Compilers

Des Watson
January 2009

These notes outline the main topics covered in the *Compilers* course. They are based on material in [21] and some sections are taken directly from this book.

Programming languages can be divided into two broad and overlapping categories – these are the *low-level languages* and the *high-level languages*. The key advantage offered by high-level languages is *abstraction*, allowing irrelevant detail to be ignored. Other advantages include easier solution of problems, they are easier to learn and understand, they are more likely to be self-documenting, they result in easier debugging, modification and maintenance.

1 Comparing high-level languages

This is a list of some of the aspects of high-level languages that can form a basis for comparisons.

- Broad classification – is the language imperative, functional or in the logic programming category? Is it procedural or non-procedural/declarative? Is the language object-oriented?
- Language structure – language definition, program structuring facilities, subprograms (procedures, functions, subroutines, methods) and parameter passing, facilities for separate compilation.
- Data access and manipulation – variable declaration, typing and scope, data structuring, operators.
- Control structures and executable statements – range of executable statements, extensibility, concurrency.
- Language implementation and debugging – efficiency, interaction with the environment, ease of debugging.
- Readability and writability – correct programs, complexity, portability, orthogonality, layout rules, comments.

2 Structure of a compiler

A compiler is a program that translates the source form of a program (expressed in a high-level language) into an equivalent target machine language form. This target machine can be a "real" machine (such as a Pentium processor) or a virtual machine (such as the Java Virtual Machine).

Simple languages that are suitably defined can be compiled one statement at a time (or at least in groups of small numbers of statements) by a *one-pass compiler*. Such compilers are usually simple, fast and potentially easy to write. But many languages have features that make one-pass compilation difficult or impossible. So a *multi-pass compiler* is used.

Compiling is logically split into two phases – *analysis* of the source program and *synthesis* of the object program.

2.1 Lexical analysis

This phase reads the characters of the source program and recognises the basic syntactic components that they represent. Spaces, newlines and other layout characters are normally discarded (but beware – in some languages, these "layout" characters may be syntactically significant). Comments are ignored in the lexical analyser and are not passed to later stages of compilation.

2.2 Syntax analysis

This phase recognises the syntactic structure of the sequence of basic symbols delivered by the lexical analyser. A syntactic check of the program is performed and the *abstract syntax tree* or equivalent data structure is built. The tree may be *annotated* with additional information needed later in compilation (such as type information, required by the semantic analyser and code generator).

2.3 Semantic analysis (translation)

This pass deals with the scopes of identifiers, declarations, type checking, the allocation of storage and selection of polymorphic operators and the insertion of automatic type transfers. It can flatten the tree into a linear sequence of basic operations. It usually creates a representation of the program in some form of linear intermediate language.

2.4 Code generation

This pass converts the output from the semantic analysis phase into target machine instructions. It has to deal with the problems of the allocation of machine registers, selection of machine instructions and so on.

2.5 Code optimisation

This is an optional phase, often integrated with the process of code generation, and attempts to produce smaller and/or faster code from the output of the code generator. Usually, optimisation techniques are built into the code generator itself, and also into a phase after semantic analysis and before code generation (optimisation at the intermediate code stage can be particularly profitable), but there are optimisation techniques that can be applied as a separate pass once code has been generated.

This multi-module and potentially multi-pass approach outlined above has several important advantages:

- It makes a large task feasible to handle – consider software engineering issues. The job of writing the compiler may be more easily shared between a group of programmers, each working on separate passes.
- It is possible to reduce storage requirements of the compiler by overlaying passes.
- Modifications to the compiler often require modification to one pass only, and are thus simpler to make.
- The compiler is easier to describe and understand.
- More of the compiler can be target machine independent. Hence portability can be enhanced.

3 Language description

Typically, the specification of a language will be in three parts:
(a) the set of symbols that can be used in valid programs,
(b) the set of valid programs, and
(c) the "meaning" of each valid program.
Loosely, *syntax* is concerned with (a) and (b) and the *semantics* of the language is concerned with (c).

3.1 Syntax description

A language used to talk about another language is called a *metalanguage*.

3.2 Backus-Naur Form (BNF)

BNF is a metalanguage which was used to used to describe the syntax of ALGOL 60 and many other more recent languages. The metasymbols of BNF are:
::= separates a phrase name from its definition
| separates alternative definitions of a phrase
<> indicates that the intervening characters are to be considered as a unit.

For example:

```
<expression>::=<term>|<expression>+<term>
<term>::=<primary>|<term>*<primary>
<primary>::= x | y | z
```

An example of an expression is therefore `x*y+z`.

Syntax can also be specified using *syntax diagrams*. For example, the syntax diagram defining the syntax of a Pascal constant could be written as in figure 1.

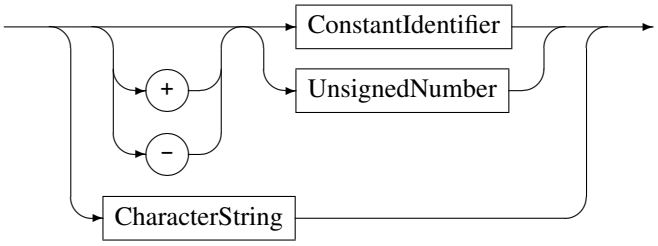

Figure 1: Syntax diagram defining a Pascal constant.

Extended BNF (EBNF) is very similar to BNF but includes several additional metasymbols which result in more compact and often more comprehensible syntax rules. For example, an *AssignmentStatement* could be defined as:

AssignmentStatement = (*Variable* | *FunctionIdentifier*) ":=" *Expression*.

3.3 Terminology

The *alphabet* of a language is the set of all characters that may appear in strings of the language. These are the *terminal symbols* (symbols of the object language).

Symbols in the metalanguage that denote strings in the object language are called *non-terminal symbols*. For example, in the grammar defined above, the terminal symbols are "+", "*", "x", "y" and "z", and the non-terminals are "`expression`", "`term`" and "`primary`".

The *starting symbol* is a distinguished non-terminal symbol from which all strings in the language are defined.

A *production* is a string transformation rule having a left-hand side that is a pattern to match a substring (possibly all) of the string to be transformed and a right-hand side that indicates a replacement for the matched part of the string. The grammar above consists of three productions.

The *grammar* of a language consists of a set of productions having a unique starting symbol. Formally, a grammar is a 4-tuple $G = (N, T, \Sigma, P)$ where
N is the set of non-terminal symbols
T is the set of terminal symbols
Σ is the starting symbol, $\Sigma \in N$
P is the set of productions $\alpha \rightarrow \beta$ ($\alpha \neq$ null)
$N \cap T = \emptyset$ (i.e. a symbol cannot be both a terminal and a non-terminal).

A *sentential form* is any string that can be derived from the starting symbol. A *sentence* is a sentential form consisting only of terminal symbols.

3.4 Chomsky classification

Type 0

$$\alpha \rightarrow \beta$$

α must be a member of U^+ and β must be a member of U^*. U^+ is the positive closure of the set U – i.e. the set of all non-empty strings that can be formed by the concatenation of members of U. U^* is the closure of U – that is, the set $U^+ \cup \{\varepsilon\}$.

Type 1 (context sensitive)

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

α, β, γ are members of U^*, γ is not null, A is a single non-terminal symbol.

Type 2 (context free)

$$A \rightarrow \gamma$$

A is a single non-terminal, γ is a member of U^*.

Type 3 (finite-state)

$A \rightarrow a$
or $A \rightarrow aB$

A and B are non-terminal symbols and a is a terminal symbol. The right hand side consists only of a terminal symbol or a terminal followed by a non-terminal.

3.5 Two-level grammars

BNF (and other similar metalanguages) seems to be well-suited to describe the syntax of many programming languages. But specification of semantics is also important. The semantics of a language can be considered in two categories:

- *dynamic semantics*, defining what will happen when the program actually runs, and
- *static semantics*, which gives all the information about the form of the program, obviously directed towards the compiler writer or to the programmer, such as the necessity for the declaration of variables and so on.

A reasonable aim is to extend the tools of syntax specification to include some of the above semantic notions. In order to do this, the conventional context-free tools (like BNF) have to be abandoned.

Van Wijngaarden grammars allow the syntactic treatment of context dependencies. One-level Van Wijngaarden grammars are equivalent to the conventional context-free grammars. Two-level Van Wijngaarden grammars generate the productions of a one-level Van Wijngaarden grammar by means of a grammar. ALGOL 68 is a language that has been described in this way.

3.6 Derivation and parsing

It is helpful to look first at the process of *derivation*; that is, the process of taking the starting symbol and repeatedly replacing non-terminals according to the production rules. The derivation where the leftmost non-terminal is replaced at each step is called the **leftmost derivation**; similarly, the **rightmost derivation** involves the replacement of the rightmost non-terminal at each step.

Consider the syntax definition:

<expression> → <term> | <expression> + <term>
<term> → <primary> | <term> * <primary>
<primary> → $a|b|c$

Using this grammar, the leftmost derivation of $a*b+c$ from <expression> is:

<expression>
<expression> + <term>
<term> + <term>
<term> * <primary> + <term>
<primary> * <primary> + <term>
$a*$ <primary> + <term>
$a*b+$ <term>
$a*b+$ <primary>
$a*b+c$

The rightmost derivation is

<expression>
<expression> + <term>
<expression> + <primary>
<expression> $+c$
<term> $+c$
<term> * <primary> $+c$
<term> $*b+c$
<primary> $*b+c$
$a*b+c$

Parsing or *syntax analysis* is the process of finding the syntactic structure associated with an input sentence. A parse should specify which productions are being used, and in which order. The process of parsing is the reverse of the process of derivation.

A *canonical parse* starts at the left hand end of the sentential form and first applies the production that reduces the characters furthest to the left (in the reverse order of the rightmost derivation). This then produces a second sentential form, and the process is repeated as required. The substring that is reduced by the first reduction in the canonical parse is called the *handle* of the sentential form.

A language is said to be *unambiguous* if there exists one and only one canonical parse for every sentence of the language. One of the most famous examples of ambiguity is the *if...then...else* statement of several high-level languages.

Parsing is seldom simple – powerful and/or ad hoc methods may be required to deal with some aspects of some languages. It may be necessary to employ techniques involving backtracking, but this should be avoided if at all possible in order to maintain efficiency and simplicity.

4 Lexical analysis

The lexical analyser is the first step of the analysis phase of the compiler and some care in the choice of implementation algorithms has to be taken since it can account for a significant proportion of the compilation time. The function of the lexical analyser is to split up the source program into a set of *basic symbols (tokens, atoms, lexemes)*. Information about these symbols is then passed to the syntax analyser.

The syntax analyser could, in theory, perform the functions of the lexical analyser, but these two modules are usually separated for several reasons:

- The compiler is split into more manageable pieces.
- Since the syntax of these basic symbols can be specified by a very simple grammar, there is generally no need to use the heavyweight techniques that usually have to be adopted by the syntax analyser.

- It makes the syntax analyser simpler and faster.

If the syntax of the symbols can be specified by a *type 3 grammar* (a *regular* grammar), then it is possible to construct a simple and efficient lexical analyser.

A *regular expression* is composed of characters and operators for concatenation (space), alternation (|), repetition (*) and parentheses for grouping. For example, `(a b | c)* d` is the specification for a set of strings including `d`, `abd`, `cd`, `ababd`, `abcd`, etc.

Using simple rules, regular expressions can be converted to *transition diagrams*. A transition diagram can be used as the basis of a recogniser algorithm.

The transition diagram can be drawn as a *directed graph with labelled branches*. For example, `(a b | c)* d` can be shown as in Figure 2:

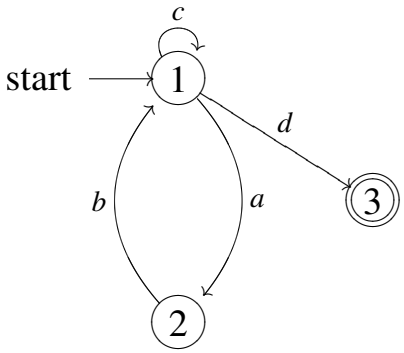

Figure 2: Directed graph with labelled branches

Each of the nodes is called a *state*, and each labelled branch is called a *transition*. Note the close relationship between this type of diagram and a type 3 grammar.

Implementation of a parser based on the directed graph is made easier by writing the graph as a *transition matrix* (the states marked '—' indicate a syntax error). This is illustrated in Figure 3.

	input symbol			
state	a	b	c	d
1	2	—	1	3
2	—	1	—	—
3		finished		

Figure 3: Transition matrix for the regular expression $(ab|c)*d$

In a program using such a technique, each matrix entry could identify a routine to handle the transition and set up the next matrix row to be used.

This technique is fast and simple, but if used on a large scale, some method should be used for storing the sparse matrix in a reasonable amount of storage – the transition matrix can become very large.

This parser can be considered as a *finite-state automaton* which is specified by:

- a finite set of control states,
- the allowable input symbols,
- the initial state,
- the set of final states (i.e. the states indicating acceptance of the input), and

- the state transition function (i.e. given the current state and the current input symbol, the function that specifies the set of possible next states).

If the state transition function is such that for any pair of current state/input symbol there is only one possible next state, then the parser is a *deterministic finite-state automaton* (DFA). However, if several next states are possible given a current state and an input symbol, then a *non-deterministic finite-state automaton* (a NFA) is the result.

An example of a NFA is shown in Figure 4:

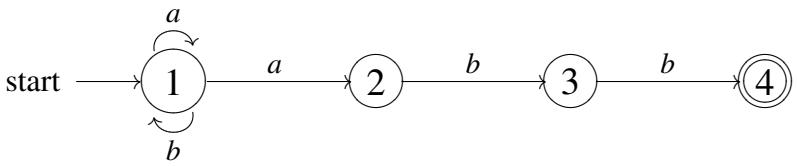

Figure 4: A non-deterministic finite-state automaton

This is nondeterministic because there are two branches labelled "a" emerging from state 1.

Suppose the above machine is in state 1 and it receives an "a". Then it goes into states 1 and 2 *simultaneously*, replicating itself so that one instance of itself exists for each of its possible new states. The device accepts the input if *any* of its parallel existences reaches a final accepting state.

Given a regular grammar, it is possible to construct a state diagram and then a NFA or a DFA. Any sentence of the grammar is accepted by the automaton, so that running the machine is equivalent to performing a *bottom-up parse*.

Simple techniques exist for converting a NFA to a DFA – see the standard textbooks for details.

To summarise – it is possible to construct a lexical analyser as a collection of recognisers for tokens where each token is defined as a regular expression, the regular expression is converted into a NFA, the NFA is converted into a DFA, the DFA is state-minimised and this final DFA is implemented in software. This forms a formal and potentially automated technique for constructing a lexical analyser.

4.1 Programming a lexical analyser

If the syntax of the symbols of a programming language can be described as a set of regular expressions, then the techniques described above may be used. Once the state transition diagram has been constructed, the actual coding of the program becomes relatively simple. Remember that for most real programming languages, the transition matrix for all the possible symbols may prove to be too unwieldy to handle and so other more heuristic methods may have to be adopted, such as using these matrices only when recognising certain symbols (such as floating point numbers).

Care should be taken to define exactly what the lexical analyser should regard as "atomic symbols". Also, some thought should be given to the problems of error detection and recovery, but it is usually the syntax analyser that takes the main responsibility for handling syntactic errors.

Several lexical analyser generating programs exist. These programs usually work by generating the state transition matrix given the syntax (expressed in the form of regular expressions or BNF or equivalent) of the basic symbols. These programs are of general applicability and offer a potentially simple way of generating an efficient and reliable lexical analyser. Sometimes they are integrated with programs that generate syntax analysers too.

In practice, lexical analysers for real programming languages are usually either written by hand in an ad hoc manner or written by machine using a lexical analyser generating program. Hybrid methods are sometimes used too. Syntactic complexity of tokens encourages the use of formal techniques for implementation, making use of automated tools.

5 Syntax analysis

The aim of the syntax analyser is to take the output of the lexical analyser, making sure that the sequence of symbols conforms to the syntax of the language, and to construct some representation of the program suitable for input to the semantic analyser (or direct to the code generator).

Chomsky type 3 grammars are in general too restrictive for any reasonable general purpose programming languages, and so this section is primarily concerned with type 2 and similar grammars.

A *parse* of a sentential form is the construction of a derivation and possibly a syntax tree for it. We are only concerned here with *left-to-right parsing*. There are many possible parsing strategies. One group of methods starts at the "top" of the grammar (at the starting symbol) and works down towards the sentential form. This is *top-down parsing*.

The other major group of parsers starts "at the bottom" and works up towards the goal symbol. The operation of many of these *bottom-up parsers* is based on the syntactic relationships between individual (adjacent) tokens.

Most bottom-up techniques are deterministic since the parser makes a series of definite decisions leading directly to the correct parse. Some top-down techniques are non-deterministic since they involve some form of guessing or exhaustive search, requiring backtracking when the guesses are wrong. But backtracking should be avoided if at all possible.

Working with type 2 grammars where the meaning of a symbol is not dependent on the context of the symbol makes many parsing problems disappear. Fortunately, this is the case (or at least very nearly the case) in most current popular programming languages.

Top-down parsing is a comparatively simple idea, but practical difficulties often appear. The goal of recognising the starting symbol is repeatedly broken down into a series of sub-goals. For example, in order to recognise S defined as S $\rightarrow$ AB, A has to be recognised followed by B. If a sub-goal starts with terminal symbols, these can be matched with the corresponding terminal symbols in the input string. If the symbols do not match, the sub-goal is not achieved. *Backtracking* may be necessary – for example, consider A $\rightarrow$ B|C. If the input string does not match the sub-goal B, then the parser has to "put back" the symbols read before attempting to recognise C.

In contrast, bottom-up parsing starts with the input string and repeatedly matches strings that appear on the right hand sides of productions with the input string, replacing them by the corresponding symbols on the left hand sides, until just the starting symbol remains.

Consider the process of *derivation* – the repeated replacement of non-terminal symbols. A *leftmost derivation* is one in which the leftmost non-terminal is replaced at each stage. A *rightmost derivation* is one in which the rightmost non-terminal is replaced at each stage. A left-to-right bottom-up parser which reduces characters furthest to the left first produces the rightmost derivation in reverse.

A grammar is said to be **LL(k)** if a parser can be written for that grammar that can make a decision of which production to apply at any stage simply by looking at most at the next k symbols of the input. LL(1) grammars are a simple and important category – simple and efficient top-down parsers can be written for them.

A grammar is said to be **LR(k)** if a parser can be written for that grammar that makes a single left to right pass over the input with a lookahead of at most k symbols. These grammars can be parsed with bottom-up parsers, requiring no backtracking.

5.1 Top-down parsing

The abstract syntax tree is constructed by starting at the root node (which corresponds to the starting symbol) and working down towards the sentence.

5.1.1 A general approach

It is natural to associate a function or procedure to recognise each non-terminal symbol. Implementing grammar rules of the form A $\rightarrow$ BC is trivial (the routine A consists of a call to B followed by a call to C), but handling alternation may give rise to problems requiring the use of backtracking. Managing the exhaustive

following of all paths through the grammar together with the backtracking may result in a complex and inefficient parser.

Productions of the form $E \rightarrow E + T$ are called *left recursive* and a naive implementation of a top-down parser for such productions will cause problems. The procedure E will immediately call itself recursively without consuming any input. Left recursion can also arise because of mutually recursive productions. It can be shown that given a left-recursive grammar, there is an equivalent grammar that is not left-recursive. Simple formal techniques exist for transforming such grammars, but it may be possible to use repetition to express the recursion. For example, $E \rightarrow E + T|T$ can be rewritten as $E \rightarrow T\{+T\}$ using the notation of EBNF. This form using repetition can be translated into recognising code really easily.

It is also possible to transform a grammar to make the provision of backtracking unnecessary in some circumstances using techniques such as *factoring*.

Note also the problems of *ambiguity* which can confuse any parser – either rewrite the grammar rules to remove the ambiguity or use *ad hoc* rules within the parser to sort the problem out.

After removing the ambiguity and the left recursion, it is possible to write a *top-down backtracking parser* for the grammar. But this is not a popular technique due to complexities and inefficiencies in its implementation. The need for backtracking prevents this from becoming a practically useful technique. The grammar has to be modified to make backtracking unnecessary.

5.1.2 Recursive descent parsing (predictive parsing)

These are top-down parsers which need no backtracking. The parser is made deterministic by designing the grammar so that it is possible to determine which of the alternative productions to use by just knowing the identity of the current input symbol. Instead, practical parsers have to be written so that they do no backtracking – they are *predictive* parsers.

For example, consider the grammar:

$$< \text{assignment} > \rightarrow < \text{identifier} > = < \text{expression} >;$$
$$< \text{expression} > \rightarrow < \text{expression} > + < \text{term} > | < \text{term} >$$
$$< \text{term} > \rightarrow < \text{identifier} > | (< \text{expression} >)$$
$$< \text{identifier} > \rightarrow \text{x|y|z}$$

The second production is left-recursive and it can be written as

$$< \text{expression} > \rightarrow < \text{term} > \{+ < \text{term} >\}$$

Writing in a language something like C, a recogniser for this grammar could be coded as follows:

```
char token;

/* We assume the presence of a routine nexttoken() which performs the
   lexical analysis, returning the next symbol from the input in the
   variable token each time it is called.  However, a lexical analyser
   for anything but the simplest of programming languages would probably
   return an 'int' rather than a 'char'. */

extern char nexttoken();
extern void error(char*);
void expression();

void identifier()
{
  if ((token=='x') || (token == 'y') || (token == 'z'))
    token=nexttoken();
  else
    error("Identifier expected");
}
```

```
void term()
{
  if (token == '(') {
    token=nexttoken();
    expression();
    if (token != ')') error(") expected");
    else token=nexttoken();
  }
  else identifier();
}

void expression()
{
  term();
  while (token == '+') {
    token=nexttoken();
    term();
  }
}

void assignment()
{
  identifier();
  if (token != '=') error("= expected");
  else {
    token=nexttoken();
    expression();
    if (token != ';') error("; expected");
    else token=nexttoken();
  }
}
```

In this example, the single token lookahead makes the parser deterministic so there is no need for backtracking. One character lookahead is sufficient. Note also that the parser contains no code for the production of a parse tree, or indeed, for any form of code generation. Code to construct the parse tree can be inserted comparatively easily within the recognising procedures. Furthermore, for very simple target architectures, it may be possible to combine code generation with the parsing process.

5.1.3 Error recovery

In the example above, the error recovery mechanism is far too crude to be adopted for a practical compiler. The assumption is that the `error` routine outputs the error message and then causes the parsing to halt. Error detection is easy in a recursive descent parser, but error recovery is far more difficult. A standard technique is to ensure that the parser is "synchronised" with the input at the beginning and at the end of each recognising procedure. Each procedure works with two sets of symbols – those that can start a construct being recognised and those that can terminate the construct. This second set may be augmented by additional symbols passed to the procedure by its caller.

5.2 Bottom-up parsing

In bottom-up parsing, we start with the tokens in the input string rather than with the starting symbol of the grammar. The problem is really that of the identification of the handle – the substring to be matched by the RHS of a production and replaced by the LHS.

The bottom-up parsers described here all fall into the category of *shift-reduce parsers*. The *shift operation* reads and stores an input symbol, and the *reduce* operation matches a group of adjacent stored symbols with the RHS of a production and replaces the stored group with the LHS of the production.

For example, given the grammar

$$E \rightarrow E + T|T$$
$$T \rightarrow T * P|P$$
$$P \rightarrow a|b|c$$

it is possible to show the actions of a shift-reduce parser when parsing the sentence a*b+c as shown in Figure 5.

Stack	*input string*	*action*
	a*b+c	shift a
a	*b+c	reduce (P→a)
P	*b+c	reduce (T→P)
T	*b+c	shift *
T*	b+c	shift b
T*b	+c	reduce (P→b)
T*P	+c	reduce (T→T*P)
T	+c	reduce (E→T)
E	+c	shift +
E+	c	shift c
E+c		reduce (P→c)
E+P		reduce (T→P)
E+T		reduce (E→E+T)
E		

Figure 5: Actions of a shift-reduce parser

Parsing is now complete and successful because the stack contains the starting symbol and there is no further input. But this example makes no attempt to illustrate a technique to determine *when* to shift and *when* to reduce.

5.2.1 Syntax analysis using precedence

There is a certain class of grammars called *precedence grammars* for which it is possible to write relatively simple parsers. Given a sentential form of this type of grammar, formal methods may be defined for determining the handle, enabling us to reduce the sentential form in the canonical parse.

The problem is stated in the following terms. Given the string ....RS...., is R always the tail of the handle, or can both R and S appear in the handle together, or what?

There are three possibilities.

1. R is part of the handle, S is not. Write $R \gtrdot S$. R has *precedence* over S because it must be reduced first. R must be the tail symbol of some production.

2. R and S are both in the handle. Write $R \doteq S$. They have the same precedence so they have to be reduced at the same time. There must be a rule U→...RS...

3. S is part of the handle, R is not. Write $R \lessdot S$. R has lower precedence than S. S must be the head of some production.

If there is no canonical sentential form ...RS..., then no relationship exists between the ordered pair (R,S).

5.2.2 Construction of the precedence matrix

A matrix can be constructed showing the precedence relationships between all pairs of symbols. The following rules are used:

1. A $\doteq$ B if there exists a production P $\rightarrow \alpha$ABβ (α, β can be null).
2. A $\lessdot$ B if there exists a production P $\rightarrow \alpha$A$\gamma\beta$ and γ produces Bπ for some string π.
3. A $\gtrdot$ B if there exists a production P $\rightarrow \alpha\gamma$Bβ and γ produces πA for some π,
   or
   if there exits a production P $\rightarrow \alpha\gamma\delta\beta$ where γ produces μA, δ produces Bπ.

A language where a unique relationship exists between all pairs of alphabet characters (either $\doteq$, $\lessdot$, $\gtrdot$ or no relationship at all) is called a *precedence grammar*.

We define the *left set* of a symbol P (written L(P)) as the set of symbols that can occur on the left (the head) of any production from P. Similarly, the *right set* of a symbol P (written R(P)) is the set of symbols that can occur on the right hand end (the tail) of any production from P.

Finally, we are in a position to define the rules for determining the precedence relations.

1. If there is a production A $\rightarrow \ldots s_i s_{i+1} \ldots$ then $s_i \doteq s_{i+1}$.

2. $s_i \lessdot s_k$ where $s_k \in \mathrm{L}(s_{i+1})$.

3a. $s_k \gtrdot s_{i+1}$ where $s_k \in \mathrm{R}(s_i)$.

3b. $s_k \gtrdot s_l$ where $s_k \in \mathrm{R}(s_i)$ and $s_l \in \mathrm{L}(s_{i+1})$.

Sparse matrix techniques may have to be used in order to store a precedence matrix in the parser. But it may be possible to assign a numerical value to each symbol, so that when the corresponding values are compared, the appropriate precedence relation is obtained. Two functions $f(X)$ and $g(X)$ can be defined such that:

if X $\lessdot$ Y then $f(X) < g(Y)$
if X $\doteq$ Y then $f(X) = g(Y)$
if X $\gtrdot$ Y then $f(X) > g(Y)$

Instead of the table being of size n^2, just $2n$ numerical values have to be stored. Note that these functions are not unique, and that there are many precedence matrices for which no such functions exist. Also note that in this representation, the information about pairs of symbols that *lack* a precedence relationship is lost – an alternative approach must be used to handle error detection.

5.2.3 Example

Consider the grammar:

Z $\rightarrow$ bMb
M $\rightarrow$ (L
M $\rightarrow$ a
L $\rightarrow$ Ma)

The left and right sets are:

L(Z) = {b}, L(M) = {(a}, L(L) = {M (a}
R(Z) = {b}, R(M) = {L) a}, R(L) = {)}

The precedence matrix is therefore as shown in Figure 6.

	Z	b	M	L	a	(	)
Z							
b			$\doteq$		$\lessdot$	$\lessdot$	
M		$\doteq$			$\doteq$		
L		$\gtrdot$			$\gtrdot$		
a		$\gtrdot$			$\gtrdot$		$\doteq$
(			$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	
)		$\gtrdot$			$\gtrdot$		

Figure 6: Precedence matrix

5.2.4 Parsing using precedence relations

The method of implementation of a shift-reduce parser presented above can be used. Symbols are read from the input and pushed one by one onto a stack until there is a $\gtrdot$ relationship between the top of the stack and the next input symbol. Then the symbols forming the handle must be on the stack, and symbols are taken off the stack until there is a $\lessdot$ relationship between the top of the stack and the symbol just removed. The symbols just removed constitute the handle, and the LHS of the corresponding production is pushed onto the stack.

It can be shown that a precedence grammar is unambiguous.

5.2.5 Problems with precedence grammars

At first sight, precedence parsing looks like a good technique – it is simple and implementations can be very efficient. However, it is a technique that is now rarely used in practice because it is difficult, if not impossible, to transform an "average" programming language grammar into a precedence form. For example, consider the grammar:

$$\begin{aligned}
&\mathrm{P} \rightarrow \{\mathrm{E}\} \\
&\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T} | \mathrm{T} \\
&\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F} | \mathrm{F} \\
&\mathrm{F} \rightarrow (\mathrm{E}) | \mathrm{I}
\end{aligned}$$

From the first two rules, we have $\{ \doteq \mathrm{E}$ and $\{ \lessdot \mathrm{E}$ showing that this cannot be a precedence grammar. However, the grammar may be transformed thus:

$$\begin{aligned}
&\mathrm{P} \rightarrow \{\mathrm{E}'\} \\
&\mathrm{E}' \rightarrow \mathrm{E} \\
&\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T}' | \mathrm{T}' \\
&\mathrm{T}' \rightarrow \mathrm{T} \\
&\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F} | \mathrm{F} \\
&\mathrm{F} \rightarrow (\mathrm{E}') | \mathrm{I}
\end{aligned}$$

This is, in fact, a precedence grammar. Unfortunately in this transformation, the grammar has become significantly more complicated, and large precedence matrices are somewhat difficult to handle.

5.2.6 Operator precedence grammars

In an ordinary precedence grammar as described above, relationships may be defined between all pairs of symbols. In an operator grammar, we are only concerned with relationships between *operators*.

An *operator grammar* is one in which there are no productions containing consecutive non-terminal symbols. In such a grammar, no sentential form contains consecutive non-terminals (simple to prove this). In deciding whether an operator grammar is an operator precedence grammar, we only have to consider the relationships between terminal symbols.

An *operator precedence grammar* is an operator grammar where a unique relationship occurs between pairs of terminal symbols. These relationships are called the *precedence relations* of the terminal symbols.

The three relationships are defined as follows:

1. $a \doteq b$ if there exists a production $\mathrm{P} \rightarrow \alpha a b \beta$ or $\mathrm{P} \rightarrow \alpha a \mathrm{A} b \beta$ (A non-terminal, a, b terminal).
2. $a \lessdot b$ if there exists a production $\mathrm{P} \rightarrow \alpha a \mathrm{A} \beta$ and A produces $b\pi$ for some π or A produces $\mathrm{D}b\pi$ where D is non-terminal.
3. $a \gtrdot b$ if there exists a production $\mathrm{P} \rightarrow \alpha \mathrm{A} b \beta$ and A produces πa for some π or A produces $\pi a \mathrm{D}$ where D is non-terminal.

Note that the rule 3b for simple precedence grammars has no equivalent here since consecutive non-terminals are not allowed.

For example:

$S \rightarrow [E]$
$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow (E) | x$

The LEFT and RIGHT sets for all the non-terminal symbols are:

LEFT(S) = $\{[\}$	RIGHT(S) = $\{]\}$
LEFT(E) = $\{+ \ * \ (\ x\}$	RIGHT(E) = $\{+ \ * \ ) \ x\}$
LEFT(T) = $\{* \ (\ x\}$	RIGHT(T) = $\{* \ ) \ x\}$
LEFT(F) = $\{(\ x\}$	RIGHT(F) = $\{) \ x\}$

The precedence matrix can now be constructed:

	[	]	+	*	(	)	x
[		$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$
]							
+		$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
*		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
(			$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
)		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
x		$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$	

This grammar is an operator precedence grammar because there are no adjacent non-terminals in any of the productions (hence an operator grammar) and there are no clashes of precedence.

5.2.7 Parsing of operator precedence grammars

The parser is constructed in a similar manner to the simple precedence parser described above, but note that in the operator precedence grammar, non-terminals are not directly involved in the parsing process. This makes parsing much simpler, but puts a greater burden of checking operands etc. on other parts of the compiler. Unfortunately, the error detecting capabilities of these parsers tend to be rather poor. Again, these operator precedence parsers are of limited applicability, but there are some constructs commonly found in programming languages for which they can be used very successfully.

5.2.8 LR(k) parsers

LR parsers are efficient bottom-up parsers which can be constructed for a large class of context-free grammars. An LR(k) grammar is one that generates strings each of which can be parsed during a single deterministic scan from left to right without looking ahead more than k symbols.

These parsers are generally very efficient and good at error reporting, but unfortunately they are very difficult to write without the help of special parser-generating programs.

The parser maintains a stack containing numerical state values – the state is a coded indication of the current left context. A convenient way of implementing an LR(1) parser is via a parsing table. Each entry (indexed by the current input symbol and the state number at the top of the stack) contains a description of the next action the parser should perform. The possible actions are *shift, reduce, accept* and *error*. The table contains all the information on the grammar – the parsing program itself is then grammar independent.

6 Semantic analysis (translation)

The role of the semantic analyser is to derive methods by which the structures constructed by the syntax analyser may be evaluated or executed. For example, the semantic analyser in a C compiler must be able to define an evaluation procedure for expressions by determining the type attributes of the components, selecting appropriate forms of the operators, issuing error messages if the types are inappropriate and so on. It must also check that all names have been declared. The semantic analyser or translator phase of the compiler takes the syntax tree produced by the syntax analyser and "flattens" it ready for the code generator. In this flattening process, types and declarations are checked, type transfers are inserted where necessary, etc. It is often very difficult to distinguish the semantic analyser from parts of the code generator since in some compilers, the functions are combined.

Many compilers use some form of intermediate code as an interface between the analysis and synthesis phases. There are good portability reasons for the use of an intermediate code that is independent of the target machine.

6.1 Symbol tables

The symbol table is a central data structure of the compiler. It contains all the names declared within the program, together with other information such as types, locations, scopes and so on. Type information may have to be quite complex so that the storage of a single type identifier will not generally suffice. The means for identifying the run-time location of the name (for example, if it is a variable) obviously depends on the method used for run-time storage allocation, but scope rules and other semantic or hardware/operating system features will almost certainly prevent the simple storage of absolute addresses.

7 Code generation

One of the major problems that has to be solved before the code generator is written is that of *storage allocation*. For some simple languages, storage allocation is comparatively straightforward – where all variable storage information can be obtained during compilation. Other languages need more sophisticated techniques.

Static allocation can be used for some simple languages where it is possible to decide at compile time the address that each object will occupy at run time. This implies that the number and size of all possible objects is known at compile time. But in languages where one can write code of the form:

```
procedure x(n:integer);
var a:array[1..n] of integer;
begin
        .
        .
```

there are problems because in such an example, it would not be possible to determine the value of `n` and hence the size of the local array `a` at compile time.

Consider the function `fact` (written in C):

```
int fact(int n)
{
  if (n==0) return 1;
```

```
  else return n*fact(n-1);
}
```

An implementation based on static storage allocation would not work here. For each recursive application of `fact`, a new location for storing the local variable `n` is required, and since the depth of recursion is not known at compile time, static allocation cannot cope. In order to implement such routines, *dynamic allocation* is required.

Dynamic allocation is required for recursive routines and also for arrays with calculated bounds. Usually a *stack* is involved, and all local variables are allocated on the stack. The code generated by the compiler when accessing any of the local variables has to be via a stack pointer. Normally a machine register is used to contain the value of the current stack pointer.

Some *linkage information* also has to be stored on the stack, such as a routine return address and the old stack pointer so that the old environment may be restored on return.

Consider a language like C where variable declarations are allowed in each block.

```
{
   int a,b,c;
   .
   .
   {
      int c,d,e;
        .
        .
   }
}
```

In the outer block, variables `a`, `b` and `c` are accessible and in the inner block, variables `a`, `b`, `c`, `d` and `e` are accessible, but the variable `c` is not the same variable `c` that was accessible in the outer block. This means that the compiler must maintain some sort of block count associated with each variable so that the code generator can access the variable in the current stack frame or in one further down the stack. Alternatively, organising the symbol table as a stack may help. One never makes any reference to any stack frames further up the stack. For example, if `a` is accessed in the inner block, the code generator will produce code to access the stack frame just 'below' the current stack frame.

With a single stack pointer, it is possible to:

- access the variables in the current stack frame,
- access the variables in the enclosing blocks (by accessing frames further down the stack – the old stack pointers in the linkage information permit this), and
- restore the environment on block exit or procedure return.

In order to access the variables of an enclosing block, it is tempting simply to follow the chain of pointers down the stack. It is clear that if the nesting is deep, this may take an appreciable time. It therefore may seem reasonable to use several stack pointer registers, one pointing to the current stack frame, another pointing to the outermost stack frame (the 'global' variables) and others pointing to frames in between. Note that this implementation may help support the use of dynamic subscripts in array declarations.

Unfortunately, this simple scheme is unworkable since the compiler cannot determine at compile time how far down the stack to go in order to access, for example, a global variable (since it cannot predict the depth of the run-time procedure nesting). One can, of course, store a block number with each stack frame so that the right frame can be identified in such a circumstance but this yields a very inefficient solution. Some extra structure has to be set up to speed such variable access.

The old stack pointers (the P pointers) described above form a *dynamic chain*, indicating the dynamic flow of control of the program. We also need a *static chain*, reflecting the scopes of variables, etc. The dynamic chain is used for returning to the caller's environment and the static chain is used to access non-local variables.

Unfortunately, this has not solved all the problems since if the static block nesting is deep, the static chain has to be followed for a long way in order to access some variables. In order to overcome this inefficiency, a *display* may be used. The display is a table containing pointers to the currently active data blocks corresponding to each block in the program. This takes the place of the static chain. It is possible to maintain a number of displays on the stack in order to overcome the problems of display space management. Each time a block (with declarations) or a procedure is entered, space is allocated on the stack for the display. In this display, the only pointers to be stored correspond to the blocks containing data currently accessible by the block. In such a display scheme, variables are referenced by means of an integer pair – the nesting depth and the relative offset.

Ideally, one would like the stack pointer and the entire display to be held in machine (index) registers. Since the number of available registers is usually limited, some sort of compromise has to be achieved.

The implementation of GOTO's is often a problem when the destination of the jump is outside the current block/environment. In such circumstances, the stack pointer has to be reset before performing the jump in order to reflect the environment of the target of the jump.

Remember that pointers *up* the stack must be avoided. Such references may arise from the use of pointer variables or by passing procedures as arguments.

A storage area not managed on a dynamic basis is also often required. This area is often called the *heap*, and it may be necessary to implement a heap garbage collector.

7.1 Example of dynamic allocation of variable storage using a stack

This example is taken from [21].

Consider a Pascal program of the form:

```
program static(output);
var i,j:integer;

procedure p1;
var a,b:integer;

    procedure p1a;
    var c,d:integer;
    begin ...; p1a; ... end;

begin ...; p1a; ... end;

procedure p2;
var p,q:integer;
    begin ...; p1; ... end;

begin (* main program *)
    .
    .
    p2;
    .
    .
end.
```

If the stack now incorporates the static chain, then three locations have to be set aside in each frame for linkage information. Immediately after *p1a* has called itself recursively for the first time – that is, there has been two activations of *p1a* – the stack has the form shown in Figure 7.

By examining the static chain from the latest invocation of *p1a*, it can be seen that the variables defined in the enclosing procedure *p1* and in the main program can be accessed by following the static chain back down the stack. Note that in following this particular chain, the variables for *p2* are bypassed since they are not in scope within *p1a*.

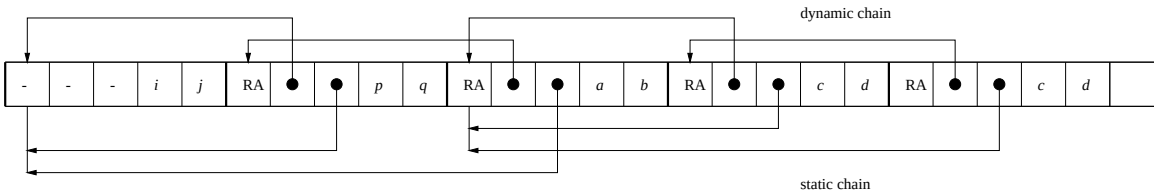

Figure 7: The static chain

7.2 Code generator design

A vital question to be answered at an early stage is what form of output from the code generator is required. Possibilities include absolute memory images, assembly language programs, object modules (perhaps relocatable) and so on. The choice depends on many factors, but the production of some form of object modules is now the norm.

One of the major aspects of code generation is the treatment of arithmetic expressions. Much of the theory of such code generation can be carried over into other structures and so such techniques play a very important part in code generation. The following algorithms are taken from [15].

Consider first a very simple machine with a single accumulator and the following instructions:

```
LOAD X     load accumulator from memory location X
STORE X    store accumulator in location X
ADD X      add contents of location X to accumulator
SUB X
MUL X
DIV X
```

For example, the code generated for $(a+b)/(c+d)$ may be:

```
LOAD a
ADD b
STORE T1
LOAD c
ADD d
STORE T2
LOAD T1
DIV T2
```

or

```
LOAD c
ADD d
STORE T1
LOAD a
ADD b
DIV T1
```

In this section, no attempt is made to remove common sub-expressions or exploit special properties of operators or operands in order to optimise the code.

Under these circumstances, if the expression has n operators, then its code will have exactly n ADD, SUB, MUL, DIV instructions (operator instructions). Only the number of LOAD's and STORE's will vary. Also note that the number of LOAD's must be one greater than the number of STORE's so that the answer may be left in the accumulator. Optimisation is therefore concerned with the minimisation of the number of LOAD's or STORE's.

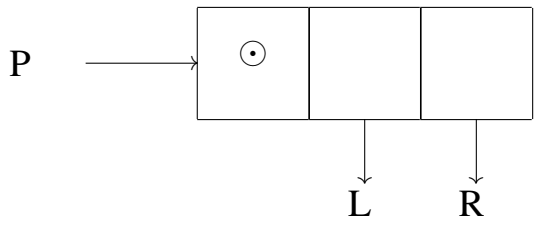

Figure 8: Tree node structure

We consider code generation from a tree. Suppose P is an internal node of the tree. L and R are its left and right subtrees respectively. The operator at node P is $\odot$. See figure 8.

In order to compute L $\odot$ R, L and R have to be computed independently and then L $\odot$ R is computed. C(L) and C(R) is the code generated for the left and right expression trees. There are several possibilities in the calculation of L $\odot$ R.

1. Both L and R are leaves (variables a and b respectively).

   LOAD a; $\odot$ b

2. L is a leaf with variable a, R is not a leaf.

   C(R); STORE T1; LOAD a; $\odot$ T1

3. R is a leaf with variable a, L is not a leaf.

   C(L); $\odot$ a

4. Neither R nor L are leaves. L computed before R.

   C(L); STORE T1; C(R); STORE T2; LOAD T1; $\odot$ T2

5. Neither R nor L are leaves. R computed before L.

   C(R); STORE T1; C(L); $\odot$ T1

Since the code of 5 is shorter than that of 4, the method of computing R before L is always used.

Consider the tree for the expression $(a+b)/(c+d)$. Starting at the root, it can be seen that the code generated by the recursive algorithm defined above is:

```
LOAD c; ADD d; STORE T1; LOAD a; ADD b; DIV T1
```

If nothing is assumed about the special properties of the operators, then this algorithm generates optimal code for the machine described above. However, if we can make use of the commutative properties of operators, the optimal code may be different.

Consider the tree for the expression $a+b*c$. The code generated by the algorithm above is:

```
LOAD b; MUL c; STORE T1; LOAD a; ADD T1
```

If it is assumed that the + operator is commutative (i.e. $a+b=b+a$), the optimal code becomes:

```
LOAD b; MUL c; ADD a
```

The above algorithm may be modified fairly easily to reflect such possibilities.

The machine used above is extremely simple in that it only has one register (accumulator). The effect of the availability of several registers obviously complicates the generation of code for arithmetic expressions, but the derivation of the method is comparatively simple – see [15] for further details.

We have assumed throughout that the left and right operands of an operator have to be computed independently. If the expression has common sub-expressions, then the "expression tree" becomes a *graph*, and generating optimal code under such circumstances becomes very much more difficult.

In general, it is not too difficult to code generate once the tree has been constructed. It is, however, very difficult to produce excellent code for any particular machine – a great number of constraints have to be considered. For example, it may be possible to condense pairs of the form `LOAD A; STORE B` to `MOVE A,B`.

8 Code optimisation

Optimisation is potentially a very difficult area, full of serious pitfalls. Here, one can only give the outline of some standard techniques and warn of some of the problems.

There are several general techniques that can be applied at some stage of the code generation process. Some of these techniques are well-suited for application *before* code generation starts – i.e. they can be applied to the intermediate representation. Many production compilers perform aggressive optimisation at this stage, carrying out an extensive analysis of the program. In particular, *flow analysis* can be carried out here. Some techniques can be applied during code generation and others should be applied after target machine instructions have been generated – target machine dependent optimisations are usually done at this stage.

8.1 Flow optimisation

Here's a simple example of how code can be optimised at the intermediate code stage, performing common sub-expression elimination and the removal of redundant variables.

Consider the two statements:

```
x = a/(b*b + c*c);
y = b/(b*b + c*c);
```

Straightforward translation into linear intermediate code yields:
$t_1 = b * b$
$t_2 = c * c$
$t_3 = t_1 + t_2$
$x = a/t_3$
$t_4 = b * b$
$t_5 = c * c$
$t_6 = t_4 + t_5$
$y = b/t_6$
Common sub-expression elimination yields:
$t_1 = b * b$
$t_2 = c * c$
$t_3 = t_1 + t_2$
$x = a/t_3$
$t_4 = t_1$
$t_5 = t_2$
$t_6 = t_4 + t_5$
$y = b/t_6$
Removal of redundant variables yields:
$t_1 = b * b$
$t_2 = c * c$
$t_3 = t_1 + t_2$

$x = a/t_3$
$t_6 = t_1 + t_2$
$y = b/t_6$
Common sub-expression elimination applied again yields:
$t_1 = b * b$
$t_2 = c * c$
$t_3 = t_1 + t_2$
$x = a/t_3$
$t_6 = t_3$
$y = b/t_6$
Finally, removal again of redundant variables yields:
$t_1 = b * b$
$t_2 = c * c$
$t_3 = t_1 + t_2$
$x = a/t_3$
$y = b/t_3$

8.2 Folding

Folding involves the performing of operations whose operands are known at compile time. For example, $k := k + 2 + 3$ should be folded to $k := k + 5$. Another useful example is as follows. Suppose x is a real variable. If the assignment $x := 3$ is made, the conversion of the integer 3 to the real 3.0 should be done at compile time rather than at run time. In general, as many of such conversions as possible should be done at compile time, especially since they are usually quite expensive operations.

8.3 Elimination of redundant expressions

An example of this technique is the factoring out of common sub-expressions. For example, most programmers are aware of the fact that it may be better to avoid statements like $x := sin(y) * sin(y) * sin(y)$ and that it would be better to write it as $t := sin(y); x := t * t * t$.

It is, of course, possible for the compiler to recognise such constructs and make the shown improvement automatically. But this may have unfortunate effects. Suppose the programmer was using a special routine called `sin` that had some side-effect (such as incrementing some global count each time it was called). By writing the assignment in the first form, the programmer would expect the `sin` routine to be called three times. If the compiler made the suggested optimisation, the `sin` routine would be called just once, against the wishes of the programmer.

Optimisation of the computation of such expressions can be somewhat more difficult – consider $x := a + b + b + a$ where it is perhaps not immediately obvious that the calculation of $a + b$ need be performed only once.

It may also be possible to remove unreachable code. Techniques of *flow analysis* can be applied, usually at the intermediate code stage (see above).

8.4 Statement rearrangement

Consider $a := b * c; d := e; f := a + b;$
If the statements are taken in order, the code generated for the above statements may be:

```
LOAD b; MUL c; STORE a;
LOAD e; STORE d;
LOAD a; ADD b; STORE f;
```

But if the statements are rearranged thus: $d := e; a := b * c; f := a + b;$ the code becomes:

```
LOAD e; STORE d; LOAD b; MUL c; STORE a; ADD b; STORE f;
```

But such rearrangements cannot always be done – consider $a := b*c; c := e; f := a+c;$. Again, techniques of flow analysis can be applied to determine the validity of such approaches.

So far, only source program optimisation has been considered. It is of course possible to optimise the object program, and in order to do this, a good knowledge of the target machine is required.

8.5 Register usage

Registers are usually in short supply, and so the code generator must keep careful track of the contents of the registers at all times in order to use them effectively. When there are few registers available, they may have to be shared between many different functions, and doing this effectively may be quite difficult.

8.6 Peephole optimisation

Code generators usually operate locally, so that code generators may produce code fragments that can be locally optimal but suboptimal when juxtaposed. For example, the local code for a source program conditional ends with a branch – so does the local code for the end of a loop. It is usually easier to implement a separate optimising pass after code generation to remove such redundant code and so on rather than to build it all into the code generator.

There are several important optimisations that can be performed in this manner.

- Delete unwanted tests (a previous instruction may set the condition codes).
- Exploit special-case instructions or exotic address calculations.
- Collapse chains of branches.
- Delete unreachable code.
- Simulate register contents so that a register reference may be substituted for a memory reference if possible.
- Perform pipeline scheduling to ensure good performance on pipelined processors.

Programs that inspect the output of the code generator in this way are called peephole optimisers.

9 Java compilers

Compiling object-oriented languages may pose particular problems if efficient target code is required. It may be necessary for the compiler to generate runtime type-checking code. Java implementations are traditionally carried out by translating the Java source into an intermediate representation – code for the *Java Virtual Machine*. This JVM code is then *interpreted* on the target machine. This approach offers many advantages, including simplicity of implementation, flexibility and portability but the advantages are at the cost of reduced runtime performance. Java compilers of a more traditional design, producing target code directly, are now also available.

Another approach to Java implementation is via the *just-in-time compiler*, where the JVM bytecode is translated to native code on the fly, on calling a particular method or on loading a class file. Here, compilation is only done when necessary, and repeated calls to the same method result in just a single compilation overhead. Significant performance gains are possible here over the simpler interpreted approach.

10 Compiler implementation

Firstly, one has to decide the language in which the compiler is to be written. There are many obvious reasons for writing it in a high-level language.

10.1 Method 1

Suppose a Pascal compiler for a particular machine is required, and it is to be written in C. If there is already a C implementation on the machine, the Pascal compiler may be written straight away, taking Pascal as input and generating machine code as output.

If there is no C system available, there are clearly two options - either implement C and then Pascal or use another machine on which C is already available. In general, the second option involves less effort.

10.2 Method 2

We want a Pascal compiler for machine 1, written in C. C is available on machine 2 and not on machine 1. So the compiler is written (in C) on machine 2, taking Pascal source as input and producing 1's machine language as output. In order to run a Pascal program on machine 1, it has to be compiled first on machine 2 and the machine code thus generated transferred to machine 1. Both machines 1 and 2 must always be available.

10.3 Method 3

Suppose now that the Pascal compiler for machine 1 is not to be written in C, but instead in Pascal. There is nothing particularly special about this - the compiler is simply a Pascal program that takes Pascal source statements as input and translates them to the equivalent statements in 1's machine code. We assume that Pascal is available on machine 2.

On machine 2, write a compiler in Pascal which takes Pascal source as input and produces 1's machine code as output. Once this has been completed, we are in a position to compile Pascal programs on machine 2 to be run on machine 1. Note that the Pascal compiler just written is nothing more that a large Pascal program. So we take the source of the compiler (for it is written in Pascal) and feed it as input to the newly-written compiler. This produces a representation of the new Pascal compiler in 1's machine code. This machine code can then be run on machine 1 – this is the Pascal compiler running on machine 1, producing 1's machine code as output. Note that machine 2 is only used in the compiler bootstrapping process, and is not required subsequently.

10.4 Method 4

In order to reduce further the total amount of work that has to be done, we make use of the fact that a compiler may be written in two parts – a machine independent "frontend" and a machine-dependent code generator. The interface between the two parts is via some intermediate language (machine independent). We call this intermediate language in the case of our Pascal compiler "Pcode".

Write a Pcode to 1's machine code translator (code generator) in Pascal on machine 2. Obtain the Pcode version of the standard "frontend" (which can always be obtained by passing the source of the frontend through the frontend of the compiler on machine 2). Using the code generator from the step above, produce the frontend in 1's machine code. Obtain the Pcode for the new code generator by using the existing compiler on machine 2, and by using the code generator from the first step, produce the code generator for machine 1 in 1's machine code. We now have a frontend and a code generator for Pascal in 1's machine code – i.e. a Pascal compiler for machine 1. Note again that machine 2 is no longer required.

This method has an important consequence. The code generator written in the first step need not be a particularly good one. Once the Pascal system has been implemented on machine 1 with the poor code generator, it can be improved at leisure, producing better and better versions of the system.

Remember the acid test – a compiler compiling itself should produce itself!

10.5 Method 5

There is yet another technique which is often very useful. This again makes use of the intermediate Pcode, and it enables an implementation of the language to be performed very quickly indeed.

Instead of writing a Pcode to 1's machine code translator straight away, we write a Pcode *interpreter*. This program is written in any language already available on machine 1 – in assembler if necessary – and it

simulates the Pcode instructions. This implies that given the Pcode version of a Pascal program, it can be run interpretively on machine 1. Once the interpreter has been completed, we obtain the Pcode version of the frontend of the Pascal compiler. This can be interpreted by our new interpreter, enabling us to produce Pcode from Pascal sources on machine 1. We can then input such Pcode into the interpreter and interpretively execute any of our Pascal programs – i.e. we now have an interpretive version of Pascal running on machine 1. We can now write a Pcode to 1's machine code translator (code generator) in Pascal, and interpret it using the interpreter, enabling us to produce 1's machine code from any Pcode segment. Using this code generator, a complete compiler in 1's machine code can be produced.

This method has two important advantages. Firstly, only one machine is involved – no other computer is required for the bootstrapping operation. Secondly, since the Pcode interface is fairly close to a typical machine's architecture, writing a Pcode interpreter is usually a fairly simple task, perhaps just a few pages of FORTRAN or assembler. If the Pcode were written in, say, FORTRAN, the language system is made very portable indeed, since it is most likely that machine 1 will already have a FORTRAN compiler available.

11 Compiler generating tools

A compiler generating tool is a program that takes as input some specification of the syntax of a source language, the desired translation (i.e. the correspondence between the source language statements and the target machine code statements) and constructs an appropriate compiler. These programs can save a great deal of time in the writing of a compiler. Some of these tools generate compilers that may be a little bulky and slow but the compilers produced may generate very good code.

Two of the most popular tools are lex and yacc (or flex and bison). Lex generates lexical analysers from a formal (regular expression) specification of the lexical tokens. Yacc generates syntax analysers from a BNF-like specification in which syntactic patterns are associated with actions (expressed in C code). Java tools such as JavaCC (lexer and top-down parser generator), CUP (bottom-up parser generator) and Jlex (lexer generator) are all popular.

12 Portability

The question of portability is a very important aspect of software design. Writing portable programs can be very difficult, and the design of some programming languages makes the problem especially difficult in some cases - e.g. moving a Pentium assembly language program to a SPARC system.

There are many things to consider when determining whether a program can be transported to a new machine. These include:

- Language incompatibilities – one manufacturer's C is very unlikely to be exactly the same as that of another manufacturer. Care has to be taken to write the program in a widely-understood subset of the language. International standardisation is an important issue here.
- Word lengths – a program should not assume, for example, that 4 characters may be stored in a machine word. Character sizes are also not always the same.
- Numerical representation and accuracy – in C, a `float` may give satisfactory accuracy on one machine but a `double` is required on another.
- Operating system interfaces – there are vast differences in the type and extent of operating system services provided by different systems. A program relying on operating system support for network operations, for example, is not portable.
- Character sets – one must never assume anything about the internal representations of characters.
- Other machine constraints – for example, addressing – some systems have difficulty in handling very large arrays, and there may be other incompatibilities due to fundamental machine design.

- The actual process of data transfer from one machine to another may prove to be unexpectedly difficult. Software and hardware incompatibilities in a communications network may cause significant difficulties.

Remember that if a compiler is written in a transportable manner, the language it compiles becomes a transportable language.

It is sometimes possible to use a macroprocessor to write portable software. A *macro* is a facility for replacing one sequence of symbols by another, and the *macroprocessor* is a piece of software for supporting such macros. The macroprocessor takes source text and a set of macro definitions as input. These macro definitions define patterns of symbols that are to be replaced and also what they are to be replaced by.

A compiler could be implemented as a series of macro calls, and it is probably then up to the implementor to write the actual macros (probably expanding into machine target language) for any particular machine.

13 Reading list and references

This is a list of a few books containing material relevant to the Compilers course. Note that the library contains many others of relevance.

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques & Tools*. Pearson Education, second edition, 2007.

[2] Henk Alblas and Albert Nymeyer. *Practice and Principles of Compiler Building with C*. Prentice Hall, 1996.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[4] Henri E. Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.

[5] P. Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice Hall International, London, 1985.

[6] John Elder. *Compiler Construction: A Recursive Descent Model*. Prentice Hall International, 1994.

[7] Charles N. Fischer and Richard J. LeBlanc Jr. *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, 1988.

[8] Charles N. Fischer and Richard J. LeBlanc Jr. *Crafting a Compiler With C*. The Benjamin/Cummings Publishing Company, 1991.

[9] D. Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, New York, 1971.

[10] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. Wiley, 2000.

[11] John L. Hennessy and David A. Patterson. *Computer Organization and Design – The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[13] Allen I. Holub. *Compiler Design in C*. Prentice Hall International, second edition, 1994.

[14] E. Horowitz, editor. *Programming Languages: A Grand Tour*. Computer Science Press, second edition, 1985.

[15] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Pitman, 1978.

[16] Ronald Mak. *Writing Compilers and Interpreters – An Applied Approach*. Wiley, 1991.

[17] Thomas Pittman and James Peters. *The Art of Compiler Design – Theory and Practice*. Prentice Hall International, Englewood Cliffs, New Jersey, 1992.

[18] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill Book Company, New York, 1985.

[19] Julian R. Ullman. *Compiling in Modula-2: A First Introduction to Classical Recursive Descent Compiling*. Prentice Hall International, 1994.

[20] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction*. Harper Collins, 1993.

[21] Des Watson. *High-Level Languages and their Compilers*. International Computer Science Series. Addison-Wesley Publishing Company, Wokingham, England, 1989.

[22] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.

[23] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[24] Niklaus Wirth. *Compiler Construction*. International Computer Science Series. Addison-Wesley, 1996.

14 How to choose a programming language

This is an article that appears regularly in various forms on netnews. It gives a perceptive and accurate view of many programming languages.

> The proliferation of modern programming languages which seem to have stolen countless features from each other sometimes makes it difficult to remember which language you are using. This guide is offered as a public service to help programmers in such dilemmas.
>
> **APL** You hear a gunshot, and there is a hole in your foot, but you do not remember enough linear algebra to understand what happened.
>
> **Ada** If you are dumb enough to actually use this language, the United States Department of Defense will kidnap you, stand you up in front of a firing squad, and tell the soldiers, "Shoot at his feet."
>
> **Algol** You shoot yourself in the foot with a musket. The musket is aesthetically fascinating, and the wound baffles the adolescent medic in the emergency room.
>
> **Assembly** You crash the OS and overwrite the root disk. The system administrator arrives and shoots you in the foot. After a moment of contemplation, the administrator shoots himself in the foot and then hops around the room rabidly shooting at everyone in sight.
>
> **BASIC** Shoot self in foot with water pistol. On big systems, continue until entire lower body is waterlogged.
>
> **C++** You accidentally create a dozen instances of yourself and shoot them all in the foot. Emergency medical care is unavailable because you cannot tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."
>
> **CLIPPER** You grab a bullet, get ready to insert it in the gun so that you can shoot yourself in the foot, and discover that the gun that the bullet fits has not yet been built, but should be arriving in the mail Real Soon Now.
>
> **COBOL** USEing a COLT45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER, and SQUEEZE. THEN return HANDGUN to HOLSTER. Check whether shoelace needs to be retied.
>
> **C** You shoot yourself in the foot.
>
> **DBase IV v 1.0** You pull the trigger, but it turns out that the gun was a poorly-designed grenade and the whole building blows up.
>
> **DBase** You squeeze the trigger, but the bullet moves so slowly that by the time your foot feels the pain you have forgotten why you shot yourself anyway.
>
> **English** You put your foot in your mouth, then bite it off.
>
> **FORTRAN** You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-processing ability.
>
> **Forth** Yourself foot shoot.
>
> **Java** You attempt to shoot yourself in the foot using a bullet that will work in any gun in the world. But you discover that the "Microsoft Gun" is actually a cross bow.
>
> **Lisp** You shoot yourself in the appendage which holds the gun with which
> you shoot yourself in the appendage which holds the gun with which
> you shoot yourself in the appendage which holds the gun with which
> you shoot yourself in the appendage which holds...
>
> **Modula-2** After realising that you cannot actually accomplish anything in the language, you shoot yourself in the head.
>
> **Pascal** The compiler won't let you shoot yourself in the foot.

PL/I You consume all available system resources, including all the offline bullets. The Data Processing & Payroll Department doubles its sizes, triples it budget, acquires four new mainframes, and drops the original one on your foot.

Prolog You attempt to shoot yourself in the foot, but the bullet, failing to find its mark, backtracks to the gun which then explodes in your face.

SNOBOL You grab your foot with your hand, then rewrite your hand to be a bullet. The act shooting the original foot then changes your hand/bullet into yet another foot (a left foot).

SQL You cut your foot off, send it out to a service bureau and when it returns it has a hole in it, but will no longer fit the attachment at the end of your leg.

Scheme You shoot yourself in the appendage which holds the gun with which
you shoot yourself in the appendage which holds the gun with which
you shoot yourself in the appendage which holds the gun with which
you shoot yourself in the appendage which holds...
...but none of the other appendages are aware of this happening.

sh, csh, etc. You cannot remember the syntax for anything, so you spend five hours reading man pages before giving up. You then shoot the computer and switch to C.

Smalltalk You spend so much time playing with the graphics and windowing system that your boss shoots you in the foot, takes away your workstation, and makes you develop in COBOL on a character terminal.

Visual Basic You'll really only appear to have shot yourself in the foot, but you'll have had so much fun doing it that you won't care.

15 Sample questions

Here are some questions to help with revision in the Compilers course. Most of the questions come from past exam papers (this course used to be called *Languages and Compilers*). The last section of questions is very mixed. Some are fairly easy, but others are rather long and involved. Perhaps some are more suited to thought and discussion rather than to written solution.

15.1 Informatics – 2008

1. (a) What is the role of the lexical analyser in a typical compiler? Why separate lexical and syntax analysis? [5 marks]

   What high-level language characteristics make lexical analysis difficult? What factors determine whether a construct is recognised by the lexical analyser or by the syntax analyser? [5 marks]

   (b) Explain carefully why it can be a good idea to implement a compiler for a particular language in that language. What other factors influence the choice of a programming language for the coding of a compiler? Illustrate your answer by assessing the suitability of Java (or any other programming language you know) for compiler implementation. [10 marks]

2. (a) A language includes a definition of a non-terminal A as follows (upper case letters are non-terminals, lower case letters are terminals):

   $$A \rightarrow a \mid bA$$

   Outline a method for a predictive top-down parser that recognises the construct specified by A.

   Suppose the definition of A is changed to:

   $$A \rightarrow aB \mid abC$$

   Outline a recognising method for this new A.

   Suppose the definition of A is now changed to:

   $$A \rightarrow a \mid Ab$$

   Again, outline a recognising method for this new A.

   Suppose finally that the definition of A is changed to:

   $$A \rightarrow a \mid Ab \mid cA$$

   Using this definition, show how the sentence cab can be constructed via a rightmost derivation. Hence explain carefully why a parser would have difficulty with this definition of A. [15 marks]

   (b) Why do hand-written parsers nearly always use the top-down, predictive approach? [5 marks]

3. (a) What are the primary aims of code optimisation? At what stages of the compilation process should code optimisation be performed? [5 marks]

   (b) What is peephole optimisation? Describe some of the optimisations that can be performed effectively using this approach and suggest a strategy for its implementation. [10 marks]

   (c) What are the advantages in code generating from a linear intermediate representation rather than directly from the syntax tree? [5 marks]

15.2 Informatics – 2007

1. (a) Explain carefully the role of the lexical analyser in a typical compiler. Why is it usually sensible to separate lexical and syntax analysis? [5 marks]

   (b) Produce a state transition diagram for a recogniser for a comment in a high-level language where a comment starts with `/*` and ends with `*/`. Suggest how this recogniser could be implemented in software. [7 marks]

   (c) A tool is required to convert all `switch` statements to equivalent `if` statements in a Java program. Outline the design of such a tool (don't worry about low-level syntactic detail in your answer). [8 marks]

2. (a) What is the definition of a Chomsky type 2 grammar? What is the relevance of this formalism to compiler construction? [4 marks]

   (b) Show that the grammar defined below is a precedence grammar. Include your working. [10 marks]

   $$S \rightarrow S(E) \mid E$$
   $$E \rightarrow T$$
   $$T \rightarrow P+T,P \mid P$$
   $$P \rightarrow x$$

   Show how a shift/reduce parser can be controlled by the data in the precedence matrix when parsing the sentence $x(x+x,x)$. Verify that this parse is indeed the same as the rightmost derivation in reverse. [6 marks]

3. (a) Describe a scheme for run-time memory allocation based on a stack and static and dynamic chains for the implementation of a block-structured procedural language. Show how both local and non-local variables may be accessed at run-time. [12 marks]

   (b) You have been asked to investigate why a large number-crunching application written in Java (for which the source code is available) takes so long to produce its results. How would you tackle this investigation? Suggest strategies you could adopt to achieve a shorter execution time for this program. [8 marks]

15.3 Informatics – 2006

1. (a) What are regular expressions and why are they relevant to the design of lexical analysers in compilers for high-level languages? [4 marks]

   (b) Explain in English the meaning of the regular expression

   ```
   a(aa)*(b|c|d)*a
   ```

   Show how this construct could be expressed in BNF. Outline a formal method for the implementation of a recogniser for tokens defined by such a regular expression. When is this a reasonable approach for a practical implementation? [16 marks]

2. (a) Construct a grammar for simple arithmetic expressions involving the operators + and *, parentheses for grouping and the operands x, y and z. [5 marks]
   Using this grammar, produce the rightmost derivation of the sentence x+y*z. What is the relevance of the rightmost derivation to bottom-up parsing? [4 marks]

   (b) A programming language definition contains a rule with the following alternatives:

   ```
   <command> ::=  . . .
                | while <expression> do <command>
                | <command> repeatuntil <expression>
                | . . .
   ```

   Why would these productions cause difficulty to a top-down predictive parser? [4 marks]
   Consider the string:

   ```
   while <expression> do <command> repeatuntil <expression>
   ```

   Is this a valid example of a `<command>`? Using the productions above, show how this example string can be parsed to yield two different parse trees. What does this imply about the rule defining `<command>`? [5 marks]
   Suppose the rule is changed to:

   ```
   <command> ::=  . . .
                | while <expression> do <command>
                | repeat <command> until <expression>
                | . . .
   ```

Does this remove the difficulties? [2 marks]

3. Describe an algorithm for the code generation of algebraic expressions using the operators `+`, `-`, `*` and `/` stored in the form of a tree. Assume that the machine for which you are generating code has a single register and instructions are available to load this register from memory, store this register to memory and to perform add, subtract, multiply and divide operations on the values in the register and a memory location (leaving the result in the register). [10 marks]

   What code is generated by your algorithm when presented with the tree for the expression `(a+b*c)*(c*b+a)`? [4 marks]

   How could the code generated for this expression be improved? Discuss where in the compiler this optimisation should be done. [6 marks]

15.4 Informatics – 2005

1. (a) Explain carefully the key functions of the lexical analyser, the syntax analyser, the semantic analyser and the code generator of a typical compiler, and outline appropriate interfaces between these phases. Why should the analysis phase of compilation ideally be target machine independent? Why is this sometimes difficult? [10 marks]

   (b) Using BNF, define a `<comment>` for a high-level language. A comment consists of the character { followed by a string of zero or more arbitrary characters (not containing }) followed by the character }. You can assume that a non-terminal `<any>` has already been defined that matches any single character other than { and }. [6 marks]

   (c) Modify your definition of `<comment>` to allow comments to be nested. Suggest why the nesting of comments is rarely supported in programming languages. [4 marks]

2. (a) What are the fundamental differences between top-down and bottom-up parsers? [4 marks]

   (b) Describe the operation of a predictive top-down parser, and suggest why it is such a popular technique. Explain why left recursive productions can cause difficulties in the writing of a predictive parser. Show how these particular difficulties may be overcome. [8 marks]

   (c) Show how an abstract syntax tree can be constructed by a top-down parser. How can type information be added to leaf nodes in the tree and then propagated throughout the tree? [8 marks]

3. (a) What are the primary aims of code optimisation? At what stages of the compilation process should code optimisation be performed? [4 marks]

   (b) Why is it important for the code generator to generate code making effective use of target machine registers? Why is it in general hard to write a good register allocator? [6 marks]

   (c) What is peephole optimisation? Describe some of the optimisations that can be performed effectively using this approach and suggest a strategy for its implementation. [10 marks]

15.5 Informatics – 2004

1. (a) What are Chomsky type 3 grammars? What are regular expressions? Why are these formalisms important to the design of lexical analysers for high-level language compilers? [5 marks]

   (b) Suppose that a construct A is defined as follows:

   $$\begin{aligned} A &\rightarrow !B \\ B &\rightarrow 0C \mid 1C \\ C &\rightarrow 0C \mid 1C \mid ! \end{aligned}$$

   Express A as (i) a regular expression, (ii) a state transition diagram, and (iii) an English description of the sentences that it generates. [9 marks]

   (c) By first considering a formal approach for specifying the syntax of an email address, outline a possible design for a software tool to search text files for character strings that could represent email addresses. [6 marks]

2. (a) What is a shift/reduce parser? What is a precedence grammar? What is bottom-up parsing? [6 marks]

   (b) Consider the precedence grammar:

   $S \rightarrow S(E) \mid E$
   $E \rightarrow T$
   $T \rightarrow T + x \mid x$

   This grammar has the precedence matrix:

	S	E	T	(	)	+	x
S				$\doteq$			
E				$\gtrdot$	$\doteq$		
T				$\gtrdot$	$\gtrdot$	$\doteq$	
(		$\doteq$	$\lessdot$				$\lessdot$
)				$\gtrdot$			
+							$\doteq$
x				$\gtrdot$	$\gtrdot$	$\gtrdot$	

   Show how a shift/reduce parser can be controlled by the data in this precedence matrix when parsing the sentence x(x+x). Show also the parse tree that would be generated. [8 marks]

   (c) Suppose you had to write a parser for Java. Explain why an approach based on precedence is unlikely to succeed. Which other approach would you use, and why? [6 marks]

3. (a) Outline advantages and disadvantages of programming in a high-level language rather than in a low-level language. [6 marks]

   (b) One criticism sometimes made of the use of high-level languages for time- or space-critical applications is that their compilers can't produce sufficiently fast or compact code – instead, the coding should be done directly in assembly language. To what extent do you believe that this is true? [8 marks]

   (c) Why is Java often implemented via an interpreted intermediate representation? Discuss whether this means that Java is always an inappropriate language for time-critical applications? [6 marks]

15.6 Others

1. Although the syntax rules of Java have context-dependent aspects (e.g. names have to be declared), a Chomsky type 1 grammar is not used to define the syntax of Java. Why is this?

2. Devise a grammar in which addition takes precedence over multiplication. Devise a grammar in which neither addition nor multiplication have precedence but in which they are performed (a) in the order encountered (left-to-right) and (b) right-to-left.

3. Produce a grammar for the definition of an identifier consisting of one to four letter or digits, where the first character must be a letter.

4. What is a finite-state grammar and what is its relevance to lexical analysis?

   Define a DFA describing the syntax of a reasonable representation for floating point numbers in a programming language. Show how it can be implemented in software. How can this parser be modified to compute the value of the floating point number it is parsing?

5. Develop the procedures for predictive (recursive descent parsing for the following grammar:

   $S \rightarrow aAB \mid b$
   $A \rightarrow a \mid bBA$
   $B \rightarrow c \mid bAB$

   What features of a grammar make it difficult or impossible to construct a predictive parser for the grammar?

6. What are LL(k) and LR(k) grammars? Explain why an LL(1) grammar is not ambiguous.

7. Can a predictive (recursive descent) parser be used in a lexical analyser, recognising a Chomsky type 3 grammar? Would there be any disadvantages in implementing a lexical analyser in this way?

8. State what is meant by the following terms: *terminal vocabulary, non-terminal vocabulary, production, context-free grammar, sentence, sentential form, operator grammar, precedence grammar, operator precedence grammar.*

   Determine whether the following grammar is an operator precedence grammar:

   $$\begin{aligned} S &\rightarrow A \\ A &\rightarrow A * B \mid B \\ B &\rightarrow B + C \mid C \\ C &\rightarrow (A) \mid x \end{aligned}$$

9. Derive the rules for filling in the precedence matrix (using the left and right sets etc.) from the definitions of the precedence relations.

10. Describe how variables may be accessed in a block-structured language using a display. Explain how the display may be set up and restored on procedure call and return using (a) a static environment chain on the stack, and (b) complete copies of the display in the stack at each level. Compare the relative merits of these two methods for the implementation of a block-structured language on a machine with which you are familiar.

11. A C compiler is to be implemented on a machine with eight general-purpose registers. These registers may be used as "accumulators" for arithmetic operations, as index registers to access main memory, and so on. Suggest for what purposes these registers could be used in the code generated by the compiler. Discuss the advantages and disadvantages of having a rigid predefined allocation strategy. Describe a suitable storage allocation scheme (based on a stack) for this C system, showing how both local and global variables may be accessed at run-time. What information has to be stored on the stack during a procedure call?

12. Outline an algorithm to compile assembly code from a suitable internal representation (such as a tree) of an arithmetic expression composed of integer variables and the basic operations `+`, `-`, `*` and `/`. The algorithm should attempt to minimise the number of locations used for anonymous results. Give the code sequence that your algorithm would generate for the expression `(a+b*c)/(f*g-(d+e)/(h+k))`.

13. "It is one thing to write a compiler that translates syntactically correct programs in a high-level language into a low-level target machine code; it is quite another matter to produce a workable system." Why?

14. List some software tools which should be included in a package for the development of software in a high-level language such as Java. Estimate the effort required to implement the various components of such a package.

15. Describe the overall structure of a typical compiler. Which part or parts would you prefer to write? Which part or parts would you least like to write? Why?

Des Watson
January 2009