University of Wales Swansea

Department of Computer Science

Compilers

Course notes for module CS_218

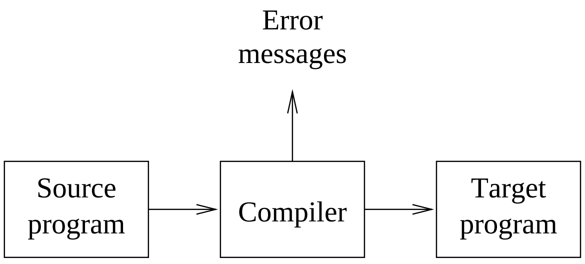
Dr. Matt Poole 2002, edited by Mr. Christopher Whyley, 2nd Semester 2006/2007

`www-compsci.swan.ac.uk/~cschris/compilers`

1 Introduction

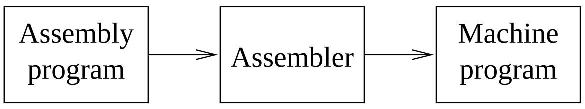
1.1 Compilation

Definition. *Compilation* is a process that translates a program in one language (the *source language*) into an *equivalent* program in another language (the *object* or *target language*).

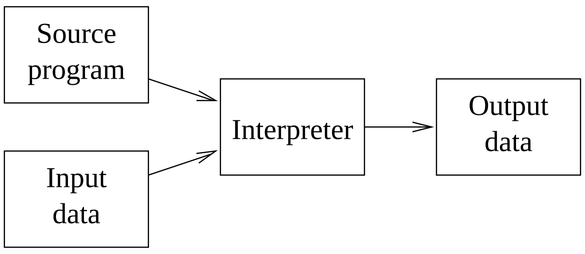


An important part of any compiler is the detection and reporting of *errors*; this will be discussed in more detail later in the introduction. Commonly, the source language is a high-level programming language (i.e. a *problem-oriented language*), and the target language is a machine language or assembly language (i.e. *a machine-oriented language*). Thus compilation is a fundamental concept in the production of software: it is the link between the (abstract) world of application development and the low-level world of application execution on machines.

Types of Translators. An *assembler* is also a type of translator:



An *interpreter* is closely related to a compiler, but takes both source program and input data. The translation and execution phases of the source program are one and the same.



Although the above types of translator are the most well-known, we also need knowledge of compilation techniques to deal with the recognition and translation of many other types of languages including:

- Command-line interface languages;
- Typesetting / word processing languages (e.g.TEX);
- Natural languages;
- Hardware description languages;
- Page description languages (e.g. PostScript);
- Set-up or parameter files.

Early Development of Compilers.

1940's. Early stored-program computers were programmed in machine language. Later, assembly languages were developed where machine instructions and memory locations were given symbolic forms.

1950's. Early high-level languages were developed, for example FORTRAN. Although more problem-oriented than assembly languages, the first versions of FORTRAN still had many machine-dependent features. Techniques and processes involved in compilation were not well-understood at this time, and compiler-writing was a huge task: e.g. the first FORTRAN compiler took 18 man years of effort to write.

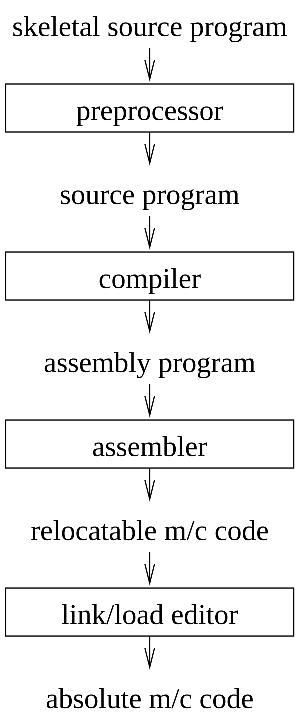
Chomsky's study of the structure of natural languages led to a classification of languages according to the complexity of their *grammars*. The *context-free languages* proved to be useful in describing the syntax of programming languages.

1960's onwards. The study of the *parsing problem* for context-free languages during the 1960's and 1970's has led to efficient algorithms for the recognition of context-free languages. These algorithms, and associated software tools, are central to compiler construction today. Similarly, the theory of finite state machines and regular expressions (which correspond to Chomsky's *regular languages*) have proven useful for describing the lexical structure of programming languages.

From Algol 60, high-level languages have become more problem-oriented and machine independent, with features much removed from the machine languages into which they are compiled. The theory and tools available today make compiler construction a managable task, even for complex languages. For example, your compiler assignment will take only a few weeks (hopefully) and will only be about 1000 lines of code (although, admittedly, the source language is small).

1.2 The Context of a Compiler

The complete process of compilation is illustrated as:



1.2.1 Preprocessors

Preprocessing performs (usually simple) operations on the source file(s) prior to compilation. Typical preprocessing operations include:

(a) Expanding *macros* (shorthand notations for longer constructs). For example, in C,

```
#define foo(x,y) (3*x+y*(2+x))
```

defines a macro `foo`, that when used in later in the program, is expanded by the preprocessor. For example, `a = foo(a,b)` becomes

```
a = (3*a+b*(2+a))
```

(b) Inserting named files. For example, in C,

```
#include "header.h"
```

is replaced by the contents of the file `header.h`

1.2.2 Linkers

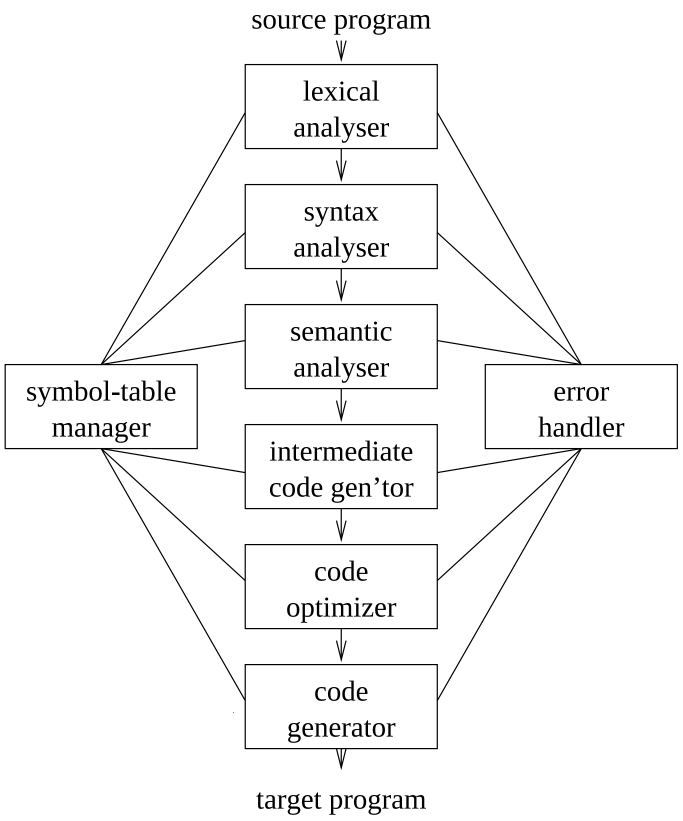
A linker combines object code (machine code that has not yet been linked) produced from compiling and assembling many source programs, as well as standard library functions and resources supplied by the operating system. This involves resolving references in each object file to external variables and procedures declared in other files.

1.2.3 Loaders

Compilers, assemblers and linkers usually produce code whose memory references are made relative to an undetermined starting location that can be anywhere in memory (relocatable machine code). A loader calculates appropriate absolute addresses for these memory locations and amends the code to use these addresses.

1.3 The Phases of a Compiler

The process of compilation is split up into six phases, each of which interacts with a *symbol table manager* and an *error handler*. This is called the *analysis/synthesis model of compilation*. There are many variants on this model, but the essential elements are the same.



1.3.1 Lexical Analysis

A *lexical analyser* or *scanner* is a program that groups sequences of characters into *lexemes*, and outputs (to the syntax analyser) a sequence of tokens. Here:

(a) *Tokens* are symbolic names for the entities that make up the text of the program; e.g. *if* for the keyword `if`, and *id* for any identifier. These make up the output of the lexical analyser.

(b) A *pattern* is a rule that specifies when a sequence of characters from the input constitutes a token; e.g the sequence `i, f` for the token *if*, and *any sequence of alphanumerics starting with a letter* for the token *id.*

(c) A *lexeme* is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example `if` matches the pattern for *if*, and `foo123bar` matches the pattern for *id.*

For example, the following code might result in the table given below.

```
program foo(input,output);var x:integer;begin
readln(x);writeln('value read =',x) end.
```

Lexeme	*Token*	*Pattern*
`program`	*program*	`p, r, o, g, r, a, m`
		newlines, spaces, tabs
`foo`	*id* (`foo`)	*letter followed by seq. of alphanumerics*
`(`	*leftpar*	*a left parenthesis*
`input`	*input*	`i, n, p, u, t`
`,`	*comma*	*a comma*
`output`	*output*	`o, u, t, p, u, t`
`)`	*rightpar*	*a right parenthesis*
`;`	*semicolon*	*a semi-colon*
`var`	*var*	`v, a, r`
`x`	*id* (`x`)	*letter followed by seq. of alphanumerics*
`:`	*colon*	*a colon*
`integer`	*integer*	`i, n, t, e, g, e, r`
`;`	*semicolon*	*a semi-colon*
`begin`	*begin*	`b, e, g, i, n`
		newlines, spaces, tabs
`readln`	*readln*	`r, e, a, d, l, n`
`(`	*leftpar*	*a left parenthesis*
`x`	*id* (`x`)	*letter followed by seq. of alphanumerics*
`)`	*rightpar*	*a right parenthesis*
`;`	*semicolon*	*a semi-colon*
`writeln`	*writeln*	`w, r, i, t, e, l, n`
`(`	*leftpar*	*a left parenthesis*
`'value read ='`	*literal* (`'value read ='`)	*seq. of chars enclosed in quotes*
`,`	*comma*	*a comma*
`x`	*id* (`x`)	*letter followed by seq. of alphanumerics*
`)`	*rightpar*	*a right parenthesis*
		newlines, spaces, tabs
`end`	*end*	`e, n, d`
`.`	*fullstop*	*a fullstop*

It is the sequence of tokens in the middle column that are passed as output to the syntax analyser.

This token sequence represents *almost* all the important information from the input program required by the syntax analyser. Whitespace (newlines, spaces and tabs), although often important in separating lexemes, is usually not returned as a token. Also, when outputting an *id* or *literal* token, the lexical analyser must also return the value of the matched lexeme (shown in parentheses) or else this information would be lost.

1.3.2 Symbol Table Management

A *symbol table* is a data structure containing all the *identifiers* (i.e. names of variables, procedures etc.) of a source program together with all the *attributes* of each identifier.

For variables, typical attributes include:

- its type,
- how much memory it occupies,
- its scope.

For procedures and functions, typical attributes include:

- the number and type of each argument (if any),
- the method of passing each argument, and
- the type of value returned (if any).

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table during the lexical analysis and/or syntax analysis phases.

1.3.3 Syntax Analysis

A *syntax analyser* or *parser* is a program that groups sequences of tokens from the lexical analysis phase into *phrases* each with an associated *phrase type*.

A phrase is a logical unit with respect to the rules of the source language. For example, consider:

```
a := x * y + z
```

After lexical analysis, this statement has the structure

$$id_1 \; assign \; id_2 \; binop_1 \; id_3 \; binop_2 \; id_4$$

Now, a syntactic rule of Pascal is that there are objects called 'expressions' for which the rules are (essentially):

(1) Any constant or identifier is an expression.

(2) If exp_1 and exp_2 are expressions then so is exp_1 *binop* exp_2.

Taking all the identifiers to be variable names for simplicity, we have:

- By rule (1) $exp_1 = id_2$ and $exp_2 = id_3$ are both phrases with phrase type 'expression';
- by rule (2) $exp_3 = exp_1 \; binop_1 \; exp_2$ is also a phrase with phrase type 'expression';
- by rule (1) $exp_4 = id_4$ is a phase with type 'expression';
- by rule (2), $exp_5 = exp_3 \; binop_2 \; exp_4$ is a phrase with phrase type 'expression'.

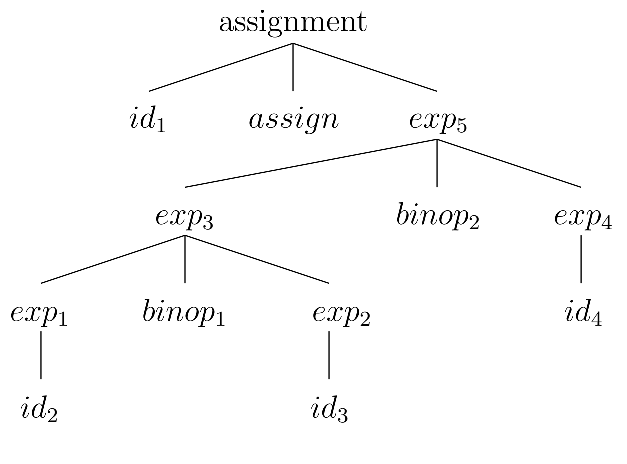
Of course, Pascal also has a rule that says:

$$id \; assign \; exp$$

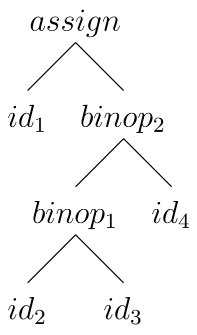
is a phrase with phrase type 'assignment', and so the Pascal statement above is a phrase of type 'assignment'.

Parse Trees and Syntax Trees. The structure of a phrase is best thought of as a *parse tree* or a *syntax tree*. A parse tree is tree that illustrates the grouping of tokens into phrases. A syntax tree is a compacted form of parse tree in which the operators appear as the interior nodes. The construction of a parse tree is a basic activity in compiler-writing.

A parse tree for the example Pascal statement is:



and a syntax tree is:



Comment. The distinction between lexical and syntactical analysis sometimes seems arbitrary. The main criterion is whether the analyser needs recursion or not:

- lexical analysers hardly ever use recursion; they are sometimes called *linear analysers* since they scan the input in a 'straight line' (from from left to right).
- syntax analysers almost always use recursion; this is because phrase types are often defined in terms of themselves (cf. the phrase type 'expression' above).

1.3.4 Semantic Analysis

A *semantic analyser* takes its input from the syntax analysis phase in the form of a parse tree and a symbol table. Its purpose is to determine if the input has a well-defined meaning; in practice semantic analysers are mainly concerned with *type checking* and *type coercion* based on *type rules*. Typical type rules for expressions and assignments are:

Expression Type Rules. Let *exp* be an expression.

(a) If *exp* is a constant then *exp* is well-typed and its type is the type of the constant.

(b) If *exp* is a variable then *exp* is well-typed and its type is the type of the variable.

(c) If *exp* is an operator applied to further subexpressions such that:

  (i) the operator is applied to the correct number of subexpressions,

  (ii) each subexpression is well-typed and

  (iii) each subexpression is of an appropriate type,

  then *exp* is well-typed and its type is the result type of the operator.

Assignment Type Rules. Let *var* be a variable of type T_1 and let *exp* be a well-typed expression of type T_2. If

(a) $T_1 = T_2$ and

(b) T_1 is an assignable type

then *var assign exp* is a well-typed assignment.

For example, consider the following code fragment:

```
intvar := intvar + realarray
```

where `intvar` is stored in the symbol table as being an integer variable, and `realarray` as an array or reals. In Pascal this assignment is syntactically correct, but semantically incorrect since `+` is only defined on numbers, whereas its second argument is an array. The semantic analyser checks for such type errors using the parse tree, the symbol table and type rules.

1.3.5 Error Handling

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

- report the error in a helpful way,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

Types of Error. Errors are either *syntactic* or *semantic*:

Syntax errors are errors in the program text; they may be either *lexical* or *grammatical*:

(a) A lexical error is a mistake in a lexeme, for examples, typing `tehn` instead of `then`, or missing off one of the quotes in a literal.

(b) A grammatical error is a one that violates the (grammatical) rules of the language, for example `if x = 7 y := 4` (missing `then`).

Semantic errors are mistakes concerning the meaning of a program construct; they may be either *type errors*, *logical errors* or *run-time errors*:

(a) Type errors occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.

(b) Logical errors occur when a badly conceived program is executed, for example: `while x = y do ...` when `x` and `y` initially have the same value and the body of loop need not change the value of either `x` or `y`.

(c) Run-time errors are errors that can be detected *only* when the program is executed, for example:

```
var x : real; readln(x); writeln(1/x)
```

which would produce a run time error if the user input 0.

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

1.3.6 Intermediate Code Generation

After the analysis phases of the compiler have been completed, a source program has been decomposed into a symbol table and a parse tree both of which may have been modified by the semantic analyser. From this information we begin the process of generating object code according to either of two approaches:

(1) generate code for a specific machine, or

(2) generate code for a 'general' or *abstract* machine, then use further translators to turn the abstract code into code for specific machines.

Approach (2) is more modular and efficient provided the abstract machine language is simple enough to:

(a) produce and analyse (in the optimisation phase), and

(b) easily translated into the required language(s).

One of the most widely used intermediate languages is Three-Address Code (TAC).

TAC Programs. A *TAC program* is a sequence of optionally labelled instructions. Some common TAC instructions include:

(i) $var_1 := var_2\ binop\ var_3$

(ii) $var_1 := unop\ var_2$

(iii) $var_1 := num$

(iv) `goto` *label*

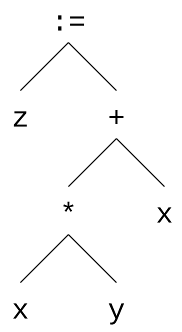
(v) `if` var_1 *relop* var_2 `goto` *label*

There are also TAC instructions for addresses and pointers, arrays and procedure calls, but will will use only the above for the following discussion.

Syntax-Directed Code Generation. In essence, code is generated by recursively walking through a parse (or syntax) tree, and hence the process is referred to as *syntax-directed* code generation. For example, consider the code fragment:

```
z := x * y + x
```

and its syntax tree (with lexemes replacing tokens):



We use this tree to direct the compilation into TAC as follows.

At the root of the tree we see an assignment whose right-hand side is an expression, and this expression is the sum of two quantities. Assume that we can produce TAC code that computes the value of the first and second summands and stores these values in `temp1` and `temp2` respectively. Then the appropriate TAC for the assignment statement is just

```
z := temp1 + temp2
```

Next we consider how to compute the values of `temp1` and `temp2` in the same top-down recursive way.

For `temp1` we see that it is the product of two quantities. Assume that we can produce TAC code that computes the value of the first and second multiplicands and stores these values in `temp3` and `temp4` respectively. Then the appropriate TAC for the computing `temp1` is

```
temp1 := temp3 * temp4
```

Continuing the recursive walk, we consider `temp3`. Here we see it is just the variable `x` and thus the TAC code

```
        temp3 := x
```

is sufficient. Next we come to `temp4` and similar to `temp3` the appropriate code is

```
        temp4 := y
```

Finally, considering `temp2`, of course

```
        temp2 := x
```

suffices.

Each code fragment is output when we leave the corresponding node; this results in the final program:

```
        temp3 := x
        temp4 := y
        temp1 := temp3 * temp4
        temp2 := x
        z := temp1 + temp2
```

Comment. Notice how a compound expression has been broken down and translated into a sequence of very simple instructions, and furthermore, the process of producing the TAC code was uniform and simple. Some redundancy has been brought into the TAC code but this can be removed (along with redundancy that is not due to the TAC-generation) in the optimisation phase.

1.3.7 Code Optimisation

An *optimiser* attempts to improve the time and space requirements of a program. There are many ways in which code can be optimised, but most are expensive in terms of time and space to implement.

Common optimisations include:

- removing redundant identifiers,
- removing unreachable sections of code,
- identifying common subexpressions,
- unfolding loops and

- eliminating procedures.

Note that here we are concerned with the general optimisation of abstract code.

Example. Consider the TAC code:

```
     temp1 := x
     temp2 := temp1
     if temp1 = temp2 goto 200
     temp3 := temp1 * y
     goto 300
200  temp3 := z
300  temp4 := temp2 + temp3
```

Removing redundant identifiers (just `temp2`) gives

```
     temp1 := x
     if temp1 = temp1 goto 200
     temp3 := temp1 * y
     goto 300
200  temp3 := z
300  temp4 := temp1 + temp3
```

Removing redundant code gives

```
     temp1 := x
200  temp3 := z
300  temp4 := temp1 + temp3
```

Notes. Attempting to find a 'best' optimisation is expensive for the following reasons:

- A given optimisation technique may have to be applied repeatedly until no further optimisation can be obtained. (For example, removing one redundant identifier may introduce another.)
- A given optimisation technique may give rise to other forms of redundancy and thus *sequences* of optimisation techniques may have to be repeated. (For example, above we removed a redundant identifier and this gave rise to redundant code, but removing redundant code may lead to further redundant identifiers.)
- The order in which optimisations are applied may be significant. (How many ways are there of applying n optimisation techniques to a given piece of code?)

1.3.8 Code Generation

The final phase of the compiler is to generate code for a specific machine. In this phase we consider:

- memory management,
- register assignment and
- machine-specific optimisation.

The output from this phase is usually assembly language or relocatable machine code.

Example. The TAC code above could typically result in the ARM assembly program shown below. Note that the example illustrates a mechanical translation of TAC into ARM; it is not intended to illustrate compact ARM programming!

```
.x      EQUD  0             four bytes for x
.z      EQUD  0             four bytes for z
.temp   EQUD  0             four bytes each for temp1,
        EQUD  0             temp3, and
        EQUD  0             temp4.


.prog   MOV   R12,#temp     R12 = base address
        MOV   R0,#x         R0 = address of x
        LDR   R1,[R0]       R1 = value of x
        STR   R1,[R12]      store R1 at R12
        MOV   R0,#z         R0 = address of z
        LDR   R1,[R0]       R1 = value of z
        STR   R1,[R12,#4]   store R1 at R12+4
        LDR   R1,[R12]      R1 = value of temp1
        LDR   R2,[R12,#4]   R2 = value of temp3
        ADD   R3,R1,R2      add temp1 to temp3
        STR   R3,[R12,#8]   store R3 at R12+8
```

2 Languages

In this section we introduce the formal notion of a language, and the basic problem of recognising strings from a language. These are central concepts that we will use throughout the remainder of the course.

Note. *This section contains mainly theoretical definitions; the lectures will cover examples and diagrams illustrating the theory.*

2.1 Basic Definitions

- An *alphabet* Σ is a finite non-empty set (of *symbols*).
- A *string* or *word* over an alphabet Σ is a finite *concatenation* (or *juxtaposition*) of symbols from Σ.
- The *length* of a string w (that is, the number of characters comprising it) is denoted $|w|$.
- The *empty* or *null* string is denoted ϵ. (That is, ϵ is the unique string satisfying $|\epsilon| = 0$.)
- The set of all strings over Σ is denoted Σ^*.
- For each $n \geq 0$ we define
$$\Sigma^n = \{w \in \Sigma^* \mid |w| = n\}.$$
- We define
$$\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$
(Thus $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.)
- For a symbol or word x, x^n denotes x concatenated with itself n times, with the convention that x^0 denotes ϵ.
- A *language over* Σ is a set $L \subseteq \Sigma^*$.
- Two languages L_1 and L_2 over common alphabet Σ are equal if they are equal as sets. Thus $L_1 = L_2$ if, and only if, $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$.

2.2 Decidability

Given a language L over some alphabet Σ, a basic question is: For each possible word $w \in \Sigma^*$, can we *effectively* decide if w is a member of L or not? We call this the *decision problem* for L.

Note the use of the word 'effectively': this implies the mechanism by which we decide on membership (or non-membership) must be a finitistic, deterministic and mechanical procedure that can be carried out by some form of computing agent. Also note the decision problem asks if a given word is a member of L *or not*; that is, it is not sufficient to be only able to decide when words *are* members of L.

More precisely then, a language $L \subseteq \Sigma^*$ is said to be *decidable* if there exists an algorithm such that for *every* $w \in \Sigma^*$

(1) the algorithm terminates with output 'Yes' when $w \in L$ and

(2) the algorithm terminates with output 'No' when $w \notin L$.

If no such algorithm exists then L is said to be *undecidable*.

Note. Decidability is based on the notion of an 'algorithm'. In standard theoretical computer science this is taken to mean a *Turing Machine*; this is an abstract, but extremely low-level model of computation that is equivalent to a digital computer with an *infinite* memory. Thus it is sufficient in practice to use a more convenient model of computation such as Pascal programs provided that any decidability arguments we make assume an infinite memory.

Example. Let $\Sigma = \{0, 1\}$ be an alphabet. Let L be the (infinite) language

$$L = \{w \in \Sigma^* \mid w = 0^n 1 \text{for some } n\}.$$

Does the program below solve the decision problem for L?

```
read( char );
if char = END_OF_STRING then
  print( "No" )
else     /* char must be '0' or '1' */
  while char = '0' do
    read( char )
  od;    /* char must be '1' or END_OF_STRING */
  if char = '1' then
    print( "Yes" )
  else
    print( "No" )
  fi
fi
```

Answer: ....

2.3 Basic Facts

(1) Every finite language is decidable. (Hence every undecidable language is infinite.)

(2) Not every infinite language is undecidable.

(3) Programming languages are (usually) infinite but (always) decidable. (*Why?*)

2.4 Applications to Compilation

Languages may be classified by the means in which they are defined. Of interest to us are *regular languages* and *context-free languages*.

Regular Languages. The significant aspects of regular languages are:

- they are defined by ‘patterns’ called *regular expressions*;
- every regular language is decidable;
- the decision problem for any regular language is solved by a *deterministic finite state automaton* (DFA); and
- programming languages’ lexical patterns are specified using regular expressions, and lexical analysers are (essentially) DFAs.

Regular languages and their relationship to lexical analysis are the subjects of the next section.

Context-Free Languages. The significant aspects of context-free languages are:

- they are defined by ‘rules’ called *context-free grammars*;
- every context-free language is decidable;
- the decision problem for any context-free language of interest to us is solved by a *deterministic push-down automaton* (DPDA);
- programming language syntax is specified using context-free grammars, and (most) parsers are (essentially) DPDAs.

Context-free languages and their relationship to syntax analysis are the subjects of sections 4 and 5.

3 Lexical Analysis

In this section we study some theoretical concepts with respect to the class of *regular languages* and apply these concepts to the practical problem of lexical analysis. Firstly, in Section 3.1, we define the notion of a *regular expression* and show how regular expressions determine regular languages. We then, in Section 3.2, introduce *deterministic finite automata* (DFAs), the class of algorithms that solve the decision problems for regular languages. We show how regular expressions and DFAs can be used to specify and implement lexical analysers in Section 3.3, and in Section 3.4 we take a brief look at Lex, a popular lexical analyser generator built upon the theory of regular expressions and DFAs.

Note. *This section contains mainly theoretical definitions; the lectures will cover examples and diagrams illustrating the theory.*

3.1 Regular Expressions

Recall from the Introduction that a lexical analyser uses *pattern matching* with respect to rules associated with the source language's tokens. For example, the token *then* is associated with the pattern `t, h, e, n`, and the token *id* might be associated with the pattern "an alphabetic character followed by any number of alphanumeric characters". The notation of regular expressions is a mathematical formalism ideal for expressing patterns such as these, and thus ideal for expressing the lexical structure of programming languages.

3.1.1 Definition

Regular expressions represent patterns of strings of symbols. A regular expression r *matches* a set of strings over an alphabet. This set is denoted $L(r)$ and is called the language *determined* or *generated* by r.

Let Σ be an alphabet. We define the set $RE(\Sigma)$ of *regular expressions over* Σ, the strings they match and thus the languages they determine, as follows:

- $\emptyset \in RE(\Sigma)$ matches no strings. The language determined is $L(\emptyset) = \emptyset$.
- $\epsilon \in RE(\Sigma)$ matches only the empty string. Therefore $L(\epsilon) = \{\epsilon\}$.
- If $a \in \Sigma$ then $a \in RE(\Sigma)$ matches the string a. Therefore $L(a) = \{a\}$.
- if r and s are in $RE(\Sigma)$ and determine the languages $L(r)$ and $L(s)$ respectively, then
  - $r|s \in RE(\Sigma)$ matches all strings matched either by r or by s. Therefore, $L(r|s) = L(r) \cup L(s)$.

- $rs \in RE(\Sigma)$ matches any string that is the concatenation of two strings, the first matching r and the second matching s. Therefore, the language determined is

  $$L(rs) = L(r)L(s) = \{uv \in \Sigma^* \mid u \in L(r) \text{ and } v \in L(s)\}.$$

  (Given two sets S_1 and S_2 of strings, the notation S_1S_2 denotes the set of all strings formed by appending members of S_1 with members of S_2.)
- $r^* \in RE(\Sigma)$ matches all finite concatenations of strings which all match r. The language denoted is thus

  $$\begin{aligned} L(r^*) = (L(r))^* &= \bigcup_{i \in \mathbf{N}} (L(r))^i \\ &= \{\epsilon\} \cup L(r) \cup L(r)L(r) \cup \cdots \end{aligned}$$

3.1.2 Regular Languages

Let L be a language over Σ. L is said to be a *regular language* if $L = L(r)$ for some $r \in RE(\Sigma)$.

3.1.3 Notation

- We need to use parentheses to overide the convention concerning the precedence of the operators. The normal convention is: $*$ is higher than concatenation, which is higher than $|$. Thus, for example, $a|bc^*$ is $a|(b(c^*))$.
- We write r^+ for rr^*.
- We write $r?$ for $\epsilon|r$.
- We write r^n as an abbreviation for $r \ldots r$ (n times r), with r^0 denoting ϵ.

3.1.4 Lemma

Writing '$r = s$' to mean '$L(r) = L(s)$' for two regular expressions $r, s \in RE(\Sigma)$, the following identities hold for all $r, s, t \in RE(\Sigma)$:

- $r|s = s|r$ ($|$ is commutative)
- $(r|s)|t = r|(s|t)$ ($|$ is associative)
- $(rs)t = r(st)$ (concatenation is associative)
- $r(s|t) = rs|rt$ (concatenation
- $(r|s)t = rt|st$ distributes over $|$)

- $\emptyset r = r\emptyset = \emptyset$
- $\emptyset | r = r$
- $\emptyset^* = \epsilon$
- $r?^* = r^*$
- $r^{**} = r^*$
- $(r^* s^*)^* = (r|s)^*$
- $\epsilon r = r\epsilon = r$.

3.1.5 Regular definitions

It is often useful to give names to complex regular expressions, and to use these names in place of the expressions they represent. Given an alphabet comprising all ASCII characters,

$$\begin{aligned} \mathit{letter} &= A|B|\cdots|Z|a|b|\cdots|z \\ \mathit{digit} &= 0|1|\cdots|9 \\ \mathit{ident} &= \mathit{letter}(\mathit{letter}|\mathit{digit})^* \end{aligned}$$

are examples of *regular definitions* for letters, digits and identifiers.

3.1.6 The Decision Problem for Regular Languages

For every regular expression $r \in RE(\Sigma)$ there exists a string-processing machine $M = M(r)$ such that for every $w \in \Sigma^*$, when input to M:

(1) if $w \in L(r)$ then M terminates with output 'Yes', and

(2) if $w \notin L(r)$ then M terminates with output 'No'.

Thus, every regular language is decidable.

The machines in question are Deterministic Finite State Automata.

3.2 Deterministic Finite State Automata

In this section we define the notion of a DFA without reference to its application in lexical analysis. Here we are interested purely in solving the decision problem for regular languages; that is, defining machines that say "yes" or "no" given an inputted string, depending on its membership of a particular language. In Section 3.3 we use DFAs as the basis for lexical analysers: pattern matching algorithms that output sequences of tokens.

3.2.1 Definition

A *deterministic finite state automaton* (or *DFA*) M is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite non-empty set of *states*,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to Q$ is the *transition* or *next-state function*,
- $q_0 \in Q$ is the *initial state*, and
- $F \subseteq Q$ is the set of *accepting* or *final states.*

The idea behind a DFA M is that it is an abstract machine that defines a language $L(M) \subseteq \Sigma^*$ in the following way:

- The machine begins in its start state q_0;
- Given a string $w \in \Sigma^*$ the machine reads the symbols of w one at a time from left to right;
- Each symbol causes the machine to make a transition from its current state to a new state; if the current state is q and the input symbol is a, then the new state is $\delta(q, a)$;
- The machine terminates when all the symbols of the string have been read;
- If, when the machine terminates, its state is a member of F, then the machine accepts w, else it rejects w.

Note the name 'final state' is not a good one since a DFA *does not terminate* as soon as a 'final state' has been entered. The DFA only terminates when all the input has been read.

We formalise this idea as follows:

Definition. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We define $\hat{\delta} : Q \times \Sigma^* \to Q$ by

$$\hat{\delta}(q, \epsilon) = q$$

for each $q \in Q$ and

$$\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$$

for each $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

We define the *language of* M by

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}.$$

3.2.2 Transition Diagrams

DFAs are best understood by depicting them as *transition diagrams*; these are directed graphs with nodes representing states and labelled arcs between states representing transitions.

A transition diagram for a DFA is drawn as follows:

(1) Draw a node labelled 'q' for each state $q \in Q$.

(2) For every $q \in Q$ and *every* $a \in \Sigma$ draw an arc labelled a from node q to node $\delta(q, a)$;

(3) Draw an unlabelled arc from outside the DFA to the node representing the initial state q_0;

(4) Indicate each final state by drawing a concentric circle around its node to form a double circle.

3.2.3 Examples

Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{1, 2, 3, 4\}$, $\Sigma = \{a, b\}$, $q_0 = 1$, $F = \{4\}$ and where δ is given by:

$$\begin{array}{rclcrcl} \delta(1, a) & = & 2 & & \delta(1, b) & = & 3 \\ \delta(2, a) & = & 3 & & \delta(2, b) & = & 4 \\ \delta(3, a) & = & 3 & & \delta(3, b) & = & 3 \\ \delta(4, a) & = & 3 & & \delta(4, b) & = & 4. \end{array}$$

From the transition diagram for M_1 it is clear that:

$$\begin{array}{rcl} L(M_1) & = & \{w \in \{a, b\}^* \mid \hat{\delta}(1, w) \in F\} \\ & = & \{w \in \{a, b\}^* \mid \hat{\delta}(1, w) = 4\} \\ & = & \{ab, abb, abbb, \ldots, ab^n, \ldots\} \\ & = & L(ab^+). \end{array}$$

Let M_2 be obtained from M_1 by adding states 1 and 2 to F. Then

$$L(M_2) = L(\epsilon|ab^*).$$

Let M_3 be obtained from M_1 by changing F to $\{3\}$. Then

$$L(M_3) = L((b|aa|abb^*a)(a|b)^*).$$

Simplifications to transition diagrams.

- It is often the case that a DFA has an 'error state', that is, a non-accepting state from which there are no transitions other than back to the error state. In such a case it is convenient to apply the convention that any apparently missing transitions are transitions to the error state.
- It is also common for there to be a large number of transitions between two given states in a DFA, which results in a cluttered transition diagram. For example, in an identifier recognition DFA, there may be 52 arcs labelled with each of the lower- and upper-case letters from the start state to a state representing that a single letter has been recognised. It is convenient in such cases to define a set comprising the labels of each of these arcs, for example,
$$\mathit{letter} = \{a, b, c, \ldots, z, A, B, C, \ldots, Z\} \subseteq \Sigma$$
and to replace the arcs by a single arc labelled by the name of this set, e.g. *letter*.

It is acceptable practice to use these conventions provided it is made clear that they are being operated.

3.2.4 Equivalence Theorem

(1) *For every $r \in RE(\Sigma)$ there exists a DFA M with alphabet Σ such that $L(M) = L(r)$.*

(2) *For every DFA M with alphabet Σ there exists an $r \in RE(\Sigma)$ such that $L(r) = L(M)$.*

Proof. See J.E. Hopcroft and J. D. Ullman Introduction to Automata Theory, Languages, and Computation (Addison Wesley, 1979).

Applications. The significance of the Equivalence Theorem is that its proof is *constructive*; there is an algorithm that, given a regular expression r, builds a DFA M such that $L(M) = L(r)$. Thus, if we can write a fragment of a programming language syntax in terms of regular expressions, then by part (1) of the Theorem we can automatically construct a lexical analyser for that fragment.

Part (2) of the Equivalence Theorem is a useful tool for showing that a language is regular, since if we cannot find a regular expression directly, part (2) states that it is sufficient to find a DFA that recognises the language.

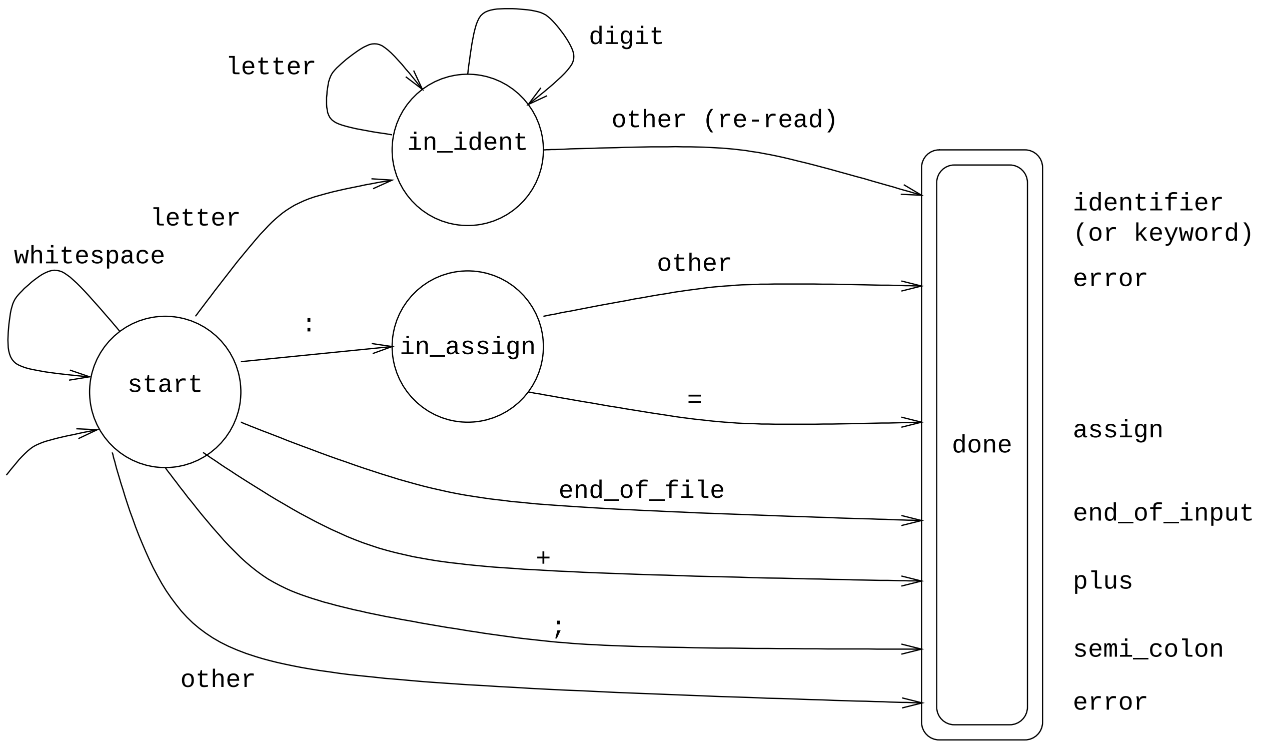
The standard algorithm for constructing a DFA from a given regular expression is not difficult, but would require that we also take a look at nondeterministic finite state automata (NFAs). NFAs are equivalent in power to DFAs but are slightly harder to understand (see the course text for details). Given a regular expression, the RE-DFA algorithm first constructs an NFA equivalent to the RE (by a method known as Thompson's Construction), and then transforms the NFA into an equivalent DFA (by a method known as the Subset Construction).

3.3 DFAs for Lexical Analysis

Let's suppose we wish to construct a lexical analyser based on a DFA. We have seen that it is easy to construct a DFA that recognises lexemes for a given programming language token (e.g. for individual keywords, for identifiers, and for numbers). However, a lexical analyser has to deal with all of a programming language's lexical patterns, and has to repeatedly match sequences of characters against these patterns and output corresponding tokens. We illustrate how lexical analysers may be constructed using DFAs by means of an example.

3.3.1 An example DFA lexical analyser

Consider first writing a DFA for recognising tokens for a (minimal!) language with identifiers and the symbols `+`, `;` and `:=` (we'll add keywords later). A transition diagram for a DFA that recognises these symbols is given by:



The lexical analyser code (see next section) consists of a procedure `get_next_token` which outputs the next token, which can be either `identifier` (for identifiers), `plus` (for `+`), `semi_colon` (for `;`), `assign` (for `:=`), `error` (if an error occurs, for example if an invalid character such as '`(`' is read or if a `:` is not followed by a `=`) and `end_of_input` (for when the complete input file has been read).

The (lexical analyser based on) the DFA begins in state `start`, and returns a token when it enters state `done`; the token returned depends on the final transition it takes to enter the `done` state, and is shown on the right hand side of the diagram.

For a state with an output arc labelled `other`, the intuition is that this transition is made on reading any character except those labelled on the state's other arcs; `re-read` denotes that the read character should not be consumed—it is re-read as the first character when `get_next_token` is next called.

Notice that adding keyword recognition to the above DFA would be tricky to do by hand and would lead to a complex DFA—why? However, we can recognise keywords as identifiers using the above DFA, and when the accepting state for identifiers is entered the lexeme stored in the buffer can be checked against a table of keywords. If there is a match, then the appropriate keyword token is output, else `identifier` token is output.

Further, when we have recognised an identifier we will also wish to output its string value as well as the `identifier` token, so that this can be used by the next phase of compilation.

3.3.2 Code for the example DFA lexical analyser

Let's consider how code may be written based on the above DFA. Let's add the keywords `if`, `then`, `else` and `fi` to the language to make it slightly more realistic.

Firstly, we define enumerated types for the sets of states and tokens:

```
state = (start, in_identifier, in_assign, done);
token = (k_if, k_then, k_else, k_fi, plus, identifier, assign,
         semi_colon, error, end_of_input);
```

Next, we define some variables that are shared by the lexical analyser and the syntax analyser. The job of the procedure `get_next_token` is to set the value of `current_token` to next token, and if this token is `identifier` it also sets the value of `current_identifier` to the current lexeme. The value of the Boolean variable `reread_character` determines whether the last character read during the previous execution of `get_next_token` should be re-read at the beginning of its next execution. The `current_character` variable holds the value of the last character read by `get_next_token`.

```
current_token : token;
current_identifier : string[100];
reread_character : boolean;
current_character : char;
```

We also need the following auxiliary functions (their implementations are omitted here) with the obvious interpretations:

```
function is_alpha(c : char) : boolean;
function is_digit(c : char) : boolean;
function is_white_space(c : char) : boolean;
```

Finally, we define two constant arrays:

```
{ Constants used to recognise keyword matches }
NUM_KEYWORDS = 4;
token_tab : array[1..NUM_KEYWORDS] of token = (k_if, k_then, k_else, k_fi);
keyword_tab : array[1..NUM_KEYWORDS] of string = ('if', 'then','else', 'fi');
```

that store keyword tokens and keywords (with associated keywords and tokens stored at the same location each array) and a function that searches the keyword array for a string and returns the token associated with a matched keyword or the token `identifier` if not. Notice that the arrays and function are easily modified for any number of keywords appearing in our source language.

```
function keyword_lookup(s : string) : token;
{If s is a keyword, return this keyword's token; else
return the identifier token}
var
  index : integer;
  found : boolean;
begin
  keyword_lookup := identifier;
  found := FALSE;
  index := 1;
  while (index <= NUM_KEYWORDS) and (not found) do begin
    if keyword_tab[index] = s then begin
      keyword_lookup := token_tab[index];
      found := TRUE
    end;
    index := index + 1
  end
end;
```

The `get_next_token` procedure is implemented as follows. Notice that within the main loop a case statement is used to deal with transitions from the current state. After the loop exits (when the `done` state is entered), if an `identifier` has been recognised, `keyword_lookup` is used to check whether or not a keyword has been matched.

```
procedure get_next_token;
{Sets the value of current_token by matching input characters. Also,
sets the values current_identifier and reread_character if
appropriate}
var
  current_state : state;
  no_more_input : boolean;
begin
  current_state := start;
  current_identifier := '';
  while not (current_state = done) do begin
    no_more_input := eof; {Check whether at end of file}
    if not (reread_character or no_more_input) then
      read(current_character);
    reread_character := FALSE;
    case current_state of
      start:
        if no_more_input then begin
          current_token := end_of_input;
          current_state := done
        end else if is_white_space(current_character) then
          current_state := start
        else if is_alpha(current_character) then begin
          current_identifier := current_identifier + current_character;
          current_state := in_identifier
        end else case current_character of
          ';' : begin
            current_token := semi_colon;
            current_state := done
          end;
          '+' : begin
            current_token := plus;
            current_state := done
          end;
          ':' :
            current_state := in_assign
          else begin
            current_token := error;
            current_state := done;
          end
        end; {case}
      in_identifier:
        if (no_more_input or not(is_alpha(current_character)
            or is_digit(current_character))) then begin
          current_token := identifier;
          current_state := done;
          reread_character := true
        end else
          current_identifier := current_identifier + current_character;
      in_assign:
        if no_more_input or (current_character <> '=') then begin
          current_token := error;
          current_state := done
        end else begin
           current_token := assign;
           current_state := done
        end
    end; {case}
  end; {while}
  if (current_token = identifier) then
    current_token := keyword_lookup(current_identifier);
end;
```

Test code (in the absence of a syntax analyser) might be the following. This just repeatedly calls `get_next_token` until the end of the input file has been reached, and prints out the value of the read token.

```
{Request tokens from lexical analyser, outputting their
values, until end_of_input}
begin
   reread_character := false;
   repeat
      get_next_token;
      writeln('Current Token is ', token_to_text(current_token));
      if (current_token = identifier) then
         writeln('Identifier is ', current_identifier);
   until (current_token = end_of_input)
end.
```

where

```
function token_to_text(t : token) : string;
```

converts token values to text.

3.4 Lex

Lex is a widely available lexical analyser *generator*.

3.4.1 Overview

Given a Lex source file comprising regular expressions for various tokens, Lex generates a lexical analyser (based on a DFA), written in C, that groups characters matching the expressions into lexemes, and can return their corresponding tokens.

In essence, a Lex file comprises a number of lines typically of the form:

pattern   *action*

where *pattern* is a regular expression and *action* is a piece of C code.

When run on a Lex file, Lex produces a C file called `lex.yy.c` (a lexical analyser).

When compiled, `lex.yy.c` takes a stream of characters as input and whenever a sequence of characters matches a given regular expression the corresponding action is executed. Characters not matching any regular expressions are simply copied to the output stream.

Example. Consider the Lex fragment:

```
a     { printf( "read 'a'\n" ); }
b     { printf( "read 'b'\n" ); }
```

After compiling (see below on how to do this) we obtain a binary executable which when executed on the input:

```
sdfghjklaghjbfghjkbbdfghjk
dfghjkaghjklaghjk
```

produces

```
sdfghjklread 'a'
ghjread 'b'
fghjkread 'b'
read 'b'
dfghjk
dfghjkread 'a'
ghjklread 'a'
ghjk
```

Example. Consider the Lex program:

```
%{
int abc_count, xyz_count;
%}

%%

ab[cC]    {abc_count++; }
xyz       {xyz_count++; }
\n        { ; }
.         { ; }

%%

main()
{
  abc_count = xyz_count = 0;
  yylex();
  printf( "%d occurrences of abc or abC\n", abc_count );
  printf( "%d occurrences of xyz\n", xyz\_count );
}
```

- This file first declares two global variables for counting the number of occurrences of `abc` or `abC` and `xyz`.
- Next come the regular expressions for these lexemes and actions to increment the relevant counters.
- Finally, there is a `main` routine to initialise the counters and call `yylex()`.

When executed on input:

```
akhabfabcdbcaxyzXyzabChsdk
dfhslkdxyzabcabCdkkjxyzkdf
```

the lexical analyser produces:

```
4 occurrences of 'abc' or 'abC'
3 occurrences of 'xyz'
```

Some features of Lex illustrated by this example are:

(1) The notation for |; for example, `[cC]` matches either `c` or `C`.

(2) The regular expression `\n` which matches a newline.

(3) The regular expression `.` which matches any character except a newline.

(4) The action `{ ; }` which does nothing except to suppress printing.

3.4.2 Format of Lex Files

The format of a Lex file is:
definitions
analyser specification
auxiliary functions

Lex Definitions. The (optional) definitions section comprises *macros* (see below) and *global declarations* of types, variables and functions to be used in the actions of the lexical analyser and the auxiliary functions (if present). All such global declaration code is written in C and surrounded by `%{` and `%}`.

Macros are abbreviations for regular expressions to be used in the analyser specification. For example, the token 'identifier' could be defined by:

```
IDENTIFIER      [a-zA-Z][a-zA-Z0-9]*
```

The shorthand *character range* construction '`[`x`-`y`]`' matches any of the characters between (and including) x and y. For example, `[a-c]` means the same as `a|b|c`, and `[a-cA-C]` means the same as `a|b|c|A|B|C`.

Definitions may use other definitions (enclosed in braces) as illustrated in:

```
ALPHA       [a-zA-Z]
ALPHANUM    [a-zA-Z0-9]
IDENTIFIER  {ALPHA}{ALPHANUM}*
```

and:

```
ALPHA      [a-zA-Z]
NUM        [0-9]
ALPHANUM   ({ALPHA}|{NUM})
IDENTIFIER {ALPHA}{ALPHANUM}*
```

Notice the use of parentheses in the definition of `ALPHANUM`. What would happen without them?

Lex Analyser Specifications. These have the form:

r_1 { *action*$_1$ }
r_2 { *action*$_2$ }
... ...
r_n { *action*$_n$ }

where $r_1, r_2, \ldots, r_n$ are regular expressions (possibly involving macros enclosed in braces) and *action*$_1$, *action*$_2$, ..., *action*$_n$ are sequences of C statements.

Lex translates the specification into a function `yylex()` which, when when called, causes the following to happen:

- The current input character(s) are scanned to look for a match with the regular expressions.
- If there is no match, the current character is printed out, and the scanning process resumes with the next character.
- If the next m characters match r_i then
  (a) the matching characters are assigned to string variable `yytext`,
  (b) the integer variable `yyleng` is assigned the value m,
  (c) the next m characters are skipped, and
  (d) *action*$_i$ is executed. If the last instruction of *action*$_i$ is `return n;` (where `n` is an integer expression) then the call to `yylex()` terminates and the value of `n` is returned as the function's value; otherwise `yylex()` resumes the scanning process.
- If end-of-file is read at any stage, then the call to `yylex()` terminates returning the value 0.
- If there is a match against two or more regular expressions, then the expression giving the longest lexeme is chosen; if all lexemes are of the same length then the first matching expression is chosen.

Lex Auxiliary Functions. This optional section has the form:

fun_1
fun_2
...
fun_n

where each fun_i is a complete C function.

We can also compile `lex.yy.c` with the lex library using the command:

```
gcc lex.yy.c -ll
```

This has the effect of automatically including a standard `main()` function, equivalent to:

```
main()
{
  yylex();
  return;
}
```

Thus in the absence of any `return` statements in the analyser's actions, this one call to `yylex()` consumes all the input up to and including end-of-file.

3.4.3 Lexical Analyser Example

The lex program below illustrates how a lexical analyser for a Pascal-type language is defined. Notice that the regular expression for identifiers is placed at the end of the list (why?).

We assume that the syntax analyser requests tokens by repeatedly calling the function `yylex()`. The global variable `yylval` (of type integer in this example) is generally used to pass tokens' attributes from the lexical analyser to the syntax analyser and is shared by both phases of the compiler. Here it is being used to pass integers' values and identifiers' symbol table positions to the syntax analyser.

```
%{
   definitions (as integers) of IF, THEN, ELSE, ID, INTEGER, ...
%}

delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
integer   [+\-]?{digit}+

%%

{ws}      { ; }
if        { return(IF); }
then      { return(THEN); }
else      { return(ELSE); }
...       ...
{integer} { yylval = atoi(yytext); return(INTEGER); }
{id}      { yylval = InstallInTable(); return(ID); }

%%

int InstallInTable()
{
   put yytext in symbol table and return
   the position it has been inserted.
}
```

4 Syntax Analysis

In this section we will look at the second phase of compilation: syntax analysis, or parsing. Since parsing is a central concept in compilation and because (unlike lexical analysis) there are many approaches to parsing, this section makes up most of the remainder of the course.

In Section 4.1 we discuss the class of *context-free languages* and its relationship to the syntactic structure of programming languages and the compilation process. Parsing algorithms for context-free languages fall into two main categories: *top-down* and *bottom-up* parsers (the names refer to the process of parse tree construction). Different types of top-down and bottom-up parsing algorithms will be discussed in Sections 4.2 and 4.3 respectively.

4.1 Context-Free Languages

Regular languages are inadequate for specifying all but the simplest aspects of programming language syntax. To specify more-complex languages such as

- $L = \{w \in \{a, b\}^* \mid w = a^n b^n \text{ for some } n\}$,
- $L = \{w \in \{(,)\}^* \mid w \text{ is a well-balanced string of parentheses}\}$ and
- the syntax of most programming languages,

we use context-free languages. In this section we define context-free grammars and languages, and their use in describing the syntax of programming languages. This section is intended to provide a foundation for the following sections on parsing and parser construction.

Note Like Section 3, this section contains mainly theoretical definitions; the lectures will cover examples and diagrams illustrating the theory.

4.1.1 Context-Free Grammars

Definition. A *context-free grammar* is a tuple $G = (T, N, S, P)$ where:

- T is a finite nonempty set of (*terminal*) symbols (tokens),
- N is a finite nonempty set of (*nonterminal*) symbols (denoting phrase types) disjoint from T,
- $S \in N$ (the *start* symbol), and
- P is a set of (*context-free*) *productions* (denoting 'rules' for phrase types) of the form $A \to \alpha$ where $A \in N$ and $\alpha \in (T \cup N)^*$.

Notation. In what follows we use:

- $a, b, c, \ldots$ for members of T,
- $A, B, C, \ldots$ for members of N,
- $\ldots, X, Y, Z$ for members of $T \cup N$,
- $u, v, w, \ldots$ for members of T^*, and
- $\alpha, \beta, \gamma, \ldots$ for members of $(T \cup N)^*$.

Examples.

(1) $G_1 = (T, N, S, P)$ where
  - $T = \{a, b\}$,
  - $N = \{S\}$ and
  - $P = \{S \to ab, S \to aSb\}$.

(2) $G_2 = (T, N, S, P)$ where
  - $T = \{a, b\}$,
  - $N = \{S, X\}$ and
  - $P = \{S \to X, S \to aa, S \to bb, S \to aSa, S \to bSb, X \to a, X \to b\}$.

Notation. It is customary to 'define' a context free grammar by simply listing its productions and assuming:

- The terminals and nonterminals of the grammar are exactly those terminals appearing in the productions. (It is usually clear from the context whether a symbol is a terminal or nonterminal.)
- The start symbol is the nonterminal on the left-hand side of the first production.
- Right-hand sides separated by '|' indicate alternatives.

For example, G_2 above can be written as

$$\begin{array}{rcl} S & \to & X \mid aa \mid bb \mid aSa \mid bSb \\ X & \to & a \mid b \end{array}$$

BNF Notation. The definition of a grammar as given so far is fine for simple theoretical grammars. For grammars of real programming languages, a popular notation for context-free grammars is *Backus-Naur Form.* Here, non-terminal symbols are given a descriptive name and placed in angled brackets, and "|" is used, as above, to indicate alternatives. The alphabet will also commonly include keywords and language symbols such as `<=`. For example, BNF for simple arithmetic expressions might be

$$\begin{array}{rcl} <\mathit{exp}> & \to & <\mathit{exp}> + <\mathit{exp}> \quad | \quad <\mathit{exp}> - <\mathit{exp}> \quad | \\ & & <\mathit{exp}> * <\mathit{exp}> \quad | \quad <\mathit{exp}> \,/\, <\mathit{exp}> \quad | \\ & & (<\mathit{exp}>) \quad | \quad <\mathit{number}> \\ <\mathit{number}> & \to & <\mathit{digit}> \quad | \quad <\mathit{digit}><\mathit{number}> \\ <\mathit{digit}> & \to & 0 \;|\; 1 \;|\; 2 \;|\; 3 \;|\; 4 \;|\; 5 \;|\; 6 \;|\; 7 \;|\; 8 \;|\; 9. \end{array}$$

and a typical rule for while loops would be:

$$\begin{array}{rcl} <\mathit{while\ loop}> & \to & \texttt{while} <\mathit{exp}> \texttt{do} <\mathit{command\ list}> \texttt{od} \\ <\mathit{command\ list}> & \to & <\mathit{assignment}> \quad | \quad <\mathit{conditional}> \ \ldots \end{array}$$

Note that the symbol `::=` is often used in place of $\to$.

Derivation and Languages. A (context-free) grammar G defines a language $L(G)$ in the following way:

- We say $\alpha A \beta$ *immediately derives* $\alpha\gamma\beta$ if $A \to \gamma$ is a production of G. We write $\alpha A \beta \Longrightarrow \alpha\gamma\beta$.
- We say α *derives* β *in zero or more steps*, in symbols $\alpha \overset{*}{\Longrightarrow} \beta$, if
  - (i) $\alpha = \beta$ or
  - (ii) for some γ we have $\alpha \overset{*}{\Longrightarrow} \gamma$ and $\gamma \Longrightarrow \beta$.
- If $S \overset{*}{\Longrightarrow} \alpha$ then we say α is a *sentential form.*
- We say α *derives* β *in one or more steps*, in symbols $\alpha \overset{+}{\Longrightarrow} \beta$, if $\alpha \overset{*}{\Longrightarrow} \beta$ and $\alpha \neq \beta$.
- We define $L(G)$ by
$$L(G) = \{w \in T^* \mid S \overset{+}{\Longrightarrow} w\}.$$
- Let $L \subseteq T^*$. L is said to be a *context-free language* if $L = L(G)$ for some context-free grammar G.

4.1.2 Regular Grammars

A grammar is said to be regular if all its productions are of the form:

(a) $A \to uB$ or $A \to u$ (a right-linear grammar)

(b) $A \to Bu$ or $A \to u$ (a left-linear grammar).

A language $L \subseteq T^*$ is regular (see Section 3) if $L = L(G)$ for some regular grammar G.

4.1.3 ϵ-Productions

A grammar may involve a production of the form

$$A \to$$

(that is, with an empty right-hand side). This is called an *ϵ-production* and is often written $A \to \epsilon$. ϵ-productions are often useful in specifying languages although they can cause difficulties (*ambiguities*), especially in parser design, and may need to be removed–see later.

As an example of the use of ϵ-productions,

$$< stmt > \to \ \texttt{begin} \ < stmt > \ \texttt{end} \ < stmt > \quad | \quad \epsilon$$

generates well-balanced '`begin/end`' strings (and the empty string).

4.1.4 Left- and Rightmost Derivations

A derivation in which the leftmost (rightmost) nonterminal is replaced at each step in the derivation is called a *leftmost* (*rightmost*) *derivation.*

Examples. Let G be a grammar with productions

$$E \to E + E \mid E * E \mid x \mid y \mid z.$$

(a) A leftmost derivation of $x + y * z$ is

$$E \Longrightarrow E + E \Longrightarrow x + E \Longrightarrow x + E * E \Longrightarrow x + y * E \Longrightarrow x + y * z.$$

(b) A rightmost derivation of $x + y * z$ is

$$E \Longrightarrow E + E \Longrightarrow E + E * E \Longrightarrow E + E * z \Longrightarrow E + y * z \Longrightarrow x + y * z.$$

(c) *Another* leftmost derivation of $x + y * z$ is

$$E \Longrightarrow E * E \Longrightarrow E + E * E \Longrightarrow x + E * E \Longrightarrow x + y * E \Longrightarrow x + y * z$$

(thus left- and rightmost derivations need not be unique).

(d) The following is neither left- nor rightmost:

$$E \Longrightarrow E + E \Longrightarrow E + E * E \Longrightarrow E + y * E \Longrightarrow x + y * E \Longrightarrow x + y * z.$$

4.1.5 Parse Trees

Let $G = (T, N, S, P)$ be a context-free grammar.

(1) A *parse* (or *derivation*) *tree* in grammar G is a pictorial representation of a derivation in which

  (a) every node has a label that is a symbol of $T \cup N \cup \{\epsilon\}$,

  (b) the root of the tree has label S,

  (c) interior nodes have nonterminal labels,

  (d) if node n has label A and descendants $n_1 \ldots n_k$ (in order, left-to-right) with labels $X_1 \ldots X_k$ respectively, then $A \rightarrow X_1 \cdots X_k$ must be a production of P, and

  (e) if node n has label ϵ then n is a leaf and is the only descendant of its parent.

(2) The *yield* of a parse tree with root node r is $y(r)$ where:

  (a) if n is a leaf node with label a then $y(n) = a$, and

  (b) if n is an interior node with descendents $n_1 \ldots n_k$ then

$$y(n) = y(n_1) \cdots y(n_k).$$

Theorem. *Let $G = (T, N, S, P)$ be a context-free grammar. For each $\alpha \in (T \cup N)^*$ there exists a parse tree in grammar G with yield α if, and only if, $S \overset{*}{\Longrightarrow} \alpha$.*

Examples.

1. Consider the parse trees for the derivations (a) and (c) of the above examples. Notice that the derivations are different and so are the parse trees, although both derivations are leftmost.

2. Let G be a grammar with productions

$$\begin{array}{lcl} <\mathit{stmt}> & \rightarrow & \texttt{if}\ <\mathit{boolexp}>\ \texttt{then}\ <\mathit{stmt}>\ \ | \\ & & \texttt{if}\ <\mathit{boolexp}>\ \texttt{then}\ <\mathit{stmt}>\ \texttt{else}\ <\mathit{stmt}>\ \ | \\ & & <\mathit{other}> \\ <\mathit{boolexp}> & \rightarrow & \ldots \end{array}$$

$$< other > \;\rightarrow\; \ldots$$

A classical problem in parsing is that of the *dangling else*: Should

$$\texttt{if} \; < boolexp > \; \texttt{then if} \; < boolexp > \; \texttt{then} \; < other > \texttt{else} \; < other >$$

be parsed as

$$\texttt{if} \; < boolexp > \; \texttt{then} \; (\; \texttt{if} \; < boolexp > \; \texttt{then} \; < other > \; \texttt{else} \; < other > \;)$$

or

$$\texttt{if} \; < boolexp > \; \texttt{then} \; (\; \texttt{if} \; < boolexp > \; \texttt{then} \; < other > \;) \; \texttt{else} \; < other >$$

Theorem. *Let $G = (T, N, S, P)$ be a context-free grammar and let $w \in L(G)$. For every parse tree of w there is precisely one leftmost derivation of w and precisely one rightmost derivation of w.*

4.1.6 Ambiguity

A grammar G for which there is some $w \in L(G)$ for which there are two (or more) parse trees is said to be *ambiguous*. In such a case, w is known as an *ambiguous sentence*. Ambiguity is usually an undesirable property since it may lead to different interpretations of the same string. Unfortunately:

Equivalently (by the above theorem), a grammar G is ambiguous if there is some $w \in L(G)$ for which there exists more than one leftmost derivation (or rightmost derivation).

Theorem. *The decision problem: Is G an unambiguous grammar? is undecidable.*

However, for a *particular* grammar we can sometimes resolve ambiguity via *disambiguating rules* that force troublesome strings to be parsed in a particular way. For example, let G be with productions

$$\begin{array}{rcl}
< stmt > & \rightarrow & < matched > \;\; | \;\; < unmatched > \\
< matched > & \rightarrow & \texttt{if} \; < boolexp > \; \texttt{then} \; < matched > \; \texttt{else} \; < matched > \;\; | \\
 & & < other > \\
< unmatched > & \rightarrow & \texttt{if} \; < boolexp > \; \texttt{then} \; < stmt > \;\; | \\
 & & \texttt{if} \; < boolexp > < then > < matched > \; \texttt{else} \; < unmatched > \\
< boolexp > & \rightarrow & \ldots \\
< other > & \rightarrow & \ldots
\end{array}$$

($< matched >$ is intended to denote an if-statement with a matching 'else' or some other (non-conditional) statement, whereas $< unmatched >$ denotes an if-statement with an unmatched 'if'.)

Now, according to this G,

$$\texttt{if} < \textit{boolexp} > \texttt{then if} < \textit{boolexp} > \texttt{then} < \textit{other} > \texttt{else} < \textit{other} >$$

has a unique parse tree. (exercise: convince yourself that this parse tree really is unique.)

4.1.7 Inherent Ambiguity

We have seen that it is possible to remove ambiguity from a grammar by changing the (non-terminals and) productions to give an equivalent (in terms of the language defined) but unambiguous grammar. However, there are some context-free languages for which *every* grammar is ambiguous; such languages are called *inherently ambiguous.*

Formally,

Theorem. *There exists a context-free language L such that for every context-free grammar G, if $L = L(G)$ then G is ambiguous.*

Furthermore,

Theorem. *The decision problem: Is L an inherently ambiguous language? is undecidable.*

4.1.8 Limits of Context-Free Grammars

It is known that not every language is context-free. For example,

$$L = \{w \in \{a, b, c\}^* \mid w = a^n b^n c^n \text{ for some } n \geq 1\}$$

is not context-free, nor is

$$L = \{w \in \{a, b, c\}^* \mid w = a^m b^n c^p \text{ for } m \leq n \leq p\}$$

although proof of these facts is beyond the scope of this course.

More significantly, there are some aspects of programming language syntax that cannot be captured with a context-free grammar. Perhaps the most well-know example of this is the rule that variables must be declared before they are used. Such properties are often called *context-sensitive* since they *can* be specified by grammars that have productions of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

which, informally, says that A can be replaced by γ but *only* when it occurs in the context $\alpha \ldots \beta$.

In practice we still use context-free grammars; problems are resolved by allowing strings (programs) that parse but are technically invalid, and then by adding further code that checks for invalid (context-sensitive) properties.

In this section we describe a number of techniques for constructing a parser for a given context-free language from a context-free grammar for that language. Parsers fall into two broad categories: *top-down parsers* and *bottom-up parsers*, as described in the next two sections.

4.2 Top-Down Parsers

Top-down parsing can be thought of as the process of creating a parse tree for a given input string by starting at the top with the start symbol and then working depth-first downwards to the leaves. Equivalently, it can be viewed as creating a leftmost derivation of the input string.

We begin, in Section 4.2.1, by showing how a simple parser may be coded in an ad-hoc way using a standard programming language such as Pascal. In Section 4.2.2 we define those grammars for which such a simple parser may be written. In Section 4.2.3 we show how to add syntax-tree construction and (very limited) error recovery code to these parsing algorithms. Finally, in Section 4.2.4 we show how this simple approach to parsing can be made more structured by using the same algorithm for all grammars, where the algorithm uses a different *look-up table* for each particular grammar.

4.2.1 Recursive Descent Parsing

The simplest idea for creating a parser for a grammar is to construct a *recursive descent parser* (r.d.p.). This is a top-down parser, and needs *backtracking* in general; i.e the parser needs to try out different production rules; if one production rule fails it needs to backtrack in the input string (and the parse tree) in order to try further rules. Since backtracking can be costly in terms of parsing time, and because other parsing methods (e.g. bottom-up parsers) do not require it, it is rare to write backtracking parsers in practice.

We will show here how to write a non-backtracking r.d.p. for a given grammar. Such parsers are commonly called *predictive parsers* since they attempt to predict which production rule to use at each step in order to avoid the need to backtrack; if when attempting to apply a production rule, unexpected input is read, then this input must be an error and the parser reports this fact. As we will see, is not possible to write such a simple parser for all grammars: for many programming languages' grammars such a parser cannot be written. For these we'll need more powerful parsing methods which we will look at later.

We will not concern ourselves here with the construction of a parse tree or syntax tree—we'll just attempt to write code that produces (implicitly, via procedure calls) a derivation of an input string (if the string is in the language given by the grammar) or reports an error (for an incorrect string). This code will form the basis of a complete syntax analyser—syntax tree construction and error recovery code will just be inserted into it at the correct places, as we'll see in Section 4.2.3.

We will write a r.d.p. for the grammar G with productions:

$$\begin{aligned} A &\rightarrow a \mid BCA \\ B &\rightarrow be \mid cA \\ C &\rightarrow d \end{aligned}$$

We will assume the existence of a procedure `get_next_token` which reads terminal symbols (or tokens) from the input, and places them into the variable `current_token`. The tokens for the above grammar are a, b, c, d and e. We'll also assume the tokens `end_of_input` (for end of string) and `error` for a non-valid input.

We'll also assume the following auxiliary procedure which tests whether the current token is what is expected and, if so, reads the next token; otherwise it calls an error routine. Later on we'll see more about error recovery, but for now assume that the procedure `error` simply outputs a message and causes the program to terminate.

```
procedure match(expected : token);
begin
  if current_token = expected then
    get_next_token
  else
    error
end;
```

The parser is constructed from three procedures, one for each non-terminal symbol. The aim of each procedure is to try to match any of the right-hand-sides of production rules for that non-terminal symbol against the next few symbols on the input string; if it fails to do this then there must be an error in the input string, and the procedure reports this fact.

Consider the non-terminal symbol A and the A-productions $A \rightarrow a$ and $A \rightarrow BCA$. The procedure `parse_A` attempts to match either an `a` or BCA. It first needs to decide which production rule to choose: it makes its choice based on the current input token. If this is a, it obviously uses the production rule $A \rightarrow a$ and is bound to succeed since it already knows it is going to find the a(!).

Notice that any string derivable from BCA begins with either a b or a c, because of the right hand sides of the two B-productions. Therefore, if either b or c is the current token, then `parse_A` will attempt to parse the next few input symbols using the rule $A \rightarrow BCA$. In order to attempt to match BCA it calls procedures `parse_B`, `parse_C` and `parse_A` in turn, each of which may result in a call to `error` if the input string is not in the language. If the current token is neither a, b or c the input cannot be derived from A and therefore `error` is called.

```
procedure parse_A
```

```
begin
  case current_token of
    a : match(a);
    b, c : begin
             parse_B; parse_C; parse_A
           end
    else
      error
  end
end;
```

The procedures `parse_B` and `parse_C` are constructed in a similar fashion.

```
procedure parse_B;
begin
  case current_token of
    b : begin
          match(b);
          match(e)
        end;
    c : begin
          match(c);
          parse_A
        end
    else
      error
  end
end;

procedure parse_C;
begin
  match(d)
end;
```

To execute the parser, we need to (i) make sure that the first token of the input string has been read into `current_token`, (ii) call `parse_A` since this is the start symbol, and if an error is not generated by this call (iii) check that the end of the input string has been reached. This is accomplished by

```
get_next_token;
parse_A;
```

```
if current_token = end_of_input then
  writeln('Success!')
else
  error
```

By executing the parser on a range of inputs that drive the algorithm through all possible branches (try the algorithm on *beda*, *bed* and *cbedada* for examples), it is easy to convince oneself that the algorithm is a neat and correct solution to the parsing problem for this particular grammar.

Notice how the algorithm

i. determines a sequence of productions that are equivalent to the construction of a parse tree that starts from the root and builds down to the leaves, and

ii. determines a leftmost derivation.

Although it is possible to construct compact, efficient recursive descent parsers by hand for some very large grammars, the method is somewhat ad hoc and more useful general-purpose methods are preferred.

We note that the method of constructing a r.d.p. will fail to produce a correct parser if the grammar is *left recursive*; that is if $A \overset{+}{\Longrightarrow} A\alpha$ for some nonterminal A–why?

Also notice that each for each nonterminal of the above grammar it is possible to predict which production (if any) is to be matched against, purely on the basis of the current input character.

This is clearly the case for the grammar G above, because as we have seen:

- To recognise an A: if the current symbol is a, we try the production rule $A \rightarrow a$; if the symbol is b or c, we try $A \rightarrow BCA$; otherwise we report an error;
- To recognise a B, if the current symbol is b, we try $B \rightarrow be$; if the symbol is cA we try $B \rightarrow cA$; otherwise we report an error;
- To recognise a C, if the current symbol is d, we try $C \rightarrow d$; otherwise we report an error.

The choice is determined by the so-called $First$ sets of the right-hand sides of the production rules. In general, $First(\alpha)$ for any string $\alpha \in (T \cup N)^*$ denotes the set of all terminal symbols that can begin strings derivable from α. For example, the $First$ set of the right hand side of the rules $A \rightarrow a$ and $A \rightarrow BCA$ are $First(a) = \{a\}$ and $First(BCA) = \{b, c\}$ respectively. Notice that, to be able to write a predictive parser for a grammar, for any non-terminal symbol, the $First$ sets for all pairs of distinct productions' right-hand sides must be disjoint. This is the case for G, since

$$\begin{aligned} First(a) \cap First(BCA) &= \{a\} \cap \{b, c\} \\ &= \emptyset \\ First(be) \cap First(cA) &= \{b\} \cap \{c\} \\ &= \emptyset. \end{aligned}$$

(and C has no pairs of productions since there is only one C-production rule).

For the grammar G_2, with productions

$$\begin{aligned} A &\rightarrow b \mid BCA \\ B &\rightarrow be \mid cA \\ C &\rightarrow d \end{aligned}$$

this is not the case, since

$$\begin{aligned} First(b) \cap First(BCA) &= \{b\} \cap \{b, c\} \\ &= \{b\} \\ &\neq \emptyset \end{aligned}$$

and therefore we can not write a predictive parser for G_2.

Things become slightly more complicated when grammars have ε productions. For example, let the grammar G_3 have productions:

$$\begin{aligned} A &\rightarrow aBb \\ B &\rightarrow c \mid \varepsilon. \end{aligned}$$

Consider parsing the string adb. Obviously, parse_A would be written:

```
procedure parse_A
begin
  if current_token = a then begin
    match(a); parse_B; match(b)
  end else
    error
end;
```

How do we write a procedure parse_B that would recognise strings derivable from B? Well, consider what the parser should do for the following three strings:

- *acb*. The procedure parse_A would consume the first a and then call parse_B. We would want parse_B to try the rule $B \rightarrow c$ to consume the c, to leave parse_A to consume the final b and thus result in a successful parse.

- *ab*. Again `parse_A` would consume the *a*. Now, we would like `parse_B` to apply $B \rightarrow \varepsilon$ (which does not consume any input), leaving `parse_A` to consume the *b* (success again).
- *aa*. Again `parse_A` would consume the *a*. Now, obviously `parse_B` should not try $B \rightarrow c$ (since the next symbol is *a*), nor should it try $B \rightarrow \varepsilon$ since there is no way in which the next symbol *a* is going to be matched—therefore it should (correctly) report an error.

The choice then, of whether to use the ε production depends on whether the next symbol can legally immediately follow a B in a sentential form. The set of all such symbols is denoted $Follow(B)$. For G_2, $Follow(B) = \{b\}$ and thus the code for `parse_B` should be:

```
procedure parse_B;
begin
  case current_token of
    c : match(c);
    b : (* do nothing *)
    else
      error
  end
end;
```

(Notice that the *b* is not consumed). Finally, let G_4 be a grammar with productions

$$\begin{array}{rcl} A & \rightarrow & aBb \\ B & \rightarrow & b \mid \varepsilon \end{array}$$

and consider parsing the strings *ab* and *abb*. Obviously, `parse_A` will be as above, and thus would consume the *a* in both strings. However, it is impossible for a procedure `parse_B` to choose, purely on the basis of the current character (*b* in both cases) whether to try the production $B \rightarrow b$ (which would be correct for *abb* but not for *ab*) or $B \rightarrow \varepsilon$ (which would be correct for *ab* but not for *abb*).

Formally, the problem is that ε is in the *First* set one of the B-production's right hand sides (we use the convention that $First(\varepsilon) = \{\varepsilon\}$) and that one of the other *First* sets—$First(b)$—of a righthand side is not disjoint from the set $Follow(B)$. Indeed, both sets are $\{b\}$. This condition of disjointedness must be fullfilled to be able to write a predictive parser.

Formally, grammars for which predictive parsers can be constructed are called *LL(1) grammars*. (The first 'L' is for *left*-to-right reading of the input, the second 'L' is for *leftmost* derivation, and the '1' is for *one* symbol of lookahead.) LL(1) grammars are the most widely-used examples of LL(k) grammars: those that require k symbols of lookahead. In the following section we define what it means for a grammar to be LL(1) based on the concept of *First* and *Follow* sets.

4.2.2 LL(1) Grammars

To define formally what it means for a grammar to be LL(1) we first define the two functions $First$ and $Follow$. These functions help us to construct a predictive parser for a given LL(1) grammar, and their definition should be the first step in writing such a parser.

We let $First(\alpha)$ be the set of all terminals that can begin strings derived from α ($First(\alpha)$ also includes ϵ if $\alpha \overset{*}{\Longrightarrow} \epsilon$).

To compute $First(X)$ for all symbols X, apply the following rules until nothing else can be added to any $First$ set:

- if $X \in T$, then $First(X) = \{X\}$.
- if $X \to \epsilon \in P$, then add ϵ to $First(X)$.
- if $X \to Y_1 \ldots Y_n \in P$, then add all non-$\epsilon$ members of $First(Y_1)$ to $First(X)$. If $Y_1 \overset{*}{\Longrightarrow} \epsilon$ then also add all non-ϵ members of $First(Y_2)$ to $First(X)$. If $Y_1 \overset{*}{\Longrightarrow} \epsilon$ and $Y_2 \overset{*}{\Longrightarrow} \epsilon$ then also add all non-ϵ members of $First(Y_3)$ to $First(X)$, etc. If all $Y_1, \ldots, Y_n \overset{*}{\Longrightarrow} \epsilon$, then add ϵ to $First(X)$.

To compute $First(X_1 \ldots X_m)$ for any string $X_1 \ldots X_m$, first add to $First(X_1 \ldots X_m)$ all non-ϵ members of $First(X_1)$. If $\epsilon \in First(X_1)$ then add all non-ϵ members of $First(X_2)$ to $First(X_1 \ldots X_m)$, etc. Finally, if every set $First(X_1), \ldots, First(X_m)$ contains ϵ then we also add ϵ to $First(X_1 \ldots X_m)$.

From the definition of $First$, we can give a first condition for a grammar $G = (T, N, S, P)$ to be LL(1), as follows. For each non-terminal $A \in N$, suppose that

$$A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

are all the A-productions in P. For G to be LL(1) it is necessary, but not sufficient, for

$$First(\beta_i) \cap First(\beta_j) = \emptyset$$

for each $1 \leq i, j \leq n$ where $i \neq j$.

We let $Follow(A)$ be the set of terminal symbols that can appear immediately to the right of the non-terminal symbol A in a sentential form. If A appears as the rightmost non-terminal in a sentential form, then $Follow(A)$ also contains the symbol \$.

To compute $Follow(A)$ for all non-terminals A, apply the following rules until nothing can be added to any $Follow$ set.

- Place \$ in $Follow(S)$.
- If there is a production $A \to \alpha B \beta$, then place all non-ϵ members of $First(\beta)$ in $Follow(B)$.

- If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where $\beta \overset{*}{\Longrightarrow} \epsilon$, then place everything in $Follow(A)$ into $Follow(B)$.

Now, supposing a grammar G meets the first condition above. G is an LL(1) grammar if it also meets the following condition. For each $A \in N$, let

$$A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

be all the A-productions. If $\epsilon \in First(\beta_k)$ for some $1 \leq k \leq n$ (and notice, if the first condition holds, there will only be at most one such value of k) then it must be the case that

$$Follow(A) \cap First(\beta_i) = \emptyset$$

for each $1 \leq i \leq n, \quad i \neq k$.

Example. Let G have the following productions:

$$\begin{array}{lcl} S & \to & BAaC \\ A & \to & D \mid E \\ B & \to & b \\ C & \to & c \\ D & \to & aF \\ E & \to & \epsilon \\ F & \to & f \end{array}$$

First and Follow sets for this grammar are:

$$\begin{array}{lcl@{\qquad}lcl} First(BAaC) & = & \{b\} & Follow(S) & = & \{\$\} \\ First(D) & = & \{a\} & Follow(A) & = & \{a\} \\ First(E) & = & \{\epsilon\} & Follow(B) & = & \{a\} \\ First(b) & = & \{b\} & Follow(C) & = & \{\$\} \\ First(c) & = & \{c\} & Follow(D) & = & \{a\} \\ First(aF) & = & \{a\} & Follow(E) & = & \{a\} \\ First(f) & = & \{f\} & Follow(F) & = & \{a\}. \end{array}$$

It is clear that G meets the first condition above, as

$$First(D) \cap First(E) = \{a\} \cap \{\epsilon\} = \emptyset.$$

However, G does not meet the second condition because

$$Follow(A) \cap First(D) = \{a\} \cap \{a\} \neq \emptyset.$$

Many non-LL(1) grammars can be transformed into LL(1) grammars (by removing left recursion and left factoring (see Section 4.2.5), to enable a predictive parser to be written for them. However, although left factoring and elimination of left recursion are easy to implement, they often make the grammar hard to read. Moreover, not every grammar can be transformed into an LL(1) grammar, and for such grammars we need other methods of parsing.

4.2.3 Non-recursive Predictive Parsing

A significant aspect of LL(1) grammars is that it is possible to construct non-recursive predictive parsers from an LL(1) grammar. Such parsers explicitly maintain a stack, rather than implicitly via recursive calls. They also use a *parsing-table* which is a two dimensional array M of elements $M[A, a]$ where A is a non-terminal symbol and a is either a terminal symbol or the end-of-string symbol \$. Each element $M[A, a]$ of M is either an A-production or an error entry. Essentially, $M[A, a]$ says which production to apply when we need to match an A (in this case, A will be on the top of the stack), and when the next symbol input symbol is a.

Constructing an LL(1) parsing table. To construct the parsing table M, we complete the following steps for each production $A \rightarrow \alpha$ of the grammar G:

- For each terminal a in $First(\alpha)$, let $M[A, a]$ be $A \rightarrow \alpha$.
- If $\epsilon \in First(\alpha)$, let $M[A, b]$ be $A \rightarrow \alpha$ for all terminals b in $Follow(A)$. If $\epsilon \in First(\alpha)$ and $\$ \in Follow(A)$, let $M[A, \$]$ be $A \rightarrow \alpha$.

Let all other elements of M denote an *error*.

What happens if we try to construct such a table for a non-LL(1) grammar?

The LL(1) parsing algorithm. The non-recursive predictive parsing algorithm (common to all LL(1) grammars) is shown below. Note the use of \$ to mark the bottom of the stack and the end of the input string.

```
set stack to $S (with S on top);
set input pointer to point to the first symbol of the input w$;
repeat the following step
  (letting X be top of stack and a be current input symbol):
  if X = a = $ then we have parsed the input successfully;
  else if X = a (a token) then pop X off the stack and advance input pointer;
  else if X is a token or $ but not equal to a then report an error;
  else (X is a non-terminal) consult M[X,a]:
     if this entry is a production X -> Y1...Yn then pop X from the stack
        and replace it with Yn...Y1 (with Y1 on top);
     output the production X -> Y1...Yn
     else report an error
```

Example. Consider the grammar G with productions:

$$\begin{aligned} C &\rightarrow cD \mid Ed \\ D &\rightarrow EFC \mid e \\ E &\rightarrow \epsilon \mid f \\ F &\rightarrow g \end{aligned}$$

First we define useful *First* and *Follow* sets for this grammar:

$$\begin{array}{lclclclclcr} First(cD) & = & \{c\} & First(f) & = & \{f\} & Follow(C) & = & \{\$\} \\ First(Ed) & = & \{d, f\} & First(g) & = & \{g\} & Follow(D) & = & \{\$\} \\ First(EFC) & = & \{f, g\} & First(F) & = & \{g\} & Follow(E) & = & \{d, g\} \\ First(\epsilon) & = & \{\epsilon\} & First(C) & = & \{c, d, f\} & Follow(F) & = & \{c, d, f\} \end{array}$$

This gives the following LL(1) parsing table:

	c	d	e	f	g	$\$$
C	$C \rightarrow cD$	$C \rightarrow Ed$		$C \rightarrow Ed$		
D			$D \rightarrow e$	$D \rightarrow EFC$	$D \rightarrow EFC$	
E		$E \rightarrow \epsilon$		$E \rightarrow f$	$E \rightarrow \epsilon$	
F					$F \rightarrow g$	

which, when used with the LL(1) parsing algorithm on input string $cgfd$, gives:

Stack	Input	Action
$\$C$	$cgfd\$$	$C \rightarrow cD$
$\$Dc$	$cgfd\$$	match
$\$D$	$gfd\$$	$D \rightarrow EFC$
$\$CFE$	$gfd\$$	$E \rightarrow \epsilon$
$\$CF$	$gfd\$$	$F \rightarrow g$
$\$Cg$	$gfd\$$	match
$\$C$	$fd\$$	$C \rightarrow Ed$
$\$dE$	$fd\$$	$E \rightarrow f$
$\$df$	$fd\$$	match
$\$d$	$d\$$	match
$\$$	$\$$	accept.

4.2.4 Syntax-tree Construction for Recursive-Descent Parsers

In this section we extend the recursive descent parser writing technique of Section 4.2.1 in order to build syntax trees. We consider the construction of a r.d.p. for the following simple grammar which only allows assignments and if-statements.

$$\begin{array}{rcl}
\langle \textit{command list} \rangle & ::= & \langle \textit{command} \rangle \ \langle \textit{rest commands} \rangle \\
\langle \textit{rest commands} \rangle & ::= & ; \ \langle \textit{command list} \rangle \quad | \quad \textit{null} \\
\langle \textit{command} \rangle & ::= & \langle \textit{if command} \rangle \quad | \quad \langle \textit{assignment} \rangle \\
\langle \textit{if command} \rangle & ::= & \texttt{if} \ \langle \textit{expression} \rangle \ \texttt{then} \ \langle \textit{command list} \rangle \ \texttt{else} \ \langle \textit{command list} \rangle \ \texttt{fi} \\
\langle \textit{assignment} \rangle & ::= & \texttt{identifier} := \langle \textit{expression} \rangle \\
\langle \textit{expression} \rangle & ::= & \langle \textit{atom} \rangle \ \langle \textit{rest expression} \rangle \\
\langle \textit{rest expression} \rangle & ::= & + \ \langle \textit{atom} \rangle \ \langle \textit{rest expression} \rangle \quad | \quad \textit{null} \\
\langle \textit{atom} \rangle & ::= & \texttt{identifier}
\end{array}$$

In order to construct a predictive parser, we first list the important *First* and *Follow* sets. These are (grouped for each of the eight non-terminals):

$$\begin{array}{rcl}
First(\langle \textit{command} \rangle \ \langle \textit{rest commands} \rangle) & = & \{\texttt{if}, \texttt{identifier}\} \\
\\
First(; \ \langle \textit{command list} \rangle) & = & \{;\} \\
First(\textit{null}) & = & \{\textit{null}\} \\
Follow(\langle \textit{rest commands} \rangle) & = & \{\texttt{else}, \texttt{fi}, \$\} \\
\\
First(\langle \textit{if command} \rangle) & = & \{\texttt{if}\} \\
First(\langle \textit{assignment} \rangle) & = & \{\texttt{identifier}\} \\
\\
First(\texttt{if} \ \langle \textit{expression} \rangle \ \texttt{then} \ \langle \textit{command list} \rangle \ \texttt{else} \ \langle \textit{command list} \rangle \ \texttt{fi}) & = & \{\texttt{if}\} \\
\\
First(\texttt{identifier} := \langle \textit{expression} \rangle) & = & \{\texttt{identifier}\} \\
\\
First(\langle \textit{atom} \rangle \ \langle \textit{rest expression} \rangle) & = & \{\texttt{identifier}\} \\
\\
First(+ \ \langle \textit{atom} \rangle \ \langle \textit{rest expression} \rangle) & = & \{+\} \\
First(\textit{null}) & = & \{\textit{null}\} \\
Follow(\langle \textit{rest expression} \rangle) & = & \{;, \texttt{else}, \texttt{fi}, \texttt{then}, \$\} \\
\\
First(\texttt{identifier}) & = & \{\texttt{identifier}\}
\end{array}$$

We will construct *syntax trees* rather than parse trees, since the former are much smaller and simpler yet contain all the information required to enable straightforward code generation.

Notice that:

- command sequences are better constructed as lists rather than as trees as would be natural given the grammar;
- expressions are better constructed as syntax trees that better illustrate the expression (with proper rules of precedence and associativity), rather than directly as given by the grammar;
- the grammar fails to represent the fact that '+' should be left associative; (i.e. x+y+z is parsed as x+(y+z) rather than the correct (x+y)+z—this is not really a problem for '+', but for '-' it would be, since 1-1-1 would be evaluated 1-(1-1) = 1 rather than the conventional (1-1)-1 = -1. In fact, it is impossible to write an LL(1) that achieves this task. Therefore, we will get the parser to construct the correct (left associative) syntax tree using a clever technique–see the code.

We first write some type definitions to represent arbitrary nodes in a syntax tree. Assume that `MAXC` is a constant with value 3, which represents the maximum number of children a node can have (if-commands need three children, for the condition, the then clause and the else clause).

```
{ There are command nodes and expression nodes.
  Command nodes can be if or assign nodes; expression
  nodes can be indentifiers or operations }
node_kind    = (n_command, n_expr);
command_kind = (c_if, c_assign);
expr_kind    = (e_ident, e_operation);

tree_ptr = ^tree_node;

tree_node = record
  children : array[1..MAXC] of tree_ptr;
  kind : node_kind;       { is it a command or expression node? }
  ckind : command_kind;   { what kind of command is it? }
  ident : string[20];     { for assignment lhs and simple expressions }
  sibling : tree_ptr;     { the next command in the list }
  ekind : expr_kind;      { what kind of expression is it? }
  operator : token        { expression operator }
end;
```

Note that some of the fields of this record type will be redundant for certain types of node. For example, only command nodes will have siblings and only expression nodes will have operators.

We could use variant records (or unions in C) but this would complicate the discussion slightly, so we choose not to. Also note that we do not need a field for operators, since we only have '+' in the grammar; however, using this field we can easily add extra operations to the grammar.

Recall from Section 4.2.1, that we write a procedure parse_A for each non-terminal A in the grammar, the aim of which is to recognise strings derivable from A. To write a parser that constructs syntax trees, we replace the procedure parse_A by a function whose additional role it is to construct and return a syntax tree for the recognised string. The code for the above grammar is as follows.

We assume that the lexical analyser takes the form of a procedure get_next_token that updates the variables current_token and current_identifier, and that we have a match procedure that checks tokens. The tokens for the grammar are k_if, k_then, k_else, k_fi, assign, semi_colon, identifier, emd_of_input and error.

We first define two auxiliary functions that the other functions will call in order to create nodes for expressions or commands, respectively.

```
function new_expr_node(t : expr_kind) : tree_ptr;
{Construct a new expression node of kind t}
var
 i : integer; temp : tree_ptr;
begin
  new(temp);
  for i := 1 to MAXC do temp^.children[i] := nil;
  temp^.kind := n_expr;
  temp^.ekind := t;
  new_expr_node := temp
end;

function new_command_node(t : command_kind) : tree_ptr;
{Construct a new command node of kind t}
var
 i : integer; temp : tree_ptr;
begin
  new(temp);
  for i := 1 to MAXC do temp^.children[i] := nil;
  temp^.kind := n_command;
  temp^.ckind := t;
  temp^.sibling := nil;
  new_command_node := temp
end;
```

Next, we define match and syntax_error procedures:

```
procedure match(expected : token);
{Check the current token is as expected; if so,
 read the next token in}
begin
  if current_token = expected then
    get_next_token
  else
    syntax_error('Unexpected token')
end;

procedure syntax_error(s : string);
{Output error message}
begin
  writeln(s);
  writeln('Exiting');
  halt
end;
```

For the first three rules in the BNF (those for lists of commands) we have the following functions. A linked list of commands is built using tree nodes' `sibling` fields.

Note that in `parse_command_list`, we do not bother looing at the current token to see if it is valid, since we know `parse_command` will do this. Notice also how the `parse_rest_commands` function uses the set $Follow(\langle rest\ commands\rangle) = \{\texttt{else}, \texttt{fi}, \$\}$ to determine when to use the ε-production.

```
function parse_command_list : tree_ptr;
var
  temp : tree_ptr;
begin
  temp := parse_command;
  temp^.sibling := parse_rest_commands;
  parse_command_list := temp
end;
```

```
function parse_rest_commands : tree_ptr;
var
  temp : tree_ptr;
begin
  temp := nil;
  if current_token = semi_colon then begin
    match(semi_colon);
    temp := parse_command_list
  end else if not (current_token in [k_else,
                 k_fi, end_of_input]) then
    syntax_error('Incomplete command');
  parse_rest_commands := temp
end;

function parse_command : tree_ptr;
begin
  case current_token of
    k_if       : parse_command := parse_if_command;
    identifier : parse_command := parse_assignment;
    else begin
      syntax_error('Expected if or variable');
      parse_command := nil
    end
  end
end;
```

The two types of command (if commands and assignments) are parsed using the following functions.

```
function parse_assignment : tree_ptr;
var
   temp : tree_ptr;
begin
  temp := new_command_node(c_assign);
  temp^.ident := current_identifier;
  match(identifier);
  match(assign);
  temp^.children[1] := parse_expression;
  parse_assignment := temp
end;
```

```
function parse_if_command : tree_ptr;
var
  temp : tree_ptr;
begin
  temp := new_command_node(c_if);
  match(k_if);
  temp^.children[1] := parse_expression;
  match(k_then);
  temp^.children[2] := parse_command_list;
  match(k_else);
  temp^.children[3] := parse_command_list;
  match(k_fi);
  parse_if_command := temp
end;
```

The final three functions handle expressions. Notice that the function `parse_rest_expression` takes an argument comprising the (already parsed) left hand side expression, and also that it uses the set $Follow(\langle rest\ expression \rangle) = \{;, \texttt{then}, \texttt{else}, \texttt{fi}, \$\}$ to determine whether to use the ε-production.

Follow through the functions for the case x+y+z to convince yourself that they return the correct interpretation (x+y)+z.

```
function parse_expression : tree_ptr;
{Parse an expression, and force it to be left associative}
var
  left : tree_ptr;
begin
  left := parse_atom;
  parse_expression := parse_rest_expression(left);
end;
```

```
function parse_rest_expression(left : tree_ptr) : tree_ptr;
{If there is an operator on the input, construct an expression
 with lhs left (already parsed) by parsing rhs. Otherwise, return
 left as the expression}
var
  new_e : tree_ptr;
begin
  if (current_token = plus) then begin
    new_e := new_expr_node(e_operation);
    new_e^.operator := current_token;
    new_e^.children[1] := left;
    get_next_token;
    new_e^.children[2] := parse_atom;
    parse_rest_expression := parse_rest_expression(new_e)
  end else if (current_token in [semi_colon, end_of_input,
                               k_else, k_fi, k_then]) then
    parse_rest_expression := left
  else begin
    syntax_error('Badly formed expression');
    parse_rest_expression := nil
  end
end;

function parse_atom : tree_ptr;
var
  temp : tree_ptr;
begin
  temp := nil;
  if current_token = identifier then begin
     temp := new_expr_node(e_identifier);
     temp^.ident := current_identifier
     match(identifier);
  end else
     syntax_error('Badly formed expression')
  parse_atom := temp
end;
```

4.2.5 Operations on context-free grammars for top-down parsing

As we have seen, many grammars are not suitable for use in top-down parsing. In particular, for predictive parsing only LL(1) grammars may be used. Many grammars can be transformed into more suitable forms in order to make construction of top-down parsers possible. Two useful techniques are *left-factoring* and *removal of left recursion*.

Left factoring. Many simple parsing algorithms (e.g. those we have studied so far) cannot cope with grammars with productions such as

$$\begin{array}{lcl}\langle \textit{if stmt}\rangle & \rightarrow & \texttt{if}\ \langle \textit{bool}\rangle\ \texttt{then}\ \langle \textit{stmt list}\rangle\ \texttt{fi}\quad | \\ & & \texttt{if}\ \langle \textit{bool}\rangle\ \texttt{then}\ \langle \textit{stmt list}\rangle\ \texttt{else}\ \langle \textit{stmt list}\rangle\ \texttt{fi}\end{array}$$

where two or more productions for a given non-terminal share a common prefix. This is because they have to make a decision on which production rule to use based only on the first symbol of the rule.

We will not look at the general algorithm for left-factoring a grammar. For the above example however, we simply factor-out the common prefix to obtain the equivalent grammar rules:

$$\begin{array}{rcl}\langle \textit{if stmt}\rangle & \rightarrow & \texttt{if}\ \langle \textit{bool}\rangle\ \texttt{then}\ \langle \textit{stmt list}\rangle\ \langle \textit{rest if}\rangle \\ \langle \textit{rest if}\rangle & \rightarrow & \texttt{else}\ \langle \textit{stmt list}\rangle\ \texttt{fi}\quad |\quad \texttt{fi}\end{array}$$

Removing Immediate Left Recursion. A grammar is said to be *immediately left recursive* if there is a production of the form $A \rightarrow A\alpha$. Grammars for programming languages are often immediately left recursive, or have general left recursion (see next section). However, a large class of parsing algorithms (specifically *top-down* parsers) cannot cope with such grammars. Here we shall show how immediate left recursion can be removed from a context-free grammar, and in the next section we will see how general left recursion can be removed.

Consider a grammar G with productions $A \rightarrow A\alpha \mid \beta$ where β does not begin with an A. We can make a new grammar G' that defines the same language as G but without any left recursion, by replacing the above productions with

$$\begin{array}{rcl}A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon\end{array}$$

More generally, consider a grammar G with productions

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

where none of $\beta_1, \ldots, \beta_m$ begin with an A. We can make a new grammar G' without left recursion by replacing the above productions with

$$\begin{array}{rcl}A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \epsilon\end{array}$$

Removing Left Recursion. A grammar is said to be *left recursive* if $A \overset{+}{\Longrightarrow} A\alpha$ for some nonterminal A. Let G be a grammar that has no ϵ-productions and no cycles. (A grammar G has a *cycle* if $A \overset{+}{\Longrightarrow} A$ for some nonterminal A.) The following is an algorithm that removes left recursion from G.

```
Arrange the nonterminals in some order $A_1 \dots A_n$
for i := 1 to n do
  for j := 1 to i-1 do
    replace each production of the form $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \cdots \mid \delta_k\gamma$
    where $A_j \to \delta_1 \mid \cdots \mid \delta_k$ are all the current $A_j$ productions;
  od
  remove any immediate left recursion from all current $A_i$-productions
od
```

In order to understand how the algorithm works, we begin by considering how G might be left recursive. First, we arrange the nonterminals into some order $A_1, \dots, A_n$. In order for G to be left recursive, it must be that $A_i \overset{+}{\Longrightarrow} A_i\alpha$ for some i and some α. Now consider an arbitrary A_i-production $A_i \to \beta$. Clearly, there are two possibilities for β:

(1) β is of the form $\beta = a\gamma$ for some terminal a, or

(2) β is of the form $\beta = A_j\gamma$ for some nonterminal A_j.

Notice that if every A_i-production is of the form (1) then any derivation from A_i will yield a string that starts with a terminal symbol. Since A_i is a *non*terminal, it is impossible to have $A_i \overset{+}{\Longrightarrow} A_i\alpha$ for any α — contradicting the assumption of G's left recursiveness. Thus, it must be that at least one A_i-production is of the form $A_i \to A_j\gamma$. Suppose that $j > i$ for *every* such production (for all non-terminals A_i): Then it must be that whenever A_i derives a string α and the leftmost symbol of α is a nonterminal A_k, then $k > i$. Since this means $k \neq i$, G cannot be left recursive.

The way the algorithm works is to transform the productions of G in such a way that the new productions are either of the form $A_i \to a\alpha$ or of the form $A_i \to A_j\alpha$ where $j > i$, and thus the new grammar cannot be left recursive.

In more detail, the algorithm considers the A_i - productions for $i = 1, 2, \dots, n$. The case $i = 1$ is a special case since if $A_i \to A_j\alpha$ then $j \geq i$. If $j > i$ then the production is in the right form, and if $j = i$ then we have an instance of immediate left recursion that can be removed as in the previous section.

For $i = 2, \dots, n$, it is possible to have A_i-productions of the form $A_i \to A_j\alpha$ with $j < i$. For each of these productions the algorithm substitutes for A_j each possible right hand side of an A_j-production. That is, if $A_j \to \delta_1 \mid \cdots \mid \delta_p$ are all the A_j-productions, then the algorithm replaces $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \cdots \mid \delta_p\gamma$. Now, since the algorithm has already processed the A_j-productions (think inductively here!), it must be the case that if the leftmost symbol of δ_r ($r = 1, \dots, p$) is a nonterminal A_k say, then $k > j$ (because δ_r is the right hand side of an A_j-production). Thus the lowest index of any nonterminal occurring as the leftmost symbol of the right hand side of any A_i-production has been increased from j to k.

We require this lowest index to be greater than (or equal to) i. If k is already greater than (or equal to) i then the production $A_i \rightarrow \delta_r\gamma$ is of the right form (we can remove the immediate left recursion if $k = i$). Otherwise, $j < k < i$. Now, $k - j$ iterations of the inner loop later, j will be equal to k, and so the algorithm will then consider A_i-productions of the form $A_i \rightarrow A_k\alpha$. As before, since $k < i$ the algorithm has already considered the A_k-productions, and so all the A_k-productions whose right hand sides start with a nonterminal A_l will have $l > k$. Thus, on substituting these right hand sides for A_k in $A_i \rightarrow A_k\alpha$, we obtain productions whose right hand sides begin with a terminal symbol a nonterminal A_l for $l > k$. Thus the lowest index of any nonterminal occurring as the leftmost symbol of the right hand side of any A_i-production has now been increased from j to k to l.

Continuing in this way, we see that by the time that we have dealt with the case $j = i - 1$ we must arrive at a situation where the lowest index has been increased to i or greater.

Example. Let us remove the left recursion from the following productions:

$$\begin{array}{rcl} A_1 & \rightarrow & a \mid A_2b \\ A_2 & \rightarrow & c \mid A_3d \\ A_3 & \rightarrow & e \mid A_4f \\ A_4 & \rightarrow & g \mid A_2h \end{array}$$

Consider what happens when we execute the above algorithm on these productions. First, the nonterminals are already ordered, so we proceed with the outer loop. For each value of `i` the algorithm modifies the set of A_i-productions — this set may grow as the algorithm proceeds.

Consider the A_i-productions for $i = 1, 2, 3$. Notice each is of the form $A_i \rightarrow \alpha$ where the leftmost symbol of α is either a terminal, or is A_j for $j > i$. Thus the algorithm does nothing for $i = 1, 2, 3$.

At $i = 4$, there are no A_i-productions of the form $A_4 \rightarrow A_1\gamma$ and so nothing happens for $j = 1$. At $j = 2$, $A_4 \rightarrow A_2\gamma$ is a production for $\gamma = h$. The current A_2-productions are $A_2 \rightarrow c \mid A_3d$, and so we replace $A_4 \rightarrow A_2h$ with $A_4 \rightarrow ch \mid A_3dh$. This ends the case $j = 2$, and the A_4-productions are currently $A_4 \rightarrow g \mid ch \mid A_3dh$.

At $j = 3$, we see $A_4 \rightarrow A_3\gamma$ is a current A_4-production for $\gamma = dh$. The current A_3-productions are $A_3 \rightarrow e \mid A_4f$, and so $A_4 \rightarrow A_3dh$ is replaced with $A_4 \rightarrow edh \mid A_4fdh$. This ends the case $j = 3$ and the A_4-productions are now $A_4 \rightarrow g \mid ch \mid edh \mid A_4fdh$.

This ends the inner loop, and so we remove the immediate left recursion from A_4's productions, resulting in:

$$\begin{array}{rcl} A_4 & \rightarrow & gA_4' \mid chA_4' \mid edhA_4' \\ A_4' & \rightarrow & fdhA_4' \mid \epsilon. \end{array}$$

4.3 Bottom-up Parsers

Bottom-up parsing can be thought of as the process of creating a parse tree for a given input string by working upwards from the leaves towards the root. The most well-known form of bottom-up parsing is that of *shift-reduce parsing* to which we will restrict our attention; this is a non-recursive, non-backtracking method. Shift-reduce parsing works by matching substrings of the input string against productions of a grammar G and then replacing such substrings with the appropriate nonterminal, so that the input string (provided it is in $L(G)$) is eventually 'reduced' to the start symbol S. In contrast to top-down parsing that traces out a leftmost derivation, shift-reduce parsing aims to yield a rightmost derivation (in reverse).

Example. Consider a grammar G with the following productions:

$$\begin{array}{lcl} A & \rightarrow & ab \mid BAc \\ B & \rightarrow & CA \\ C & \rightarrow & d \mid Ba \mid abc. \end{array}$$

Now consider the input string $w = dababc$. The d matches $C \rightarrow d$ so w can be reduced to $w_1 = Cababc$. Next, the first ab matches $A \rightarrow ab$ so w_1 can be reduced to $w_2 = CAabc$. Next, CA matches $B \rightarrow CA$ so w_2 can be reduced to $w_3 = Babc$. Now ab matches $A \rightarrow ab$ so w_3 can be reduced to $w_4 = BAc$. Finally, w_4 can be reduced to A (the start symbol) by the production $A \rightarrow BAc$. We have reduced w to the start symbol, and reversing this process does indeed yield a rightmost derivation of w from A:

$$A \Longrightarrow BAc \Longrightarrow Babc \Longrightarrow CAabc \Longrightarrow Cababc \Longrightarrow dababc.$$

There are other sequences of reductions that reduce w to A but they need not yield rightmost derivations. For example, we could have chosen to reduce w_2 to $CAAc$ (and then to BAc), but this yields:

$$A \Longrightarrow BAc \Longrightarrow CAAc \Longrightarrow CAabc \Longrightarrow Cababc \Longrightarrow dababc.$$

Further, notice that not all sequences of reductions lead to the start symbol. For example, w_2 can be reduced to CAC that can only be reduced to BC.

General idea of shift-reduce parsing. A shift-reduce parser uses a stack in a similar way to LL(1) parsers. The stack begins empty, and a parse is complete when S appears as the only symbol on the stack. The parser has two possible actions (besides 'accept' and 'error') at each step:

- To *shift* the current input symbol onto the top of the stack.
- To *reduce* a string (for example, α) at the top of the stack to the non-terminal A, given the production $A \rightarrow \alpha$.

Consider the above grammar and the input string $dababc$. The actions that a shift-reduce parser might make are:

Stack	Input	Action
	$dababc\$$	shift
d	$ababc\$$	reduce $C \to d$
C	$ababc\$$	shift
Ca	$babc\$$	shift
Cab	$abc\$$	reduce $A \to ab$
CA	$abc\$$	reduce $B \to CA$
B	$abc\$$	shift
Ba	$bc\$$	shift
Bab	$c\$$	reduce $A \to ab$
BA	$c\$$	shift
BAc	$\$$	reduce $A \to BAc$
A	$\$$	accept.

It is important to note that this is only an illustration of what a shift-reduce parser might do; it does not show *how* the parser makes decisions on what actions to take. The main problem with shift-reduce parsing is deciding when to shift and when to reduce (notice that in the example, we do not reduce the top of the stack every time possible—e.g we did not reduce the Ba to C when the stack contained Bab as this would have given an incorrect parse). A shift-reduce parser looks at the contents of the stack and at (some of) the input to decide what action to take. Different types of shift-reduce parsers use the available stack/input in different ways, as will will see in the next section.

Handles. The difficulty in writing a shift-reduce parser begins with deciding when reducing a substring of an input string will result in a (reverse) step in a rightmost derivation.

Informally, a *handle* of a string is a substring that matches the right hand side of a production, and whose reduction represents one (reverse) step in a rightmost derivation from the start symbol.

Formally, a handle of a right-sentential form γ is a pair $h = (A \to \beta, p)$ comprising a production $A \to \beta$ and a position p indicating where the substring β may be found and replaced by A to produce the the previous right-sentential form in a rightmost derivation of γ. That is, if $S \overset{*}{\Longrightarrow} \alpha A w \Longrightarrow \alpha\beta w = \gamma$ is a rightmost derivation, then $(A \to \beta, p)$ is a handle of $\alpha\beta w$ when p marks the position immediately after α.

For example, with reference to the example above, $(A \to ab, 2)$ is a handle of $Cababc$ since $Cababc$ is a right-sentential form, and replacing ab by A at position 2 yields $CAabc$ that is the predecessor of $Cababc$ in its rightmost derivation. In contrast, $(A \to ab, 4)$ is not a handle of $Cababc$ since although $CabAc \Longrightarrow Cababc$ is a rightmost derivation, it is not possible to derive $CabAc$ from the start symbol with a rightmost derivation.

An important fact underlying the shift/reduce method of parsing is that handles will always appear on the top of the stack. In other words, if there is a handle to reduce at all, then the last symbol of the handle must be on top of the stack; it is then a matter of tracing down through the stack to find the start of the handle.

Viable Prefixes. The second technical notion that we require in order to understand how a shift/reduce parser can be made to make choices that lead to a rightmost derivation is that of a *viable prefix.*

A viable prefix is a prefix of a right-sentential form that does not continue past the right hand end of the rightmost handle of that sentential form. For the example, the viable prefixes of $Cababc$ are ϵ, C, Ca and Cab since the rightmost handle of $Cababc$ is the first occurrence of ab.

The significance of viable prefixes is that provided the input seen up to a given point can be reduced to a viable prefix (as is always possible initially since ϵ is always a viable prefix), it is always possible to add more symbols to yield a handle at the top of the stack.

4.3.1 LR(k) Parsing

An *LR(k)* grammar is one for which we can construct a shift/reduce parser that reads input *left*-to-right and builds an *right*most derivation in reverse, using at most k symbols of lookahead. There are situations where a shift/reduce parser cannot resolve conflicts even knowing the entire contents of the stack and the next k inputs: such grammars, which include all ambiguous grammars, are called *non-LR*.

However, shift/reduce parsing can be adapted to cope with ambiguous grammars, and moreover LR parsing (i.e. LR(k) parsing for some $k \geq 0$) has a number of advantages. Most importantly, LR parsers can be constructed to recognise virtually all programming language constructs for which context-free grammars can be devised (including all LL(k) grammars). In fact, it is rarely the case that more than one lookahead character is required. Also, the LR parsing method can be implemented efficiently, and LR parsers can detect syntax errors as soon as theoretically possible.

The process of constructing an LR parser is however relatively involved, and in general, too difficult to do by hand. For this reason *parser generators* have been devised to automate the process of constructing an LR parser where this is possible. One well-known such tool is the program *yacc*. The remainder of this section is devoted to explaining the LR(k) parsing method in order to provide a thorough understanding of the ideas behind this tool.

4.3.2 LR, SLR and LALR parsing

The LR parsing algorithm is a stack-based shift/reduce parsing algorithm involving a parsing table that dictates an appropriate action for a given state of the parse. We will look at four different kinds of LR parser that use at most one symbol of lookahead. Each kind is an instance of the general LR parsing algorithm but is distinguished by the kind of parsing table employed.

The four types are:

- LR(0) parsers. As implied by the 0, the actions taken by LR(0) parsers are not influenced by lookahead characters.
- SLR(1) parsers ("S" meaning "simple").
- LALR(1) parsers ("LA" meaning "lookahead").
- LR(1) parsers.

The grammars for which each of the types of parsers may be constructed is described by the hierarchy:

$$\text{LR}(0) \subset \text{SLR}(1) \subset \text{LALR}(1) \subset \text{LR}(1)$$

such that every LR(0) grammar is also an SLR(1) grammar, etc. Roughly speaking, the more general the parser type, the more difficult the (manual) construction of the parsing table (i.e. LR(0) parsing tables are the easiest to construct, LR(1) tables are the most difficult). Finally, although the parsing tables for LR(0), SLR(1) and LALR(1) grammars are essentially the same size, LR(1) parsing tables can be extremely large. Because many programming language constructs cannot be described by SLR(1) grammars (and hence neither by LR(0) grammars), but almost all can be described (with effort) by LALR(1) grammars, it is the LALR(1) class of parser that has proven the most popular for bottom-up parsing; *yacc*, for example, is an LALR(1) parser generator.

We will first look at the LR parsing algorithm (or 'LRP' for 'LR parser' for short) and then look at how to construct the parsing table in the case of each of the others.

4.3.3 The LR Parsing Algorithm

The LRP is essentially a shift/reduce parser that is augmented with *states*. The stack holds strings of the form

$$s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m$$

(with s_m on the top) where $X_1, \ldots, X_m$ are grammar symbols and $s_0, \ldots, s_m$ are states. The idea behind a state in the stack is that it summarises the information in the stack below it: to be more precise, a state represents a viable prefix that has occurred before that state was reached. One of these states represents 'nothing has yet been read' (note that the empty string is always a viable prefix) and is designated the *initial state*. In fact, this initial state is the initial state of a DFA that recognises viable prefixes as we shall see below.

The states are used by the LRP's *goto function*: this is a function that takes a state and a grammar symbol as arguments and produces a new state; the idea is that if $goto(s, X) = s'$ then s' represents the viable prefix obtainable by appending an X to the viable prefix represented by s. (In other words, the goto function is the transition function of the DFA mentioned above.)

The behaviour of the LRP is best described in terms of *configurations*; that is, a pair

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, a_i \ldots a_n)$$

comprising the contents of the stack and the input yet to be read. The idea behind such a configuration is that it represents the LRP having up to this point constructed the right-sentential form

$$X_1 X_2 \ldots X_m a_i \ldots a_n$$

and about to read a_i.

The LRP starts with the initial state on its stack and the input pointer at the start of the input. It then repeatedly looks at the current (topmost) state and input symbol and takes the appropriate action before considering the next state and input, until processing is complete.

For a given configuration such as the above (with current state s_m and input symbol a_i) the action $\mathit{action}(s_m, a_i)$ taken by the LRP is one of the following:

- $\mathit{action}(s_m, a_i) = \mathit{shift}\ s$. Here the parser puts a_i and s onto the stack (in that order) and advances the input pointer. Thus from the configuration above, the LRP enters the configuration:
$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, a_{i+1} \ldots a_n).$$
- $\mathit{action}(s_m, a_i) = \mathit{reduce\ by}\ A \to \beta$. This action occurs when the top r (say) grammar symbols constitute β; that is, when $X_{m-r+1} \ldots X_m = \beta$. When this does occur, the topmost $2r$ symbols are replaced with A and $s = \mathit{goto}(s_{m-r}, A)$ (in that order); thus, in this case the configuration becomes:
$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, a_i \ldots a_n).$$
- $\mathit{action}(s_m, a_i) = \mathit{accept}$. Here parsing is completed and the LRP terminates.
- $\mathit{action}(s_m, a_i) = \mathit{error}$. Here a syntax error has been detected and the LRP calls an error recovery routine.

4.3.4 LR(0) State Sets and Parsing

We begin our discussion of the different types of LR parser by looking at LR(0) parsers. As implied by the '0', an LR(0) parser makes decisions on whether to shift or reduce based purely on the contents of the stack, not on lookahead. Although few useful grammars are LR(0), the construction of LR(0) parsing tables illustrates most of the concepts that we will need to build SLR(1), LR(1) and LALR(1) parsing tables.

As mentioned previously, the idea behind the 'states' on the stack of an LR parser is that a state represents a viable prefix that has occurred on the stack up to a given point in the parse.

We begin to make this idea more precise by defining an *LR(0) item* (or 'item' for short) to be a string of the form $A \to \alpha \bullet \beta$ where $A \to \alpha\beta$ is a production. (If $A \to$, that is, $A \to \epsilon$, then $A \to \bullet$ is the only item for this production.) Thus, for example, the production $A \to XY$ yields the three items: $A \to \bullet XY$, $A \to X \bullet Y$ and $A \to XY\bullet$.

In general, the dot ($\bullet$) is a marker denoting the current position in parsing a string; that is, an item of the form $A \to \alpha \bullet \beta$ represents a parser having parsed α and about to (try and) parse β.

We begin the process of constructing LR(0) parsing tables by augmenting the grammar G with a new production $S' \to S$ (where S' is a new start symbol) to form an *augmented grammar* G'. The reason for this is that we can use the item $S' \to \bullet S$ to represent the start of the parse (that is, the initial state).

Item Closures. Notice that if $A \to \alpha \bullet B\beta$ is an item, this represents a state in a shift/reduce parser where α is the current viable prefix on the top of the stack, and we expect to see (a string derivable from) $B\beta$ in the input. However, if $B \to \gamma$ is a production then (a string derivable from) γ is also acceptable input at this point.

Thus, in this situation $A \to \alpha \bullet B\beta$ and $B \to \bullet\gamma$ are equivalent in the sense that α is a viable prefix for sentential forms derivable from either item. In order to compute all the items of the parse equivalent to a given item (or set of item) we define the *closure* of a set of items I for a grammar using the following algorithm:

```
closure($I$)
  let $J$ := $I$
  repeat
    for each item $A \to \alpha \bullet X\beta$ in $J$
      for each $X \to \gamma$ add $X \to \bullet\gamma$ to $J$
  until $J$ does not change
  return $J$
```

The reason why the closure operation is defined on *sets* of items will be clearer later.

State Transitions. If the parser is in a certain state and then reads a symbol X, then intuitively the parser will change state in the sense that the set of strings it expects to see having read the X will be different (in general) to the set of strings the parser was expecting to see before the X. More formally, if $A \to \alpha \bullet X\beta$ is an item then reading an X results in the new item $A \to \alpha X \bullet \beta$ (and all the items equivalent to it).

We formalise these transitions between states of the parse with the function *goto*(I, X) that takes a set of items and a grammar symbol X as input, and returns a set of (equivalent) items as output, using the following algorithm:

```
goto($I$,$X$)
```

```
let $J$ be the empty set
for any item $A \to \alpha \bullet X\beta$ in $I$
  add $A \to \alpha X \bullet \gamma$ to $J$
return closure($J$)
```

LR(0) State Set Construction. We now construct all possible states that an LR(0) parser for an LR(0) grammar $G = (T, N, S, P)$ can be in. Initially, the parser will be in the state

$$closure(\{S' \to \bullet S\}).$$

We construct all other states from this initial state by using the *goto* and *closure* operations above, until no new states can be generated. The following algorithm accomplishes this task:

```
$C$ := $\{closure(\{S' \to \bullet S\})\}$
repeat
  for each state $I \in C$
    for each item $A \to \alpha \bullet X\beta$ in $I$
      let $J$ be $goto(I, X)$
      add $J$ to $C$
until $C$ does not change
return $C$
```

Example. Consider the LR(0) grammar

$$S \to (S) \mid a$$

which we augment with the production $S' \to S$. We begin construction of the LR(0) state set by computing the closure of the item $\{S' \to \bullet S\}$,

$$\{S' \to \bullet S, \quad S \to \bullet(S), \quad S \to \bullet a\}$$

which we will refer to as I_0. So $C = \{I_0\}$ after the initial step of the algorithm. In the next step of the algorithm we compute $goto(I_0, S)$, $goto(I_0, a)$ and $goto(I_0, ()$:

$$\begin{aligned}
goto(I_0, S) &= closure(\{S' \to S\bullet\}) \\
&= \{S' \to S\bullet\} \\
&= I_1 \quad (say) \\
goto(I_0, a) &= closure(\{S \to a\bullet\}) \\
&= \{S \to a\bullet\} \\
&= I_2 \quad (say) \\
goto(I_0, () &= closure(\{S \to (\bullet S)\}) \\
&= \{S \to (\bullet S), \quad S \to \bullet(S), \quad S \to \bullet a\} \\
&= I_3 \quad (say)
\end{aligned}$$

and thus $C = \{I_0, I_1, I_2, I_3\}$ after the first iteration of the repeat-forever loop.

On the next iteration, (noticing that neither I_1 nor I_2 contain items of the form $A \to \alpha \bullet X\beta$) we consider I_3 and compute $goto(I_3, S)$, $goto(I_3, a)$ and $goto(I_3, ()$:

$$\begin{array}{rcl}
goto(I_3, S) & = & closure(\{S \to (S\bullet)\}) \\
 & = & \{S \to (S\bullet)\} \\
 & = & I_4 \quad (say) \\
goto(I_3, a) & = & closure(\{S \to a\bullet\}) \\
 & = & I_2 \\
goto(I_3, () & = & closure(\{S \to (\bullet S)\}) \\
 & = & I_3
\end{array}$$

and now $C = \{I_0, I_1, I_2, I_3, I_4\}$. On the next iteration, we need only compute $goto(I_4,))$:

$$\begin{array}{rcl}
goto(I_4,)) & = & closure(\{S \to (S)\bullet\}) \\
 & = & \{S \to (S)\bullet\} \\
 & = & I_5 \quad (say)
\end{array}$$

and now $C = \{I_0, I_1, I_2, I_3, I_4, I_5\}$. It is clear that a further iteration of the loop will not yield any further states, so we have finished.

The states and *goto* functions can be depicted as a DFA....

Constructing LR(0) Parsing tables. We construct an LR(0) parsing table as follows:

(1) Construct the set $\{I_0, \ldots, I_n\}$ of states for the augmented grammar G'.

(2) The parsing actions for state I_i are determined as follows:

  (a) If $A \to \alpha \bullet a\beta \in I_i$ and $goto(I_i, a) = I_j$ then $action(i, a) = shift\ j$.

  (b) If $A \to \alpha\bullet \in I_i$ then $action(i, a) = reduce\ A \to \alpha$, for each $a \in T \cup \{\$\}$.

  (c) If $S' \to S\bullet \in I_i$ then $action(i, \$) = accept$.

(3) The *goto* actions for state i are as given by directly from the *goto* function; that is, if $goto(I_i, A) = I_j$ then let $goto(i, A) = j$.

(4) All entries not defined by the above rules are made *error*.

(5) The initial state is the one constructed from $\{S' \to \bullet S\}$.

Rule (a) above says that, if we have parsed α and we can see a terminal symbol b then we should shift the symbol onto the stack and go into state j.

Rule (b) says that, if we have finished parsing a particular production rule α then we should reduce by that production rule for all terminal symbols.

Rule (c) says that, if we have successfully parsed the start symbol then we have finished.

This method of creating a parser will fail if there are any conflicts in the above rules, and grammars leading to such conflicts are not LR(0).

Example. For the example grammar we obtain the following parsing table, where sj means $shift\ j$, rj means reduce using production rule j (where rule 1 is $S \to (S)$ and rule 2 is $S \to a$), and *ok* means 'accept'.

	action				goto
State	a	(	)	\$	S
0	$s2$	$s3$			1
1				*ok*	
2	$r2$	$r2$	$r2$	$r2$	
3	$s2$	$s3$			4
4			$s5$		
5	$r1$	$r1$	$r1$	$r1$	

A trace of the parse of the string $((a))$ is as follows:

Stack	Input	Action
0	$((a))\$$	shift 3
0(3	$(a))\$$	shift 3
0(3(3	$a))\$$	shift 2
0(3(3a2	$))\$$	reduce $S \to a$
0(3(3S4	$))\$$	shift 5
0(3(3S4)5	$)\$$	reduce $S \to (S)$
0(3S4	$)\$$	shift 5
0(3S4)5	\$	reduce $S \to (S)$
0S1	\$	accept

Notice that from the rules for constructing an LR(0) parsing table, for a grammar to be LR(0), if a state contains an item of the form $A \to \alpha\bullet$ (signifying the fact that α should be reduced to A) then it cannot also contain another item of the form:

(a) $B \to \beta\bullet$, or

(b) $B \to \beta \bullet a\gamma$.

In the case of (a), the parser cannot decide whether to reduce using rule $A \to \alpha$ or rule $B \to \beta$; this is known as a *reduce-reduce conflict*. In the case of (b), the parser cannot decide whether to reduce using rule $A \to \alpha$ or to shift (by the item $B \to \beta \bullet a\gamma$); this is known as a *shift-reduce conflict*. A grammar is LR(0) if, and only if, all states contain either no items of the form $A \to \alpha\bullet$, or contain exactly one item.

4.3.5 SLR(1) Parsing

SLR(1) parsing is more powerful than LR(0) as it consults (by the way an SLR(1) parsing table is constructed) the next input token to direct its actions. The construction of an SLR(1) table begins with the LR(0) state set construction as with LR(0) parsers.

Let's consider an example. We first construct the LR(0) state set. We then show that the grammar is not LR(0) due to shift-reduce conflicts. We then show how to construct general SLR(1) parsing tables and apply this to our example.

Example. Consider the following productions for a grammar for arithmetic expressions.

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow a$

augmented with the production $E' \rightarrow E$.

The state set construction proceeds as follows. We begin by constructing the initial state:

$$\begin{aligned} I_0 &= \mathit{closure}(\{E' \rightarrow \bullet E\}) \\ &= \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet a\}. \end{aligned}$$

Thus $C = \{I_0\}$ after the initial step of the algorithm.

Next, we see that we need to compute $\mathit{goto}(I_0, X)$ for $X = a$, (, E, T, F:

$$\begin{aligned} \mathit{goto}(I_0, a) &= \mathit{closure}(\{F \rightarrow a\bullet\}) \\ &= \{F \rightarrow a\bullet\} \\ &= I_1 \text{ (say)} \\ \mathit{goto}(I_0, () &= \mathit{closure}(\{F \rightarrow (\bullet E)\}) \\ &= \{F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, \\ &\quad\; T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet a\} \\ &= I_2 \text{ (say)} \\ \mathit{goto}(I_0, E) &= \mathit{closure}(\{E' \rightarrow E\bullet, E \rightarrow E \bullet + T\}) \\ &= \{E' \rightarrow E\bullet, E \rightarrow E \bullet + T\} \\ &= I_3 \text{ (say)} \\ \mathit{goto}(I_0, T) &= \mathit{closure}(\{E \rightarrow T\bullet, T \rightarrow T \bullet * F\}) \end{aligned}$$

$$\begin{array}{rcl} & = & \{E \to T\bullet, T \to T \bullet *F\} \\ & = & I_4 \text{ (say)} \\ goto(I_0, F) & = & closure(\{T \to F\bullet\}) \\ & = & \{T \to F\bullet\} \\ & = & I_5 \text{ (say)}. \end{array}$$

Thus now $C = \{I_0, I_1, I_2, I_3, I_4, I_5\}$.

On the next iteration of the loop we need to consider I_1, I_2, I_3, I_4, I_5. First, notice that there are no items in I_1 that have any symbols following the marker, and hence $goto(I_1, X) = \emptyset$ for each possible symbol X. We now compute $goto(I_2, X)$ for $X = a, (, E, T, F$ as follows:

$$\begin{array}{rcl} goto(I_2, a) & = & closure(\{F \to a\bullet\}) \\ & = & I_1 \\ goto(I_2, () & = & closure(\{F \to (\bullet E)\}) \\ & = & I_2 \\ goto(I_2, E) & = & closure(\{F \to (E\bullet), E \to E \bullet +T\}) \\ & = & \{F \to (E\bullet), E \to E \bullet +T\} \\ & = & I_6 \text{ (say)} \\ goto(I_2, T) & = & closure(\{E \to T\bullet, T \to T \bullet *F\}) \\ & = & I_4 \\ goto(I_2, F) & = & closure(\{T \to F\bullet\}) \\ & = & I_5. \end{array}$$

For I_3 we only need to compute $goto(I_3, +)$:

$$\begin{array}{rcl} goto(I_3, +) & = & closure(\{E \to E + \bullet T\}) \\ & = & \{E \to E + \bullet T, T \to \bullet T * F, T \to \bullet F, F \to \bullet(E), F \to \bullet a\} \\ & = & I_7 \text{ (say)}. \end{array}$$

For I_4 we need only compute $goto(I_4, *)$:

$$\begin{array}{rcl} goto(I_4, *) & = & closure(\{T \to T * \bullet F\}) \\ & = & \{T \to T * \bullet F, F \to \bullet(E), F \to \bullet a\}) \\ & = & I_8 \text{ (say)}. \end{array}$$

We also see that $goto(I_5, X) = \emptyset$ for all X, as in the case of I_1. Thus, after the second iteration of the loop we have $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8\}$.

In the next iteration we need to consider I_6, I_7, I_8. For I_6, we see that we need to compute $goto(I_6, +)$ and $goto(I_6,))$:

$$\begin{array}{rcl} goto(I_6, +) & = & closure(\{E \to E + \bullet T\}) \\ & = & I_7 \\ goto(I_6,)) & = & closure(\{F \to (E)\bullet\}) \end{array}$$

$$\begin{aligned} &= \{F \to (E)\bullet\} \\ &= I_9 \text{ (say).} \end{aligned}$$

For I_7, we see that we need to compute $goto(I_7, X)$ for $X = a, (, T, F$:

$$\begin{aligned} goto(I_7, a) &= closure(\{F \to a\bullet\}) \\ &= I_1 \\ goto(I_7, () &= closure(\{F \to (\bullet E)\}) \\ &= I_2 \\ goto(I_7, T) &= closure(\{E \to E + T\bullet, T \to T \bullet *F)\}) \\ &= \{E \to E + T\bullet, T \to T \bullet *F\}) \\ &= I_{10} \text{ (say)} \\ goto(I_7, F) &= closure(\{T \to F\bullet\}) \\ &= I_5. \end{aligned}$$

For I_8, we see that we need to compute $goto(I_8, X)$ for $X = a, (, F$:

$$\begin{aligned} goto(I_8, a) &= closure(\{F \to a\bullet\}) \\ &= I_1 \\ goto(I_8, () &= closure(\{F \to (\bullet E)\}) \\ &= I_2 \\ goto(I_8, F) &= closure(\{T \to T * F\bullet\}) \\ &= \{T \to T * F\bullet\} \\ &= I_{11} \text{ (say).} \end{aligned}$$

Thus, after the third iteration of the loop we have $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}\}$.

Next, we need to consider I_9, I_{10}, I_{11}. For I_9 and I_{11}, we see that $goto(I_9, X) = goto(I_{11}, X) = \emptyset$ for all X. For I_{10} we only need to compute $goto(I_{10}, *)$:

$$\begin{aligned} goto(I_{10}, *) &= closure(\{T \to T * \bullet F\}) \\ &= I_8. \end{aligned}$$

Thus, C did not change during the fourth iteration of the loop and we have finished(!).

Now, consider an attempt at constructing an LR(0) parsing table from this state set. Specifically, consider the state I_4 (we could also have chosen I_{10}). This state contains the items $E \to T\bullet$ and $T \to T \bullet *F$ and $goto(I_4, *) = I_8$. This is a shift-reduce conflict (we would attempt to place both $s8$ and $r2$ at location $(4, *)$ in the table).

Constructing SLR(1) Parsing Tables.

In LR(0) parsing a reduce action is performed whenever the parser enters a state i (i.e. I_i) whose (only) item is of the form $A \to \alpha\bullet$ regardless of the next input symbol. In SLR(1) parsing, if a state contains an item of this form then the reduction of α to A is only performed if the next token is in $Follow(A)$ (the set of all terminal symbols that can follow A in a sentential form), else it performs a different action (e.g. a shift, reduce, or error) based on the other items of state I_i.

The rules for constructing SLR(1) parsing tables are the same as for LR(0) tables giving in the previous section, except that we replace rule 2(b):

If $A \to \alpha\bullet \in I_i$ then $action(i, a) = reduce\ A \to \alpha$, for each $a \in T \cup \{\$\}$

by

If $A \to \alpha\bullet \in I_i$ then $action(i, a) = reduce\ A \to \alpha$, for each $a \in Follow(A)$.

Lets consider the example grammar. The $Follow$ sets are

$$\begin{aligned} Follow(E) &= \{\$, +,)\} \\ Follow(T) &= \{\$, *, +,)\} \\ Follow(F) &= \{\$, *, +,)\}. \end{aligned}$$

Notice first that the shift-reduce conflict mentioned above has been resolved. Recall the items $E \to T\bullet$ and $T \to T \bullet *F$ of I_4 where $goto(I_4, *) = I_8$. As $* \notin Follow(E)$, we do not choose to reduce in this state (i.e. we place only $s8$ in the parsing table at position $(4, *)$).

The complete SLR(1) parsing table for this grammar is as shown below.

	action						goto		
State	a	$+$	$*$	(	)	\$	E	T	F
0	$s1$			$s2$			3	4	5
1		$r6$	$r6$		$r6$	$r6$			
2	$s1$			$s2$			6	4	5
3		$s7$				ok			
4		$r2$	$s8$		$r2$	$r2$			
5		$r4$	$r4$		$r4$	$r4$			
6		$s7$			$s9$				
7	$s1$			$s2$				10	5
8	$s1$			$s2$					11
9		$r5$	$r5$		$r5$	$r5$			
10		$r1$	$s8$		$r1$	$r1$			
11		$r3$	$r3$		$r3$	$r3$			

A parse of the string $a + a * a$ is:

Step	Stack	Input	Action
1	0	$a + a * a$	shift 1
2	$0a1$	$+a * a$	reduce by $F \to a$
3	$0F5$	$+a * a$	reduce by $T \to F$
4	$0T4$	$+a * a$	reduce by $E \to T$
5	$0E3$	$+a * a$	shift 7
6	$0E3 + 7$	$a * a$	shift 1
7	$0E3 + 7a1$	$*a$	reduce by $F \to a$
8	$0E3 + 7F5$	$*a$	reduce by $T \to F$
9	$0E3 + 7T10$	$*a$	shift 8
10	$0E3 + 7T10 * 8$	a	shift 1
11	$0E3 + 7T10 * 8a1$	\$	reduce by $F \to a$
12	$0E3 + 7T10 * 8F11$	\$	reduce by $T \to T * F$
13	$0E3 + 7T10$	\$	reduce by $E \to E + T$
14	$0E3$	\$	accept

It can be seen from the SLR(1) table construction rules that a grammar is SLR(1) if, and only if, the following conditions are satisfied for each state i:

(a) For any item $A \to \alpha \bullet a\beta$, there is no other item $B \to \gamma\bullet$ with $a \in \mathit{Follow}(B)$;

(b) For any two items $A \to \alpha\bullet$ and $B \to \beta\bullet$, $\mathit{Follow}(A) \cap \mathit{Follow}(B) = \emptyset$.

Violation of (a) will lead to a shift-reduce conflict: if the parser is in state i and a is the next token, the parser cannot decide whether to shift the a (and $\mathit{goto}(i, a)$) onto the stack or to reduce the top of the stack using the rule $B \to \gamma$. Violation of (b) will lead to a reduce-reduce conflict: if the parser is in state i and the next token a is in $\mathit{Follow}(A)$ and $\mathit{Follow}(B)$, the parser cannot decide whether to reduce using rule $A \to \alpha$ or rule $B \to \beta$. For non-SLR(1) grammars that are still LR(1), we need to look at LR(1) and LALR(1) parsers.

5 Javacc

Javacc is a *compiler compiler* which takes a grammar specification and writes a parser for that language in Java. It comes with a useful tool called **Jjtree** which in turn produces a Javacc file with extra code added to construct a tree from a parse. By running the output from Jjtree through Javacc and then compiling the resulting Java files we get a full top-down parser for our grammar. The process is shown in figure 1

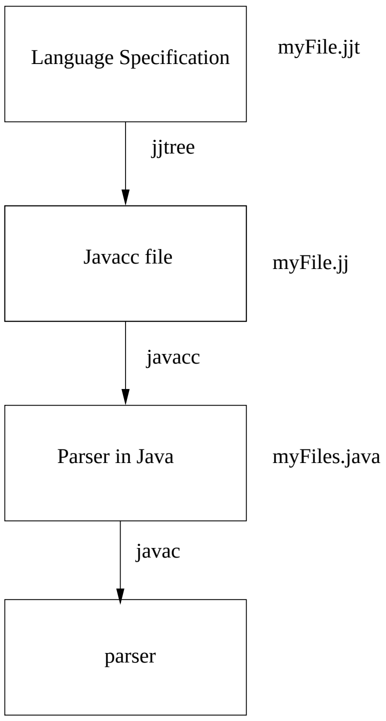


Figure 1: The JavaTree Compilation Process

Because the parser is top-down we can only use Javacc on LL(k) grammars, meaning that we have to remove any left-recursion from the grammar before writing the parser.

5.1 Grammar and Coursework

The grammar we will be compiling, both in these notes and in your coursework, is defined here:
program $\rightarrow$ *decList* **begin** *statementList* | *statementList*
decList $\rightarrow$ *dec decList* | *dec*
dec $\rightarrow$ *type* **id**
type $\rightarrow$ **float** | **int**
statementList $\rightarrow$ *statement* **;** *statementList* | *statement*
statement $\rightarrow$ *ifStatement* | *repeatStatement* | *assignStatement* | *readStatement* | *writeStatement*
ifStatement $\rightarrow$ *ifPart* **end** | *ifPart elsePart* **end**

ifPart → **if** *exp* **then** *statementList*
elsePart → **else** *statementList*
repeatStatement → **repeat** *statementList* **until** exp
assignStatement → **id :=** *exp*
readStatement → **read id**
writeStatement → **write** *exp*
exp → *simpleExp boolop simpleExp* | *simpleExp*
boolOp → **<** | **=**
simpleExp → *term addOp simpleExp* | *term*
addOp → **+** | **-**
term → *factor mulOp factor* | *factor*
mulOp → ***** | **/**
factor → **id** | **float** | **int** | **(** *exp* **)**

So we have an optional list of variable declarations, followed by a series of statements. There are 5 statements, a conditional, an iterational, an assignment, and a read and write statement. All variables must be declared before use and each variable can be a floating point number or an integer. We also have various expressions. Note how the grammar forces all arithmetic expressions to have the correct operator precedence. Other ways of achieving this will be discussed in the lectures. There are no boolean variables but two boolean operators are available for conditional and repeat statements, less than, and equals. The terminals in this grammar are precisely those written in bold in the above list.

Your coursework comes in seven parts with a deadline at midnight on the Friday of weeks 2 - 6, 8 and 11. The coursework is designed to take 30 hours to complete in total, so you should spend about 3 hours each week. The early parts shouldn't take that long so you are encouraged to work ahead. I hope to put a model answer on Blackboard by Monday evening each week, together with your grade, so if you achieved a disappointing grade for one assignment you can use the model answer for the next. Each week the coursework will build upon the previous work. Because I am giving model answers each week there will be no extensions. If you have a reasonable excuse for not submitting on time you will be exempted from that coursework. You are recommended to spend each Friday afternoon working on this assignment. Although there will not be formal lab classes you are **strongly** recommended to use Friday afternoons as such. You plan each week should be to read the relevant section of this document, try out the examples, ensure that you understand fully what is going on, and complete the coursework for that week. If this takes less than three hours you should start work on the next one. The first five courseworks are worth 10% each, the last two are worth 25% each. The coursework in total is worth 30% of the marks for the module.

5.2 Recognising Tokens

Our first job is to build a recogniser for the terminal symbols in the grammar. Lets look at a minimal javacc file:

```
options{
  STATIC = true;
}

PARSER_BEGIN(LexicalAnalyser)
   class LexicalAnalyser{
      public static void main(String[] args){
         LexicalAnalyser lexan = new LexicalAnalyser(System.in);
         try{
            lexan.start();
         }//end try
         catch(ParseException e){
            System.out.println(e.getMessage());
         }//end catch
         System.out.println("Finished Lexical Analysis");
      }//end main
   }//end class
PARSER_END(LexicalAnalyser)

//Ignore all whitespace
SKIP:{" "|"\t"|"\n"|"\r"}
//Declare the tokens
TOKEN:{<INT: (["0"-"9"])+>}
//Now the operators
TOKEN:{<PLUS: "+">}
TOKEN:{<MINUS: "-">}
TOKEN:{<MULT: "*">}
TOKEN:{<DIV: "/">}

void start():
{}
{
   <PLUS>    {System.out.println("Plus");} |
   <MINUS>   {System.out.println("Minus");} |
   <DIV>     {System.out.println("Divide");} |
   <MULT>    {System.out.println("Multiply");}|
   <INT>     {System.out.println("Integer");}
}
```

First we have a section for *options*. Most of the options available default to acceptable values for this example but we want this program to be **static** because we are going to write a main method here rather than in another file. Therefore we set this to true because it defaults to false.

Next we have a block of code delimited by `PARSER_BEGIN(LexicalAnalyser)` and `PARSER_END(LexicalAnalyser)`. The argument is the name which Javacc will give the finished parser. Within that there is a class declaration. All code within the PARSER_BEGIN and PARSER_END delimiters will be output verbatim by Javacc. Now we write the main method, which constructs a new parser and calls its **start()** method. This has to be placed inside a try-catch block because Javacc automatically generates its own internally defined ParseExceptions and throws them if something goes wrong with the parse. Finally we print a message to the screen announcing that we have finished the parse successfully.

We then have some `SKIP`ped items. In this case we are simply telling the final parser to ignore all whitespace, spaces, tabs, newlines and carriage returns. This is normal as we usually want to encourage users of our language to use whitespace to indent their programs to make them easier to read. There are languages in which whitespace has meaning, e.g. Haskell, but these tend to be rare.

We then have to declare the **tokens** or terminal symbols of the grammar. The first declaration is simply a regular expression for integers. We can put in square brackets a range of values, in this case `0 - 9`, separated by a hyphen. This means "anything between these two values inclusive". Then wrapping the range expression in parentheses and adding a "+" symbol we say that an `INT` (our name for integers) is one or more instances of a digit. Note that literals are always included in quote marks.

We then include definitions for the arithmetic operators. For such a simple example this is not necessary since we could use literals instead, but we will be making this more complex later so it's as well to start off the right way. Besides, it sometimes makes the code difficult to read if we mix literals and references freely. A TOKEN declaration consists of:

- The keyword TOKEN.
- A colon.
- An open brace.
- An open angle bracket.
- The name by which we will be referring to this token.
- A colon.

- A regular expression defining the pattern for recognising this token.
- The close angle bracket.
- The close brace.

We can have multiple token (or skip) declarations in a single line by including the | (or) symbol, as in the skip declarations in the example above.

We finally define the start() method in which all the work is done. A method declaration in Javacc consists of the return type (in this case void), the name of the method (in this case start) and any arguments required, followed by a colon and two (!) sets of braces. The first one contains variables local to this method (in this case none). The second one contains the grammar's production rules. In the above example we are saying that a legal program consists of a single instance of one of the grammar terminals.

Each statement inside the start() method follows a format whereby a previously defined token is followed by a set on one or more (one in this example) statements inside braces. This states that when the respective token is identified by the parser the code inside the braces (which must be legal Java code) is executed. Thus, if the parser recognises an **int** it will print **integer** to the terminal.

We first run the code through Javacc :**Javacc lexan.jj**. (The file-type .jj is compulsory). This gives us a series of messages. Most of these relate to the files being produced by Javacc. We then compile the resulting java files: **javac *.java**. This gives us the .class files which can be run from the command line with **Java LexicalAnalyser**.

N.B. You are strongly recommended to type in the above code and compile it. You should do this with all the examples given in this chapter. Test your parser with both legal and illegal input to see what happens.

At the moment we can only read a single token. The first improvement is to continue to read tokens until either we reach the end of the input or we read illegal input. We can do this by simply creating another regular expression inside the start() method.

```
void start():
{}
{
   (<PLUS>   |
    <MINUS>  |
    <DIV>    |
    <MULT>   |
    <INT>)+
}
```

Also we can redirect the input from the console to an input file:

```
java LexicalAnalyser < input.txt
```

This file will now, when compiled, produce a parser which recognises any number of our operators and integers, printing to the screen what it has found. I will continue to do this until it reaches the end of input or finds an error.

It is time to do assignment 1. The deadline for this is Friday midnight of week two. You must write a Javacc file which recognises all legal terminal symbols in the grammar defined on page 77.

5.3 JjTree

So far all we have done is create a lexical analyser which merely decides whether each input is a token (terminal symbol) or not. The first time it finds an error it exits. The above program demonstrates how each token is specified and given a name. Then when each token is recognised we insert a piece of java code to execute. For a complete compiler we need to construct a tree. Having done that we can traverse the tree generating machine-code for our target machine. Luckily Javacc comes with a tree builder which inserts all the code necessary for the construction of such a tree. We will now build a complete parser using this tool – Jjtree.

First of all it should be noted that Jjtree is a *pre-processor* for Javacc, i.e. it takes our input file and outputs Javacc code with tree building code built in. The process is shown in figure 1.

The input file to Jjtree needs a file-type of **jjt** so let's just change the file we have been using to Lexan.jjt (from Lexan.jj) and add a few lines of code:

```
options{
   STATIC = true;
}

PARSER_BEGIN(LexicalAnalyser)
   class LexicalAnalyser{
      public static void main(String[] args)throws ParseException{
         LexicalAnalyser lexan = new LexicalAnalyser(System.in);
         SimpleNode root = lexan.start();
         System.out.println("Finished Lexical Analysis");
         root.dump("");
      }
   }
PARSER_END(LexicalAnalyser)

//Ignore all whitespace
```

```
SKIP:{" "|"\t"|"\n"|"\r"}

//Declare the tokens
TOKEN:{<INT: (["0"-"9"])+>}

//Now the operators
TOKEN:{<PLUS: "+">}
TOKEN:{<MINUS: "-">}
TOKEN:{<MULT: "*">}
TOKEN:{<DIV: "/">}

SimpleNode start():
{}
{
   (<PLUS>  {System.out.println("Plus");}     |
    <MINUS> {System.out.println("Minus");}    |
    <DIV>   {System.out.println("Divide");}   |
    <MULT>  {System.out.println("Multiply");} |
    <INT>{System.out.println("Integer");
   )+
   {return jjtThis;}
}
```

We have stated that calling lexan.start() will return an object of type **SimpleNode** and assigned this to a variable called **root**. This will be the root of our tree. **SimpleNode** is the default type of node returned by JavaTree methods. We will be seeing later how we can change this behaviour. The final line of the main method now calls `root.dump(‘‘’’)`, passing it an empty string. This is a method defined within the files produced automatically by JavaTree and its behaviour can be changed if we wish. At the moment it prints to the output a text description of the tree which has been built by the parse if successful. The start() method now has a return value of type simpleNode and the final line of the method returns **jjtThis**. jjtThis is the node currently under construction when it is executed. Running this code (Lexan.jjt) through jjtree produces a file called Lexan.jj. Run this through Javacc and we get the usual series of .java files. Finally invoking javac *.java produces our parser, this time with tree building capabilities built in. Running the compiled code with the following input file...

```
123
+
- * /
```

...produces the following output:

```
Integer
Plus
Minus
Multiply
Divide
Finished Lexical Analysis
start
```

It is the same as before except that we now see the result of printing out the tree (dump("")). It produces the single line **start**. This is in fact the name of the root of the tree. We have constructed a tree with a single node called start. Let's add a few production rules to get a clearer idea of what is happening.

```
SimpleNode start():
{}
{
   (multExp())*   {return jjtThis;}
}

void multExp():
{}
{
   intExp() <MULT> intExp()
}

void intExp():
{}
{
   <INT>
}
```

What we are doing here is stating that all internal nodes in our finished tree take the form of a method, while all terminal nodes are tokens which we have defined.

The code is the same until the definition of the start() method. The we have a series of methods, each of which contains a production rule for that non-terminal symbol. The start() rule expects 0 or more multExp's. A multExp is an intExp followed by a MULT terminal followed by another intExp. An intExp is simply an INT terminal. The grammar in BNF form is as follows:

$start \ \rightarrow multExp \mid multExp \ start \mid \epsilon$

$multExp \ \rightarrow \ intExp \ MULT \ intExp$

$intExp \ \rightarrow \ INT$

where INT and MULT are terminal symbols.

The output from this after being run on the input:

```
1*12
3 *14
45 * 6
```

is more interesting:

```
start
 multExp
  intExp
  intExp
 multExp
  intExp
  intExp
 multExp
  intExp
  intExp
```

Pictorially the tree we have built from the parse of the input is shown in figure 2

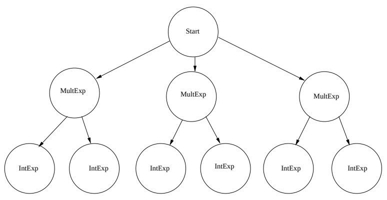


Figure 2: The tree built from the parse

We have a start node containing three multExp's, each of which contains two children, both of which are intExp's. **It is time for assignment 2. All you have to do is change the file you produced for assignment 1 into a jjtree file which contains the complete specification for the module grammar given on page 77**

Notice how the nodes take the same name as the method name in which they are defined? Let's change this behaviour:

```
SimpleNode start() #Root:{}{...}
void multExp() #MultiplicationExpression:{}{...}
void intExp() #IntegerExpression:{}{...}
```

All we have done is precede the method body with a local name (indicated by the # sign). This gives us the following, more easily understandable output:

```
Root
 MultiplicationExpression
  IntegerExpression
  IntegerExpression
 MultiplicationExpression
  IntegerExpression
  IntegerExpression
 MultiplicationExpression
  IntegerExpression
  IntegerExpression
```

To see the full effect of this change we can add two options to the first section of the file:

```
MULTI = true;
NODE_PREFIX = "";
```

The first tells the compiler builder to create different types of node, rather than just SimpleNode. If we don't add the second line the files thus generated will all have a default prefix. We will be using these different names later.

It is time for assignment 3. Change all the methods you have written so far so that they produce node names and types which are other than the names of the methods.

5.4 Information within nodes

With the addition of the above method of changing node names we have a great deal of information about interior nodes. What about terminal nodes? We know that we have, for example, an **identifier** or perhaps a **floating point number** but what is the name of that identifier, or the actual value of the float? We will need to know that information in order to be able to use it when, for example, generating code.

Looking at the list of files that Javacc generates for us we can see lots of nodes. Examining the code we can see that all of them extend **SimpleNode**. Let's look at the superclass:

```
/* Generated By:JJTree: Do not edit this line. SimpleNode.java */

public class SimpleNode implements Node {
  protected Node parent;
protected Node[] children;
protected int id;
protected LexicalAnalyser parser;

public SimpleNode(int i) {
  id = i;
}

public SimpleNode(LexicalAnalyser p, int i) {
  this(i);
  parser = p;
}

public void jjtOpen() {
}

public void jjtClose() {
}

public void jjtSetParent(Node n) { parent = n; }
public Node jjtGetParent() { return parent; }

public void jjtAddChild(Node n, int i) {
  if (children == null) {
    children = new Node[i + 1];
  } else if (i >= children.length) {
    Node c[] = new Node[i + 1];
    System.arraycopy(children, 0, c, 0, children.length);
    children = c;
  }
  children[i] = n;
}

public Node jjtGetChild(int i) {
  return children[i];
}

public int jjtGetNumChildren() {
  return (children == null) ? 0 : children.length;
}

/* You can override these two methods in subclasses of SimpleNode to
   customize the way the node appears when the tree is dumped.  If
```

```
     your output uses more than one line you should override
     toString(String), otherwise overriding toString() is probably all
     you need to do. */

  public String toString() { return LexicalAnalyserTreeConstants.jjtNodeName[id]; }
  public String toString(String prefix) { return prefix + toString(); }

  /* Override this method if you want to customize how the node dumps
     out its children. */

  public void dump(String prefix) {
    System.out.println(toString(prefix));
    if (children != null) {
      for (int i = 0; i < children.length; ++i) {
        SimpleNode n = (SimpleNode)children[i];
        if (n != null) {
          n.dump(prefix + " ");
        }
      }
    }
  }
}
```

The child nodes for this node are kept in an Array. **jjtgetChild(i)**, returns the i_{th} child node, **jjtGetNumChildren()** returns the number of children this node has and **dump()** alters how the nodes are printed to the screen when we call the Root note. Notice that dump() in particular recursively invokes dump() on all its children after printing itself, giving a **pre-order** printout of the tree. This can easily be changed to in-order or post-order.

We can add functionality to the node classes by adding it to SimpleNode, since all other nodes inherit from this. Add the following to the SimpleNode file:

```
protected String type = "Interior node";
protected int intValue;

public void setType(String type){this.type = type;}
public void setInt(int value){intValue = value;}

public String getType(){return type;}
public int getInt(){return intValue;}
```

By adding these members to the SimpleNode class all subclasses of SimpleNode will inherit them. Now change the code in the intExp method as follows:

```
void factor() #Factor:
{Token t;}
{
<ID> |
<FLOAT> |
t = <INT> {jjtThis.setType("int"); jjtThis.setInt(Integer.parseInt(t.image));} |
<OPENPAR> exp() <CLOSEPAR>
}
```

Each time javacc recognises a predefined token it produces an object of type **Token**. The class Token contains some useful methods of which two are seen here, **kind** and **image**. This is exactly the information we need to add to our nodes.

What we are saying here is that if the method **factor()** is called then it will be an **ID**, and **int**, a **float** or a parenthesised expression. If it is an int we add to the factor node the type and the value of this particular int.

Now, when we have a factor node we know, if it is an integer, what value it has. We can also, by adding to this code, know exactly which type of factor we are dealing with. Look at the **dump()** method:

```
public void dump(String prefix) {
   System.out.println(toString(prefix));
if(toString().equals("Factor")){
      if(getType().equals("int")){
         System.out.println("I am an integer and my value is " + getInt());}
      }
      if (children != null) {
         for (int i = 0; i < children.length; ++i) {
         SimpleNode n = (SimpleNode)children[i];
         if (n != null) {
            n.dump(prefix + " ");
         }
      }
   }
}
```

Now when we call the dump() method of our root node we will get all the information we had previously, plus, for each terminal **factor** node, if it is an integer we will know its value as well.

Time for assignment 4. Alter your code (in the factor method and the SimpleNode class) so that variables, integers and floats will all print out what they are, plus their value or name.

5.5 Conditional statements in jjtree

Many of the methods we have defined so far for the internal nodes have an optional part. This means that, for example, a statement list is a single statement followed by an optional semicolon and statement list. This use of recursion allows us to have as many statements in our grammar as we want. Unfortunately it means that when the tree is constructed it will not be possible to tell what exactly we have when we are examining a statement list node without counting to see how many child nodes there are. This unnecessarily complicates our code so let's see how to alter it:

```
void multExp() #MultiplicationExpression(>1):
{}
{
intExp() (<MULT> intExp())?
}
```

The production rule now states that a multExp is a single intExp followed by 0 or 1 intExps (indicated by the question mark). The (>1) expression says to return a MultiplicationExpression node **only** if there is more than one child. If the expression consists of just a single intExp the method returns that, rather than a MultiplicationExpression.

This very much simplifies the output of dump(), and also makes our job much easier when walking the tree to, for example, type check assignment statements, or generate machine code. Now when we visit an expression or statement we know **exactly** what sort of node we are dealing with.

5.6 Symbol Tables

Because our terminal nodes are treated differently from interior nodes we need to maintain a **symbol table** to keep track of them. We will use this during all phases of our finished compiler so we should look at it now. Of course we will implement a symbol table as a **hash table**, so that we can access our variables in constant time.

This would also be a convenient time to add some demands to our language, such as the requirement to declare variables before using them, and forbidding, or at least detecting, multiple declarations.

Our Variables have a **name**, a **type** and a **value**. This is a trivial class to implement:

```
public class Variable{
   private String type;
   private String name;
   private int intValue = 0;
   private double floatValue = 0.0;

   public Variable(int value, String name){
      this.name = name;
      type = "integer";
      intValue = value;
   }

   public Variable(double value, String name){
      this.name = name;
      type = "float";
      floatValue = value;
   }

   public String getType(){return type;}
   public String getString(){return name;}
   public int getInt(){return intValue;}
   public double getDouble(){return floatValue;}
}
```

Now that we have **Variable**s we need to declare a HashMap in which to put them. Once we have done that we add to our LexicalAnalyser class a method for adding variables to our symbol table:

```
public static void addToSymbolTable(Variable v){
   if(symbolTable.containsKey(v.getName())){
      System.out.println("Variable " + v.getName() + " multiply declared");
   }
   else{
      symbolTable.put(v.getName(), v);
   }
}
```

This quite simply interrogates the symbol table to see if the Variable we have just read exists or not. If it does we print out an error message, otherwise we add the new variable to the table. All that is left to do is invoke our function when we read a variable declaration:

```
void dec() #Declaration:
{String tokType;
 Token tokName;
 Variable v;}
```

```
{
   tokType = type() tokName = <ID>{
      if(tokType.equals("integer"))v = new Variable(0, tokName.image);
      else v = new Variable(0.0, tokName.image);
      addToSymbolTable(v);
   }
}

String type() #Type:
{Token t;}
{
   t = <FPN> {return "float";} |
   t = <INTEGER> {return "integer";}
}
```

Read the above carefully, type it in and make sure you understand exactly what is going on. Then start this week's assignment. **Assignment 5. Alter the code you have written so far so that methods with alternative numbers of children return the appropriate type of node. Then add the symbol table code given above. (Ideally you should declare the main method in a class of its own to avoid having all the extra methods and fields declared as static.) Then add code which detects if the writer is trying to use a variable which has not been declared, printing an error message if (s)he does so.**

5.7 Visiting the Tree

In altering the dump() method in the previous example we merely altered the **toString()** method. But let's think about what we are trying to do, especially with the abilities you have all demonstrated by progressing thus far in your studies.

Let us consider what we have here. We have a data structure (a tree) which we want to traverse, operating upon each node (for the time being we will merely print out details of the node). Those of you who took CS-211 - Programming with Objects and Threads will recognise immediately the **Visitor** pattern. (See, you always knew that module would come in handy.) Fortunately the writers of Javacc also took that module so they have already thought of this. If we add a VISITOR switch to the options section:

```
options{
MULTI = true;
NODE_PREFIX = "";
STATIC = false;
VISITOR = true;
}
```

...jjtree writes a **Visitor** interface for us to implement. It defines an **accept** method for each of the nodes allowing us to pass our implementation of the Visitor to each node. This is the Visitor interface for our current example:

```
/* Generated By:JJTree: Do not edit this line. ./LexicalAnalyserVisitor.java */

public interface LexicalAnalyserVisitor
{
  public Object visit(SimpleNode node, Object data);
  public Object visit(Root node, Object data);
  public Object visit(DeclarationList node, Object data);
  public Object visit(Declaration node, Object data);
  public Object visit(Type node, Object data);
  public Object visit(StatementList node, Object data);
  public Object visit(Statement node, Object data);
  public Object visit(IfStatement node, Object data);
  public Object visit(IfPart node, Object data);
  public Object visit(ElsePart node, Object data);
  public Object visit(RepeatStatement node, Object data);
  public Object visit(AssignmentStatement node, Object data);
  public Object visit(ReadStatement node, Object data);
  public Object visit(WriteStatement node, Object data);
  public Object visit(Expression node, Object data);
  public Object visit(BooleanOperator node, Object data);
  public Object visit(SimpleExpression node, Object data);
  public Object visit(AdditionOperator node, Object data);
  public Object visit(Term node, Object data);
  public Object visit(MultiplicationOperator node, Object data);
  public Object visit(Factor node, Object data);
}
```

Warning 1. Jjtree does not consistently rewrite all of its classes. This means that if you have been writing these classes and experimenting with them as you have read this (highly recommended) you should delete all files except the .jjt file, add the new options and then recompile with Jjtree.

Warning 2. We will be writing an implementation of the Visitor interface. This is our own file and is not therefore rewritten by Jtree. Therefore from now on if we alter the .jjt file and then compile byte code with javac *.java we may get errors if we have not checked our implementation of the Visitor first.

Now instead of **dump()**ing our tree we **visit** it with a visitor. By implementing the visit() method of each particular type of node in a different way we can do anything we want, without altering the construct of the nodes themselves, exactly what the Visitor pattern was designed to do.

```
public class MyVisitor implements ParserVisitor{

   public Object visit(Root node, Object data){
   System.out.println("Hello. I am a Root node");
   if(node.jjtGetNumChildren() == 0){
      System.out.println("I have no children");
   }
   else{
      for(int i = 0; i < node.jjtGetNumChildren(); i++){
         node.children[i].jjtAccept(this, data);
      }
   }
}
```

Simple isn't it! **Assignment 6. Write a Visitor class, implementing LexicalAnalyserVisitor, which visits each node in turn, printing out the type of node it is and how many children it has.**

5.8 Semantic Analyses

Now that we have our symbol table and a Visitor interface it is easy to implement a Visitor object which can do anything we wish with the data held in the tree which our parser builds. In the last assignment all we did was print out details of the type of node. In a real compiler we would want eventually to generate machine code for our target machine. We will not be doing that as an assignment, though if you are enjoying this module I hope you might want to do this out of interest. For your final assignment you are going to be doing some semantic analysis, to ensure that you do not attempt to assign incorrect values to variables, e.g. a floating point number to an integer variable.

You have three weeks to do this assignment so there will be plenty of help given during lectures and you have plenty of time to come and see me, or ask Program Advisory for help. It should be clear thet we only need to write code for assignment statements. Let's look at this:

```
void assignStatement() #AssignmentStatement:
{}
{
```

```
<ID> <ASSIGN> exp()
}
```

When we read the ID token we can check its type by looking it up in the symbol table as we have done before. All we need to do is decide what the type of the exp is. This is your assignment. Start from the bottom nodes of your tree and work your way up to the expression.

6 Tiny

For your coursework you will be writing a compiler which compiles a **while** language, similar to the one you met in Theory of Programming Languages, into an assembly language called **tiny**. A tiny machine simulator can be downloaded from the course web-site to test your compiler. This section describes the tiny language.

6.1 Basic Architecture

tiny consists of a read–only instruction memory, a data memory, and a set of eight general–purpose registers. These all use nonnegative integer addresses, beginning at 0. Register 7 is the program counter and is the only special register, as described below.

The following C declarations will be used in the descriptions which follow:

```
#define IADDR_SIZE...
   /* size of instruction memory */
#define DADDR_SIZE...
   /* size of data memory */
#define NO_REGS 8
   /* number of registers */
#define PC_REG 7

Instruction iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

tiny performs a conventional fetch–execute cycle:

```
do
   /* fetch */
   currentInstruction = iMem[reg[pcRegNo]++];
   /* execute current instruction */
   ...
   ...
while (!(halt|error));
```

At start–up the tiny machine sets all registers and data memory to 0, then loads the value of the highest legal address (namely DADDR_SIZE - 1) into dMem[0]. This allows memory to easily be added to the machine since programs can find out during execution how much

RO Instructions Format : opcode r, s, t	
Opcode	Effect
HALT	stop execution (operands ignored)
IN	reg[r] $\leftarrow$ integer value read from the standard input. (s and t ignored)
OUT	reg[r] $\rightarrow$ the standard output (s and t ignored)
ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t] (may generate ZERO_DIV)
RM Instructions Format : opcode r, d(s)	a = d + reg[s]; any reference to dMem[a] generates DMEM_ERR if $a < 0$ or $a \geq$ DADDR_SIZE
Opcode	Effect
LD	reg[r] = dMem[a] (load r with memory value at a)
LDA	reg[r] = a (load address a directly into r)
LDC	reg[r] = d (load constant d directly into r. s is ignored)
ST	dMem[a] = reg[r] (store value in r to memory location a)
JLT	if (reg[r] < 0) reg[PC_REG] = a (jump to instruction a if r is negative, similarly for the following)
JLE	if(reg[r] $\leq$ 0) reg[PC_REG] = a
JGE	if(reg[r] $\geq$ 0) reg[PC_REG] = a
JGT	if(reg[r] > 0) reg[PC_REG] = a
JEQ	if(reg[r] == 0) reg[PC_REG] = a
JNE	if(reg[r] != 0) reg[PC_REG] = a

Table 1: The tiny instruction set

memory is available. The machine then starts to execute instructions beginning at iMem[0]. The machine stops when a HALT instruction is executed. The possible error conditions include IMEM_ERR, which occurs if reg[PC_REG] < 0 or reg[PC_REG] $\geq$ IADDR_SIZE in the fetch step, and the two conditions DMEM_ERR and ZERO_DIV, which occur during instruction execution execution as described below.

The instruction set of the machine is given in table 1, together with a brief description of the effect of each instruction.

There are two basic instruction formats: register–only, or RO instructions, and register–memory, or RM instructions. A register–only instruction has the format

opcode r,s,t

where the operands **r, s, t** are legal registers (checked at load time). Thus, such instructions are three–address, and all three addresses must be registers. All arithmetic instructions are limited to this format, as are the two primitive input/output instructions.

A register–memory instruction has the format

opcode r,d(s)

In this code **r** and **s** must be legal registers (checked at load time), and **d** is a positive or negative integer representing an offset. This instruction is a two–address instruction, where the first address is always a register and the second address is a memory address **a** given by **a** = **d** + **reg**[**r**], where **a** must be a legal address ($0 \leq a <$ DADDR_SIZE). If **a** is out of the legal range, then **DMEM_ERR** is generated during execution.

RM instructions include three different load instructions corresponding to the three addressing modes "load constant" (**LDC**), "load address" (**LDA**), and "load memory" (**LD**). In addition there is one store instruction and six conditional jump instructions.

In both RO and RM instructions, all three operands must be present, even though some of them may be ignored. This is due to the simple nature of the loader, which only distinguishes between the two classes of instruction (RO and RM) and does not allow different formats within each class. (This actually makes code generation easier since only two different routines will be needed).

Table 1 and the discussion of the machine up to this point represent the complete tiny architecture. In particular there is no hardware stack or other facilities of any kind. No register except the PC is special in any way, there is no sp or fp. A compiler for the machine must therefore maintain any runtime environment organisation entirely manually. Although this may be unrealistic it has the advantage that all operations must be generated explicitly as they are needed.

Since the instruction set is minimal we should give some idea of how they can be used to achieve some more complicated programming language operations. (Indeed, the machine is adequate, if not comfortable, for even very sophisticated languages).

i. The target register in the arithmetic, **IN**, and load operations comes first, and the source register(s) come second, similar to the 80x85 and unlike the Sun SparcStation. There is no restriction on the use of registers for sources and targets; in particular the source and target registers may be the same.

ii. All arithmetic operations are restricted to registers. No operations (except load and store operations) act directly on memory. In this the machine resembles RISC machines such as the Sun SparcStation. On the other hand the machine has only 8 registers, while most RISC processors have many more.

iii. There are no floating point operations or floating point registers. (But the language you are compiling has no floating point variables either).

iv. There is no restriction on the use of the PC in any of the instructions. Indeed, since there is no unconditional jump, it must be simulated by using the PC as the target register in an **LDA** instruction:

LDA 7, d(s)

This instruction has the effect of jumping to location **d** + **reg**[**s**].

v. There is also no indirect jump instruction, but it too can be imitated if necessary, by using an **LD** instruction. For example,

LD 7,0(1)

jumps to the instruction whose address is stored in memory at the location pointed to by register 1.

vi. The conditional jump instructions (JLT etc.), can be made relative to the current position in the program by using the PC as the second register. For example

JEQ 0,4(7)

causes the machine to jump five instructions forward in the code if register 0 is 0. An unconditional jump can also be made relative to the PC by using the PC twice in an **LDA** instruction. Thus

LDA 7,-4(7)

performs an unconditional jump three instructions backwards.

vii. There are no procedure calls or **JSUB** instruction. Instead we must write

LD 7,d(s)

which has the effect of jumping to the procedure whose entry address is **dMem**[**d** + **reg**[**s**]]. Of course we need to remember to save the return address first by executing something like

LDA 0,1(7)

which places the current PC value plus one into **reg**[**0**].

6.2 The machine simulator

The machine accepts text files containing TM instructions as described above with the following conventions:

- An entirely blank line is ignored
- A line beginning with an asterisk is considered to be a comment and is ignored.
- Any other line must contain an integer instruction location followed by a colon followed by a legal instruction. Any text after the instruction is considered to be a comment and is ignored.

The machine contains no other features–in particular there are no symbolic labels and no macro facilities.

We now have a look at a TM program which computes the factorial of a number input by the user.

```
* This program inputs an integer, computes
* its factorial if it is positive,
* and prints the result

0: IN 0,0,0      r0 = read
1: JLE 0,6(7)    if 0 < r0 then
2: LDC 1,1,0        r1 = 1
3: LDC 2,1,0        r2 = 1
                    * repeat
4: MUL 1,1,0           r1 = r1 * r0
5: SUB 0,0,2           r0 = r0 - r2
6: JNE 0,-3(7)      until r0 = 0
7: OUT 1,0,0        write r1
8: HALT 0,0,0    halt
* end of program
```

Strictly speaking there is no need for the **HALT** instruction at the end of the code since the machine sets all instruction locations to **HALT** before loading the program, however it is useful to put it in as a reminder, and also as a target for jumps which wish to end the program.

If this program were saved in the file **fact.tm** then this file can be loaded and executed as in the following sample session. (The simulator assumes the file extension **.tm** if one is not given).

tm fact TM simulation (enter h for help) ... Enter command: g Enter value for IN instruction: 7 OUT instruction prints: 5040 HALT: 0,0,0 Halted Enter command: q Simulation done.

The **g** command stands for "go", meaning that the program is executed starting at the current contents of the PC (which is 0 after loading), until a **HALT** instruction is read. The complete list of simulator commands can be obtained by using the command **h**.