookup is called. This takes as its parameter the lexical symbol for the identifier and scans the identifier stack from top to bottom to find the first identifier record with that symbol. The effect is to find the most recently declared identifier of that name. Whenever the syntax analyser encounters the start of a new scope, either a procedure or a block, it calls the procedure enterscope to record the value of the identifier stack pointer when the scope was entered. On exit from the procedure or block exitblock or exit_proc is called to restore the identifier stack to the state that it was in when the scope was entered.

.

Chapter 5

Code generation

5.1 Modification of syntax analyser

Up to now we have described a parser that is capable of checking if a program is valid in terms of the syntax and type rules of the language. It does not produce any output other than error messages for incorrect programs.

We will now look at how to modify the parser to generate an equivalent machine code program. The technique used in the Toolbox is to decorate the parser procedures with calls to the code generator. This can be illustrated by looking at a couple of simple examples.

```
{ -------------------------------------------------------------- }

{       WHILE CLAUSE                                              }

{             recognises: while <bool> do <void>                 }

{ -------------------------------------------------------------- }

procedure while_clause;

var t:typerec;

begin

     mustbe(while_sy); clause (t);

     if have (do_sy) then

     begin clause(t); match(t,VOID); end;

end;
```

Listing 9.1

```
{ ------------------------------------------------------------ }

{       WHILE CLAUSE                                            }

{            recognises: while <bool> do <void>                }

{ ------------------------------------------------------------ }

procedure while_clause;

var t:typerec;

    l1,l2,l3:labl;

begin

     l1:=newlab; l3:=newlab;l2:=newlab;

     plant(l1);

     mustbe(while_sy); clause (t); condify(t); jumpt(l3);jumpop(l2);

     plant(l3);

     if have (do_sy) then

     begin clause(t); match(t,VOID); end;

     bjump(l1);plant(l2);

end;
```

Listing 9.2

Listing 9.1 shows the original form of the parsing procedure for while clauses. It simply checks the grammar of the clause. Listing 9.2 shows how this is modified to handle code generation. It has been augmented with calls to newlab, plant, condify, jumpt, jumpop and bjump. As can be deduced from their names these procedures are responsible for generating jump instructions and handling labels. Suppose that we have the while statement:

while C1 do C2

with the Ci standing for clauses. The effect of the decorated parsing procedure is to generate machine code that looks like listing 9.3.

l1:

C1 code

condify code
jumpt l3
jump l2
l3: C2 code
jump l1
l2:
Listing 9.3

5.2 Notion of an abstract machine

The code shown in listing 9.3 is not for any one particular type of CPU. It is an abstract machine code. It abstracts from the details of particular machines. The syntax analyser assumes it is producing instructions for this abstract machine. The abstract machine is a general purpose computer whose instruction set includes all of the operations necessary to implement the semantics of the language that is being translated. On some computers the operations of the abstract machine can be implemented with single instructions. In others, several real machine instructions may be needed to achieve the same effect as the abstract machine instructions. What is shown in listing 9.3 is a fairly simple set of abstract machine instructions that are likely to be available to most machines. A full listing of the instructions executed by the abstract machine is given in

Appendix G, but we will give a brief outline of the machine here. The machine is assumed to have four registers:

PC Program Counter points at current instruction.
GP Globals Pointer, points at the start of the global variables
FP Frame pointer, points at the local variables of a procedure
SP Stack Pointer points at the top of the stack.
There are three areas of store:
CS The Code Store holds instructions
STACK This holds variables and temporary results
HEAP This holds objects like arrays, strings or structures.

All instructions are defined in terms of the effect that they produce on the registers and the stores.

5.3 Expressions and reverse polish notation

The S abstract machine is a stack machine. That is to say arithmetic instructions operate on the top two words on stack. Consider the following expression:

2+4

This works by placing two words on the stack and then adding them. The abstract machine instructions that do this would be:

llint(2)
llint(4)
add

This form of arithmetic in which the operator comes after its operands is termed reverse polish notation. It is a particularly easy notation to compile into. The general rule for generating code for any binary expression

e1 op *e2*

becomes :

generate code for *e1*

generate code for *e2*

generate code for op

Reverse polish notation combined with a recursive descent compiler will automatically generate the right code for expressions with operators of mixed priorities. The expression:

4+2*3

should yield 10. Given the syntax:

<exp3>::=<exp4>[<addop><exp4>]*

<exp4>::=<exp5>[<multop><exp5>]*

<exp5>::=<int-literal>

we obain the parse

```
Parse                   Code produced Stack
exp3                                  ...
exp4 addop exp4                       ...
exp5 addop exp4                       ...
4 addop exp4             llint(4)     ... 4
4 addop exp5 multop exp5              ... 4
4 addop 2 multop exp5    llint(2)     ... 4 2
4 addop 2 multop 3       llint(3)     ... 4 2 3
4 addop 2 * 3            mult         ... 4 6
4 + 2 * 3                add          ... 10
```

It is easy to translate these abstract instructions into concrete 8x86 instructions since the 8x86 supports a hardware stack. The previous sequence of instructions would generate:

```
push 4
push 2
push 3
pop ecx
pop eax
imul ecx
x:push eax
pop ecx
y:pop eax
add eax,cx
push eax
```

The instructions labeled x and y in the above sequence are strictly speaking redundant, and if the compiler has an optimising phase they should be deleted.

5.4 Handling of conditionals

The generation of arithmetic instructions is fairly straight forward since computers always have a set of arithmetic machine codes. Handling boolean operations is more problematic. Consider the operation $<$ which takes two numbers and returns a truth value. In a high level language like S-algol or Pascal truth values have the type boolean, and are represented in memory by a word which contains some non zero value for true and zero for false. Some modern CPUs like the AMD 29000 have opcodes that directly compute this operation, but older ones like the 80x86 series do not. Instead they have comparison instructions which compare two values and set some CPU flags on the result. In particular the sign and carry flags are set according to the result of comparison. The 80x86 series then provide jump instructions that will conditionally jump on the flags : JL for Jump Lessthan, JG for Jump Greater than etc.

Suppose we have the source code

```
if a<b do X
```

we want to generate code something like

```
push a
push b
pop ecx
pop eax
cmp eax,ecx
jl label1
jump label2
label1: ... code for X
label2:
```

For this sort of construct the setting of CPU flags is quite efficient as a control mechanism. For boolean assignment this is not so suitable. For the statement

```
p:= a<b
```

we need something like

```
push a
push b
pop ecx
pop eax
cmp eax,ecx
; code to generate a boolean on the stack
jl label1
push 0 ;*
jump label2
label1:push 1 ;*
; code to perform the assignment
label2:pop p
```

The instructions marked with * have to be inserted to convert the values in the flags into a boolean value on the stack. With a recursive descent compiler the syntax analyser procedure that looks for comparison operations does not know if this comparison is to be called in an if statement or in a boolean assignment or any one of a number of other contexts. What the procedure that analyses comparison expressions does is plant code for a compare instruction and return 'conditional' rather than 'boolean' as the type produced by the expression. When the code generator is asked to perform a conditional operation it remembers what comparison it was: less than, greater than etc. If at a later stage the syntax analyser discovers that it has a conditional and needs a boolean it calls the code generator to convert the conditional into a boolean by planting code that will plant the appropriate truth value on the stack.

5.4.1 If clauses

If clauses provide an illustration of how conditionals are handled. The syntax analysis procedure for an if clause is given in listing 30. Note how the procedure condify is called to ensure that the condition codes have been set. This is necessary to deal with examples like:

```
a:= z>y
if a do write "z > y"
```

The if clause tests the boolean variable a. After the compilet has matched the if it calls the procedure clause to parse the condition. This returns to indicate that the result on the stack is a boolean. The condify procedure finds that the top of the stack is a boolean so it plants code to compare the top of stack with zero. This sets the condition codes and allows the jump to be made. If on the other hand the source had been:

if x<y do write "x < y "

then the call on clause would have set the variable t to condition. Finding that the condition codes were already set, the condify procedure would do nothing.

Listing 9.7

5.4.2 For loops

For loops in programming languages come in two main forms. The simples is the Pascal variant where you write something like for i:=x to y do ... . Within the body of the loop, i will take on all the values in the range x to y in turn. Other languages, including S-algol allow a more general variant of the for loop : for i=x to y by z do ... . In this case z provides the step by which i is to be incremented. It is necessary to evaluate the expressions for the start, finish and the step at the top of the loop. Consider the following discriminating example:

```
let x:=1
let y:=3
```

Algorithm 30 If clauses

```
{ -------------------------------------------------------------- }

{     IF_CLAUSE

               this parses the rule
               <ifclause> ::= if <clause> do <clause> |

                              if <clause> then <clause> else <clause>
 }

{ -------------------------------------------------------------- }
 procedure if_clause ;
 var t1:typerec;l,l1,l3:labl;
 begin
     l1:=newlab; l:=newlab;l3:=newlab;
     next_symbol;
     clause(t);  condify(t);
     jumpt(l);jumpop(l3);plant(l);
     if have(do_sy) then begin clause(t);plant(l3); match(t,VOID) end
else

     begin
          mustbe(then_sy);
          clause(t1);jumpop(l1);decsp(t1);
          mustbe(else_sy);
          plant(l3);
          clause(t);balance(t,t1); plant(l1); release_label(l1);
     end;
     release_label(l3);

end;
```

```
for i=x to y do begin
write i
y:=y+i
end
write y
--> 1 2 3 9
```

If the expression y were evaluated each time round the loop, then the program would never terminate. In S-algol the loop variable (i in the above example) is implicitly declared as a cint for the duration of the for loop. Knowing this we can see thatthe above is equivalent to the code:

```
let x:=1
let y:=3
let induction:=x
let finish=y
while induction <= finish do begin
let i=induction
write i
y:=y+i
induction:=induction+1
end
write y
```

Note :
1) The induction variable and the end point are computed before the loop starts
2) That i is declared as a cint each time round the loop
The semantics of the generalised for loop with a variable step size are more complex.

```
for i=x to y by z do begin
write i
y:=y+i
end
write y
```

is equivalent to

```
let induction:=x
let finish=y
let step=z
while if step>0 then induction <= finish else induction>= finish do begin
let i=induction
write i
y:=y+i
induction:=induction+step
end
write y
```

In this general case, the direction in which the loop is going is must be determined at run time. We can not assume that the direction will be upwards. If z was -1 the direction of the loop has to be downwards in which case we have to check whether the induction variable is greater than the finish. What the compiler actually does is similar to translating the for loop into a while loop and then compiling this into machine code. What is done is shown in listing 31.

Algorithm 31 for loop code generation

```
procedure for_clause;

var t:typerec;l1,l2:labl;id:lextoken; os,   n:namedesc;
    complex:boolean;

begin
     enterscope(os);
     l1:=newlab;l2:=newlab;
     next_symbol;
     id:=symbol;
     mustbe(identifier);
     mustbe(eq_sy);
     clause(t);match(t,int_type);
     n:=newid(id,cint_type);
     mustbe(to_sy);
     clause(t);match(t,int_type);
     if have(by_sy) then begin complex:=true; clause(t);match(t,int_type);end
     else complex:=false;
     mustbe(do_sy); if not complex then forprepop;
     plant(l1);fortestop(complex,l2);
     clause(t);match(t,VOID);
     forstepop(complex,l1);plant(l2);
     exitblock(os,VOID);
 end;
```

This looks at the for loop to see if the loop is a simple one or a complex one. It calls the code generator to output either a forprep sequence if it is a simple loop. The code generator routines fortest and forpstep are then called with a parameter to indicate if the loop is complex or simple.We can see the code generated for the simple loop:

```
for i=1 to 10 do ...
```

in listing 32.

Note that this takes advantage of special instructions included in the x86 instruction set to handle simple loops. The loop instruction, expects the counter register ECX to hold the number of times it is to go round a loop.

Algorithm 32 code generated for a for loop

```
push 1 ; this location on the stack will be the variable i
push 10
; forprep sequence
pop ecx
pop eax
push eax
sub ecx,eax
add ecx,2 ; precompute the number of times round loop
;minfortest sequence
l1: loop m1 ; this is a machine code instruction which
; tests the CX register
; if non zero it goes to m1 and decrements CX
pop eax
jmp l2
m1: push ecx ; CX held induction variable
;--------------------- Main body of loop goes here
; minforstep sequence
pop ecx ; induction variable back in CX register
pop eax ; increment i
inc eax
push eax
jmp l1 ; go back to the top of the loop
l2:
```

We precompute this and load it into CX before the loop starts. During the body of the loop, CX is pushed onto the stack to prevent it being corrupted by a nested loop.

5.5 Variable access

An abstract machine specifies a set of stores and a set of operations on these stores. These stores can have a number of possible types. One class of store is predesignated variables capable of holding an individual word of data. We generally call these registers. In an actual hardware machine the registers will often be implemented by using particularly fast memory chips, or in a microprocessor, by using on chip memory cells. From the standpoint of abstract machine design this is not important, since an abstract machine is concerned only with the functional specification of a computer. The speed of access to different parts of the store is an implementation optimization.

Some abstract machines support a random access memory. The PS-algol machine does not. Instead it uses forms of structured memory: stack and heap. On a given implementation these may actually be implemented in a common random access store, but this is not necessary. Indeed it might be advantageous from a performance point of view to implement the heaps and stacks as physically distinct memories.

The areas of memory defined by the PS-algol abstract machine are the registers, the code store, the stack, and the heap.

5.5.1 Stack variables

PS-algol, like all Algols is a recursive language. It is recursive in two senses. It is defined by a recursive grammar and it allows the recursive calling of procedures. This imposes special constraints on the store of the language that are best satisfied by a stack structured memory. Consider the fragment of code in algorithm 33.

In this example four variables are define a,i,x,y, but at no point are more than 3 of the variables in scope at once. At position 2 the variables x, y,a are in scope and at position 3 the variables x,y,i are in scope. In other words, different variables persist for different periods of time. Variables are only in scope between the point at which they are declared and the end of the block. Because the grammar of Algol allows blocks to be nested it generates a Last In First Out discipline on the scope rules. The variables in the outermost block remain in scope for the entire program whereas the variables in innermost blocks are discarded first. This lends itself naturally to a stack implementation in algorithm 34.

Variables are accessed by specifying their address relative to the current base of the stack. The variable x is accessed using the operator global(0) since it is at the base of the stack, y is addressed as global(1) as it is at position 1 on the

Algorithm 33 example of nested scopes

```
begin
      let x:=3
      let y:=x*readi
      ! position 1
      begin
       let a = x
       x:=y; y:=a
       ! position 2
      end
      begin
       let i:=9+x
       if i>y do y:=x
       ! position 3
      end
    end
```

Algorithm 34 Code generated for the nested scopes in algorigthm33

```
1 ll.int(3)    ! let x:=3
2 global(0)    ! x -> top of stack
3 readi        ! readi -> top of stack
4 times        ! let y:= x*readi
! position 1
5 global(0)    ! let a=x
6 global(1)    ! y -> top of stack
7 globalassign(0) ! x:=y
9 global(2)    ! a-> top of stack
10 gloabalassign(1) ! y:=a
! position 2
11 retract(1)  ! get rid of a
12 ll.int(9)   ! 9 -> top of stack
13 global(0)   ! x-> top of stack
14 plus        ! let i:=9+x
15 global(2)   ! i->top of stack
16 global(1)   ! y->top of stack
17 le.i        ! i<y -> top of stack
18 jumpf(23)   ! if top of stack
               ! false goto 23
19 global(1)   ! y -> top of stack
20 globalassign(0) ! x:=y
! position 3
21 retract(1) ! get rid of i
22 retract(2) ! get rid of x and y
```

stack etc. It is worth noting that the combination of the PS-algol initializing assignment statement

let <variable>:= <expression>

with the stack allocation discipline means that many of the store instructions that would be required in a conventional machine architecture are dispensed with. The initial value is simply calculated and then left on the stack. The compiler then just remembers where on the stack it was left.

If the variable was declared at the outer most level the address associated with the variable is given relative to the GP or global pointer register [1]. If a variable is declared in a procedure, then its address is specified relative to the FP or frame pointer register [2]. When generating code, variables are consistently dealt with in terms of their addresses relative to some base register.

5.6 Procedure calls

The most complicated useof the stack in an Algol-based language is the way in which it is used to implement procedure calls. We will start by looking at how to implement procedure calls in languages like C that do not allow nested procedures.

5.6.1 Simple procedures

Global variables are accessed by offset from some global base register.

Local variables are accessed by an offset from the frame pointer. Suppose we have the following C procedure:

```
swap(a,b)int *a,*b;
{int temp;temp= *a; *a= *b; *b = temp; ]
```

This might generate the folowing abstract machine code:

```
49 push(fp) ! tos <- fp
49 copy(fp,sp) ! fp <- sp
50 retract(-1) ! reserve space for temp
51 local.i(-3) ! push a, tos<- [FP-3]
52 deref ! change to *a tos<-[tos]
53 localass(1) ! store in temp [FP+1]<-tos
54 local.i(-3) ! a to top of stack
55 local.i(-2) ! b to top of stack
56 deref ! change to *b
57 store ! store in *a [tos]<-tos
58 locali(-2) ! push b
59 local(1) ! push temp
60 store ! *b <-temp
```

[1] Typically the DS register on an intel machine.

[2] This would typically be the EBP register on an intel machine.

Figure 5.1: Local variable and parameter access

```
  +----------------+ <-----------SP
  | work space     |
  +----------------+
  | temp           |
  +----------------+
+-| old FP         | <-----------FP
| +----------------+
| | return address |
| +----------------+
| | b              |
| +----------------+
| | a              +
V +----------------+
```

```
61 retract(1) ! get rid of temp
62 pop(fp)
63 return
```

The important thing to note about this is the procedure entry and exit code. When the procedure is entered the FP is saved on the stack and reset to point at the current top of stack. The stack pointer is then advanced to create sufficient space for the local variables (only one in this case). On exit from the procedure the space is released and the FP restored to its previous value before returning. The stored copy of FP on the stack is termed the dynamic link . It links a procedure to the environment in which it was called.

The meaning of the code is made clearer by figure 5.1. The local variables are accessed by a positive offset from the FP and the parameters by a negative one. A procedure call to swap might go as follows:

```
swap(&x,&y)
```

translating into

```
100 local.addr(4) ! tos <- &x means tos<- FP+4
101 local.addr(5) ! tos <- &y means tox<- FP+5
102 call(49) ! call swap
103 retract(2) ! get rid of parameters
```

The parameters are pushed onto the stack followed immediately by a call to the start address of the procedure. The call itself pushes the return address onto the stack so that when the procedure has been entered and the last parameter (b in this case) will be at a local address of -2 relative to the FP.

Stack direction

In the abstract machine examples given above it is assumed that the stack grows upwards from low addresses to high addresses. This is true on some hardware but not on all. On intel machines like the 8086, the stack grows downwards from high addresses to low addresses. The actual machine code generated by the toolbox must take into account which direction the stack grows in. On a machine with a downward growing stack the addresses of parameters will be a positive offset from the FP and the addresses of local variables a negative offset from the FP.

5.6.2 Nested procedures

Algol like languages allow procedures to be nested within one another. The problem is to devise a calling mechanism that will:

a) Enable procedures to have space for local variables
b) Allow these to be called recursively
c) Allow procedures to access variables that are in a surrounding scope

We will consider the example shown in listing 35. Here we have 3 procedures A, B and C with procedures B and C within procedure A. B must have access to the variables of A , to its own variables and to the global variables. Access to the globals is no problem, since the global pointer register can beused for this. Access to the locals can be handled as shown in the previous example. The difficulty comes with access to intermediate level variables, those of A. In our example X is a global, and M and P are intermediate level variables with respect to B. B is called by C. How is B to access M?

Algorithm 35

```
let X:=3 ! a global variable
procedure A(int P)
begin
 let M=P+X ! an intermediate variable
procedure B
begin
 write M
end
procedure C
begin
 B; ! call B
end
 if P>0 then A(P-1) else C
end
```

The technique used to handle this problem is called a display. In an algol like langauge variables can be accessed using a 2 component addressing technique.

Figure 5.2: Formation of a display

address	contents	comment
135	132	display 2
134	122	display 1
133	100	display 0
132	127	dynamic link
131		return address for b
130	127	display 2
129	122	display 1
128	100	display 0
127	122	dynamic link
126		return address for c
125		M
124	122	display 1
123	100	display 0
121		return address for a
120		P

For the global context we have

address	contents	comment
101		X
100	100	global base

Each variable is identified by the combination of what is called its lexical level with an offset. Variables at the global level are said to be at lexical level 0. Variables in global procedures like A are said to be at lexical level 1. Variables in a nested procedure like B would be at lexical level 2 etc. If we write down the addresses of the variables in the example in this way we get the following:

```
Variable Address
X        0,1
P        1,-2
M        1,3
```

A display is an array of pointers that points at the start of the store for different lexical levels. If the display is indexed using the lexical level portion of the address, then it is possible to find a variable at any lexical level in two stages. If an address is given in the form (ll,offset) then the variable will be at location:
display[ll]+offset

Whenever a procedure is entered a display is created pointing at the enclosing lexical levels. This is done using the abstract machine instruction prologop.

The semantics of the prolog operation are given below.

```
prologop(ll:integer);
FP->S[++SP]; SP->FP;
```

```
S[S[FP]:S[FP]+ll-1]->S[SP+1:SP+ll]; SP+ll->SP;
FP->S[++SP]
```

Prologop takes one parameter: ll, the lexical level of the procedure that is being entered. The first thing it does is to push the dynamic link just as we did for a C procedure. It then copies ll elements of the previous display onto the stack and follows that by pushing the frame pointer onto the stack. What happens in practice is shown in figure5.2 .

The display of A is the array [100, 122], where 100 is the base address for globals, and 122 is the address of the current frame.

The display of C is [100, 122, 127]. That is a copy of the display of A followed by the address of the frame for C = 127.

The display of B is [100, 122, 132]. It shares elements 0 and 1 with the display of C, since both of these procedures are nested at the same lexical level. On the other hand it has a new element 2, since the new lexical level 2 is the frame of B. This dynamically constructed display gives access to all of the variables in scope from B. Thus

X = (0,1) = display[0]+1 = 100 +1 = 101

P = (1,-2) = display[1]-2 = 122 -2 = 120

M = (1,3) = display[1]+3 = 122 +3 = 125

The abstract machine instructions that access intermediate variables take two parameters lexical level and offset. The compiler translates these into a series of instructions that access the display and find the variable. For instance on an x86 the abstract instruction to assign an integer to a variable could be translated into:

```
assi     macro
;  Assign an integer
; invoke with first parameter lexical level
; second parameter the offset of the variable within frame
          mov esi,[ebp+#1]   ; ebp points at the display
                             ;esi<-display[lexlevel]
          pop dword[esi+#2]  ; esi now points at the desired frame
                             ; esi[offset]<- tos
          #em
```

At the end of a procedure an epilog sequence has to be generated to return to the context the procedure came from. Unlike C, S-algol procedures discard their parameters before they return from the stack. This is done by epilogop.

```
epilogop (Discard:integer)
FP->SP; S[SP--]->FP; S[SP--]->PC; SP-Discard ->SP
```

5.7 Structure of the code generator

For the PS-algol compiler the code generator is a collection of procedures in the module SAGEN.PAS.

These are organised one per abstract machine instruction. Each procedure has the the name of an abstract machine instruction and when called places this abstract machine instruction into the compiled binary program.

It actually implements the abstract machine instructions by calling the macro assembler module ASSEMBLE.PAS, to output one or more 8086 instructions.

We will examine how the assembler works in the next chapter. We will now examine how the abstract machine instructions are implemented using the physical resources made available to us by the x86.

5.7.1 Register use

The abstract machine has a small collection of registers that have to be implemented on the physical register set of the intel x86 series machines. A description of the Intel processor architecture is not provided here. Those who are unfamiliar with it are advised to consult a reference book [3].

On the x86 the following conventions are used for register allocation in the compiler toolbox. The frame pointer is implemented using the intel EBP register. The global pointer is the intel EBX register. Since the stack grows downwards variables are accessed with negative offsets from these registers.

The display mechanism is directly supported in the intel hardware for processor models iAPX 186 and upwards and on the NEC V series processors. On these machines there is a single instruction ENTER that implements prologop.

The assignment of abstract machine registers to physical registers is sumarised below

Abstract	Real
PC	PC
GP	EBX
FP	EBP
SP	ESP

Arithmetic is done using the EAX register as the destination. The ECX register is used as a loop counter.

Note that this contrasts with the exercise for PLDI3 which uses the floating point stack for all arithmetic.

5.7.2 Keeping track of the stack

The code generator maintains an internal variable called stack_ptr which is used to keep track current displacement between the SP and FP registers. The pro-

[3] A particularly clear explanation of the original 8086 is given in chapter 5 of 'Osborne 16 bit Microprocessor Handbook' by Adam Osborne from McGraw-Hill. Alternatively one can consult the processor manuals published by Intel, AMD or NEC for their CPU chips. It should be born in mind that the register naming conventions used in NEC literature differ slightly from that used by Intel and AMD. In what follows, the Intel names will be used. For more recent machines the manuals can be downloaded from http://developer.intel.com/design/pentium4/manuals/.

cedures which output abstract machine instructions should increment or decrement stack_ptr to mimic the effects that will be produced at run time on the real stack. To help in doing this a collection of utility routines are provided to increment or decrement the stack by the space that would be taken up by a value of a given type. The procedure incsp should be called when a value is pushed onto the stack and decsp when a value is poped from the stack. These procedures use information about the sizes of types that are expressed in strides. Strides are the smallest amount by which the stack can be adjusted. On 80386 machines and above strides are 4 bytes long. It is important when making any alterations to the codegenerator to ensure that these procedures are called whenever an opcode is produced that will affect the real time stack.

Chapter 6

The Assembler

The assembler phase of a compiler is responsible for generating the binary machine code that will be executed by the cpu. As such it has to know about the details of an individual machine code. A conventional assembler is a stand alone program that takes in a file of ascii text with an assembly language source program in it and outputs either a linkable object file or an executable binary file. Many compilation systems use a stand alone assembler as the last phase of the compilation process. This has several advantages:

1. The compiler writers need not bother themselves with the binary formats of the machine instructions, all they have to know about is how to generate the assembler mnemonics, which is usually much easier.

2. Several compilers can share the same assembler which encourages software reuse.

On Unix this use of a stand alone assembler for the back end of a compiler seems to be standard practice. This is partly because an assembler is provided with the C compiler on every Unix system. Since all Unix systems used for software development are bound to have a C compiler, writers of other compilers are free to use the same back end. On MSDOS, assemblers are not provided as standard features of the operating system. Since each compiler writer then has to provide their own version of the assembler, they have to consider whether to make the assembler a stand alone program or a bound in module of their compiler.

The disadvantage of using a stand alone program is obviously that it has to communicate with the compiler via intermediate text files. The output of these, followed by their input and reanalysis by the assembler will be time consuming. If you have to provide your own assembler it you might as well produce a fast one. That is the strategy adopeted in the Toolbox. The assembler in the toolbox is a pascal module assemble.pas that is bound into the main compiler program. We can see the relationship between the two in figure 6.1. In the left hand example you see the arangement used in the toolbox. The compiler is a single program containing several modules, of which only the assembler and the front end are shown. These communicate via shared buffers. In the right hand example the

Figure 6.1: Two possible arrangements for the compiler and assembler

compiler and the assembler are stand alone programs which communicate via intermediate files.

The two main tasks that a conventional assembler has to deal with are

a) Converting mnemonics into binary code.

b) Keeping track of the addresses associated with labels.

Both of these become considerably easier when the assembler is a program module and communicates with other modules using internal buffers. It is no longer necessary for the mnemonics to be human readable. An assembly language instruction would normally look something like:

operator parm1 parm2

ENTER 2, 20

There would be an opcode, followed by one or more operands. Each of these fields would be encoded as a sequence of ascii characters. Allowing for newline characters and spaces, the whole line might take 15 to 20 bytes of file space. When we want to assemble code directly in memory, we do not want to waste this much space. Instead of holding the assembly language source as lines of text, we can hold it as records. These can be much more compact. Listing 36 shows the format of the pseudo instruction record used in the toolbox to represent a line of assembler source. Like the textual source line it contains three fields: an operator and two parameters. The whole thing takes up only 5 bytes, 2 bytes for each integer and one for the operator.

Algorithm 36

```
type pseudo = record abstract instruction
operator:opcode;
parm2:integer;
parm1:integer;
end;
```

The operator belongs to the enumerated datatype opcode declared in the opcodes.pas module. A partial listing of the type opcode is shown below.

```
type opcode = (
jl,
jle,
jg,
jge,
je,
jnz,
jump,
jumpt,
jumptt,
```

Figure 6.2: finding the binary code for an opcode

```
    cjump,
    fortest,
    forstep,
    outbyte,
    shrink,
    enterframe,
    exitframe,
```

It is an enumerated type whose members are opcode mnemonics. Since there are fewer than 255 members of the enumerated type, the pascal compiler will represent its members as byteintegers of 8 bits. This enables the opcode field of the pseudo instruction to only take up one byte.

The assembler strategy is for the code generator phase of the compiler to deposit a sequence of these pseudo instructions into an array. At the end of the code generation phase the assembler is then called to convert these into binary code which can be output to a file.

6.1 Converting the opcodes

The central task of the assembler is to expand out the opcode menmonics into the series of bytes that implement the operations on an 80x86 processor. It does this using the three arrays : codelen, codeoffset, codelib. The fist two of these are mappings from opcodes to integers. Given an opcode mnemonic, codelen will tell you how many bytes there are in the equivalent machine code. Codelib is a big array of bytes that contains all of the binary opcodes. To obtain the sequence of bytes equivalent to a particular opcode the process shown in figure 6.2 is followed. The codeoffset array is indexed to find the start of the code sequence in the codelib array, and the codelen array is used to determine how many bytes to copy from the codelib to the binary file.

The other task that the assembler has to achieve is to plant the parameter fields of the instructions into the outgoing code stream. If we consider the instructionset of the 80x86 we find that some of the opcodes take 2 parameters, some take 1 and others take none. As against this the format of the pseudo instructions always contains two parameters whether they are needed or not. The assembler has to find some way of determining how many parameters an instruction will really need. this information is encoded in an array codeparams, shown in listing 10.3. This specifies the different types of parameters that instructions can have, as follows:

nonadic The instruction has no parameters.

monadic The instruction has a single parameter, this will be the 16 bit number held in param 1.

dyadic The instruction has 2 parameter field each of 16 bits, to be obtained from param1 and param 2 of the pseudo instruction.

byteadic The instruction has a single 8 bit parameter obtained from param1 of the pseudo instruction

relative The instruction has a single parameter which is a 16 bit relative offset from the current value of the program counter.

byterel The instruction has a single parameter which is an 8 bit relative offset from the current value of the program counter.

abslabel The instruction contains a single parameter which is a 16 bit absolute address from the start of the code segment.

Algorithm 37

```
optype=(
nonadic,monadic,dyadic,stringadic,byteadic,relative,byterel,
abslabel);
const codeparams:array[opcode]of optype =(
{jl}byterel,
{jle}byterel,
{jg}byterel,
{jge}byterel,
{je}byterel,
{jnz}byterel,
{jump}relative,
...
{fortest}relative,
...
{outbyte}nonadic,
...
{prolog}byteadic,
```

6.2 Address handling

Several of the addressing modes require the assembler to generate addresses to words in the code segment. These are usually the destinations of jump instructions. The address formats required in the final code will either be relative to the start of the code segment or relative to the program counter. In the 'assembly source', which in this case is a sequence of pseudo instructions, the destination address will be indicated by a label. To understand how this works it is helpful at first to think of what the assembler would have to do if it were taking its input from a conventional source file.

Consider the following example
mov ecx,10

```
l1: push ecx
call proc1
pop eax
dec ecx
jnz l1
...
proc1: ......
.....
```

The assembly language sequence will call procedure proc1 10 times. The assembler needs to know what addresses correspond to these labels if it is to be able to generate the correct binary code. What an assembler does is to maintain a label table as shown below.

label	address
l1	100
proc1	124
labn	230

This associates with each label its machine code address. Assemblers conventionally have two passes over the source. During the first pass they calculate what address is associated with each label and put it into the label table. During the second they use the addresses thus calculated to fill in the address fields of the code.

With the ram resident assembler we are using the labels are simply numbers that are used to index the label table. No lexical representation of the labels is needed. A special pseudo instruction called PLANT is used to plant a label in the code. The procedure FIXLABELS shown in listing 3810.3 calculates what address is associated with each label.

6.3 The interface

The interface to the assembler is provided by 3 procedures :

```
procedure pass2(var comfile:codefile);
procedure assem(op:opcode;p1:integer;p2 : integer);
procedure initassem;
```

Algorithm 38

```
procedure fixlabels;

      var i:word;

      begin
           machinepc:=start;
           for i:=0 to pc do
               with pseudocode^[i] do begin
                    if operator=plant_label then labels[parm1]:=machinepc;
                    machinepc:=machinepc+codelen[operator];
               end
       end;
```

Before anything else is called the assembler must be intialised using INITASM. Then for each instruction to be assembled, the procedure ASSEM is called. The arguments to this are an opcode and two parameters. These can be either integer constants or indices into the label table. When all the instructions are placed, PASS2 is called to output the machine code to the specified file.