An Introduction to Compilers

Part II

Suresh K. Basandra

It is worth noting that programming languages with the structure of ('standard') ALGOL, where each program is a single procedure, do not lend themselves to one-pass compiling, since forward jumps, for example, may traverse the entire length of the program. In contrast, languages like PL/1 or FORTRAN lend themselves to a programming style in which large programs can be created as a sequence of relatively small procedures or subroutines. In these cases, most backpatching can be done on a procedure together (thus providing a hidden extra pass not thought of as part of the compiler). In fact, for this reason, among others, most ALGOL implementations add to the 'standard' by allowing programs to consist of a sequence of procedures which may be linked.

In this article, it is not intended to consider how much processing should be done in one pass or how big a given pass should be. The answer to this question is dependent on the particular environment of a given compiler. Rather, it is intended to study each phase of the compilation process shown in Fig. 3 (Part 1 of the article) as a process in itself, investigating algorithms and tradeoffs that are applicable to the phase alone. It should be kept in mind, however, that in any real compiler, all phases must act in concert, and that a strategy adopted for one phase can affect the type of processing that must be done in a subsequent phase.

Lexical analysis

The performance of lexical analysis on source program text is perhaps the most well understood and easiest part of the compilation process. As a large part of the total compilation time is spent in performing lexical analysis, an attempt must be made to simplify it as far as possible.

The lexical analyser is the interface between the source program and the compiler. The lexical analyser reads the source program one character at a time, carving the source program into a sequence of atomic units called 'tokens'. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. For example, in the FORTRAN statement

IF (5. EQ. MAX) GO TO 100 (1)

we find the following eight tokens: IF; (;5;.EQ.;MAX;); GOTO; 100.

What is called a token depends on the language at hand and, to some extent, on the discretion of the compiler designer. But in general each token is a substring of the source program that is to be treated as a single unit. For example, it is not reasonable to treat M or MA (of the identifier MAX above) as an independent entity.

There are two kinds of tokens: specific strings such as IF or a semicolon; and classes of strings such as identifiers, constants, or labels. To handle both cases we shall treat a token as a pair consisting of two parts: a token type and a token value.

For convenience, a token consisting of a specific string such as a semicolon will be treated as having a type (the string itself) but no value. A token such as the identifier MAX above has a type 'identifier' and a value consisting of the string MAX. Frequently, we shall refer to the type or value as the token itself. Thus, when we talk about identifier being a token, we are referring to a token type. And when we talk about MAX being a token, we are referring to a token whose value is MAX.

The lexical analyser and the following phase, the syntax

analyser, are often grouped together into the same pass. In that pass, the lexical analyser operates either under the control of the parser or, as a coroutine with the parser. The parser asks the lexical analyser for the next token, whenever the parser needs one. The lexical analyser returns to the parser a code for the token that it finds. In the case that the token is an identifier or another token with a value, the value is also passed to the parser.

The usual method of providing this information is for the lexical analyser to call a book-keeping routine which installs the actual value in the symbol table if it is not already there. The lexical analyser then passes the two components of the token to the parser. The first is a code for the token type (identifier), and the second is the value—a pointer to the place in the symbol table reserved for the specific value found.

Finding tokens

To find the next token, the lexical analyser examines successive characters in the source program, starting from the first character not yet grouped into a token. The lexical analyser may be required to search many characters beyond the next token in order to determine what the next token actually is.

Syntax analysis

We now turn to the most important part of the analysis phase. The problem on hand is, given a sequence of symbols, to find the syntactic structure that binds all of them to form the source program. When this structure is discovered, we can consider the problem of semantic transformation of the source program into an internal form.

The first thing to be done is to find a 'derivation' for the given sentence. An equivalent task is the construction of the unique 'syntax tree' for the sentence. Another synonymous term for this work is 'parsing'. This is the job for the syntax analyser.

A natural way of processing the input sequence of symbols is from left-to-right. The only question which remains open is whether the parsing should be from the symbols in the sentence to the distinguished symbol, i.e. bottom-up or whether it should be from the distinguished symbol to the sentence.

Both methods are used in syntax analysis. Each of the methods is named accordingly: the one which constructs syntax trees upwards from the input symbols to the distinguished symbol is called 'bottom-up' parsing, and the one which constructs syntax trees downwards from the distinguished symbol to the symbols in the sentence is called 'top-down' parsing.

Top-down approaches to problem-solving and, especially, top-down presentations of solutions to problems are much easier to understand. Hence top-down parsing strategies will only be considered.

The parser has two functions: First, it checks that the token appearing in its input, which is the output of the lexical analyser, occurs in patterns that are permitted by the specification for the source language. Secondly, it imposes on the token a tree-like structure that is used by the subsequent phases of the compiler.

For example, if a PL/1 program contains the expression

A + / B

then after lexical analysis this expression might appear to the syntax analyser as the token sequence

id + / id

On seeing the /, the syntax analyser should detect an error situation, because the presence of these two adjacent binary operators violates the formulation rules of a PL/1 expression.

The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped together. For example, the expression

A / B * C

has two possible interpretations:

(a) divide A by B and then multiply by C (as in FORTRAN); or

(b) multiply B by C and then use the result to divide A (as in APL).

Each of these two interpretations can be represented in terms of parse tree, a diagram which exhibits the syntactic structure of the expression. Parse trees that reflect orders (a) and (b) are shown in Figs 4(a) and 4(b), respectively. Note how in each case the operands of the first operation to be performed meet each other at a lower level than that at which they meet the remaining operand.

The language specification must tell us which of the interpretations (a) and (b) is to be used and, in general, what hierarchical structure each source program has. These rules form the syntactic specification of a programming language. It can be mentioned that context-free grammars are particularly helpful in specifying the syntactic structure of a language. Moreover, efficient syntactic analysers can be constructed automatically from certain types of context-free grammars.

Intermediate code generation

On a logical level the output of the syntax analyser is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate-language representation of the source program.

Three-address code

One popular type of intermediate language is what is called 'three-address code'. A typical three-address code statement is

A : = B op C

where A, B and C are operands and op is a binary operator.

The parse tree in Fig. 4(a) might be converted into the three-address code sequence:

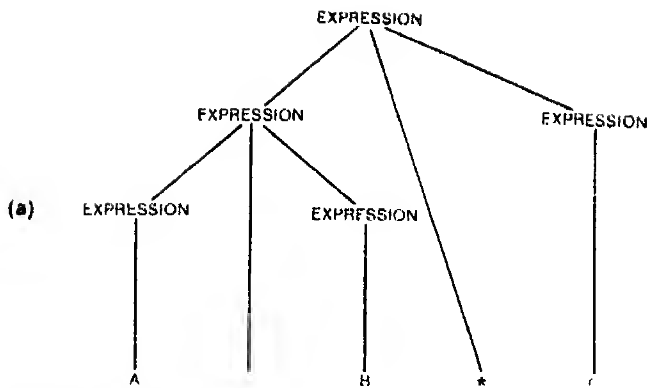

Fig. 4(a): A parse tree.

T1: = A / B
T2: = T1 * C

where T1 and T2 are names of temporary variables.

In addition to statements that use arithmetic operators, an intermediate language needs unconditional and simple conditional branching statements, in which at most one relation is tested to determine whether or not a branch is to be made. Higher-level flow of control statements such as while-do statements, or if-then-else statements are translated into these lower-level conditional three-address statements.

Optimisation

Object programs that are frequently executed should be fast and small. Certain compilers have within them a phase that tries to apply transformations to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object-language program can ultimately be produced. This phase is popularly called the 'optimisation phase'.

The term 'optimisation' in this context is a complete misnomer, since there is no algorithmic way of producing a target language program that is the best possible under any reasonable definition of 'best'. Optimising compilers merely attempt to produce a better target program than would be produced with no optimisation. A good optimising compiler can improve the target program by perhaps a factor of two in overall speed, in comparison with a compiler that generates code carefully but without using the specialised techniques generally referred to as code optimisation.

Local optimisation

There are 'local' transformations that can be applied to a program to attempt an improvement. For example, we can have instances of jumps over jumps in the intermediate code, such as

if A > B goto L2
goto L3
L2: (2)

This sequence could be replaced by the single statement

if A < B goto L3 (3)

Sequence (2) would typically be replaced in the object program by machine statements which:

(a) compare A and B to set the condition codes,
(b) jump to L2 if the code for > is set, and
(c) jump to L3.

Sequence (3), on the other hand, would be translated to machine instructions which:

(d) compare A and B to set the condition codes, and
(e) jump to L3 if the code for < or = is set.

If we assume A > B is true half the time, then for (2) we execute (a) and (b) all the time and (c) half the time, for an average of 2.5 instructions. For (3) we always execute two instructions, a 20% savings. Also (3) provides a 33% space saving if we crudely assume that all instructions require the same space.

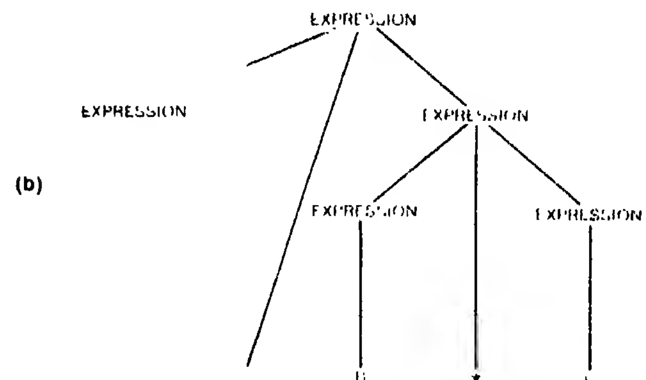

Fig. 4(b): Another parse tree.

Another important local optimisation is the elimination of common sub-expressions. Provided A is not an alias for B or C, the assignment

A : = B + C + D
E : = B + C + F

might be evaluated as

T1 : = B + C
A : = T1 + D
E : = T1 + F

taking advantage of the common sub-expression B+C. Common sub-expressions written explicitly by the programmer are relatively rare, however. A more productive source of common sub-expressions arises from computations generated by the compiler itself. Chief among these is subscript calculation. For example, the assignment

A [I] : = B [I] + C [I]

will, if the machine memory is addressed by bytes and there are, say, four bytes per word, require (4 * I) to be computed three times.

An optimising compiler can modify the intermediate program so that the calculation of (4 * I) is done only once. Note that it is impossible for the programmer to specify that this calculation of (4 * I) be done only once in the source program, since these address calculations are not explicit at the source level.

Loop optimisation

Another important source of optimisation concerns speed-ups of loops. Loops are especially good targets for optimisation because programs spend most of their time in inner loops. A typical loop improvement is to move a computation that provides the same result each time around the loop to a point in the program just before the loop is entered, rather than once for each interaction of the loop. Such a computation is called 'loop invariant'.

Various kinds of optimisation can be performed, and a list of a few of them is given below. Optimisation resulting in reduced execution time for the resultant program are:

(a) compile-time evaluation of expressions involving only constant operands; this optimising transformation is called 'folding';

(b) eliminating common sub-expressions,

(c) Boolean expression optimisation;

(d) moving invariant computations outside loops (from within);

(e) strength reduction of operators within loops;

(f) initialising values of variables at compile time;

(g) replacing a call to a produce by its body; and

(h) unrolling a loop; etc.

While all the above optimisations save execution time, it is worth noting that (a), (b) and (f) also save space, viz, they result in shorter programs. Optimisations (c), (d) and (e) result in faster programs by either not computing unrequired or unchanging values repeatedly, or by using simpler and hence faster operations. However, they do not contribute to any saving in space at all. Optimisations (g) and (h), result in faster programs but, clearly, at the expense of using more space to store longer programs.

Optimisation aimed at primarily reducing the amount of storage used are not many. Usually, they involve the problem of reducing the use of temporary storage in some form or the other, e.g. rearranging the operations in an arithmetic-expression to minimise the number of temporary locations used.

Code generation

Code generation is one of the least formalised subjects in compiler construction. While some cohesion seems evident in the trends in the evolution of higher-level programming languages, the same cannot be said of machine architecture. As a result, each machine seems to require a separate and exhaustive case analysis before code generation can be attempted for it.

The task set for the code generation phase of a compiler is, normally, to take as input a given internal form representation of the source program and to produce as output an equivalent sequence of instructions in the language of the object machine. In the case of one-pass load-and-go compiler, the luxury of an internal form representation of source programs cannot be afforded. The code generation phase of such compilers is therefore more complex compared to those of multi-pass compilers and they interface directly with the source program semantic-analysis actions.

It is undoubtedly appreciated that there is a great gap in the levels of detail of specification of the same algorithm in source and object languages. Object languages make explicit mention of locations used for temporary storage of values whereas corresponding names do not exist in the source language. A programmer programming in an object language is expected to be aware of internal registers of the machine, their function and the saving in time obtained from their use. Once again, the notion of such 'word area' is nearly non-existent as far as programming in a source language is concerned.

It is not intended to exhaustively list the differences between source and object languages. The point made here is that a burden of the code generator of a compiler is to make a note of those resources in the object language that are not visible in the source language. Next, the code generator should preserve and update a model of these resources and their status as it produces code (instructions in the object language that make use of these (and other) resources of the object machine). The object of this exercise is to make an efficient use of the resources of the object machine.

The code generation phase converts the intermediate code into a sequence of machine instructions. A simple-minded code generator might map the statement

```
A := B + C
```

into the machine code sequence

```
LOAD B
ADD C
STORE A
```

However, such a straightforward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores, and thus utilises the resources of the target machine inefficiently. To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of registers. Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary.

Many computers have only a few high-speed registers in which computations can be performed particularly quickly. A good code generator would, therefore, attempt to utilise these registers as efficiently as possible. This aspect of code generation, called 'register allocation', is particularly difficult to do optimally, but some heuristic approaches can give reasonably good results.

Book-keeping

A compiler needs to collect information about all the data objects that appear in the source program. For example, a compiler needs to know whether a variable represents an integer or a real number, what size an array has, how many arguments a function expects, and so forth.

The information about data objects may be explicit, as in declarations, or implicit, as in the first letter of an identifier or in the context in which an identifier is used. For example, in FORTRAN, A(I) is a function call if A has not been declared to be an array.

The information about data objects is collected by the early phases of the compiler lexical and syntactic analysis and entered into the symbol table. For example, when a lexical analyser sees an identifier, MAX, say, it may enter the name MAX into the symbol table if it is not already there, and produce as output a token whose value component is an index to this entry of the symbol table. If the syntax analyser recognises a declaration 'integer' MAX, the action of the syntax analyser will be to note in the symbol table that MAX has type 'integer'. No intermediate code is generated for this statement.

The information collected about the data objects has a number of uses. For example, if we have the expression A + B, where A is of type integer and B of type real, and if the language permits an integer to be added to a real, then on most computers code must be generated to convert A from type integer to type real before the addition can take place. The addition must be done in floating point, and the result is real. If mixed-mode expressions of this nature are forbidden by the language, then the compiler must issue an error message when it attempts to generate code for this construct.

The term 'semantic analysis' is applied to the determination of the type of intermediate results, the check that arguments are of types that are legal for an application of an operator, and the determination of the operation denoted by the operator. (For example, '+' could denote fixed or floating add, perhaps logical 'or' and possibly other operations as well.) Semantic analysis can be done during the syntax analysis phase, the intermediate code generation phase, or the final code generation phase.

Error handling

One of the most important functions of a computer is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by virtually all the phases of a compiler. For example,

1. The lexical analyser may be unable to proceed because the next token in the source program is mis-spelled.

2. The syntax analyser may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred.

3. The intermediate code generator may detect an operator whose operands have incompatible types.

4. The code optimiser, doing control flow analysis, may detect that certain statements can never be reached.

5. The code generator may find a compiler-created constant that is too large to fit in a word of the target machine.

6. While entering information into the symbol table, the book-keeping routine may discover an identifier that has been multiply declared with contradictory attributes.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors.

Good error handling is difficult because certain errors can mask subsequent errors. Other errors, if not properly handled, can spawn an avalanche of spurious errors.

Factors affecting compiler design and implementation

Several aspects have to be considered in the implementation of a programming language. Amongst them, the chief ones are syntax, semantics and pragmatic issues (like the relationship between the language and the users). For instance, whether symbol tables should be printed out and how errors in programs are to be reported are pragmatic issues which in no way affect the correctness of a compiler.

Additionally, there are questions, such as, should the compiler be very fast and not bother too much about the kind of code it generates, or should it take its time in producing the object program taking care that the result is an optimised object program with substantially reduced storage and run-time requirements? All such factors play an important part in the design of a compiler.

Other major factors affecting the implementation strategy are the language in which the compiler is written, the structure of the group that writes it and the resources available, i.e. the size of the implementation machine, the time for the completion of the project, etc.

Compiler-writing tools

A number of tools have been developed to help construct compilers. These tools range from scanner and parser generators to complex systems, variously called 'compiler-compilers' 'compiler-generators' or 'translator-writing systems' which produce a compiler from some form of specification of a source language and target machine.

The input specification for these systems may contain:

(a) a description of the lexical and syntactic structure of the source language;

(b) a description of what output is to be generated for each source language construct; and

(c) a description of the target machine.

In many cases the specification is merely a collection of programs fitted together into a framework by the compiler-compiler. Some compiler-compilers, however, permit a portion of the specification of a language to be non-procedural rather than procedural. For example, instead of writing a program to perform syntax analysis, the user writes a context-free grammar and the compiler-compiler automatically converts that grammar into a program for syntax analysis.

While a number of useful compiler-compilers exist, they have limitations. The chief problem is that there is a trade-off between how much work the compiler-compiler can do automatically for its user and how flexible the system can be.

For example, it is tempting to assume that lexical analysers for all languages are really the same, except for the particular keywords and signs recognised. Many compiler-compilers do in fact produce fixed lexical analysis routines for use in the generated compiler. These routines differ only in the list of keywords recognised, and this list is supplied by the user. This approach is quite valid, but may be unworkable if it is required to recognise non-standard tokens such as identifiers that may include characters other than letters and digits.

More general approaches to the automatic generation of lexical analysers exist but these require the user to supply more input to the compiler-compiler, i.e. to do more work.

The principal aids provided by existing compiler-compilers are:

1. **Scanner generators.** The 'built-in' approach described above and regular expression based techniques are the most common approaches.

2. **Parser generators.** Almost every compiler-compiler provides one. The reason is twofold. First, while parsing represents only a small part of compiler construction, with a fixed framework in which parsing is done can be a great aid in the organisation of the entire compiler. Secondly, the parsing phase is unique among the compiler phases in that a notation known as the context-free grammar exists which is sufficiently non-procedural to reduce the work of the compiler writer significantly, sufficiently general to be of use in any compiler, and sufficiently developed to permit efficient implementations to be generated automatically. One significant advantage of using a parser generator is increased reliability. A mechanically generated parser is more likely to be correct than one produced by hand.

3. **Facilities for code generation.** Often a high-level language especially suitable for specifying the generation of intermediate, assembly or object code is provided by the compiler-compiler. The user writes routines in this language and, in the resulting compiler, the routines are called at the correct times by the automatically generated parser. A common feature of compiler-compilers is a mechanism for specifying decision tables that select the object code. These tables become part of the generated compiler, along with an interpreter for these tables supplied by the compiler-compiler.

A compiler is characterised by three languages: its source language, its object language, and the language in which it is written. These languages may be quite different. For example, a compiler may run on one machine and produce object code for another machine. Such a compiler is often called a 'cross-compiler'. Many minicomputer and microprocessor compilers are implemented this way; they run on a bigger machine and produce object code for the smaller machine. □