UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Computer Science Tripos
Part IB [P]
Part II General [C]
Diploma in Computer Science [D]

Compiler Construction

(Part 1 of 2)

http://www.cl.cam.ac.uk/Teaching/2001/CompConstr

Alan Mycroft am@cl.cam.ac.uk

2001–2002 (Lent Term)

Summary

The first part of this course covers the design of the various parts of a fairly basic compiler. The second part of the course considers various language features and concepts common to many programming languages, together with an outline of the kind of run-time data structures and operations they require.

The course is intended to study compilation of a range of languages and accordingly syntax for example constructs will be taken from various languages (with the intention that the particular choice of syntax is reasonably clear).

In terms of programming languages in which parts of compilers themselves are to be written, the preference varies between pseudo-code (as per the 'Data Structures and Algorithms' course) and language features (essentially) common to C/C++/Java. The language Standard ML (which the Diploma and Part II (general) students will see as part of the 'Functional Programming' course) is used when it significantly simplifies the code compared to C.

The following books contain material relevant to the course.

Compilers—Principles, Techniques, and Tools
A.V.Aho, R.Sethi and J.D.Ullman
Addison-Wesley (1986)
Ellis Horwood (1982)

Compiler Design in Java/C/ML (3 editions)
A.Appel
Cambridge University Press (1996)

Compiler Design
R.Wilhelm and D.Maurer
Addison Wesley (1995)

Introduction to Compiling Techniques
J.P.Bennett
McGraw-Hill (1990)

A Retargetable C Compiler: Design and Implementation
C.Frazer and D.Hanson
Benjamin Cummings (1995)

Compiler Construction
W.M.Waite and G.Goos
Springer-Verlag (1984)

High-Level Languages and Their Compilers
D.Watson
Addison Wesley (1989)

Acknowledgments

Various parts of these notes are due to or based on material developed by Dr M. Richards of the Computer Laboratory. Dr G. Bierman provided the call-by-value lambda-form for Y.

Teaching and Learning Guide

[This is a Computer Laboratory mandated section of all lecture courses.]

The lectures largely follow the syllabus for the course which is as follows.

```
Survey of execution mechanisms.
  The spectrum of interpreters and compilers; compile-time and run-time.
  Structure of a simple compiler. Java virtual machine.

Lexical analysis and syntax analysis.
  Regular expressions and finite state machine implementations. Grammars,
  Chomsky classification of phrase structured grammars. Parsing
  algorithms: recursive descent, precedence and SLR(k). Syntax error
  recovery.

Simple type-checking.
  Type of an expression determined by type of subexpressions; inserting
  coercions.  Polymorphism.

Translation phase.
  Intermediate code design. Translation of commands, expressions and
  declarations. Translating variable references into access paths.

Code generation.
  Typical machine codes. Code generation from the parse tree and from
  intermediate code. Simple optimisation.

Compiler compilers.
  Summary of Lex and Yacc.

Runtime.
  Object modules and linkers. Resolving external references.
  Static and dynamic linking.  Debuggers, break points and
  single step execution. Profiling, portability.

Survey of language constructs and their implementation.
  Expressions, applicative structure, lambda expressions. Environments
  and a simple lambda evaluator. Evaluation of function calls using
  static and dynamic chains, Dijkstra displays. Situations where a simple
  stack is inadequate. Objects and inheritance. Implementation of labels,
  jumps, arrays and exceptions. L-value and r-value; choices for argument
  passing and free-variable association. Dynamic and static binding.
  Dynamic and static types, polymorphism.
```

A good source of exercises is the past 10 or 20 years' (sic) Tripos questions in that most of the basic concepts of block-structured languages and their compilation to stack-oriented code were developed in the 1960s. The course 'Optimising Compilation' in CST (part II) considers more sophisticated techniques for the later stages of compilation and the course 'Comparative Programming Languages' considers programming language concepts in rather more details.

Note: these notes have this year been re-written to use Java and the JVM as the intermediate code. I would be grateful for comments identifying errors or readability problems.

1 Introduction and Overview

A *compiler* is a program to translate the source form of a program into its equivalent machine code or relocatable binary form. The number of compilers that exist is very large and considerable human effort has been expended in constructing them. As a result most parts of the compilation process have become well understood and the job of writing a compiler is no longer the difficult task it once was. A compiler tends to be a large program (typically 10,000 to 50,000 machine instructions for a simple compiler and more that a million for serious optimising compilers) and it is wise to structure it in order to make its individual components small enough to think about and handle conveniently.

If a language is sufficiently simple it and is designed suitably, it can be compiled by a *one pass compiler*, that is, it can be compiled a small piece (often just a statement) at a time. During the compilation process the data types and allocated locations of variables are remembered together with any other information that may be needed later on during the compilation. The space required for this is substantially less than the space needed to hold the entire program and there is usually no limit on the size of the program than may be compiled in this way. Although such a compiler can be simple and fast, such a route is generally avoided because (a) it is hard to optimise the code and (b) the techniques developed for multi-pass compilation are better known.

Most languages have features that make compilation in a single pass either difficult or impossible. For example, in Java the definition of names may occur many lines after they are first used, as in:

```
class A {
  public int g() { return f(); }
  // ... many lines before we find ...
  public int f() { ... }
}
```

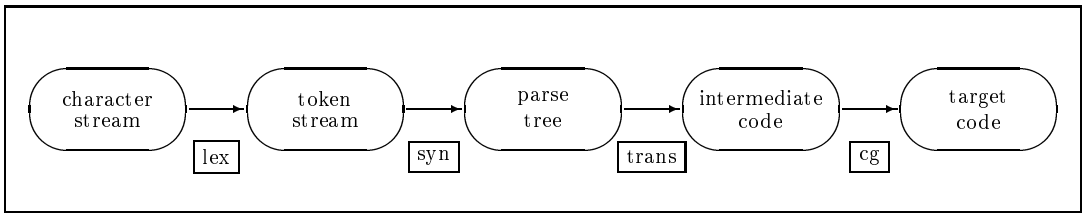
Similarly in ML, consider programs like:

```
val g = 3;
fun f(x) = ... g ...
and g(x) = ...;
```

For most current programming languages, it is normal to compile in a number of stages (or phases, or passes) with the output of one pass being the input of the next. A compiler designed in this way is called a *multi-pass compiler*.

1.1 The structure of a typical multi-pass compiler

We will take as an example a compiler with four passes.



1.2 The lexical analyser

This reads the characters of the source program and recognises the basic syntactic components that they represent. It will recognise identifiers, reserved words, numbers, string constants and all other basic symbols (or tokens) and throw away all other ignorable text such as spaces, newlines and comments. For example, the result of lexical analysis of the following program phrase:

```
{ let x = 1;
  x := x + y;
}
```

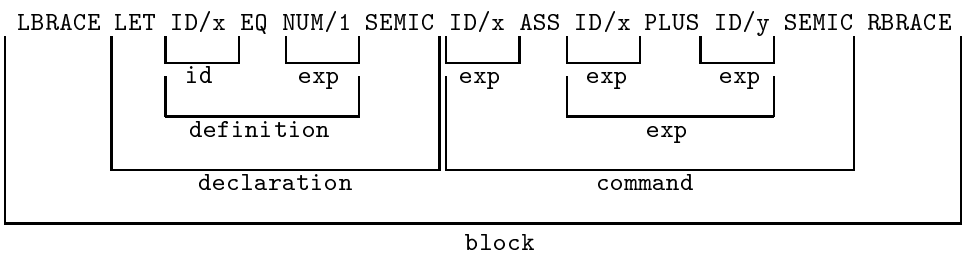
might be:

```
LBRACE LET ID/x EQ NUM/1 SEMIC ID/x ASS ID/x PLUS ID/y SEMIC RBRACE
```

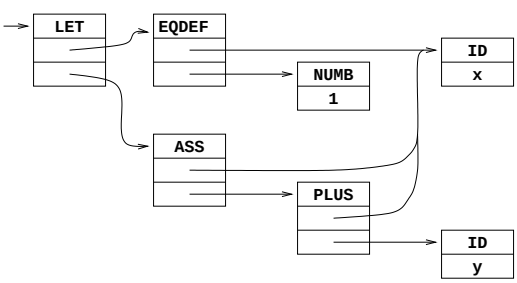
Lexical tokens are often represented in a compiler by small integers; for composite tokens such as identifiers, numbers, etc. additional information is passed by means of pointers into appropriate tables; for example calling a routine `lex()` might return the next token while setting a global variable `lex_aux_string` to the string form of an identifier when `ID` is returned, similarly `lex_aux_int` might be set to the binary representation of an integer when `NUM` is returned.

1.3 The syntax analyser

This will recognise the syntactic structure of the sequence of tokens delivered by the lexical analyser. The result of syntax analysis is often a tree representing the syntactic structure of the program. This tree is sometime called an *abstract syntax tree*. The syntax analyser would recognise that the above example parses as follows:



and it might be represented within the compiler by the following tree structure:



where the tree operators (e.g. `LET` and `EQDEF`) are represented as small integers.

In order that the tree produced is not unnecessarily large it is usually constructed in a condensed form as above with only essential syntactic features included. It is, for instance, unnecessary to represent the expression `x` as a `<sum>` which is a `<factor>` which is a `<primary>` which is an `<identifier>`. This would take more tree space space and would also make later processing less convenient. Similarly, nodes representing identifiers are stored uniquely—this saves store and reduces the problem of comparing whether identifiers are equal to simple pointer equality. The phrase 'abstract syntax tree' refers to the fact the only semantically important items are incorporated into the tree; thus `a+b` and `((a)+(((b)))` might have the same representation, as might `while (e) C` and `for(;e;) C`.

1.4 The translation phase

This pass flattens the tree into a linear sequence of intermediate object code. At the same time it can deal with

1. the scopes of identifiers,
2. declaration and allocation of storage,
3. selection of overloaded operators and the insertion of automatic type transfers.

However, nowadays often a separate 'type-checking' phase is run on the syntax tree below translation. This phase at least conceptually modifies the syntax tree to indicate which version of an overloaded operator is required. We will say a little more about this in section 6.5.

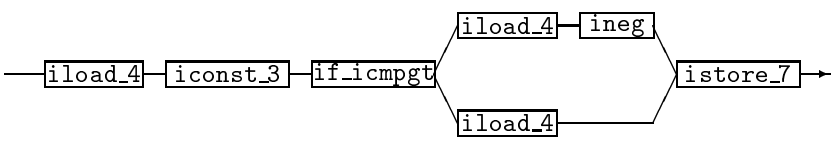
The intermediate object code for the statement:

```
y := x<=3 ? -x : x
```

might be as follows (using for JVM as an example intermediate code):

```
iload_4        load x (4th load variable)
iconst_3       load 3
if_icmpgt L36  if greater (i.e. condition false) then jump to L36
iload_4        load x
ineg           negate it
goto L37       jump to L37
label L36
iload_4        load x
label L37
istore_7       store y (7th local variable)
```

Alternatively, the intermediate object code could be represented within the compiler as a directed graph[1] as follows:



1.5 The code generator

This pass converts the intermediate object code into machine instructions and outputs them in either assembly language or relocatable binary form in so-called *object files*. The code generator is mainly concerned with local optimisation, the allocation of target-machine registers and the selection of machine instructions. Using the above intermediate code form of

```
y := x<=3 ? -x : x
```

we can easily produce (simple if inefficient) ARM code of the form using the traditional downwards-growing ARM stack:

```
     LDR   r0,[fp,#40-16]    load x (4th local variable, out of 10 say)
     MOV   r1,#3             load 3
     CMP   r0,r1
     BGT   L36               if greater then jump to L36
     LDR   r0,[sp,#40-16]    load x
     RSB   r0,r0,#0          negate it
     STMDB sp!,{r0}          i.e. PUSH r0 (to local stack)
     B     L37               jump to L37
L36: LDR   r0,[sp,#40-16]    load x
     STMDB sp!,{r0}          i.e. PUSH r0 (to local stack)
L37: LDMIA sp!,{r0}          i.e. POP r0 (from local stack)
     STR   r0,[sp,#40-28]    store y (7th local variable)
```

[1] The Part II course on optimising compilers will take this approach but here it would merely add to the weight of concepts for no clear gain.

This code has the property that it can simply be generated from the above JVM code on an instruction-by-instruction basis (which explains why I have not hand-optimised the `PUSH`es and `POP`s away).

When compilers produce textual output in a file (for example `gcc`) it is necessary to have a separate program (usually called an *assembler*) to convert this into an object file. An assembler is effectively a simple compiler which reads an instruction at a time from the input stream and writes the binary representation of these into the object file output. One might well argue that this is a separate pass, which separates the formatting of the binary file from the issue of deciding which code to generate.

1.6 Compiler Summary

The four passes just described form a clear-cut logical division of a compiler, but are not necessarily applied in sequence. It is, for instance, common for the lexical analyser to be a subroutine of the syntax analyser and for it to be called whenever the syntax analyser requires another lexical token. In simple compilers It is also quite common for the translation phase and the code generator to be merged into one pass. Some compilers have additional passes, particularly for complex language features or if a high degree of optimisation is required. Examples might be separating the type-checking phase from the translation phase (for example as code which replaces source-level types in the syntax tree with more machine-oriented type information), or by adding additional phases to optimise the intermediate code structures (e.g. common sub-expression elimination which reworks code to avoid re-calculating common subexpressions).

The advantages of the multi-pass approach can be summarised as.

1. It breaks a large and complicated task into smaller, more manageable pieces. [Anyone can juggle with one ball at a time, but juggling four balls at once is much harder.]
2. Modifications to the compiler (e.g. the addition of a synonym for a reserved word, or a minor improvement in compiler code) often require changes to one pass only and are thus simple to make.
3. A multi-pass compiler tends to be easier to describe and understand.
4. More of the design of the compiler is language independent. It is sometimes possible to arrange that all language dependent parts are in the lexical and syntax analysis (and type-checking) phases.
5. More of the design of the compiler is machine independent. It is sometimes possible to arrange that all machine dependent parts are in the code generator.
6. The job of writing the compiler can be shared between a number of programmers each working on separate passes. The interface between the passes is easy to specify precisely.

1.7 Reading compiler output

Reading assembly-level output is often useful to aid understanding of how language features are implemented; if the compiler can produce assembly code directly then use this feature, for example

```
gcc -S foo.c
```

will write a file `foo.s` containing assembly instructions. Otherwise, use a *disassembler* to convert the object file back into assembler level form, e.g. in Java

```
javac foo.java
javap -c foo.java
```

Note that the '`-c`' switch seems not to work on all versions of `javap`. It works at least on the Linux PWF facility and also on the following variants of `thor`:

```
Solaris Release 8 [hammer] Linux Red Hat Release 7.1 [belt, gloves] (Thor)
```

1.8 The linker

Most programs written in high-level languages are not self-contained. In particular they may be written in several modules which are separately compiled, or they may merely use library routines which have been separately compiled. With the exception of Java (or at least the current implementations of Java), the task of combining all these separate units is performed by a *linker* (on Linux the linker is called `ld` for 'loader' for rather historical reasons). A linker concatenates its provided object files, determines which library object files are necessary to complete the program and concatenates all these to form a single *executable* output file. Thus classically there is a total separation between the idea of *compile-time* (compiling and linking commands) and *run-time* (actual execution of a program).

In Java, the tendency is to write out what is logically the intermediate language form (i.e. JVM instructions) into a `.class` file. This is then dynamically loaded (i.e. read at run-time) into the running application. Because (most) processors do not execute JVM code directly, the JVM code must be interpreted (i.e. simulated by a program implementing the JVM virtual machine). An alternative approach is to use a so-called *just in time* (JIT) compiler in which the above code-generator phase of the compiler is invoked to convert the loaded JVM code into native machine instructions (selected to match the hardware on which the Java program is running). This idea forms part of the "write once, compile once, run anywhere" model which propelled Java into prominence when the internet enabled `.class` files (applets) to be down-loaded to execute under an Internet *browser*

1.9 Compilers and Interpreters

The above discussion on Java execution mechanism highlights one final point. Traditionally user-written code is translated to machine code appropriate to its execution environment where it is executed directly by the hardware. The JVM *virtual machine* above is an alternative of an *interpreter-based* system. Other languages which are often interpreted are Basic, various scripting languages (for shells, spreadsheets and the like), perl etc. The common thread of these languages is that traditional compilation as above is not completed and some data structure analogous to the input data-structure of one of the above compiler phases. For example, some Basic interpreters will decode lexical items whenever a statement is executed (thus syntax errors will only be seen at run-time); others will represent each Basic statement as a parse tree and refuse to accept syntactically invalid programs (so that run-time never starts). What perhaps enables Java to claim to be a compiled language is that compilation proceeds far enough that all erroneous programs are rejected at compile-time. Remaining run-time problems (e.g. de-referencing a NULL pointer) are treated as *exceptions* which can be handled within the language.

I will present a rather revisionist view on compilers and interpreters in the second part of the course.

2 Lexical analysis

This is a critical part of a simple compiler since it can account for more than 50% of the compile time. This is because:

1. character handling tends to be expensive,

2. there are a large number of characters in a program compared with the number of lexical tokens, and

3. the lexical analyser usually constructs name tables and performs the binary conversion of constants.

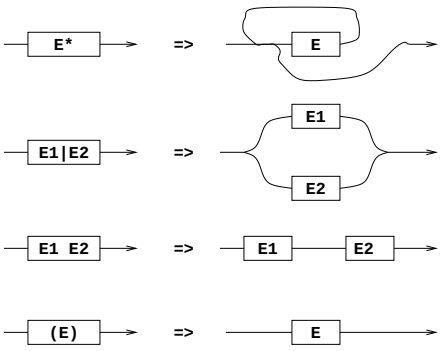
2.1 Regular expressions

The recognition of lexical tokens is straightforward and does not require a sophisticated analyser. This results from the simple syntax of lexical tokens. It is usually the case that all the lexical tokens of a language can be described by *regular expressions*, which then implies that the recognition can be performed by a *finite state algorithm*.

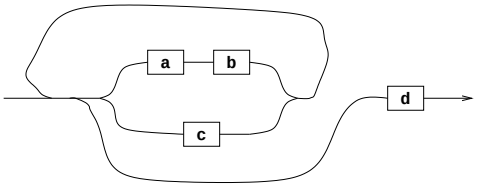
A *regular expression* is composed of characters, operators for concatenation (juxtaposition), alternation (`|`) and repetition `*`, and parentheses are used for grouping. For example, `(a b | c)* d` is a regular expression. It can be regarded as a specification of a potentially infinite set of strings, in this case:

```
d
abd      cd
ababd    abcd    cabd    ccd
etc.
```

This is best derived by constructing the corresponding *transition diagram* by repeated application of the following rules.



The transition diagram for the expression "`(a b | c)* d`" is:



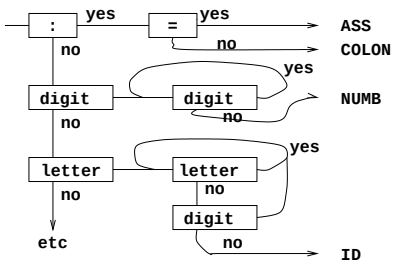
This can be regarded as a generator of strings by applying the following algorithm:

1. Follow any path from the starting point to any accessible box.

2. Output the character in the box.

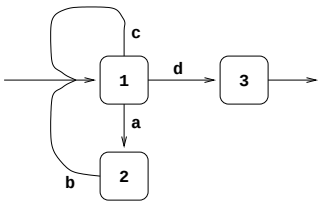
3. Follow any path from that box to another box (possibly the same) and continue from step (2). The process stops when the exit point is reached.

We can also use the transition diagram as the basis of a recogniser algorithm. For example, an analyser to recognise `:=`, `:`, `<numb>` and `<id>` might have the following transition diagram:



Optimisation is possible (and needed) in the organisation of the tests. This method is only satisfactory if one can arrange that only one point in the diagram is active at any one time.

It is sometimes convenient to draw the transition diagram as a directed graph with labelled edges. For example, the graph for the expression "`(a b | c)* d`" can be represented as follows:

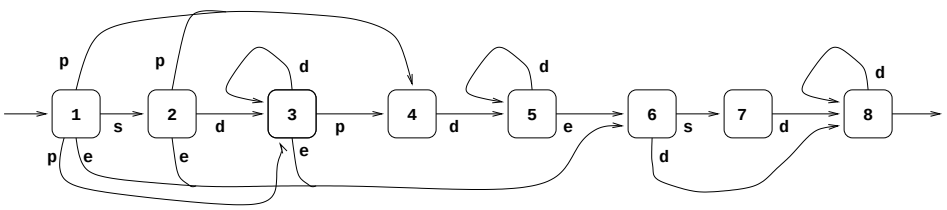


With state 3 designated an accepting state, this graph is a finite state acceptor for the given regular expression. The acceptor is easily implemented using a *transition matrix* to represent the graph.

We will demonstrate the method by considering the following syntax of floating point numbers (the '$\longrightarrow$' notation is introduced below in section 3):

```
N  —→  U  |  s U              Number
U  —→  D  |  E  |  D E        Unsigned number
D  —→  J  |  F  |  J F        Unsigned decimal number
E  —→  e I                    Exponent part
F  —→  p J                    Decimal fraction
I  —→  J  |  s J              Integer
J  —→  d  |  d J              Unsigned integer

where  s  is a sign                + or -
       e  is the exponent symbol   E
       p  is the decimal point     .
       d  is a digit               0-9
```

The corresponding graph is:



The corresponding matrix is as follows:

	s	d	p	e	other
S1	S2	S3	S4	S6	.
S2	.	S3	S4	S6	.
S3	.	S3	S4	S6	acc
S4	.	S5	.	.	.
S5	.	S5	.	S6	acc
S6	S7	S8	.	.	.
S7	.	S8	.	.	.
S8	.	S8	.	.	acc

In a program that uses this technique each matrix entry would specify the address of some code to deal with the transition[2] and note the next matrix row to be used. The entry `acc` would point to the code that processes a complete floating point number. Blank entries correspond to syntactic error conditions.

In general, this technique is fast and efficient, but if used on a large scale it requires skill and cunning to reduce the size of the matrix and to reduce the number of separate transition routines.

3 Phrase structured grammars

A *grammar* consists of an *alphabet* of symbols (think of these as characters to lexing or tokens resulting from lexing) and set of rules for generating a *language* (set of strings) of such symbols. For example, if the alphabet were the set of all letters `{a ... z}` and the rule were "generate all strings of length three" we would have a language whose strings are:

```
aaa, aab, ... zzy, zzz
```

A more useful form of grammar is the *phrase structured grammar* where the generation rule is given as a set of *productions*. It is first necessary to break the alphabet into two sets of symbols: *terminal symbols* like `a`, `b`, `c` above which may occur in the input text and *non-terminals* like `Term` or `Declaration` which do not occur in input text but summarise the structure of a sequence of symbols. The most general form of a production is:

$$A_1\ A_2 \cdots\ A_m \longrightarrow B_1\ B_2 \cdots\ B_n$$

where the A_i and B_i are symbols and $A_1\ A_2 \cdots\ A_m$ contains at least one non-terminal. This rule specifies that if $A_1\ A_2 \cdots\ A_m$ occurs in a string belonging to the grammar then the string formed by replacing $A_1\ A_2 \cdots\ A_m$ by $B_1\ B_2 \cdots\ B_n$ also belongs to the grammar (note that the symbol '`::=`' is sometimes used as an alternative to '$\longrightarrow$'). There must be a unique non-terminal `S`, say, called the *sentence symbol* that occurs by itself on the left hand side of just one production. Any string that can be formed by the application of productions is called a *sentential form*. A sentential form containing no non-terminals is called a *sentence*. The problem of syntax analysis is to discover which series of applications of productions that will convert the sentence symbol into the given sentence.

It is useful to impose certain restrictions on $A_1\ A_2 \cdots\ A_m$ and $B_1\ B_2 \cdots\ B_n$ and this has been done by Chomsky to form four different types of grammar. The most important of these in the Chomsky Type 2 grammar.

3.1 Type 2 grammar

In the Chomsky type 2 grammar the left hand side of every production is restricted to just a single non-terminal symbol. Such symbols are often called *syntactic categories*. Type 2 grammars are known as *context free grammars* and have been used frequently in the specification of the syntax of programming languages, most notably Algol 60 where it was first used. The notation is sometime called *Backus Naur Form* or BNF after two of the designers of Algol 60. A simple example of a type 2 grammar is as follows:

[2] E.g. multiply the current total by 10 and add on the current digit

```
S  ⟶  A B
A  ⟶  a
A  ⟶  A B b
B  ⟶  b c
B  ⟶  B a
```

A slightly more convenient way of writing the above grammar is:

```
S  ⟶  A B
A  ⟶  a    |  A B b
B  ⟶  b c  |  B a
```

The alphabet for this grammar is `{S, A, B, a, b, c, d}`. The non-terminals are `S, A, B` being the symbols occurring on the left-hand-side of productions, with `S` being identified as the start symbol. The terminal symbols are `a, b, c, d`, these being the characters that only appear on the right hand side. Sentences that this grammar generates include, for instance:

```
abc
abcbbc
abcbca
abcbbcaabca
```

Where the last sentence, for instance, is generated from the sentence symbol by means of the following productions:

```
S
|
A---------------B
|               |
A-------B-----b B---a
|       |     | |   |
A-B---b B---a | b-c |
| |   | |   | | | | |
a b-c | b-c | | | | |
| | | | | | | | | | |
a b c b b c a b b c a
```

A grammar is ambiguous if there are two or more ways of generating the same sentence. Convince yourself that the follow three grammars are ambiguous:

```
a)  S  ⟶  A B
    A  ⟶  a  |  a c
    B  ⟶  b  |  c b

b)  S  ⟶  a T b  |  T T
    T  ⟶  a b  |  b a

c)  C  ⟶  if E then C else C  |  if E then C
```

Clearly every type 2 grammar is either ambiguous or it is not. However, it turns out that it is not possible to write a program which, when given an arbitrary type 2 grammar, will terminate with a result stating whether the grammar is ambiguous or not. It is surprisingly difficult for humans to tell whether a grammar in ambiguous. One example of this is that the productions in (c) above appeared in the original Algol 60 published specification. As an exercise, determine whether the example grammar given above is ambiguous.

For completeness, the other grammars in the Chomsky classification are as follows.

3.2 Type 0 grammars

Here there are no restrictions on the sequences on either side of productions. Consider the following example:

```
S    ⟶  a S B C  |  a B C
C B  ⟶  B C
a B  ⟶  a b
b B  ⟶  b b
b C  ⟶  b c
c C  ⟶  c c
```

This generates all strings of the form $a^n b^n c^n$ for all $n \geq 1$.

To derive `aaaaabbbbbccccc`, first apply `S` ⟶`aSBC` four times giving:

```
aaaaSBCBCBCBC
```

Then apply `S` ⟶`aBC` giving:

```
aaaaaBCBCBCBCBC
```

Then apply CB ⟶BC many times until all the `C`s are at the right hand end.

```
aaaaaBBBBBCCCCC
```

Finally, use the last four productions to convert all the `B`s and `C`s to lower case giving the required result. The resulting parse tree is as follows:

```
S
a-S-----------------------B-C
| a-S-----------------B-C | |
| | a-S-----------B-C | | | |
| | | a-S-----B-C | | | | | |
| | | | a-B-C | | | | | | | |
| | | | a-b B-C B-C B-C B-C |
| | | | | b-b B-C B-C B-C | |
| | | | | | b-b B-C B-C | | |
| | | | | | | b-b B-C | | | |
| | | | | | | | b-b | | | | |
| | | | | | | | | b-c | | | |
| | | | | | | | | | c-c | | |
| | | | | | | | | | | c-c | |
| | | | | | | | | | | | c-c |
| | | | | | | | | | | | | c-c
| | | | | | | | | | | | | | | |
a a a a a b b b b b c c c c c
```

As a final remark on type 0 grammars, it should be clear that one can write a grammar which essentially specifies the behaviour of a Turing machine, and syntax analysis in this case is equivalent to deciding whether a given string is the answer to some program. This is undecidable and syntax analysis of type 0 grammars is thus, in general, undecidable.

3.3 Type 1 grammars

A production in a type 1 grammar takes the following form:

$$\underbrace{L_1 \cdots L_l}\ A\ \underbrace{R_1 \cdots R_r} \longrightarrow \underbrace{L_1 \cdots L_l}\ \overbrace{B_1 \cdots B_n}\ \underbrace{R_1 \cdots R_r}$$

where A is a single non-terminal symbol, and the $L_1 \cdots L_l$, $R_1 \cdots R_r$ and $B_1 \cdots B_n$ are sequences of terminal and non-terminal symbols. The sequence $B_1 \cdots B_n$ may not be empty. These grammars are called *context sensitive* since A can only be replaced by $B_1 \cdots B_n$ if it occurs in a suitable context (the L_i are the left context and the R_i the right context).

3.4 Type 3 grammars

This is the most restrictive of the phrase structured grammars. In it all productions are limited to being one of the following two forms:

$$\begin{array}{lcl} A & \longrightarrow & a \\ A & \longrightarrow & a\ B \end{array}$$

That is, the right hand side must consist of a single terminal symbol possibly followed by a single non-terminal. It is sometimes possible to convert a type 2 grammar into an equivalent type 3 grammar. Try this for the grammar for floating point constants given earlier.

Type 3 grammars can clearly be parsed using a finite state recogniser, and for this reason they are often called *regular grammars*. [To get precise correspondence to regular languages it is necessary also to allow the empty production `S` $\longrightarrow \epsilon$ otherwise the regular language consisting of the empty string (accepted by an automaton whose initial state is accepting, but any non-empty input sequence causes it to move to a non-accepting state) cannot be represented as a type 3 grammar.]

Finally, note that clearly every Type 3 grammar is a Type 2 grammar and every Type 2 grammar is a Type 1 grammar etc. Moreover these inclusions are strict in that there are languages which can be generated by (e.g.) a Type 2 grammar and which cannot be generated by any Type 3 grammar. However, just because a particular language can be described by (say) a Type 2 grammar does not automatically mean that there is no Type 3 grammar which describes the language. An example would be the grammar G given by

```
S  —→  a
S  —→  S a
```

which is of Type 2 (and not Type 3) but the grammar G' given by

```
S  —→  a
S  —→  a S
```

clearly generates the same set of strings (is *equivalent* to G) and is Type 3.

4 Syntax analysis

The type 2 (or context free) grammar is the most useful for the description of programming languages since it is powerful enough to describe the constructions one typically needs and yet is sufficiently simple to be analysed by a small and generally efficient algorithm. Some compiler writing systems use BNF (often with slight extensions) as the notation in which the syntax of the language is defined. The parser is then automatically constructed from this description.

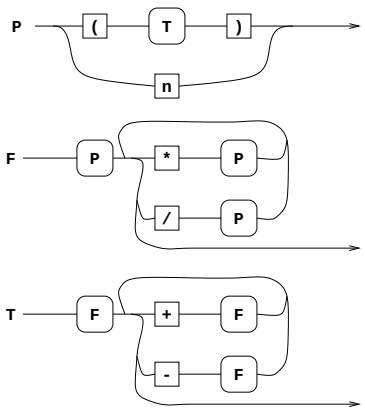
We will now look at three main parsing techniques, namely: *recursive descent*, *precedence* and *SLR(1)*.

4.1 Recursive descent

In this method the syntax is converted into transition diagrams for some or all of the syntactic categories of the grammar and these are then implemented by means of recursive functions. Consider, for example, the following syntax:

```
P  —→  ( T )  |  n
F  —→  F * P  |  F / P  |  P
T  —→  T + F  |  T - F  |  F
```

where the terminal symbol `n` represents name or number token from the lexer. The corresponding transition diagrams are:



Notice that the original syntax has been modified to avoid left recursion[3] to avoid the possibility of a recursive loop in the parser. The recursive descent parsing functions are outlined below (implemented in C):

```
void RdP()
{ switch (token)
  { case '(': lex(); RdT();
              if (token != ')') error("expected ')');
              lex(); return;
    case 'n': lex(); return;
    default:  error("unexpected token");
  }
}

void RdF()
{ RdP();
  for (;;) switch (token)
  { case '*': lex(); RdP(); continue;
    case '/': lex(); RdP(); continue;
    default:  return;
  }
}

void RdT()
{ RdF();
  for (;;) switch (token)
  { case '+': lex(); RdF(); continue;
    case '-': lex(); RdF(); continue;
    default:  return;
  }
}
```

[3] By replacing the production `F ⟶F * P | F / P | P` with `F ⟶P * F | P / F | P` which has no effect on the strings accepted, although it does affect their parse tree—see later.

4.2 Data structures for parse trees

It is usually best to use a data structure for a parse tree which corresponds closely to the *abstract syntax* for the language in question rather than the *concrete syntax*. The abstract syntax for the above language is

```
E  ⟶  E + E  |  E - E  |  E * E  |  E / E  |  ( E ) | n
```

This is clearly ambiguous seen as a grammar on strings, but it specifies parse trees precisely and corresponds directly to ML's

```
datatype E = Add  of E * E  |  Sub of E * E  |
             Mult of E * E  |  Div of E * E  |
             Paren of E  |  Num of int;
```

Indeed one can go further and ignore the `( E )` construct in the common case parentheses often have no semantic import beyond specifying grouping. In C the construct tends to look like:

```
struct E {
  enum { E_Add, E_Sub, E_Mult, E_Div, E_Paren, E_Numb } flavour;
  union { struct { struct E *left, *right; } diad;
            // selected by E_Add, E_Sub, E_Mult, E_Div.
          struct { struct E *child; } monad;
            // selected by E_Paren.
          int num;
            // selected by E_Numb.
         } u;
};
```

It is not generally helpful to reliability and maintainability to make a single datatype which can represent all sub-structures of a parse tree. For parsing C, for example, one might well expect to have separate abstract parse trees for `Expr`, `Cmd` and `Decl`.

It is easy to augment a recursive descent parser so that it builds a parse tree while doing syntax analysis. The ML datatype definition defines constructor functions, e.g. `Mult` which maps two expression trees into one tree which represents multiplying their operands. In C one needs to work a little by defining such functions by hand:

```
struct E *mkE_Mult(E *a, E *b)
{   struct E *result = malloc(sizeof (struct E));
    result->flavour = E_Mult;
    result->u.diad.left = a;
    result->u.diad.right = b;
    return result;
}
```

A recursive descent parser which builds a parse tree for the parsed expression is given in Figure 1.

When there are many such operators like `+`, `-`, `*`, `/` with similar syntax it can often simplify the code to associate a binding power with each operator and to define a single routine `RdE(int n)` which will read an expression which binds at least as tightly as `n`. In this case `RdT()` might correspond to `RdE(0)`, `RdT()` to `RdE(1)` and `RdP()` to `RdE(2)`.

This idea is can be pushed further to produce a table-driven parser to which topic we now turn.

```
    struct E *RdP()
    {   struct E *a;
        switch (token)
        {   case '(': lex(); a = RdT();
                      if (token != ')') error("expected ')');
                      lex(); return a;
            case 'n': a = mkE_Numb(lex_aux_int); lex(); return a;
/* do names by
**          case 'i': a = mkE_Name(lex_aux_string): lex(); return a;
*/
            default:  error("unexpected token");
        }
    }

/* This example code includes a right associative '^' operator too...   */
/* '^' binds more tightly than '*' or '/'.  For this example, The rule  */
/*          F ::= P | F * P | F / P                                     */
/* has been changed into the two rules                                  */
/*          F ::= G | F * G | F / G          G ::= P | P ^ G           */
    struct E *RdG()
    {   struct E *a = RdP();
        switch (token)
        {   case '^': lex(); a = mkE_Mult(a, RdG()); return a;
            default:  return a;
        }
    }

    struct E *RdF()
    {   struct E *a = RdG();
        for (;;) switch (token)
        {   case '*': lex(); a = mkE_Mult(a, RdG()); continue;
            case '/': lex(); a = mkE_Div(a, RdG()); continue;
            default:  return a;
        }
    }

    struct E *RdT()
    {   struct E *a = RdF();
        for (;;) switch (token)
        {   case '+': lex(); a = mkE_Add(a, RdF()); continue;
            case '-': lex(); a = mkE_Sub(a, RdF()); continue;
            default:  return a;
        }
    }
```

Figure 1: Recursive descent parser yielding a parse tree

4.3 Simple precedence

For simple arithmetic grammars a parser based on the precedence of the operators is possible. Consider the token stream:

$\boxed{\texttt{bof}}$ `x * y + a / t - c ** d` $\boxed{\texttt{eof}}$

(We use the special terminals $\boxed{\texttt{bof}}$ and $\boxed{\texttt{eof}}$ to represent respectively beginning and end of a file (stream).) Let us define two relations $\lessdot$ and $\gtrdot$ which hold as follows:

	+	-	*	/	**	eof
bof	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	
+	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
-	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
/	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
**	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$

then we can parse the above expression by means of the following steps:

				We start with:		bof x * y + a / t - c ** d eof
bof	$\lessdot$	*	$\gtrdot$	+	=>	bof (x*y) + a / t - c ** d eof
+	$\lessdot$	/	$\gtrdot$	-	=>	bof (x*y) + (a/t) - c ** d eof
bof	$\lessdot$	+	$\gtrdot$	-	=>	bof ((x*y)+(a/t)) - c ** d eof
-	$\lessdot$	**	$\gtrdot$	eof	=>	bof ((x*y)+(a/t)) - (c**d) eof
bof	$\lessdot$	-	$\gtrdot$	eof	=>	bof (((x*y)+(a/t))-(c**d)) eof

It is worth noting that this method allows both the precedence and associativity of operators to be specified. For example `a-b-c` parses as `(a-b)-c`, but `a**b**c` as `a**(b**c)`.

4.4 General precedence

For some grammars it is possible to define relations $\doteq$, $\lessdot$ and $\gtrdot$ between alphabet characters (terminal or non-terminal) of the grammar in such a way that, if

`... U A B C D V ...`

is a sentential form, and

$\texttt{U} \lessdot \texttt{A} \doteq \texttt{B} \doteq \texttt{C} \doteq \texttt{D} \gtrdot \texttt{V}$

then there must be a production $\texttt{X} \longrightarrow \texttt{A B C D}$ and so

`... U X V ...`

is a simpler sentential form. A matrix defining these relations then forms the basis of a very simple parser.

The definitions of $\doteq$, $\lessdot$ and $\gtrdot$ are as follows:

1) $\texttt{A} \doteq \texttt{B}$ <=> $\texttt{P} \longrightarrow \texttt{... A B ...}$ is a production

2) $\texttt{A} \lessdot \texttt{B}$ <=> $\texttt{P} \longrightarrow \texttt{... A U ...}$ is a production
and $\texttt{U} \xrightarrow{1+} \texttt{B ...}$ (using one or more productions)

3a) $\texttt{A} \gtrdot \texttt{B}$ <=> $\texttt{P} \longrightarrow \texttt{... U B ...}$ is a production
and $\texttt{U} \xrightarrow{1+} \texttt{... A}$

3b) $\texttt{A} \gtrdot \texttt{B}$ <=> $\texttt{P} \longrightarrow \texttt{... U V ...}$ is a production
and $\texttt{U} \xrightarrow{1+} \texttt{... A}$
and $\texttt{V} \xrightarrow{1+} \texttt{B ...}$

The grammar is a *precedence grammar* if for all pairs of alphabet characters at most one of the relations holds. The following grammar is not a precedence grammar.

```
S  ⟶  [bof] E [eof]
E  ⟶  T  |  E + T
T  ⟶  P  |  T * P
P  ⟶  ( E )  |  I
```

since, for instance [bof]≐E and [bof]<·E. It can, however, be modified into the following equivalent grammar which is a precedence grammar.

```
S  ⟶  [bof] E' [eof]
E' ⟶  E
E  ⟶  T'  |  E + T'
T' ⟶  T
T  ⟶  P  |  T * P
P  ⟶  ( E' )  |  I
```

4.4.1 Construction of the precedence matrix

Before we construct the matrix it is convenient to form, for each non-terminal U in the grammar, two sets `Left`(U) and `Right`(U) of symbols that can start and end strings derived from U. If $U \xrightarrow{1+} B_1 \cdots B_n$ then B_1 is in `Left`(U) and B_n is in `Right`(U).
`Left`(U) can be derived for all non-terminals in the grammar by the following algorithm:

1. Initialise all sets `Left`(U) to empty.
2. For each production $U \longrightarrow B_1 \cdots B_n$ enter B_1 into `Left`(U).
3. For each production $U \longrightarrow B_1 \cdots B_n$ where B_1 is also a non-terminal enter all the elements of `Left`(B_1) into `Left`(U)
4. Repeat 3. until no further change.

`Right`(U) can be derived similarly. For the example grammar the sets are as follows:

U	`Left`(U)	`Right`(U)
E'	E T' T P (I	E T' T P) I
E	E T' T P (I	T' T P) I
T'	T P (I	T P) I
T	T P (I	P) I
P	(I	) I

The following algorithm constructs the precedence matrix. For each pair A B occurring consecutively in a production (i.e. $P \longrightarrow \cdots AB \cdots$ is a production) do the following:

1. Enter $A \doteq B$ into the matrix
2. Enter $A <\cdot X$ into the matrix for all X in `Left`(B)

3a. Enter $X \cdot> B$ into the matrix for all X in `Right`(A)

3b. Enter $X \cdot> Y$ into the matrix for all X in `Right`(A) and all Y in `Left`(B)

For the example grammar, there are 8 pairs to consider:

[bof] E'	E' [eof]	E' +	+ T'	T *	* P	(E'	E')

and the resulting matrix is:

	E'	E	T'	T	P	(	I	*	+	)	eof
E'	-	-	-	-	-	-	-	-	-	$\doteq$	$\doteq$
E	-	-	-	-	-	-	-	-	$\doteq$	$\cdot>$	$\cdot>$
T'	-	-	-	-	-	-	-	-	$\cdot>$	$\cdot>$	$\cdot>$
T	-	-	-	-	-	-	-	$\doteq$	$\cdot>$	$\cdot>$	$\cdot>$
P	-	-	-	-	-	-	-	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
)	-	-	-	-	-	-	-	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
I	-	-	-	-	-	-	-	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$
*	-	-	-	-	$\doteq$	$\cdot<$	$\cdot<$	-	-	-	-
+	-	-	$\doteq$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	-	-	-	-
(	$\doteq$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	-	-	-	-
bof	$\doteq$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	$\cdot<$	-	-	-	-

For this grammar it is possible to find two functions `f` and `g` with the property that:

$$\begin{aligned} A \doteq B &\iff f(A) = g(B) \\ A \lessdot B &\iff f(A) < g(B) \\ A \gtrdot B &\iff f(A) > g(B) \end{aligned}$$

Define such functions for this grammar and show that it is not possible in general. Why are such functions useful?

The following program will perform the parse using the precedence matrix. `P[`k`]` is the kth symbol of source text, and `S[`i`]` is the ith element of a stack (storing part-parsed input)

```
S[0] = P[0]; i = 0, k = 1;
while (P[k] != eof)
{ S[++i] = P[k++];
  while (S[i] ⋗ P[k])
  { int j = i;
    while (S[j-1] ≐ S[j]) j--;
    S[j] = Leftpart(S[j] ,..., S[i]);
    i = j;
  }
}
```

The function `Leftpart` finds the subject of the production whose right part is given as its argument(s). Syntactic errors are detected by either encountering a blank entry in the matrix or by a failure in the function `Leftpart`.

NB. Note that the parsing program above is independent of the grammar to be used—all the grammatical details are stored in the tables. One can say that here the grammar is coded as data whereas in the recursive descent parser it was coded as program.

4.5 SLR parsing

Various parsing algorithms based on the so called LR(k) approach have become become popular. These are specifically LR(0), SLR(1), LALR(1) and LR(1). These four methods can parse a source text using a very simple program controlled by a table derived from the grammar. The methods only differ in the size and content of the controlling table.

To exemplify this style of syntax analysis, consider the following grammar (here `E`, `T`, `P` abbreviate 'expression', 'term' and 'primary'—an alternative notation would use names like `<expr>`, `<term>` and `<primary>` instead):

```
#0    S  ⟶  E  eof
#1    E  ⟶  E  +  T
#2    E  ⟶  T
```

```
#3     T  ⟶  P  **  T
#4     T  ⟶  P
#5     P  ⟶  i
#6     P  ⟶  (  E  )
```

The form of production #0 is important. It defines the sentence symbol S and its RHS consists of a single non-terminal followed by the special terminal symbol eof which must not occur anywhere else in the grammar. (When you revisit this issue you will note that this ensures the value parsed is an E and what would be a reduce transition using rule #0 is used for the acc accept marker.)

We first construct what is called the *characteristic finite state machine* or CFSM for the grammar. Each state in the CFSM corresponds to a different set of *items* where an *item* consists of a production together with a position marker (represented by .) marking some position on the right hand side. There are, for instance, four possible items involving production #1, as follows:

```
E  ⟶ .E  +  T
E  ⟶  E .+  T
E  ⟶  E  + .T
E  ⟶  E  +  T .
```

If the marker in an item is at the beginning of the right hand side then the item is called an *initial* item. If it is at the right hand end the the item is called a *completed* item. In forming item sets a *closure* operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, E say, then initial items must be included in the set for all productions with E on the left hand side.

The first item set is formed by taking the initial item for the production defining the sentence symbol (S ⟶.E eof) and then performing the closure operation, giving the item set:

```
1: {  S  ⟶ .E  eof
      E  ⟶ .E  +  T
      E  ⟶ .T
      T  ⟶ .P  **  T
      T  ⟶ .P
      P  ⟶ .i
      P  ⟶ .(  E  )
   }
```

States have *successor* states formed by advancing the marker over the symbol it precedes. For state 1 there are successor states reached by advancing the marker over the symbols E, T, P, i or (. Consider, first, the E successor (state 2), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non terminal). State 2 is thus:

```
2: {  S  ⟶  E .eof
      E  ⟶  E .+  T
   }
```

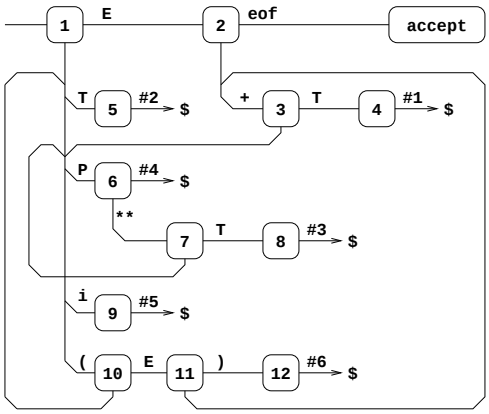
The other successor states are defined similarly, except that the successor of eof is always the special state accept. If a new item set is identical to an already existing set then the existing set is used. The successor of a completed item is a special state represented by $ and the transition is labeled by the production number (#i) of the production involved. The process of forming the complete collection of item sets continues until all successors of all item sets have been formed. This necessarily terminates because there are only a finite number of different item sets.

For the example grammar the complete collection of item sets given in Figure 2. Note that for completed items the successor state is reached via the application of a production (whose number is given in the diagram).

```
1: {  S  -> .E  eof          \
      E  -> .E  +  T         /    E  => 2
      E  -> .T                    T  => 5
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
2: {  S  ->  E .eof               eof => accept
      E  ->  E .+  T              +  => 3
   }
3: {  E  ->  E  + .T              T  => 4
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
4: {  E  ->  E  +  T .            #1 => $
   }
5: {  E  ->  T .                  #2 => $
   }
6: {  T  ->  P .**  T             ** => 7
      T  ->  P .                  #4 => $
   }
7: {  T  ->  P  ** .T             T  => 8
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
8: {  P  ->  P  **  T .           #3 => $
   }
9: {  P  ->  i .                  #5 => $
   }
10:{  P  ->  ( .E  )          \
      E  -> .E  +  T         /    E  => 11
      E  -> .T                    T  => 5
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
11:{  P  ->  (  E .)              )  => 12
      E  ->  E .+  T              +  => 3
   }
12:{  P  ->  (  E  ) .            #6 => $
   }
```

Figure 2: CFSM item sets

The CFSM can be represented diagrammatically as follows:



Before we can construct an SLR(1) parser we must define and compute the sets `FOLLOW`(A) for all non-terminal symbols A. `FOLLOW`(A) is defined to be the set of all symbols (terminal and non-terminal) that can immediately follow the non-terminal symbol `A` in a sentential form. They can be formed iteratively by repeated application of the following rules.

1. If there is a production of the form $X \longrightarrow \ldots YZ \ldots$ put Z and all symbols that can start Z into `FOLLOW`(Y).

2. If there is a production of the form $X \longrightarrow \ldots Y$ put all symbols in `FOLLOW`(X) into `FOLLOW`(Y).

We are assuming here that no production in the grammar has an empty right hand side. For our example grammar, the `FOLLOW` sets are as follows:

A	`FOLLOW`(A)
E	eof +)
T	eof +)
P	eof +) **

From the CFSM we can construct the two matrices `action` and `goto`:

1. If there is a transition from state i to state j under the terminal symbol k, then set `action`$[i, k]$ to `S`j.

2. If there is a transition under a non-terminal symbol A, say, from state i to state j, set `goto`$[i, A]$ to `S`j.

3. If state i contains a transition under eof set `action`$[i,$ eof$]$ to `acc`.

4. If there is a reduce transition `#p` from state i, set `action`$[i, k]$ to `#p` for all terminals k belonging to `FOLLOW`(A) where A is the subject of production `#p`.

If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (`-`).

	action						goto		
state	eof	(	i	)	+	**	P	T	E
S1	-	S10	S9	-	-	-	S6	S5	S2
S2	acc	-	-	-	S3	-	-	-	-
S3	-	S10	S9	-	-	-	S6	S4	-
S4	#1	-	-	#1	#1	-	-	-	-
S5	#2	-	-	#2	#2	-	-	-	-
S6	#4	-	-	#4	#4	S7	-	-	-
S7	-	S10	S9	-	-	-	S6	S8	-
S8	#3	-	-	#3	#3	-	-	-	-
S9	#5	-	-	#5	#5	#5	-	-	-
S10	-	S10	S9	-	-	-	S6	S5	S11
S11	-	-	-	S12	S3	-	-	-	-
S12	#6	-	-	#6	#6	#6	-	-	-

The parsing algorithm used for all LR methods uses a stack that contains alternately state numbers and symbols from the grammar, and a list of input terminal symbols terminated by eof. A typical situation is represented below:

a A b B c C d D e E f | u v w x y z eof

Here a ... f are state numbers, A ... E are grammar symbols (either terminal or non-terminal) and u ... z are the terminal symbols of the text still to be parsed. If the original text was syntactically correct, then

A B C D E u v w x y z

will be a sentential form.

The parsing algorithm starts in state S1 with the whole program, i.e. configuration

1 | ⟨the whole program upto eof⟩

and then repeatedly applies the following rules until either a syntactic error is found or the parse is complete.

1. If action[f, u] = Si, then transform

   a A b B c C d D e E f | u v w x y z eof

   to

   a A b B c C d D e E f u i | v w x y z eof

   This is called a *shift* transition.

2. If action[f, u] = #p, and production #p is of length 3, say, then it will be of the form P ⟶ C D E where C D E exactly matches the top three symbols on the stack, and P is some non-terminal, then assuming goto[c, P] = g

   a A b B c C d D e E f | u v w x y z eof

   will transform to

   a A b B c P g | u v w x y z eof

   Notice that the symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the goto table. This is called a *reduce* transition.

3. If `action[f, u] = acc` then the situation will be as follows:

   `a Q f | eof`

   and the parse will be complete. (Here `Q` will necessarily be the single non-terminal in the start symbol production (`#0`) and `u` will be the symbol `eof`.)

4. If `action[f, u] = -` then the text being parsed is syntactically incorrect.

Note again that there is a single program for all grammars; the grammar is coded in the `action` and `goto` matrices.

As an example, the following steps are used in the parsing of `i+i`:

Stack	text	production to use
1	i + i eof	
1 i 9	+ i eof	P ⟶ i
1 P 6	+ i eof	T ⟶ P
1 T 5	+ i eof	E ⟶ T
1 E 2	+ i eof	
1 E 2 + 3	i eof	
1 E 2 + 3 i 9	eof	P ⟶ i
1 E 2 + 3 P 6	eof	T ⟶ P
1 E 2 + 3 T 4	eof	E ⟶ E + T
1 E 2	eof	acc (E is result)

In practice a tree will be produced and stored attached to terminals and non-terminals on the stack. Thus the final `E` will in reality be a pair of values: the non-terminal `E` along with a tree representing `i+i`.

4.5.1 Errors

A syntactic error is detected by encountering a blank entry in the `action` or `goto` tables. If this happens the parser can recover by systematically inserting, deleting or replacing symbols near the current point in the source text, and choosing the modification that yields the most satisfactory recovery. A suitable error message can then be generated.

4.5.2 Table compaction

In a typical language we can expect there to be over 200 symbols in the grammar and perhaps rather more states in the CFSM. The `action` and `goto` tables are thus likely to require over 40000 entries between them. There are good ways of compacting these by about a factor of ten.

5 Automated tools to write compilers

These tools are often known as compiler compilers (i.e. they compile a textual specification of part of your compiler into regular, if sordid, source code instead of you having to write it yourself).

Lex and Yacc are programs that run on Unix and provide a convenient system for constructing lexical and syntax analysers. JLex and CUP provide similar facilities in a Java environment. There are also similar tools for ML.

5.1 Lex

Lex takes as input a file (e.g. `calc.l`) specifying the syntax of the lexical tokens to be recognised and it outputs a C program (normally `lex.yy.c`) to perform the recognition. The syntax of each token is specified by means of a regular expression and the corresponding action when that

```
%%
[ \t]  /* ignore blanks and tabs  */ ;

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"mod" return MOD;
"div" return DIV;
"sqr" return SQR;
\n|.  return yytext[0]; /* return everything else */
```

Figure 3: `calc.l`

token is found is supplied as a fragment of C program that is incorporated into the resulting lexical analyser. Consider the lex program `calc.l` in Figure 3. The regular expressions obey the usual unix conventions allowing, for instance, `[0-9]` to match any digit, the character `+` to denote repetition of one or more times, and dot (`.`) to match any character other than newline. Next to each regular expression is the fragment of C program for the specified token. This may use some predefined variables and constants such as `yylval, yytext` and `NUMBER`. `yytext` is a character vector that holds the characters of the current token (its length is held in `yyleng`). The fragment of code is placed in the body of an external function called `lex`, and thus a `return` statement will cause a return from this function with a specified value. Compound tokens such as `NUMBER` return auxiliary information in suitably declared variables. For example, the converted value of a `NUMBER` is passed in the variable `lexlval`. If a code fragment does not explicitly return from `lex` then after processing the current token the lexical analyser will start searching for the next token.

In more detail, a Lex program consists of three parts separated by `%%`s.

```
declarations
%%
translation rules
%%
auxiliary C code
```

The declarations allows a fragment of C program to be placed near the start of the resulting lexical analyser. This is a convenient place to declare constants and variables used be the lexical analyser. One may also make regular expression definitions in this section, for instance:

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

These named regular expressions may be used by enclosing them in braces (`{ or }`) in later definitions or in the translations rules.

The translation rules are as above and the auxiliary C code is just treated as a text to be copied into the resulting lexical analyser.

5.2 Yacc

Yacc (yet another compiler compiler) is like Lex in that it takes an input file (e.g. `calc.y`) specifying the syntax and translation rule of a language and it output a C program (usually `y.tab.c`) to perform the syntax analysis.

Like Lex, a Yacc program has three parts separated by `%%`s.

```
declarations
%%
```

```
translation rules
%%
auxiliary C code
```

Within the declaration one can specify fragments of C code (enclosed within special brackets `%{` and `%}`) that will be incorporated near the beginning of the resulting syntax analyser. One may also declare token names and the precedence and associativity of operators in the declaration section by means of statements such as:

```
%token NUMBER
%left '*' DIV MOD
```

The translation rules consist of BNF-like productions that include fragments of C code for execution when the production is invoked during syntax analysis. This C code is enclosed in braces (`{ and }`) and may contain special symbols such as `$$`, `$1` and `$2` that provide a convenient means of accessing the result of translating the terms on the right hand side of the corresponding production.

The auxiliary C code section of a Yacc program is just treated as text to be included at the end of the resulting syntax analyser. It could for instance be used to define the main program.

An example of a Yacc program (that makes use of the result of Lex applied to `calc.l`) is `calc.y` listed in Figure 4.

Yacc parses using the LALR(1) technique. It has the interesting and convenient feature that the grammar is allowed to be ambiguous resulting in numerous shift-reduce and reduce-reduce conflicts that are resolved by means of the precedence and associativity declarations provided by the user. This allows the grammar to be given using fewer syntactic categories with the result that it is in general more readable.

The above example uses Lex and Yacc to construct a simple interactive calculator; the translation of each expression construct is just the integer result of evaluating the expression. Note that in one sense it is not typical in that it does not construct a parse tree—instead the value of the input expression is evaluated as the expression is parsed. The first two productions for '`expr`' would more typically look like:

```
expr: '(' expr ')'    { $$ = $2; }
    | expr '+' expr   { $$ = mkbinop('+', $1, $3); }
```

where `mkbinop()` is a C function which takes two parse trees for operands and makes a new one representing the addition of those operands.

6 Translation to intermediate code

The translation phase of a compiler normally converts the abstract syntax tree representation of a program into intermediate object code which is usually either a linear sequence of statements or an internal representation of a flowchart. We will assume that the translation phase deals with (1) the scope and allocation of variables, (2) determining the type of all expressions, (3) the selection of overloaded operators, and (4) generating of the intermediate code.

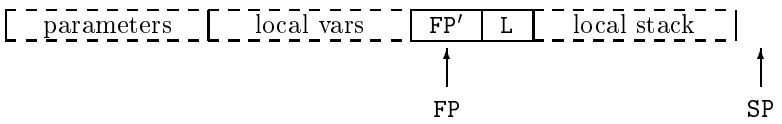
Before we can give algorithms to translate a parse-tree into linear intermediate code form, we need to be a little more precise about the particular representation of the parse tree and also the intermediate code used. We present a JVM-style[4] intermediate code, which consists of a linear sequence of simple (virtual machine) instructions which act on a *run-time* stack. Note that the explanation given here corresponds to a subset of the JVM (e.g. we will not explore exceptions or non-static method invocation) and will prefer pedagogic simplicity over precision. Nevertheless, you should be able to read disassembled (using '`javap -c`') JVM `.class` files and I recommend you do this to aid your understanding.

[4] The Microsoft language C# has a very close resemblance to Java and their `.net` virtual machine code a similar relationship to JVM code. Their divergence owes more to commercial reasons that technological ones.

```
%{
#include <stdio.h>
%}

%token NUMBER

%left '+' '-'
%left '*' DIV MOD
  /* gives higher precedence to '*', DIV and MOD  */
%left SQR

%%
comm: comm '\n'
    | /* empty  */
    | comm expr '\n'  { printf("%d\n", $2); }
    | comm error '\n' { yyerrok; printf("Try again\n"); }
    ;

expr: '(' expr ')'    { $$ = $2; }
    | expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | expr DIV expr   { $$ = $1 / $3; }
    | expr MOD expr   { $$ = $1 % $3; }
    | SQR expr        { $$ = $2 * $2; }
    | NUMBER
    ;

%%

#include "lex.yy.c"

yyerror(s)
char *s;
{ printf("%s\n", s);
}

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

Figure 4: `calc.y`

In these notes the stack is *upwards growing*, so that subsequent parameters and local variables occupy increasing memory locations (this is not necessary for the JVM but makes the pictures easier to understand). The version of the JVM we present has a stack-pointer `SP` register which points to the first free location on the stack. In addition, it has a so-called frame-pointer `FP` register which points to a constant place during the activation of a procedures. Although strictly not necessary, this simplifies explanations and aids debuggers, since local variables will remain at the same constant offset from `FP` but their offset from `SP` will vary during pushes and pops to the stack. (N.B. on both the Pentium and the ARM, the convention instead is to have the stack *downward growing*, and `SP` to hold the lowest currently used location on the stack.) Evaluation of a procedure uses a *stack frame* which will be addressed from `FP`. A stack frame has the following form:



Here `FP` points to (or rather into) the current stack frame. `FP` points to the so-called *linkage information* which allows a function call to return. This consists of the previous value of `FP`, here called `FP′` and a pointer `L` (the `return address`) to the machine instruction to be next evaluated after the function's return. To the left (at lower addresses) of the linkage informations are stored the functions parameters and (user-declared) local variables; on the right is the local stack (used for evaluating temporaries and for storing arguments), `SP` is a pointer to first free cell of this (which coincides with the first free cell of the whole stack). To clarify explanations of function call, we will use the word *parameters* to refer to the identifiers holding arguments to the current procedure and restrict the word *arguments* to refer to expressions are being, or have been, evaluated on the stack in order to be passed to a further function. At the end of each Java statement we can expect the local stack to be empty and hence `SP` to point exactly two words further along the stack from `FP`. In general these notes will assume that the JVM stack is an array of (32-bit) words, and so addresses of stack items will differ by one.

One more point is worthy of note. A Java routine has a fixed number of parameters (say n_p) and local variables (say n_v) and similarly the maximum depth of its local stack can be determined in advance. The latter means it is simple to check, at procedure invocation, that there is enough space beyond `SP` for a new stack frame, thus avoiding over-writing issues. The JVM does not have distinct instructions for addressing parameters and local variables, they are just accessed with the instruction `iload_`i, using offsets $0..(n_p + n_v - 1)$.

Turning to the JVM instructions, variables and parameters are accessed by instructions `iload_`i and `istore_`i, with operand $0..(n_p + n_v - 1)$. Thus "read the first parameter to a function" would be written `iload_0`.

More formally, and showing the use of the local stack, these JVM instructions have the following effects:

```
iload_i          SP[0] = FP[i − (n_p + n_v)]; SP++
istore_i         SP--; FP[i − (n_p + n_v)] = SP[0];
```

Note that we have just written, following C, the name of a pointer as an array name, in particular `SP[0]` means "the cell pointed to by `SP`". Some of the other JVM instructions are as follows:

```
iadd             SP[-2] = SP[-2] + SP[-1]; SP--;
isub             SP[-2] = SP[-2] - SP[-1]; SP--;
ineg             SP[-1] = - SP[-1];
if_icmpgt L      if (SP[-2] > SP[-1]) PC = L;    SP-=2;
if_icmpeq L      if (SP[-2] == SP[-1]) PC = L;    SP-=2;
goto L           PC = L;
```

Function calls and return are more tricky because they have to create or deallocate a stack frame (the general principle is to avoid trampling on one's own feet, especially in `ireturn` if $n_p + n_v = 0$).

`invokestatic` L	`SP = SP +` $n_v^{\texttt{called-fn}} + 2$`; SP[-2] = FP; SP[-1] = PC;`
	`FP = SP - 2; PC = L;`
`ireturn`	`tempSP = FP -` $(n_p + n_v)$`; FP = FP[-2]; PC = FP[-1];`
	`tempSP[0] = SP[-1]; SP = tempSP+1;`

The use of the JVM instructions can be illustrated by considering the following Java program fragment:

```
class fntest {
public static void main(String args[]) {
    System.out.println("Hello World!" + f(f(1,2),f(3,4)));
    }
static int f(int a, int b) { int y = a+b; return y*a; }
}
```

The JVM function code generated for the function `f` might be:

```
iload_0       ; load a
iload_1       ; load b
iadd
istore_2      ; store result to y
iload_2       ; re-load y
iload_0       ; re-load a
imul
ireturn       ; return from fn with top-of-stack value as result
```

and the series of calls in the `println` in `main` as:

```
iconst_1
iconst_2
invokestatic f
iconst_3
iconst_4
invokestatic f
invokestatic f
```

Note how two-argument function calls just behave like binary operators in that they remove two items from the stack and replace them with one; the instructions `invokestatic` and `ireturn` both play their part in the conspiracy to make this happen. You really must work through the above code step-by-step to understand the function call/return protocol.

Instuctions, such the first few instructions of `f` above, will be generated by translation phase of the compiler using a series of calls such as:

```
Gen2(OP_iload, 0);
Gen2(OP_iload, 1);
Gen1(OP_iadd);
Gen2(OP_istore, 2);
```

where `OP_iload` etc. will be represented by different values in an enumeration, for example (using the barbarous Java syntax for this)

```
static final const OP_iload = 1;
static final const OP_istore = 2;
static final const OP_iadd = 3;
static final const OP_isub = 4;
static final const OP_itimes = 5;
```

The magic numbers can correspond directly to the bit patterns in a `.class` file, or can be decoded in the `Gen`i routines into readable strings. Alternatively successive instructions can be stored in memory ready to be translated into native machine instructions in the CG phase.

6.1 Example tree form used in this section

In this section we will use a simple example language reflecting a subset of Java without types or classes. It has the following abstract syntax tree structure (expressed in ML for conciseness):

```
type N = string;   (* shorthand for 'name' *)
datatype E = Var of N | Num of int | Apply of N * (E list) |
             Neg of E  |  Pos of E  |  Not of E  |
             Add  of E * E  |  Sub of E * E  |
             Mult of E * E  |  Div of E * E  |
             Eq of E * E  |  Ne of E * E  |
             Lt of E * E  |  Gt of E * E  |
             Le of E * E  |  Ge of E * E  |
             And of E * E  |  Or of E * E  |  (* for && and || *)
             Cond of E * E * E |

     and C = Seq of C * C  |  Assign of N * E
             If3 of E * C * C  |  While of E * C  |
             Block of D list * C list | Return of E

     and D = Vardef of N * E  |   Fndef of N * (N list) * C;
type P = D list;   (* shorthand for 'program' *)
```

A program in this language essentially consists of an interleaved sequence of initialised variable definitions `let` x `=` e and function definitions `let` f`(`$x_1, \ldots, x_k$`)` c.

6.2 Dealing with names and scoping

To generate the appropriate instruction for a variable or function reference (e.g. `iload_7` instead of `y`) we require the compiler to maintain a table (often called a *symbol table* although beware that this sometimes is used for for other things). This table keeps a record of which variables are currently in scope and how the compiler may access them. For example, in Java

```
class A {
  public static int g;
  public int n,m;
  public int f(int x) { int y = x+1; return foo(g,n,m,x,y); }
}
```

the variables `x` and `y` will be accessed via the `iload` and `istore` as above, but there will be another pair of instructions to access a variable like `g` which is logically a global variable which can live in a fixed position in memory and be addressed using absolute addressing. Accessing per-instance variables, such as `n` above, is really beyond the scope of this part of the course which deals with translation, but how a translation might work will be covered in the second part of the course.

Essentially, the routine `trdecl` will save the current state of the symbol table and add the new declared names to the table. The routine `trname` consults the symbol table to determine the access path for a given name. Finally, the compiler will arrange, when it has concluded treatment a scope which has definitions, that the symbol table is restored to that which was extant at the start of the scope (and saved by `trdecl`).

As an example for the above the table might contain

```
"g"         static variable
"n"         class variable 0
"m"         class variable 1
"f"         method
"x"         local variable 0
"y"         local variable 1
```

when compiling the call to `foo`, but just the first four items when merely in the scope of `A`. In more detail, the symbol table will be extended by the entry (`x`, *loc*, 0) when `f`'s parameters (`x`) are scanned, and then by (`y`, *loc*, 1) when the local definition of `y` is encountered. The issue of how *environments* (the abstract concept corresponding to our symbol table) behave will be given in the second part of the course.

6.3 Translation of expressions

Some of the functions used during translation are as follows:

`trexp(x)`	translate an expression
`trexplist(x)`	translate an expression list
`trname(op,x)`	translate a name, `op` is one of `OP_iload`, `OP_istore`, `OP_invokestatic`
`jumpcond(x,b,n)`	translate a conditional jump
`trcmd(x)`	translate a command
`trdecl(x)`	translate a declaration

The argument to `trexp` is the tree for the expression being translated. An outline of its definition is as follows:[5]

```
fun trexp(Num(k))      = gen2(OP_iconst, k);
  | trexp(Id(s))       = trname(OP_iload,s);
  | trexp(Add(x,y))    = (trexp(x); trexp(y); gen1(OP_iadd))
  | trexp(Sub(x,y))    = (trexp(x); trexp(y); gen1(OP_isub))
  | trexp(Mult(x,y))   = (trexp(x); trexp(y); gen1(OP_imul))
  | trexp(Div(x,y))    = (trexp(x); trexp(y); gen1(OP_idiv))
  | trexp(Neg(x))      = (trexp(x); gen1(OP_ineg))
  | trexp(Apply(f, el))   =
                 ( trexplist(el);           // translate args
                   trname(OP_Invokestatic, f)) // Compile call to f
  | trexp(Cond(b,x,y)) =
                 let val p = ++label;       // Allocate two labels
                     val q = ++label in
                   jumpcond(b,false,p);     // (see below for jumpcond)
                   trexp(x);                // code to put x on stack
                   gen2(OP_goto,q);         // jump to common point
                   gen2(OP_Lab,p);
                   trexp(y);                // code to put y on stack
                   gen2(OP_Lab,q)           // common point; result on stack
                 end;
etc...

fun trexplist[] = ()
  | trexplist(e::es) = (trexp(e); trexplist(es));
```

6.4 Translation of short-circuit boolean expressions

In Java, the operators `&&` and `||` are required not to evaluate their second operand if the result of the expression is determined by the value of their first operand. For example, consider code like

```
if (i>=0 && A[i]==42) { ... }
```

[5] We have adopted an ML-like syntax to describe this code since we can exploit pattern matching to make the code more concise than C or Java would be. For ML experts there are still things left undone, like defining the `++` and `--` operators of type `int ref -> int`.

If `i>=0` is false then we are forbidden to evaluate `A[i]` as it may give rise to an exception. We will use the function `jumpcond` to compile such expressions. Its first argument is the tree structure of the expression, the second is a truth value stating whether a jump is to be made on true or on false, and the third argument is the number of the label to jump to. We follow C and assume that a boolean is represented as an `int` value with 0 corresponding to `false` and all other values being treated as `true`. The definition of `jumpcond` is outlined below:

```
fun jumpcond(Num(c), true, n) =  if (c!=0) gen2(OP_goto, n) else ();
  | jumpcond(Num(c), false, n) =  if (c==0) gen2(OP_goto, n) else ();
  | jumpcond(Le(x,y), b, n) = (trexp(x); trexpr(y);
                               gen2((b ? OP_if_cmple:OP_if_cmpgt), n))
  | jumpcond(Ne(x,y), b, n) = (trexp(x); trexpr(y);
                               gen2((b ? OP_if_cmpne:OP_if_cmpeq), n))
  (* the cases Lt, Eq, Gt, Gt have been omitted here *)
  | jumpcond(Not(x),  b, n) =  jumpcond(x, not b, n)
  | jumpcond(And(x, y),  true, n) =
                         let val m = ++label in
                           jumpcond(x, false, m);
                           jumpcond(y,  true, n);
                           gen2(OP_Lab, m)
                         end
  | jumpcond(And(x, y), false, n) =  (jumpcond(x, false, n);
                                      jumpcond(y, false, n))
  | jumpcond( Or(x, y),  true, n) =  (jumpcond(x,  true, n);
                                      jumpcond(y,  true, n))
  | jumpcond( Or(x, y), false, n) =
                         let val m = ++label in
                           jumpcond(x,  true, m);
                           jumpcond(y, false, n);
                           gen2(OP_Lab, m);
                         end
  | jumpcond(x, b, n) =  ( trexp(x);
                           gen2(OP_iconst, 0);
                           gen2((b ? OP_if_cmpne:OP_if_cmpeq), n))
```

6.5 Type checking

So far in this section we have ignored type information (or rather, just assumed every variable and operator is of type `int`—hence the integer operators `iadd`, `ineg`, `iload` etc). In a language like Java, every variable and function name is given an explicit type when it is declared. This can be added to the symbol table along with other (location and name) attributes. The language specification then gives a way of determining the type of each sub-expression of the program. For example, the language would typically specify that $e + e'$ would have type `float` if e had type `int` and e' had type `float`.

This is implemented as follows. Internally, we have a data type representing language types (e.g. Java types), with elements like `T_float` and `T_int` (and more structured values representing things like function and class types which we will not discuss further here). A function `typeof` gives the type of an expression. It would be coded :

```
fun typeof(Num(k))      = T_int
  | typeof(Float(f))    = F_float
  | typeof(Id(s))       = lookuptype(s)  // looks in symbol table
  | typeof(Add(x,y))    = arith(typeof(x), typeof(y));
  | typeof(Sub(x,y))    = arith(typeof(x), typeof(y));
   ...
```

```
fun arith(T_int, T_int  )   = T_int
  | arith(T_int, T_float)   = T_float
  | arith(T_float, T_int)   = T_float
  | arith(T_float, T_float) = T_float
  | arith(t, t') = raise type_error("invalid types for arithmetic");
```

So, when presented with an expression like $e + e'$, the compiler first determines (using `typeof`) the type t of e and t' of e'. The function `arith` tells us the type of $e + e'$. Knowing this latter type enables us to output either a `iadd` or a `fadd` JVM operation. Now consider an expression `x+y`, say, with `x` of type `int` and `y` of type `float`. It can be seen that the type of `x` differs from the type of `x+y`; hence a *coercion*, represented by a *cast* in Java, is applied to `x`. Thus the compiler (typically in `trexp` or in an earlier phase which only does type analysis) effectively treats `x+y` as `((float)x)+y`. These type coercions are also elementary instructions in intermediate code, for example in Java, `float f(int x, float y) { return x+y; }` generates

```
iload_0
i2f
fload_1
fadd
freturn
```

Overloading (having two simultaneous active definitions for the same name, but distinguished by type) of user defined names can require careful language design and specification. Consider the C++ `class` definition

```
class A
{   int f(int, int) { ... }
    float f(float, char) { ... }
    void main() { ... f(1,'a'); ... }
}
```

The C++ rules say (roughly) that, because the call (with arguments of type `char` and `int`) does not match any declaration of `f` exactly, the *closest in type* variant of `f` is selected and appropriate coercions are inserted, thus the definition of `main()` corresponds to one of the following:

```
void main() { ... f(1, (int)'a'); ... }
void main() { ... f((float)1, 'a'); ... }
```

Which is a matter of fine language explanation, and to avoid subtle errors I would suggest that you do not make your programs depend on such fine details.

7 Translation to machine code from intermediate code

The part II course on 'Optimising Compilation' will cover this topic in an alternative manner, but let us for now merely observe that each intermediate instruction listed above can be mapped into a small number of ARM or Pentium instructions, essentially treating JVM instructions as a `macro` for a sequence of Pentium instructions. Doing this naïvely will produce very unpleasant code, for example recalling the

```
y := x<=3 ? -x : x
```

example and its intermediate code with

```
iload_4        load x (4th load variable)
iconst_3       load 3
if_icmpgt L36  if greater (i.e. condition false) then jump to L36
iload_4        load x
ineg           negate it
goto L37       jump to L37
label L36
iload_4        load x
label L37
istore_7       store y (7th local variable)
```

could expand to (assuming a descending stack and 10 stack locations used for parameters and local variables):

```
movl   %eax,40-16(%fp) ; iload_4
pushl  %eax            ; iload_4
movl   %eax,#3         ; iconst_3
pushl  %eax            ; iconst_3
popl   %ebx            ; if_icmpgt
popl   %eax            ; if_icmpgt
cmpl   %eax,%ebx       ; if_icmpgt
bgt    L36             ; if_icmpgt
movl   %eax,40-16(%fp) ; iload_4
...
```

However, delaying output of `PUSH`es to stack by caching values in registers and having the compiler hold a table representing the state of the cache can improve the code significantly:

```
       movl   %eax,40-16(%fp) ; iload_4      stackcache=[%eax]
       movl   %ebx,#3         ; iconst_3     stackcache=[%eax,%ebx]
       cmpl   %eax,%ebx       ; if_icmpgt    stackcache=[]
       bgt    L36             ; if_icmpgt    stackcache=[]
       movl   %eax,40-16(%fp) ; iload_4      stackcache=[%eax]
       negl   %eax            ; ineg         stackcache=[%eax]
       pushl  %eax            ; (flush/goto) stackcache=[]
       b      L37             ; goto         stackcache=[]
L36:   movl   %eax,40-16(%fp) ; iload_4      stackcache=[%eax]
       pushl  %eax            ; (flush/label) stackcache=[]
L37:   popl   %eax            ; istore_7     stackcache=[]
       movl   40-28(%fp),%eax ; istore_7     stackcache=[]
```

I would claim that this code is near enough to code one might write by hand, especially when we are required to keep to the JVM allocation of local variables to `%fp`-address storage locations. The generation process is sufficiently simple to be understandable in an introductory course such as this one; and indeed we would not in general to seek to produce 'optimised' code by small

adjustments to the instruction-by-instruction algorithm we used as a basis. (For more powerful techniques see the Part II course "Optimising Compilers").

However, were one to seek to improve this scheme a little, then the code could be extended to include the concept that the top of stack cache can represent integer constants as well as registers. This would mean that the `movl #3` could fold into the `cmpl`. Another extension is to check jumps and labels sufficiently to allow the cache to be preserved over a jump or label (this is quite an effort, by the way). Register values could also be remembered until the register was used for something else (we have to be careful about this for variables accessible by another thread or `volatile` in C). These techniques would jointly give code like:

```
     movl  %eax,40-16(%fp) ; iload_4    stackcache=[%eax], memo=[]
                           ; iconst_3   stackcache=[%eax,3], memo=[%eax=local4]
     cmpl  %eax,#3         ; if_icmpgt  stackcache=[], memo=[%eax=local4]
     bgt   L36             ; if_icmpgt  stackcache=[], memo=[%eax=local4]
     negl  %eax            ; ineg       stackcache=[%eax], memo=[]
     b     L37             ; goto       stackcache=[%eax], memo=[]
 L36:                      ; (label)    stackcache=[], memo=[%eax=local4]
                           ; iload_4    stackcache=[%eax], memo=[%eax=local4]
 L37:                      ; (label)    stackcache=[%eax], memo=[]
     movl  40-28(%fp),%eax ; istore_7   stackcache=[], memo=[%eax=local7]
```

This is now about as good as one can do with this strategy.

7.1 Translation using tree matching and rewriting

This section gives an alternative method of generating code for CISC-like target architectures directly from parse trees.[6]

A slightly simplified version of the algorithm is presented here. The algorithm uses a collection of tree rewrite rules to define the resulting translation. Each rule has four components as follows:

```
replacement  <-  template        cost      code
```

where `replacement` is a single node, `template` is a tree, `cost` is the cost of using this rule, `code` is a fragment of compiled code. For example:

```
       Rule                            Cost      Code
#1  Ri  <-  Kc                           2      MOV #c,Ri
#2  Ri  <-  Ma                           2      MOV a,Ri
#3  C   <-  Ass(Ma, Ri)                  2      MOV Ri,a
#4  C   <-  Ass(Ind(Ri),Ma)              2      MOV a,*Ri
#5  Ri  <-  Ind(Add(Kc,Rj))              2      MOV c(Rj),Ri
#6  Ri  <-  Add(Ri, Ind(Add(Kc,Rj)))     2      ADD c(Rj),Ri
#7  Ri  <-  Add(Ri, Rj)                  1      ADD Rj,Ri
#8  Ri  <-  Add(Ri, K1)                  1      INC Ri
```

The tree pattern could be drawn as in Figure 5. The tree for the assignment `v[i] := x` might be as in Figure 6. A third representation of the same tree is

```
Ass(Ind(Add(Add(Kv,Rp),Ind(Add(Ki,Rp))), Mx)
```

This tree can be 'covered' by the templates in several ways, but one that gives the least cost is given in Figure 6. The total cost in this case is 7. Using this covering, the code and resulting trees produced by a depth first left to right scan are as follows:

[6] Aho, A.V., Ganapathi, M. and Tjiang, S.W.K. *Code Generation Using tree matching and Dynamic programming"*. ACM Transactions on Programming Languages and Systems, Vol 11, No 4, October 1989.

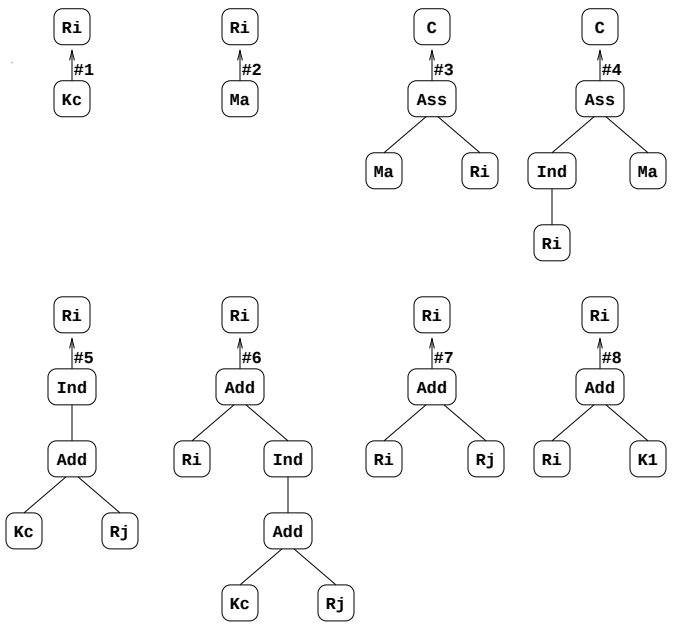


Figure 5: Tree version of rules

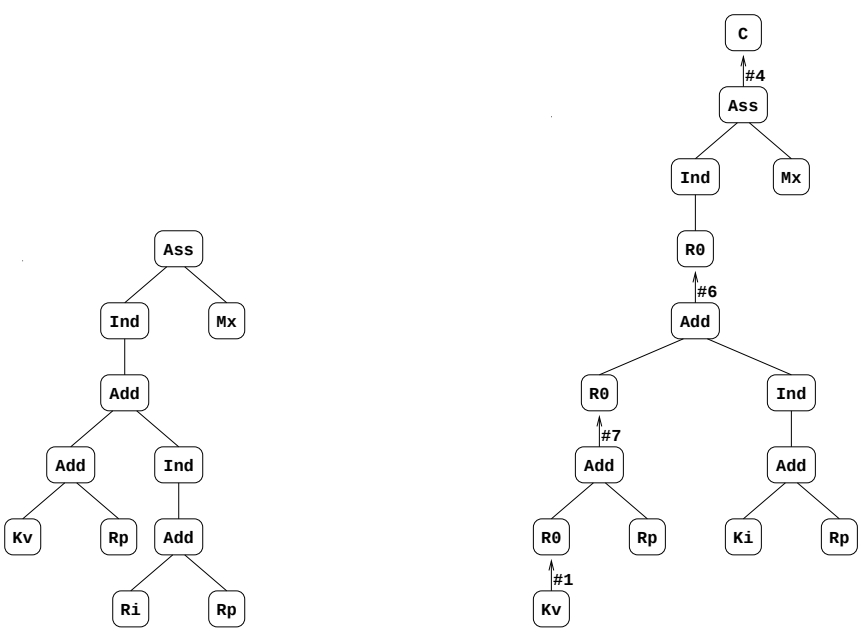


Figure 6: Tree for `v[i] := x` (left) and its minimum cost covering (right)

```
                    Ass(Ind(Add(Add(Kv,Rp), Ind(Add(Ki,Rp))), Mx)
MOV #v,R0           Ass(Ind(Add(Add(R0,Rp), Ind(Add(Ki,Rp))), Mx)
ADD Rp,R0           Ass(Ind(Add(R0,         Ind(Add(Ki,Rp))), Mx)
ADD i(Rp),R0        Ass(Ind(R0),                              Mx)
MOV x,*R0           C
```

Other coverings are possible but these would give different costs and different code sequences. The algorithm given here is designed to find a least cost covering efficiently.

The template `#6 Add(R?, Ind(Add(K?,R?)))` contains paths from the root to its three leaf nodes, namely:

```
Add.1--R?
Add.2--Ind.1--Add.1--K?
Add.2--Ind.1--Add.2--R?
```

The integers specify which edge is being taken from an operator node. The length of a path is the number of edges it contains. The path `Add.1--R?` is of length 1 and occurs in template `#6` (this is denoted by `#6/1`). We can combine all the paths from all the templates to form a tree as follows:

```
                                          Rule/Length

n00-Add.1-n01-R?--a00                     #8/1,#7/1,#6/1
  |    .2-n02-Ind.1-n03-Add.1-n04-K1-a01  #6/3,#5/2,#1/0
  |         |              |    *-K?-a02  #6/3,#5/2,#1/0
  |         |              .2-n05-R?-a03  #7/1,#6/3,#5/2
  |         *-K1--a04                     #8/1,#1/0
  |         *-R?--a05                     #7/1
  *-Ass.1-n06-Ind.1-n07-R?--a06           #4/2
  |    |    *-M?--a07                     #3/1,#2/0
  |    .2-n08-M?--a08                     #4/1,#2/0
  |         *-R?--a09                     #3/1
  *-Ind.1-n09-Add.1-n10-K1--a10           #5/2,#1/0
  |              |    *-K?--a11           #5/2,#1/0
  |              .2-n11-R?--a12           #7/1,#5/2
  *-K1--a13                               #1/0
  *-K?--a14                               #1/0
  *-M?--a15                               #2/0
```

The internal nodes of the tree are labelled `n00` to `n11`, and the leaf nodes are labelled `a00` to `a15`. Notice that the path `Add.1--R?` is of length 1 and occurs in three different templates, namely `#8`, `#7` and `#6` which accounts for why `#8/1,#7/1,#6/1` is attached to node `a00`. The node `a01` is at the end of the path `Add.2--Ind.1--Add.1--K1` which is a path of length three belonging to template `#6`. However, `a01` is also at the end of `Ind.1--Add.1--K1` which is a path of length 2 belonging to template `#5`, and it also ends a path of length zero belonging to template `#1`. This accounts for `#6/3,#5/2,#1/0` being attached to `a01`.

This tree forms the basis of a finite state acceptor with the leaf nodes being the accepting states. Extra transitions must be added, but the number of nodes remains unchanged. Consider the path string: `Ind.1--Add.2--Ind.1--Add.1--K1`. Starting from `n00`, `Ind.1` gets us to `n09`, then `add.2` gets us to `n11`, but from here the only transition in the tree is on `R?` to `a12`. This tells us that there are no path strings starting with `Ind.1--Add.2--Ind.1` belonging to any of our set of templates. What we should do is delete the first item of our path string, giving us `Add.2--Ind.1--Add.1--K1` and see if this leads to an accepting state. Thus from state `n11`, a transition on `Ind.1` should lead to `n03`. One way of viewing this is if state `n11` does not encounter `R?` then it should behave like state `n02`. Looking carefully at the tree we see that:

State	Not followed by	Behaves like
n01	R?	n00
n02	Ind.1, K1 or R?	n00
n03	Add.1 or Add.2	n09
n04	K1 or K?	n10
n05	R?	n11
n06	Ind.1 or M?	n00
n07	R?	n09
n08	M? or R?	n00
n09	Add.1 or Add.2	n00
n10	K1 or K?	n01
n11	R?	n02

This easily leads to the resulting finite state acceptor described by the following table:

	Add.1	Add.2	Ass.1	Ass.2	Ind.1	K1	K?	R?	M?	like
n00	n01*	n02*	n06*	n08*	n09*	a13*	a14*	-	a15*	-
n01	n01	n02	n06	n08	n09	a13	a14	a00*	a15	n00
n02	n01	n02	n06	n08	n03*	a04*	a14	a05*	a15	n00
n03	n04*	n05*	n06	n08	n09	a13	a14	-	a15	n09
n04	n01	n02	n06	n08	n09	a01*	a02*	a00	a15	n10
n05	n01	n02	n06	n08	n03	a04	a14	a03*	a15	n11
n06	n01	n02	n06	n08	n07*	a13	a14	-	a07*	n00
n07	n10	n11	n06	n08	n09	a13	a14	a06*	a15	n09
n08	n01	n02	n06	n08	n09	a13	a14	a09*	a08*	n00
n09	n10*	n11*	n06	n08	n09	a13	a14	-	a15	n00
n10	n01	n02	n06	n08	n09	a10*	a11*	a00	a15	n01
n11	n01	n02	n06	n08	n03	a04	a14	a12*	a15	n02

The asterisks (*) indicate transitions that are in the original tree. The non-asterisked entries of a row are copied from the row it (otherwise) behaves like. Compare, for instance, row n11 with row n02.

The rule/length information associated with accepting states are encoded as bit patterns, as shown in the following table.

	#8	#7	#6	#5	#4	#3	#2	#1	Path	Rule/Length
a00	10	10	0010	000	000	00	0	0	Add.1-R?	#8/1,#7/1,#6/1
a01	00	00	1000	100	000	00	0	1	Add.2-Ind.1-Add.1-K1	#6/3,#5/2,#1/0
a02	00	00	1000	100	000	00	0	1	Add.2-Ind.1-Add.1-K?	#6/3,#5/2,#1/0
a03	00	10	1000	100	000	00	0	0	Add.2-Ind.1-Add.2-R?	#7/1,#6/3,#5/2
a04	10	00	0000	000	000	00	0	1	Add.2-K1	#8/1,#1/0
a05	00	10	0000	000	000	00	0	0	Add.2-R?	#7/1
a06	00	00	0000	000	100	00	0	0	Ass.1-Ind.1-R?	#4/2
a07	00	00	0000	000	000	10	1	0	Ass.1-M?	#3/1,#2/0
a08	00	00	0000	000	010	00	1	0	Ass.2-M?	#4/1,#2/0
a09	00	00	0000	000	000	10	0	0	Ass.2-R?	#3/1
a10	00	00	0000	100	000	00	0	1	Ind.1-Add.1-K1	#5/2,#1/0
a11	00	00	0000	100	000	00	0	1	Ind.1-Add.1-K?	#5/2,#1/0
a12	00	10	0000	100	000	00	0	0	Ind.1-Add.2-R?	#7/1,#5/2
a13	00	00	0000	000	000	00	0	1	K1	#1/0
a14	00	00	0000	000	000	00	0	1	K?	#1/0
a15	00	00	0000	000	000	00	1	0	M?	#2/0

The number of bits allocated for a template is one greater than the length of the longest path in the template. The position of a one indicates the length of an accepted path string. Bit strings allow overlapping matches to the same tree template to be recorded.

7.1.1 The Algorithm

Perform a left to right depth first scan over the subject tree attaching (context) states of the acceptor to each node. For the given example this gives:

```
Replacement              Ri Ri   Ri   Ri   C  C Ri Ri
Rule number              #8 #7   #6   #5  #4 #3 #2 #1

Ass                 n00  00 00 0000 000 001 00  0  0   #4/0
(#4->C)                  00 00 0000 000 000 00  0  0
*-Ind               n06  00 00 0000 000 010 00  0  0   #4/1
| *-Add             n01  00 01 0001 000 000 00  0  0   #6/0, #7/0
|   (#6->Ri)             00 00 0000 000 100 00  0  0   #4/2
|   *-Add           n01  00 01 0000 000 000 00  0  0   #7/0
|   | (#7->Ri)           10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|   | *-Kv          n01  00 00 0000 000 000 00  0  1   #1/0
|   | | (#1->Ri)         10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|   | *-Rp          n02  00 10 0000 000 000 00  0  0   #7/1
|   *-Ind           n02  00 00 0010 001 000 00  0  0   #6/1,5/0
|     (#5->Ri)           00 10 0000 100 000 00  0  0   #7/1,#5/2
|     *-Add         n03  00 01 0100 010 000 00  0  0   #7/0,#6/2,#5/1
|       (#7->Ri)         00 00 0000 000 000 00  0  0
|       *-Ki        n04  00 00 1000 100 000 00  0  1   #6/3,#5/2,#1/0
|       | (#1->Ri)       10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|       *-Rp        n05  00 10 1000 100 000 00  0  0   #7/1,#6/3,#5/2
*-Mx                n08  00 00 0000 000 010 00  1  0   #4/1,#2/0
  (#2->Ri)               00 00 0000 000 000 10  0  0   #3/1
```

An item of the form `(#i->Op)` indicates that the current branch of the parse tree can be matched by rule `#i`, and if it is, the branch would be replaced by a leaf node with operator `Op`.

The bit patterns indicate for each position in the parse tree which rules are matched and to what depths. When the least significant bit is a one then that point in the tree, it can be matched by the corresponding rule. For instance, the bit pattern indicates that the root node can be matched by rule `#4`. The value obtained for a node is computed by ANDing together the bit patterns for its children and shifting the result right by one position. When a template matches that position in the tree can be replaced by a leaf node. This may lead to another accepting state whose bit pattern should be regarded as being ORed with the bit pattern for the node itself. Careful study of the above table should make the mechanism clear.

As each possible replacement is found its cost is computed and, if found to be lower than the cost of a previously discovered replacement (yielding the same leaf node), the new cost and the number of the rule that made it possible is recorded in the tree.

When the depth first scan is complete the root node will contain a list of possible leaf nodes that it can be replaced by, together with the minimum cost for each replacement and the corresponding rule that was used. A second pass over the tree can generate the code corresponding to this minimum cost covering.

8 Object Modules and Linkers

We have shown how to generate assembly-style code for a typical programming language using relatively simple techniques. What we have still omitted is how this code might be got into a state suitable for execution. Usually a compiler (or an assembler, which after all is only the word used to describe the direct translation of each assembler instruction into machine code) takes a source language and produces an *object file* or *object module* (`.o` on Unix and `.OBJ` on MS-DOS). These object files are linked (together with other object files from program libraries) to produce an *executable file* (`.EXE` on MS-DOS) which can then be loaded directly into memory for execution. Here we sketch briefly how this process works.

Consider the C source file:

```
int m = 37;
extern int h(void);
int f(int x) { return x+1; }
int g(int x) { return x+m+h(); }
```

Such a file will produce a *code segment* (often called a *text segment* on Unix) here containing code for the functions `f` and `g` and a *data segment* containing static data (here `m` only).

The data segment will contain 4 bytes probably `[0x25 00 00 00]`.

The code for `f` will be fairly straightforward containing a few bytes containing bit-patterns for the instruction to add one to the argument (maybe passed in a register like `%eax`)) and return the value as result (maybe also passed in `%eax`). The code for `g` is more problematic. Firstly it invokes the procedure `h()` whose final location in memory is not known to `g` so how can we compile the call? The answer is that we compile a 'branch subroutine' instruction with a dummy 32-bit address as its target; we also output a *relocation entry* in a *relocation table* noting that before the module can be executed, it must be linked with another module which gives a definition to `h()`.

Of course this means that the compilation of `f()` (and `g()`) cannot simply output the code corresponding to `f`; it must also register that `f` has been defined by placing an entry to the effect that `f` was defined at (say) offset 0 in the code segment for this module.

It turns out that even though the reference to `m` within `g()` is defined locally we will still need the linker to assist by filling in its final location. Hence a relocation entry will be made for the 'add `m`' instruction within `g()` like that for 'call `h`' but for 'offset 0 of the current data segment' instead of 'undefined symbol `h`'.

A typical format of an object module is shown in Figure 7 for the format ELF often used on Linux (we only summarise the essential features of ELF).

8.1 The linker

Having got a sensible object module format as above, the job of the linker is relatively straightforward. All code segments from all input modules are concatenated as are all data segments. These form the code and data segments of the executable file.

Now the relocation entries for the input files are scanned and any symbols required, but not yet defined, are searched for in (the symbol tables of) the library modules. (If they still cannot be found an error is reported and linking fails.) Object files for such modules are concatenated as above and the process repeated until all unresolved names have been found a definition.

Now we have simply to update all the dummy locations inserted in the code and data segments to reflect their position of their definitions in the concatenated code or data segment. This is achieved by scanning all the relocation entries and using their definitions of 'offset-within-segment' together with the (now know) absolute positioning of the segment in the resultant image to replace the dummy value references with the address specified by the relocation entry.

(On some systems exact locations for code and data are selected now by simply concatenating code and data, possibly aligning to page boundaries to fit in with virtual memory; we want code to be read-only but data can be read-write.)

Header information; positions and sizes of sections
`.text` segment (code segment): binary data
`.data` segment: binary data
`.rela.text` code segment relocation table: list of (offset,symbol) pairs showing which offset within `.text` is to be relocated by which symbol (described as an offset in `.symtab`)
`.rela.data` data segment relocation table: list of (offset,symbol) pairs showing which offset within `.data` is to be relocated by which symbol (described as an offset in `.symtab`)
`.symtab` symbol table: List of external symbols used by the module: each is listed together with attribute 1. undef: externally defined; 2. defined in code segment (with offset of definition); 3. defined in data segment (with offset of definition). Symbol names are given as offsets within `.strtab` to keep table entries of the same size.
`.strtab` string table: the string form of all external names used in the module

Figure 7: Summary of ELF

The result is a file which can be immediately executed by *program fetch*; this is the process by which the code and data segments are read into virtual memory at their predetermined locations and branching to the *entry point* which will also have been marked in the executable module.

8.2 Dynamic linking

Consider a situation in which a user has many small programs (maybe 50k bytes each in terms of object files) each of which uses a graphics library which is several megabytes big. The classical idea of linking (*static linking*) presented above would lead to each executable file being megabytes big too. In the end the user's disc space would fill up essentially because multiple copies of library code rather than because of his/her programs. Another disadvantage of static linking is the following. Suppose a bug is found in a graphics library. Fixing it in the library (`.OBJ`) file will only fix it in my program when I re-link it, so the bug will linger in the system in all programs which have not been re-linked—possibly for years.

An alternative to static linking is *dynamic linking*. We create a library which defines *stub* procedures for every name in the full library. The procedures have forms like the following for (say) `sin()`:

```
static double (*realsin)(double) = 0;  /* pointer to fn */
double sin(double x)
{   if (realsin == 0)
    {   FILE *f = fopen("SIN.DLL");    /* find object file */
        int n = readword(f);           /* size of code to load */
        char *p = malloc(n);           /* get new program space */
        fread(p, n, 1, f);             /* read code */
        realsin = (double (*)(double))p;  /* remember code address */
    }
    return (*realsin)(x);
}
```

Essentially, the first time the `sin` stub is called, it allocates space and loads the current version of the object file (`SIN.DLL` here) into memory. The loaded code is then called. Subsequent calls essentially are only delayed by two or three instructions.

In this scheme we need to distinguish the stub file (`SIN.OBJ`) which is small and statically linked to the user's code and the dynamically loaded file (`SIN.DLL`) which is loaded in and referenced at run-time. (Some systems try to hide these issues by using different parts of the same file or generating stubs automatically, but it is important to understand the principle that (a) the linker does some work resolving external symbols and (b) the actual code for the library is loaded (or possibly shared with another application on a sensible virtual memory system!) at run-time.)

Dynamic libraries have extension `.DLL` (dynamic link library) on Microsoft Windows and `.so` (shared object file) on Linux. Note that they should incorporate a version number so that an out-of-date DLL file cannot be picked up accidentally by a program which relies on the features of a later version.

The principal disadvantage of dynamic libraries is the management problem of ensuring that a program has access to acceptable versions of all DLL's which it uses. It is sadly not rare to try to run a Windows `.EXE` file only to be told that given DLL's are missing or out-of-date because the distributor forgot to provide them or assumed that you kept your system up to date by loading newer versions of DLL's from web sites! Probably static linking is more reliable for executables which you wish still to work in 10 years' time—even if you cannot find the a hardware form of the processor you may be able to find an emulator.