An appraisal of compiler technology

by ROBERT M. McCLURE
Consultant

INTRODUCTION

Although the last decade has seen the implementation of a very large number of compilers for every conceivable language, the literature on compiler writing is still unorganized. There are many papers on formal languages, syntactic analysis, graph theoretic register assignment, and similar topics to be sure. But, they are not very useful in helping the newcomer to the field decide how to design a compiler. Even the recent books in the field are more encyclopedic in nature than instructive. The best single book available is Gries,1 which has a good bibliography for further study.

The few open descriptions of compilers that do exist rarely are candid about the mistakes that were made. This is, after all, human nature. Moreover, the nature of scientific publishing is not conducive to papers of an evaluative or subjective nature. The principal reason, therefore, for writing this paper, is not to add to the basic body of scientific knowledge, but rather to provide a few value judgments on how compilers are being written and should be written. Since subjective statements should always be prefaced with "It is my opinion that", the indulgent reader will understand that this phrase should be applied to the entire paper.

There is an enormous amount of material to be studied in connection with compiler design. Most of it is very difficult to read. We refer to the listings of the compilers themselves and associated internal documentation. At present there is no alternative to obtaining a comprehensive knowledge of the field. Even if one is willing to put forward the effort, however, much of the material is difficult to obtain. The desire of computer manufacturers and software houses to protect their latest ideas interferes with a free flow of information. The result of this is the growth of an oral tradition for passing information much like the wandering troubadours of yore. Until this changes, there will be hardships worked on those who would like to learn the trade.

In order to reduce this paper to manageable size and to make generalizations more useful, let it be understood that we are mainly talking about so-called "production" compilers. By this we mean compilers for important languages that are intended for translating programs that are expected to be run extensively. Moreover, we restrict our consideration to compilers for medium and large scale general purpose computers. The subject of compilers for minicomputers and special purpose computers is deserving of considerable study on its own. Also research compilers, "quick and dirty" compilers, and pedagogic compilers will not be considered in this commentary. Finally, it is inevitable that someone's favorite technique will be given a short straw. For this we plead for tolerance.

SYNTACTIC ANALYSIS

In the area of syntactic analysis or parsing, the necessary solutions are clearly at hand. Parsers are now routinely constructed by graduate students as class projects. The literature on the subject is extensive, and new techniques are revealed regularly. Almost all compilers are written using only two of the many available methods however: recursive descent or precedence. Rarely are either used in pure form. Compilers that use both methods intermixed are common. Although both of these methods have their roots in the very earliest compilers, they remain the mainstays of the business.

Recursive descent

The basic idea in recursive descent is that there is a one-to-one correspondence between the syntactic units in the formal description of the language and routines for their recognition. Each routine examines the input text by either scanning for specified terminal tokens (usually produced by a separate lexical analyzer) or by calling other routines to recognize non-terminal syntactic units. Figure 1 is a flowchart of one such simple recognizer. Because most practical languages are defined recursively, the recognizers must themselves be
Recursive descent analyzers are usually written so that it is never necessary to "back up" the input stream. If an impasse is reached, an error indication is given, perhaps a correction attempted, and the scan resumed at some readily recognizable token, such as ";". The knowledge of the context at the time an error is discovered allows more meaningful diagnostics to be generated than with any other technique.

The use of recursive descent usually requires the rearrangement of the formal syntax, since left recursions must be removed. Although this is not difficult, it does somewhat spoil the clarity of the approach. For peculiar grammars, it is easier to use recursive descent than to transform the grammar into a form suitable for precedence analysis. Although the methodology is basically ad hoc, recursive descent is the most widely used method of syntactic analysis.

Recursive descent has one further advantage for a language with a large number of productions. Precedence methods seem to require table sizes proportional to the square of the number of syntactic units, whereas recursive descent recognizers are roughly proportional to the grammar size.

Precedence analysis

The idea that operators have varying degrees of "precedence" or "binding-power" is quite old, and has been used in some of the earliest compilers. The formalization of precedence relations between operators actually started with Floyd in 1963. Now it is more customary to define precedence relations between all syntactic units. While very few production compilers have used a formal precedence parser yet, the modern implementation of these techniques such as described by McKeeman, is clearly destined for wider use.

The fundamental attraction of precedence methods lies in the automatic construction of analyzers from the formal grammar for the language to be analyzed. Parsers built in this way can be quickly changed to reflect language changes simply by changing the tables without modifying the underlying interpretation algorithm. Furthermore, if the tables are constructed by formal methods, the language accepted is exactly that specified and no other. This considerably simplifies the construction of bug-free parsers.

There are some drawbacks to parsers built wholly around precedence methods. For example, many languages in everyday use, such as COBOL and FORTRAN can simply not be made into precedence languages. Even PL/I can only be fit into the correct mold with considerable difficulty. Moreover, the tables required can be quite large.

A major problem with precedence parsing methods is the problem of recovery from source program errors and the issuance of good diagnostics for such errors. Several approaches to solving this problem have been tried with only modest success. Various techniques for reducing the size of the tables required, such as the introduction of precedence functions, serve only to complicate this problem.

Nevertheless, there are several conditions that suggest strongly that a precedence parser of some variety should be tried. If, for example, a parser must be produced in the shortest possible time. Precedence parsers tend to be easy to debug. A further advantage is gained if the language is being designed at the same time as the compiler, since the language can be made into a precedence language. Finally, a precedence parser is often a most suitable alternative when recursive procedure calls required for recursive descent parsing are either impossible (as in FORTRAN) or expensive (as in PL/I).

Mixed strategies

A number of compilers use the technique of recursive descent for analysis of the larger syntactic units such as statements, and turn to precedence methods for parsing expressions. In this case, simple operator precedence analysis is customarily used. There is no conceptual or practical difficulty in accomplishing this since the expression recognizer requires relatively small tables which may easily be hand constructed. The number of calls on procedures is minimized (normally a high overhead item) and perhaps the best of both worlds is achieved.

The idea of mixing parsing techniques can be generalized. It appears (with no theoretical foundation as yet) that a good partition is achieved when the subgrammar to be parsed with a precedence parser results in a relatively dense precedence matrix, and the subgrammar to be parsed with a recursive descent parser would result in a relatively sparse (for legitimate symbol pairs) precedence matrix.
INTERNAL REPRESENTATION

Generation of code occurs simultaneously with syntactic analysis only in very small compilers and quick and dirty compilers. In most compilers the results of syntactic analysis are stored for later code generation. A number of ways have been described for the internal representation of the results of parsing, two of which have attained really wide usage: tree structures (most usually a form known as triples) and late operator Polish notation. Figure 2 illustrates several forms of the representation of a simple expression. Form (a) is a binary tree, called triples when recorded in the tabular form shown, since each line consists of an operator and two operands. The result of each operator is known implicitly by the line number of the triple. Form (b) is a tree form with variable sized nodes. Form (c) is the Polish form and is actually the string that results from traversing the terminal nodes of a parse tree in a prescribed order.

The simpler compilers prefer the Polish representation since subsequent processing usually amounts to simply a linear reading of the text. Fortunately the order in which action is to be taken by the code genera-

tor is very nearly that of tokens in the Polish string anyway. Since this is also the order of generating the Polish string, the only advantage gained from deferred generation is that obtained by a full reading of all of the text, including declarations, implicit as well as explicit. Although optimization can be implemented on internal representations in Polish form, tree forms are much easier to work with.

A further advantage of Polish strings is that they may be easily written on and read from sequential data files, are conceptually simple, and require a minimum of additional space for linkage information. If memory space is at a premium or the simplest representation is preferred, Polish strings are recommended.

Complete representation of the source program in tree form is now growing in popularity, and has appeared in quite a number of the more recent compilers. It is especially prevalent in compilers for major languages for the larger machine in which optimization is important. The ease with which the program can be searched and/or rearranged is the primary motivation for this selection. For building a generation system based upon formal semantics, an internal tree representation is a good choice. Not only does it appear that code generation can be formalized but also that optimization strategies can be formalized utilizing the idea of transformation sets defined over trees.

SYMBOL TABLES AND DICTIONARIES

Considerable effort has gone into devising symbol table routines that have all of the desirable properties of compactness of storage required, speed of access, and simplicity of construction. The result is that almost all compilers use some variant of the hash link method. In this method, the symbol to be looked up is first hashed into a reasonably small range such as 31 to 512. This hash index is then used to access the head of a list of symbols with the same hash value. This is shown schematically in Figure 3. This hash chain is frequently ordered, say alphabetically, to reduce further the look-up time.

The dictionary information associated with each symbol may either be included directly with the external representation or be contained in a separate table. Since in a multiple pass compiler, the external representation is not required after the lexical analysis is complete, the separation of symbol table and dictionary has become customary.

It is becoming increasingly important to make provision for symbol tables and dictionaries to grow beyond the bounds of the available main memory. This presents no unusual problems for hardware systems that

\[ A + B + C \ast D \]

[Figure 2—Internal representations]
The conceptual simplicity of having tables in consecutive memory locations is a major reason for adopting the idea of "floating" tables. The additional burden of referencing all entries indirectly through a current table base pointer is a small price to pay for this simplicity. First used in compilers by Digitek in a series of minicomputer compilers and by IBM in a COBOL compiler for the IBM 7090 in the early 1960's, the basic ideas have been widely adopted. Modern computer architecture that makes both base addressing and indexing simultaneously available makes implementation extremely simple.

Every table is allowed to grow (or shrink) independently. Before any table expands, a check is made to see if there is available space. If not, space is made available

\[
\begin{array}{c|c|c|c|c|c}
\text{BEFORE} & \text{AFTER} \\
\hline
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
5 & 5 \\
\end{array}
\]
by actually copying the data items in the tables to be moved into another location in memory. Although copying data in memory to reorganize storage seems inefficient at first, it turns out to be quite satisfactory in practice since it occurs rarely. Figure 4 shows how tables are rearranged to allow available space for all tables to grow.

The interesting questions about this form of allocation revolve around deciding how much of the available space to allocate to each table. Garwick suggests that space be allocated in proportion to how much each table has grown since the last allocation. CITRUS (the storage allocation package in the IBM COBOL compiler) required specification of the desired increment of space. Most common is dividing the remaining space proportional to the current size of each table (with some prescribed minimum). All methods will run out of space only when there is absolutely no more. This may not be desirable, though, since considerable time will be spent trying to squeeze in the last item before overflow occurs.

Movable tables seem to work best for relatively small memories, for machines in which internal speeds are fast relative to I-O speeds, or for systems in which all available memory is allocated to the compiler at the start of any compilation.

List structures are complicated by the necessity to use relative links rather than absolute addresses and to identify the table pointed to.

**Block allocation**

Block allocation methods are the second most popular storage management technique. In this case, tables are not kept in contiguous locations, and information is usually linked together. Each routine that may add data to a table is responsible for requesting a block of suitable size (sometimes a system fixed size) for the purpose at hand. Usually space is allocated in multiples of the basic element size of the table at hand in order to avoid calling the main allocator too often. Since OS/360 implements this form of allocation as a primitive in the operating system, this form of allocation has been quite widely used on that machine.

This technique does have the principal advantage that since table segments are not moved after allocation, absolute pointers may be used in list structures. It suffers from the drawback that storage fragmentation can occur and may prevent usage of all available memory.

Block allocation is suggested whenever memory is a resource to be shared with other simultaneous users of the machine in a multiprogramming environment. This is because it is desirable to minimize the amount of memory required at any given time, and most main memory allocation algorithms supported by operating systems do not guarantee that additional memory will be allocated contiguously with previously allocated memory.

**IMPLEMENTATION TECHNIQUES**

Early compilers were invariably written in assembly language; most still are today. It was originally felt that only assembly language could yield the efficiency that was required in such a widely used program as a compiler, and that the required operations were not well supported by higher level languages. Although it has now been generally recognized that very little of any compiler needs to be "tightly coded" to achieve nearly the same efficiency, the tradition of assembly coding is dying a slow and painful death.

A second reason usually given for writing a compiler in assembly language was to minimize the space that the compiler required for its code. Factors of 1.5 to 3 have been cited. With the growth of main memory sizes available, the almost universal availability of random access secondary storage, and the common use of dynamic loading techniques, the requirement for small code has been considerably reduced in importance.

Advocates of coding in "higher level" languages have not always been completely candid in their arguments either. It is frequently stated that one of the main motives for using a higher level language is the gain in readability that occurs. Anyone who has tried to read a compiler written in FORTRAN knows that this simply is not the case. A much stronger case may be made for PL/I or ALGOL. The fluidity of these languages plus the natural benefits of block structuring generally result in code substantially more readable than assembly code.

A major drawback to most higher level languages for coding compilers is that the native data types manipulated in these languages are neither those required in either lexical scanning, nor those required for table management. Both of these are vitally important in compiling, and result in the construction of subroutines, frequently in assembly code, to support them. The linkage overhead in using these routines can be substantial.

By all odds, the most compelling reason for using a higher level language is the conciseness with which code can be written. The sheer laboriousness of assembly code is a major obstacle to its use. One of the best measures of the suitability of a language for a particular purpose is the number of characters that must be written to achieve the desired result. Since error rates in programming are almost independent of the language being written, con-
cise programs will have fewer bugs than verbose pro-
grams.

Pops

One approach that has gained a number of particu-
larly ardent adherents is that of writing compilers in an
interpretive language. This idea seems also to have
originated in the early 1960's. Although the COBOL
compiler for the IBM 1410 was written in interpreted
code, the main source of the popularity was the series of
(mostly FORTRAN) compilers written by Digitek. A
number of syntax directed compilers of the same vintage
utilized an internal representation of a similar nature.
The increased suitability of current computer instruc-
tion sets for compiler writing has caused the technique
to largely fall from favor for large machine compilers in
recent years. The technique has much to recommend it
for some applications, though, and it is worthy of a few
comments.

Since there have been no published papers on the
Digitek POP system, which appears to be the most
highly developed system, we will include here a some-
what more complete description than for the other ideas
discussed in this paper.

The name POP derives from the Programmed OPera-
tors on the SDS 910 for which the first Digitek FOR-
TRAN compiler was written. These were a series of un-
assigned operation codes that caused a trap to a loca-
tion distinct for each such op-code. This enabled the
user to define the meaning of each of these op-codes by
a subroutine. Subsequently, for other machines which
did not have such a feature, a completely interpretive
system was substituted.

The POP system consists of a set of operations that
resemble single address instructions. The single operand
in each POP is either a single data element (usually an
integer, but perhaps a pointer into a table, etc.), a charac-
ter, a string of characters (this is handled indirectly), a
table (movable), or a flag. Additional data types are
defined as needed. Tables are implemented as contigu-
ous, movable tables as previously described. A pointer
is defined as having a table number part plus an offset
into a table. Pointers are used for forming linked struc-
tures and for referencing data in tables if other than the
last item is to be referenced. Tables are normally used
like stacks. One table is distinguished as the main stack
and is often used as an implicit second operand. Another
table is distinguished as the stack used to save return
addresses in procedure calls. Recursion is therefore quite
natural.

To illustrate how this system is used, we will define
several of the more common POP's. The first POP is
LDS (Load Stack), and is written as:

\[
\text{LDS} \quad \text{A} \\
\text{LDS} \quad \text{B}
\]

This sequence of two POP's is interpreted as follows:
First the data item A, a single work item, perhaps an
integer, is placed on the main stack, extending it by one
word in the process. Then the item B is added to the
stack, extending it once again. At the conclusion, the
stack appears as in Figure 5. The end of stack item
is referred to as item 0 and the next item (A) is referred to
as item 1.

The POP STS (Store Stack) is simply the converse.
For instance:

\[
\text{STS} \quad \text{A} \\
\text{STS} \quad \text{B}
\]

stores the end item of the stack in cell A, and shortens
the stack. The second POP stores the new end item in
cell B and shortens it once again. The net effect of the
four POP's we executed is to exchange cells A and B.

Similarly, the stack may be loaded with the end item
on any table by MOF (Move Off) which has as an
operand a table number. The effect of this is to lengthen
the stack and to shorten the table specified as the
operand of the instruction by moving one word of data.
This operation is usually defined to set a global flag to
FALSE rather than to move an item if the table speci-
fied is empty. The companion operation is MON (Move
On). Hence to move the contents of one table into
another, the four instruction loop suffices:

\[
\text{LA} \quad \text{MOF} \quad \text{TABLE1} \\
\text{BRT} \quad \text{ALLDONE} \\
\text{MON} \quad \text{TABLE2} \\
\text{BRU} \quad \text{LA}
\]

Figure 5—Picture of end of stack
The two new instructions above are BRT (Branch True) and BRU (Branch Unconditional).

Character scanning is done with primitives of the form

\[ \text{CHS} \quad \text{character} \]

The POP CHS (Character Scan) has as its operand the internal representation of a character. If the specified character is the next in the character input stream, the input scanning pointer is advanced and a global flag set to TRUE. Otherwise the global flag is set to FALSE. Similarly a complete string may be matched with SCN (String Scan), as in

\[ \text{SCN} \quad \text{"BEGIN"} \]

Subroutine calling is done with BRS (Branch and Save) which stores the return address in a stack. The natural implementation of recursion leads most POP written compilers to be of the recursive descent variety. For instance the sequence of code required to recognize the syntax \( \text{SUM} ::= \text{TERM} \{+\text{TERM}\}^* \) is as simple as

\[ \text{SUM} \quad \text{BRS} \quad \text{TERM} \quad \text{CHS} \quad \text{"+"} \quad \text{BRT} \quad \text{SUM} \quad \text{RET} \]

The POP system is fleshed out with instructions for packing and unpacking data words, creating instructions, setting and testing flags, and so on almost ad infinitum. In theory, a POP compiler can be moved from one machine to another simply by writing a new interpreter for the next machine. In practice, this is not feasible because the bulk of the work in writing a compiler is in designing the code generation strategy, which must be redone for any new machine in any event.

POP compilers are considerably more compact in code than machine coded compilers, especially where there is an addressability problem (such as in a minicomputer). Fortunately this is the place that compactness is needed most. POP written compilers are, however, slower than machine coded compilers not only because of the interpretive overhead, but also because of the redundant work done in moving data through the stack. This problem is masked in minicomputers since their computing power is very high relative to input output speeds of paper tape and teletypes.

The synopsis is that POPs are a fairly good way to write very small compilers for minicomputers and a poor way to write compilers for large machines.

Translator writing systems

As an implementation technique, the use of one of the many extant Translator Writing Systems deserves at least some mention. The idea of using a TWS is very appealing, but in practice the use of TWS has not proven much of a boon. The reason is quite simple. To date, TWS has done a good job of lexical scan, parsing, and the dictionary parts of a compiler, but has made few inroads into the most difficult parts, namely code generation and optimization. Even if the facilities provided by a TWS are valuable, the penalty of forcing a user into a prescribed mold has been too stiff for most production compiler writers to bear.

CODE GENERATION

Code generation has traditionally been one of the most ad hoc (and bug-ridden) parts of any compiler. There is some evidence, though, that this is changing rather rapidly. Formally, the generation of code is the attaching of semantics to specific syntactic structures. In the case of immediate generation, of course, opportunities for substantially altering the source order are minimal and conversion of the parse back into sequences of instructions proceeds strictly on a local basis. The actual generation of code is accomplished either by open sequences of code that construct the required instructions or by the equivalent of macro sequences selected by combination of the operator and the associated operand types. In the latter case, the macros are usually described in a format similar to the assembly language of the target machine. Conditional features similar to conditional features in macro assemblers are used to improve on the quality of code generated. Provision is normally made to test the state of the target machine as well as supplementary information about the operands. The macros also update the state of the target machine.

In multipass compilers, there is now a move to systematize code generation by formal tree transformations so that all data conversions, register loadings, and the like, are explicitly recognized in the tree structure. Most of the current work in formal semantics is along this line. Information may be collected during tree traversals which aids considerably in the production of high quality code.

Whether the code is produced by conventional or table driven techniques is far less important that the organization of the generation as a sequence of transformations rather than as a complex series of decisions based upon global data left as side effects of prior generation.

OPTIMIZATION

One of the most important differences between production compilers and experimental compilers is that
most production compilers try much harder to generate high quality code even at the expense of considerably slower compiling. This tradition dates from the very first FORTRAN compiler. At that time, it was felt necessary to compete with hand generated code in quality in order to gain acceptance. Since that time much effort has gone into finding ways to improve the quality of code generated by compilers. Unfortunately, the matter is too tightly bound up with the specifics of the language at hand. FORTRAN, for example, is relatively easy to optimize. PL/I, on the other hand, is almost impossible due to the significance of side-effects in the language. If, of course, optimization information is supplied by the programmer (which is rarely done), the problem becomes more nearly like that of FORTRAN.

After carefully sifting through all of the forms of optimization that have been used however, there are only two areas of optimization that have sufficient payoff to warrant consideration, unless super-optimization is being attempted. That is, there are only two issues in addition to strictly local improvements in generated sequences. These two areas are register assignment and address calculation.

Optimal register assignment

Also dating back to the earliest compilers is the problem of optimal assignment of the registers available in the machine. If a computer has only one register of a given class, or a very large number, the problem is minimal or does not exist. However, for common machines with 3 to 16 registers in a register class, the advantage to finding the correct data items to maintain in registers can be substantial. This is particularly true of values used for array accessing. Although there have been numerous papers on this subject, the general problem is still unsolved. Even the most satisfactory approaches require an excessive amount of computation to find an optimal solution.

The consequence of this dilemma, is that most compilers that do not do flow analysis usually simplify the problem by merely keeping a simple record of register contents. Loads and stores are then done on a strictly local basis. This works quite satisfactorily for all except the most demanding requirements.

Address calculation

One of the most important forms of non-trivial optimization and one that requires at least a modicum of flow analysis is the calculation of subscripts within a loop by repeated addition of some precalculated value to an address rather than a full recalculation of the address. This is best shown by the simple loop:

\[
\text{DO } K = 2 \text{ TO 99};
\]

\[
A(I, J, K) = A(I, J, K + 1) + A(I, J, K - 1);
\]

END;

The address calculations required can be drastically reduced with only two observations. First, all three array addresses are related to each other with only a constant difference. Second, consecutive executions of the loop body require only that the address be adjusted by a constant amount. The first of these simplifications is called common subexpression elimination. The second is called recursive address calculation. (It has nothing to do with recursion in the programming sense. Here it is really an iteration.)

Gear\(^4\) reports on a reasonably straightforward way of accomplishing this by consecutive passes over the parsed program in alternating directions. Although his technique is applicable without alteration only to explicit loops in the program with no local branching to disturb the movement of code, with more extensive flow analysis this can be generally accomplished.

Although common subexpressions other than subscript expressions can be located and moved, the benefits are not impressive and there are perils. The necessity of avoidance of altering the side effects of the evaluation is often underestimated. For this reason, more general common subexpression detection is not often done.

Since programs that do considerable accessing of multidimensional arrays spend a large part of the time in address calculation, locating common index expressions and calculating addresses recursively in loops is recommended as one of the first targets in any compiler intended to produce very high quality code.

SUMMARY

This has been intended as a brief overview of the way in which production compilers are written at the present time as seen by one of the practitioners of the art. The word art is used here as little has been accomplished in applying engineering techniques to compiler writing outside of the areas of lexical and syntactic analysis. The parts associated with declaration processing and code generation and optimization are still very ad hoc. The need for researchers to find better descriptive devices for generation strategies and the miscellaneous problems surrounding the creation of real object programs is still great. The need for compiler writers to
apply the best that is already known is perhaps even greater.

REFERENCES

1 D GRIES
   *Compiler construction for digital computers*
   John Wiley 1971
2 R W FLOYD
3 W R McKEEMAN  J J HORNING
   D B WORTMAN
   *A compiler generator*
   Prentice Hall 1970
4 C W GEAR
   *High speed compilation of efficient object code*
   Communications of the ACM August 1965 p 483 Vol 8