An Introduction to Compilers

Part I

Suresh K. Basandra

The subject matter of compiler construction can be considered as a form of language translation. The phrase 'language translation' suggests the existence of several languages and, also, the notion of translation. By translation we mean a mapping of sentences of a given language to sentences of another given language.

A necessary aspect of translation is that the mapping preserves the meaning of the sentence to be translated. Since this is a very broad constraint, it should be evident that, in general, translation is a one-to-many mapping.

As we are interested in solving problems by using computers, the act of translation can be imperatively expressed as: Take an arbitrary sentence of some given language, analyse it and, if possible, synthesise a sentence of another given language so that it completely conveys the meaning of the initial sentence.

To realise this, however, several major problems have to be carefully considered: What is a language? What do we mean by an analysis of statements in a language? How do we ascribe a meaning to a sentence in any language? In any given language, how do we synthesise sentences which will have a given meaning? If many sentences can be ascribed the same meaning, how do we choose amongst them?

The most common languages we know of are the natural languages such as English. Our understanding of such languages is rather loose and cannot be succinctly expressed. This is not at all surprising, for such languages are usually very vast. Consequently, the problem of translation between such languages has mammoth proportions. Fortunately, for the present, our interests lie in a very specific set of languages, the so-called 'programming languages'.

Programming languages provide a precise and unambiguous means for communication between man and machine. Amongst such languages, we distinguish between those which are more suited for problem-solving by man from those which are suited for direct interpretation by a machine. In this article, we will be concerned with the translators (agents which perform translation) which translate sentences from languages in the former class to sentences of languages in the latter class. Compilers are such translators.

Compilers and translators

A 'translator' is a program that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language). If the source language is a high-level language such as FORTRAN, PL/1 or COBOL, and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a 'compiler'.

Programs written in source languages are called 'source programs'. On the other hand, object languages are usually suited for machine interpretation. Programs expressed in object languages are called 'object programs'. To translate a source program, the compiler must analyse it thoroughly and synthesise an equivalent object program.

The task performed by a compiler is called 'compilation'. During compilation, the process of source program analysis yields a variety of information about source programs. This information has to be preserved by the compiler for it to produce an equivalent object program. Various data structures, such as tables, lists, trees, etc are employed by the compiler to preserve this information. The construction of an equivalent object program is directed by the information

Mr Basandra, an M. Tech in Computer Technology, unravels the system of 'compilers' and various other terms like 'translators' and 'interpreters' that the readers may have heard and wondered as to their actual meaning.

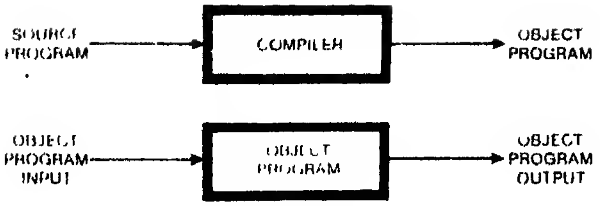

Fig. 1: Compilation and execution.

preserved in these data structures. All the actions performed by a compiler are said to occur at 'compile-time'.

Executing a program written in a high-level programming language is basically a two-step process, as illustrated in Fig. 1. The source program must first be compiled, i.e. translated into the object program. Then the resulting object program is loaded into memory and executed.

Other translators

Certain translators transform a programming language into a simplified language, called 'intermediate code', which can be directly executed using a program called an 'interpreter'. We may think of the intermediate code as the machine language of an abstract computer designed to execute the source code. For example, SNOBOL is often interpreted, the intermediate code being a language called 'polish postfix notation'.

In some cases, the source language itself can be the intermediate language. For example, most 'command languages', such as JCL, in which one communicates directly with the operating system, are interpreted with no prior translation at all.

Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs. However, the main disadvantage of interpreters is that the execution time of an interpreted program is usually more than that of a corresponding compiled object program.

There are several other important types of translators, besides compilers. If the source language is assembly language and the target language is machine language, then the translator is called an 'assembler'.

The term 'preprocessor' is sometimes used for translators that take programs in a high-level language into equivalent programs in another high-level language. For example, there are many FORTRAN preprocessors that map 'structured' versions of FORTRAN into conventional FORTRAN.

Programming languages

Here it is intended to define a programming language formally. This has to be done because, intuitively speaking, a compiler specifies a relationship between source programs and object programs. It does this for all source programs, and hence compilation is really a relationship between two languages. In other words, a compiler is not concerned with any specific set of source programs but all programs that can be written in the source language.

A programming language is a notation with which people can communicate algorithms to computers and to one another.

Syntax

A program in any language can be viewed as a string of characters chosen from some set, or 'alphabet', of characters. But how do we prescribe which strings of characters represent valid programs? The rules that tell us whether a string is a valid program or not are called the 'syntax' of the language.

It is often almost impossible to state concisely and precisely what strings are valid programs, just as it is hard to state which sentences of English are proper and which are not.

Semantics

Once we know that we have a valid program, how do we specify what the program does? It is essential to know what a program means if we are to compile it faithfully into a machine language program that does what the programmer expects. The rules that give meaning to programs are called the 'semantics' of the programming language.

The semantics of a programming language are much harder to specify than its syntax. No completely satisfactory means for specifying semantics in a way that helps construct a correct compiler for the language has been found.

The hierarchical structure of programming languages

A programming language is a notation for specifying a sequence of operations to be carried out on data objects. Both the data objects and the operations can be grouped into a hierarchy that looks like the tree of Fig. 2. Not all languages have every one of these features, and some languages such as ALGOL 68, permit statements to be in expressions. Nevertheless, the units in this hierarchy are so common that they should be familiar to all.

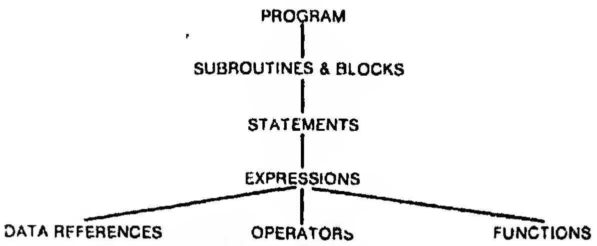

Fig. 2: Hierarchy of program elements.

At the top of the hierarchy is the program itself. The program is the basic execution unit. Next comes an entity that can be compiled but not necessarily executed—the subroutine or block. These are units which may have their own data (local names).

Subroutines differ from blocks by being callable from other portions of a program. Both subroutines and blocks are composed of statements. In turn, statements are fashioned from expressions which are made up of operators, function calls, and references to data.

The structure of a compiler

The compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of subprocesses called 'phases' as shown in Fig. 3. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

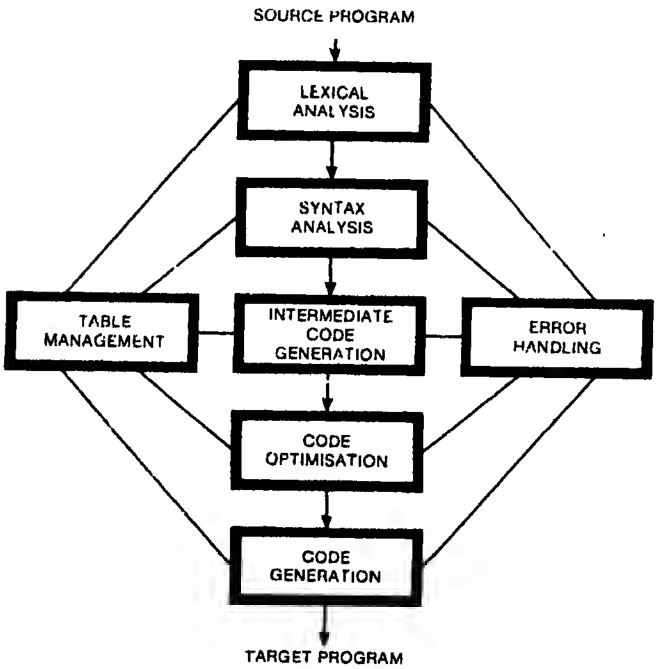

Fig. 3: Phases of a compiler.

As input to a compiler, the source program is only a string of characters. From this linear representation of the program, the process of source program analysis should detect the structure and meaning of the program.

This is very similar to the actions involved in finding constituent phrases in a sentence in English, in which case we do so by making use of the English grammar. In order to understand an English sentence, we must first know the meaning of various words used in it; grouping the words into phrases allows us to ascribe meanings to these phrases; and finally, combining the meanings of the phrases according to the grammatical structure of the sentence leads to an understanding of the sentence.

From the above analogy, it is reasonable to assume that the source language has a grammar which defines acceptable grammatical structures for source programs. In order to analyse the structure of a source program, we have to formulate a strategy for recognising its constituent sub-structures. Also, we have to detect the methods by means of which these sub-structures are bound together to form a whole source program.

Looking at the analogy of grammatical analysis of English sentences again, we note that we recognise words first and then we search for the phrases. The first phase consists of the analysis of the string of characters in the source program so as to form meaningful primitives (analogous to words and punctuation marks in an English sentence).

The first phase, called the 'lexical analyser' or 'scanner' separates characters of the source language into groups that logically belong together; these groups are called 'lexemes' or 'tokens'. The usual tokens are keywords such as D0 or IF, identifiers (the equivalent of 'names' in programming languages) such as X or NUM, operator symbols such as ':-', ' ', '=' or '+', and punctuation symbols such as parentheses or commas.

The output of the lexical analyser is a stream of tokens which is passed to the next phase. The tokens in this stream can be represented by codes which we may regard as integers. Thus, D0 might be represented by 1, + by 2, and 'identifier' by 3. In the case of a token like identifier, a second quantity that indicates which of those identifiers used by the program is represented by this instance of token identifier is passed along with the integer code for identifier.

The task of this first phase is quite simple and straightforward. Subsequent to this phase of analysis, a source program may be viewed as a sequence of tokens.

Following lexical analysis is the task of recognising grammatical phrases in a source program. This is a more complicated task: sequences of lexemes have to be grouped together to form simple phrases of the source language; these simple phrases are used to form more complex ones and, ultimately, source programs.

'All these actions are the substance of 'syntax analysis', the term usually ascribed to this second phase in the analysis of source programs. The syntax analyser groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a syntactic structure called an 'expression'. Expressions might further be combined to form statements.

Often, the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

The 'intermediate code generator' uses the structure produced by the syntax analyser to create a stream of simple instructions. Many styles of intermediate codes are possible. One common style uses instructions with one operator and a small number of operands. These instructions can be viewed as simple macros. The primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

'Code optimisation' is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space.

The final phase, known as 'code generation', produces the object code by deciding on the memory locations for data, selecting codes to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically.

The 'table-management' or 'book-keeping' portion of the compiler keeps track of the names used by the program and records essential information about each, such as its type (integer, real etc). The data structure used to record this information is called a 'symbol table'.

The 'error handler' is invoked when a flow in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed.

It is desirable that compilation be completed on flawed programs, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. Both the table-management and error handling routines interact with all phases of the compiler.

Passes

In an implementation of a compiler, portions of one or more phases are combined into a module called a 'pass' A pass reads the source program or the output of the previous pass, makes the transformation specified by its phases, and writes output into an intermediate file which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

The number of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine rather than by any mathematical optimality criterion. The structure of the source language has a strong effect on the number of passes.

Certain languages require at least two passes to generate code easily. For example, languages such as PL/I or ALGOL 68 allow the declaration of a name to occur after uses of that name Code for expressions containing such a name cannot be generated conveniently until the declaration has been seen.

The environment in which the compiler must operate can also affect the number of passes. A multi-pass compiler can be made to use less space than a single-pass compiler, since the space occupied by the compiler program for one pass can be reused by the following pass.

A multi-pass compiler is, of course, slower than a single-pass compiler, because each pass reads and writes an intermediate file. Thus, compilers running on computers with small memory would normally use several passes while, on a computer with a large random access memory, a compiler with fewer passes would be possible.

Reducing the number of passes

Since each phase is a transformation on a stream of data representing an intermediate form of the source program, it may be wondered how several phases can be combined into one pass without the reading and writing of intermediate files. In some cases one pass produces its output with little or no memory of prior inputs. Lexical analysis is typical. In this situation, a small buffer serves as the interface between passes. In other cases, we may merge phases into one pass by means of a technique known as 'backpatching'. In general terms, if the output of a phase cannot be determined without looking at the remainder of the phase's input, the phase can generate output with 'slots' which can be filled in later, after more of the input is read.

While it is not possible to deal with backpatching in detail, an example from assemblers will serve as a paradigm. An assembler might have a statement like

GOTO L

which precedes a statement with label L. A two-pass assembler uses its first pass to enter into its symbol table a list of all identifiers (statement labels and data names) together with the machine address (relative to the beginning of the program), to which these identifiers correspond. Then a second pass replaces mnemonic operation codes, such as GOTO by their machine language equivalent, and replaces uses of identifiers by their machine address.

A one-pass assembler, on the other hand, could generate a skeleton of the GOTO machine instruction the first time it saw GOTO L. It could then append the machine address for this instruction to a list of instructions to be backpatched once the machine address for L is determined. For example, when the assembler encounters a statement such as

L:ADD X

it scans the list of statements referring to L and places the machine address for statement L : ADD X in the address field of each such instruction. Subsequent assembly instructions referring to L can have the value for L substituted immediately.

In a compiler, most of the backpatching that needs to be done is done over relatively short distances. Labels, for example normally need to be backpatched as above only with one procedure or subroutine. The distance over which backpatching occurs is important since the code to be backpatched must remain accessible until backpatching is complete. Even though the object program may fit in memory when it is produced, intermediate forms of the source program may be too big to fit in memory all at once, especially as a substantial portion of memory may be occupied by the compiler program itself. **(To be continued)**