A GENERAL-PURPOSE TABLE-DRIVEN COMPILER

Stephen Warshall and Robert M. Shapiro
Computer Associates, Inc.
Lakeside Office Park
Wakefield, Massachusetts

INTRODUCTION

If a compiler is to generate efficient object code, there are several different kinds of optimization which should take place. Each of these optimization procedures has a preferred domain: that is, some algorithms prefer to operate over the input string, others over the tree which describes the syntax of the string, others over the "macro-instructions" which are generated from the tree, and so forth. In an earlier paper,[1] one of the present authors pointed out the necessity for employing the tree form in particular as a natural domain for optimizers which consider syntactic context and suggested that, just as Irons[2] and others had built general-purpose table-driven parsing algorithms, one could also build a general-purpose table-driven program for getting from trees to macro-instructions. The final compiler design presented here is the result of pursuing that kind of thinking somewhat farther.

COMPILER ORGANIZATION

The compiler is composed of five phases (not "passes," if that means pulls of the input tape):

1. A syntactic *analyzer* which converts a piece of input string into a tree-representation of its syntax.

2. A *generator,* which transforms the tree into a sequence of n-address macro-instructions, investigating syntactic context to decide the emission.

3. An *"in-sequence optimizer"* (ISO) which accumulates macros, recognizes and eliminates the redundant computation of common subexpressions, moves invariant computations out of loops, and assigns quantities to special registers.

4. A *code selector* which transforms macros into syllables of machine code, keeping complete track of what is in special registers at each stage of the computation.

5. An *assembler* which simply glues together the code syllables in whatever form is required by the system with which the compiler is to live: symbolic, absolute, or relocatable, with or without symbol tables, etc.

The first four phases are encoded as general-purpose programs; the fifth has been handled as a special-purpose job in each version of the compiler, and will therefore not be covered in the present discussion.

Phase I: Analyzer

The analyzer is of the "top-down" syntax-directed variety, driven by tables which are in effect an encodement of the source language's syntax as given by a description in the metalinguistics of the ALGOL 60 report[3] (the so-called "Backus normal form"). There are several features of interest in this encodement: rules are shortened where possible by application of the distributive law (thus, "<A>

<B> | <A> <C>" would be coded as "<A> (<B> | <C>)"; it is possible to define types by naming an arbitrary scanner which recognizes them, thus eliminating syntax-chasing when forming ordinary identifiers, etc.; left and right recursive rules are carried in a transformed condition whose effect is to force recognition of the longest possible string which satisfies the rule. The analyzer tables contain some information which is not syntactic, but rather concerned with compiler control (how much tree should be built before the generator is called, for example) or with specifying additional information to be placed in the tree for later use by the generator.

Phase II: Generator

The generator algorithm "walks" through the tree from node to node, following directions carried in its tables. As it walks, macro-instructions are emitted from time to time, also as indicated in the tables. The encodement of a set of tables (a so-called "generation strategy") is based upon the idea of a "relative tree name." A relative tree name is, formally, a function whose domain is the set of nodes in the tree and whose range is its domain union zero. At any given time, the generator is looking at (has walked to) some particular node of the tree. Every relative tree name is then interpreted (evaluated) as a function of that node as independent variable. A relative tree name is a rule for getting from "here" to some neighboring node of the tree. Thus, if we may view the tree as a genealogical one, a relative tree name might "mean" father or first son or first son of second son, for example, of the node the generator is currently considering.

A generation strategy is composed of a set of rules each of which consists of a description of some kind of node which is somehow of interest together with a list of things to be done when a node of that kind is in fact encountered. A kind of node is generally distinguished by its own syntactic type and the types of some of its neighbors. The things to be done include walking to a neighboring node and emission of macros whose variables are neighboring nodes. In all cases, neighboring nodes are named in the tables by relative tree names.

Phase III: In-Sequence Optimizer

The ISO accepts macro-instructions emitted by the generator. The processing of a macro usually results in placing the macro in a table and sending a "result" message back to the generator. Macros also instigate various bookkeeping and control operations within the ISO.

The processing of a macro is controlled by a table of macro descriptions. A macro may or may not be capable of being combined with others into a common-subexpression; the arguments of a macro may or may not be commutable, and so forth. The ISO will detect macros whose arguments are literals and in effect execute those macros at compile time, creating new literals. If a macro may be handled as (part of) a common subexpression and is not computable at compile time, the ISO will recognige a previous occurrence of the same macro as equivalent if none of its arguments have been changed in value either explicitly or implicitly by any of the messages that have been received in the interval.

At some point the ISO receives a macro-instruction that signals an "end region" control operation. This causes the ISO to perform a set of "global" optimizations over the region of macros just completed. These global optimizations include the recognition of those computations which remain "invariant" within the region and the reservation of "special registers" such as index registers to reduce the number of special register loads and stores within the region.

Phase IV: Code Selector

The code selector produces symbolic machine code for a region of macros after the ISO has collected these macros and performed its various optimizations. The code selector is driven by a table of code selection strategy. The domain of a strategy is the region of macros and a "track table" which represents the condition of the registers of the target computer.

The code selector views the macros as nodes of a tree structure; that is, certain partial orderings exist which guarantee that the code emitted preserve the computational meaning of the macros, but within these constraints the

strategy can order the computation according to its convenience, as a function of the availability of registers and results. The preferred mode of operation is to postpone making decisions about what code to generate for a macro until it is known how the result of that macro will be used.

The code selector also makes use of certain information gleaned from the macro-description tables in order to predict how the special registers (index registers, etc.) will be used. These predictions enable the code selector to use such registers intelligently. This local optimization, combined with the global reservation of special registers by the ISO, results in a fairly effective use of these registers.

BOOTSTRAP TECHNIQUE

There are three major sets of tables to be prepared if the compiler is to be particularized to a specific source language and target machine. These are the syntax tables, the generation strategy tables, and the tables of macro description and code selection. The bootstrap technique is simply a method of automating part of the process of preparing these tables.

A group of three languages was developed, corresponding to the three tables. These languages are all describable in the Backus notation and thus capable of analysis by the analyzer. A set of syntax tables and generation strategy tables for these languages was encoded by hand and installed in the compiler, which was then capable of translating from these special languages into macro-instructions. The link from the generator to the ISO was broken and replaced by a link to a special "Bootstrap ISO" which converted these apparent macro-instructions into lines of tables, namely, the three tables of interest. Thus the process of table preparation was reduced to one of writing down statements in a family of readable languages.

THE LANGUAGE BNF

The syntax of the source language is described in a language called BNF (to suggest "Backus normal form," denoted B.n.f.). BNF looks much like B.n.f. to within the limitations of the available character set and a desire for legibility. Thus syntactic type names are given as identifiers (upper case alphanumerics), the sign "::=" is replaced by "=", and the sign "|" by "/"; literal symbol strings are headed by "$" and delimited by "/" or blank. Within a symbol string "$" acts as a control character and forces inclusion of the next character as part of the string. Thus, as long as the symbol strings do not include "$", "/", or blank, everything is quite readable; if they do, legibility drops accordingly.

Examples:

B.n.f.	BNF
<arex> ::= <term> | <arex> <adop> <term>	AREX = TERM/AREX ADOP TERM
<relop> ::= GE | LE | UE | EQ	RELOP = $GE/$LE/$UE/$EQ
<mulop> ::= *|/	MULOP = $*/$$/

To each BNF type definition may also be appended tags which provide control information associated with the type.

THE LANGUAGE GSL

A statement in the generation strategy language GSL begins with a predicate to be satisfied, of the form:

$$\text{IF } \langle\text{type name}\rangle \text{ AND } \mathcal{S}_1\ (t_1) \text{ AND } \ldots \text{AND } \mathcal{S}_n\ (t_n),$$

where the $\mathcal{S}_i$ are assertions whose truth is to be tested and the t_i are relative tree names. The $\mathcal{S}_i$ are assertions about the presence or absence of a node, its syntactic type, the number of its "sons" (components), and so on.

Following this predicate is a sequence of commands to be associated with nodes of the distinguished kind. A command is either a directive to proceed to another node or an action of some sort (emit output, for example).

Example:

```
IF AREX AND SON3 = 0 AND
FATHER*FATHER*RTSIB IS ADOP,
$LFTSIB $SON1 $OUTPUT
(PLUS,SON1,FATHER*RTSIB).
```

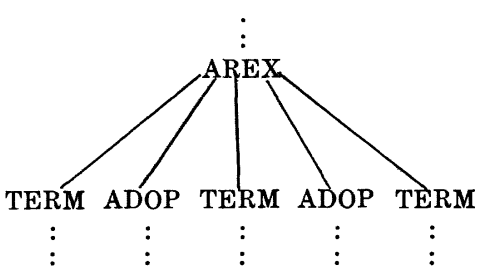
The list of commands is in effect a small program to be attached to a node when its kind is first determined. It is an odd program, in that each node keeps its own program counter. The generator may consider a node several times. The first time, its kind is determined, its generation strategy "program" attached, and the first command of the program executed. Upon subsequent reconsiderations of that particular node, the first command not yet executed is in fact performed. There are appropriate procedures for handling ill-defined commands (if a command directs walk to a non-existent node, take the next command; if there are no more commands, go to FATHER, and so on).

In the above example, upon first entry, the generator will just move consideration to the "left sibling" of the current node; at second entry, to the first son; at third, a PLUS macro will be emitted; afterward, any entry will cause automatic transfer to the father of this node.

Obviously, for a language of this sort—built around the relative tree name idea—to work at all, it is essential that the GSL programmer know in precise detail what tree structure will be output by the Analyzer. Thus the components of a type as given in the BNF and the sons of a node are in perfect one-one correspondence. The single exception is that left and right recursives come out as a "bush" of components at the same level; thus,

AREX = TERM/AREX ADOP TERM

would produce, from "A+B+C", a bush of the form



This exception is readily detectable by the BNF/GSL programmer, and thus does not violate the general philosophy.

The language GSL includes a number of commands besides "OUTPUT." These are concerned with passing information from node to node of the tree and with the emission of control messages to the ISO.

THE LANGUAGE MDL

The Macro Description Language provides a means of completely defining the semantics of the macro-instructions emitted by the generator. The definition contains the information necessary for controlling the processing of a macro by the ISO and the code selector, including a definition of the instruction repertoire and special registers of the "object" computer.

The most significant part of MDL is the sublanguage CSL (Code Selection Language). CSL permits the writer of a strategy to describe decision rules for the selection of machine instructions in terms of the status of macros (and their dependents: arguments, arguments of arguments, etc.) and the status of computer registers.

To illustrate a few of the properties of CSL and the code selector, assume the compiler is translating an algorithmic language with simple arithmetic assignment statements and has analyzed and generated over the statement

A ← B * C + A

producing the following macros:

(1) *, B, C
(2) +, (1), A
(3) ←, A, (2)

Assume that the computer has an accumulator (ACC), an addressable memory (M) and five single-address instructions:

CLA M : (M) → ACC
ADD M : (ACC) + (M) → ACC
MPY M : (ACC) * (M) → ACC
RAD M : (ACC) + (M) ACC, M
STO M : (ACC) → M

Following the principle that no code should be generated for a macro until the intended use of the result of the macro is known, the code selector would begin execution of the selection

strategy for macro line (3), the strategy for a "←" macro.

For this example the strategy would recognize that the RAD instruction was applicable and, after deciding on the execution of macro line one and verifying that its result is in the ACC, would emit an RAD A instruction.

The strategy for the "*" on macro line one would produce CLA B followed by MPY C.

Hence the code produced would be

```
CLA B
MPY C
RAD A
```

To give some idea of the appearance of a code selection strategy we append a set of statements in CSL. Each line has been numbered to facilitate referencing a set of explanatory notes. Also refer to the descriptions of EXECUTE, OUTPUT, and ALLOCATE which appear immediately after the strategies.

In the notes, the following symbols are used:

$\mathcal{V}$ variable
$\mathcal{E}$ expression
ω operation (either "+" or "x")

```
      FOR "←"
 1.                IS ARG2 = "+" MACRO
 2.                begin IS ARG1 (ARG2) = ARG1
 3.                    begin  C1 = ARG2(ARG2)
 4.                           GO TO ALPHA                    end
 5.                    IS ARG2 (ARG2) = ARG1
 6.                    begin C1 = ARG1 (ARG2)
 7.                           ALPHA .. EXECUT*(C1)
 8.                           ALLOCATE* (C1 TO ACC)
 9.                           OUTPUT* ((RAD) ARG1)
10.                           EXIT                           end end
11.               EXECUT (ARG2)
12.               ALLOCATE (ARG2 TO ACC)
13.               OUTPUT ((STO) ARG1)
14.               EXIT
```

Notes for "←"

1. Is the second argument (i.e., the value being assigned to the first argument) the result of a "+" macro?
2. If so, is the first argument of the "+" macro identical to the variable receiving the assignment; in other words do we have the form $\mathcal{V} \leftarrow \mathcal{V} + \mathcal{E}$?
3. If so, the local variable C1 is set to the macro line number for $\mathcal{E}$.
4. And transfer control to ALPHA (line 7).
5. Alternatively, is the second argument of the "+" macro identical to the variable receiving the assignment; in other words do we have the case $\mathcal{V} \leftarrow \mathcal{E} +$?
6. If so, the local variable C1 is set to the macro line number for $\mathcal{E}$.
7. ALPHA.. Execute the code selection strategy appropriate for the macro on the line specified by C1 (i.e., cause the evaluation of $\mathcal{E}$.
8. Execute the code selection strategy subroutine ALLOCATE to guarantee that the result of macro line C1 (i.e., the value of $\mathcal{E}$) is in the accumulator.
9. Output to the assembler RAD $\mathcal{V}$.
10. Exit.
11. Otherwise (the fail path of 1. or 5.) execute the code selection strategy appropriate for the macro pointed to by the second argument. (We have the case $\mathcal{V} \leftarrow \mathcal{E}$ and wish to cause the generation of code to evaluate $\mathcal{E}$).
12. Execute the code selection strategy subroutine ALLOCATE to guarantee that the value of $\mathcal{E}$ is in the accumulator.
13. Output to the assembler STO $\mathcal{V}$.
14. Exit.

```
FOR "+"
1. C2 = (ADD)
2. BETA . . EXECUTE (ARG1)
3. EXECUTE (ARG2)
4. IS ARG2 IN ACC
5. begin        C1 = ARG1
6.              GO TO GAMMA          end
7. ALLOCATE (ARG1 TO ACC)
8. C1 = ARG2
9. GAMMA . . OUTPUT ((C2) C1)
10. EXIT
```

Notes for "+"

1. The local variable C2 is set to the operation code for "ADD."
2. BETA . . Execute the code selection strategy appropriate for the macro pointed to by the first argument (we have the case $\mathcal{E}_1 \omega \mathcal{E}_2$, and wish to cause the generation of code to evaluate $\mathcal{E}_1$).
3. Execute the code selection strategy appropriate for the macro pointed to by the second argument (i.e., $\mathcal{E}_2$).
4. Is the value of $\mathcal{E}_2$ now in the accumulator?
5. If so, set local variable C1 to $\mathcal{E}_1$.
6. And transfer control to GAMMA (line 9).
7. Otherwise execute the code selection strategy subroutine ALLOCATE to guarantee that $\mathcal{E}_1$ is in the accumulator.
8. And set local variable C1 to $\mathcal{E}_2$.
9. GAMMA . . Output to the assembler the operation code specified by C2 and the address specified by C1.
10. Exit.

```
For "*"
1. C2 = (MPY)
2. GO TO BETA
```

Notes for ""*

1. The local variable C2 is set to the operation code for "MPY".
2. And transfer control to BTA (line 2 for "+" strategy).

EXECUTE (N): If N names a macro line this action causes the code selection to execute the strategy appropriate for the macro on line N and then resume where it left off.

OUTPUT (): This action outputs code to the assembler.

ALLOCATE (N_1 TO N_2): N_1 names a macro line and N_2 names a register (or register class). This code selection subroutine causes the code selection to guarantee that the value for which N_1 stands is placed in the register (or register class) named by N_2, saving the contents of the register if they are still needed.

A flow history of the processing for the example would be:

macro type	*line*	*resulting output*	
←	1		
←	2		
←	5		
←	6		
←	7		
*	1		
*	2		
+	2		
+	3		
+	4		
+	7	CLA	B
+	8		
+	9	MPY	C
+	10		
←	8		
←	9	RAD	A
←	10		

STATUS AND EVALUATION

This compiler and its associated bootstrap have been realized on a variety of machines (IBM 7090, Burroughs D-825, CDC 1604). The compiler has been used to translate from source languages JOVIAL, L_0 (the algebraic language of the CL-I System), and CXA (a BALGOL dialect) into several machine languages, including D-825 and 1604. The method of moving the compiler from machine to machine may be of interest as an indication of the power of the technique.

The compiler itself was originally written in language L_0 and compiled through the CL-I

Programming System into a running program on the IBM 7090. Then a deck of BNF/GSL/MDL cards defining the translation from L_0 into CDC 1604 was input to the bootstrap. After execution of the latter program, there existed a compiler for translating algorithms written in L_0 into an assembly language (meeting the requirements of the COOP system on the CDC 1604), operating on the IBM 7090. That compiler was fed the decks of cards in L_0 which had originally defined the compiler to CL-I. The result of this run was a compiler (and bootstrap) which could be moved to the CDC 1604. Then *that* version of the bootstrap was fed decks in BNF/GSL/MDL which defined the translation of CXA into CDC 1604, resulting in a CXA compiler on the CDC 1604.

The compiler is not as fast as some we have seen, but does not seem prohibitively slow, either. The object code is quite alarmingly good: indeed, it is frequently completely unreadable by a programmer of normal patience. In practice we prepare a BNF/GSL/MDL deck which does an adequate job. If we later want to improve the code (and are willing to slow down the compiler accordingly), we simply extend the deck.

The bootstrap method does not make compiler construction trivial, since code selection for a messy machine can be very difficult to work out and since contact with the data and control environment of the code being compiled may be more expensive than the translation process. What is now trival is the substantial modification of source syntax, minor changes in optimization rules, and the like.

The work described here has been informally reported to several agencies over the last year under various names, including "C.G.S." (for "compiler generator system"), the "bootstrap method," and the "COMPASS technique" for Computer Associates, Inc.).

BIBLIOGRAPHY

1. S. WARSHALL, "A Syntax Directed Generator", Proceedings of the EJCC, 1961, Macmillan and Co., 1961.
2. E. T. IRONS, "A Syntax Directed Compiler for ALGOL-60", *Communications of the ACM*, January, 1961.
3. NAUER (ed.) et al., "Report on the Algorithmic Language ALGOL-60," *Communications of the ACM*, Vol. 3, No. 5, May, 1960.