A solution is proposed to the problem of optimizing code generation by a large-language compiler.

A high-level definitional language is used to define the code mappings, and an interpreter executes the routines in this language during the one-pass, text-driven code-generation phase.

The technique might also be applied to extendable languages and shared-component compilers.

Code-generation technique for large-language compilers

by M. Elson and S. T. Rake

The design of an optimizing compiler for a large and rich language poses problems beyond those of sheer size and cost. One of the most serious is that the wealth and variety of the language makes it possible to express the same logical function in terms of many different source constructs. The choice may be based on naturalness of language use, program readability, ease of debugging, compile-time or object-time space-versus-time tradeoffs, or programmer whimsicality.

The problem is compounded when the target machine of the compiler is one as rich in function as the IBM System/360.[1] Again, there are many ways of expressing the same function. And if the source language is designed to be highly machine-independent, there is, naturally, no simple set of mappings between source and target constructs. The problem of optimizing in this environment is most strongly felt at code-generation time, when the mapping must be effected from a source-oriented text to a target-oriented one.

This paper describes a solution to this problem, in terms of a code-generation phase that features a high-level, special-purpose, code-generation language, and total context sensitivity, unlimited special casing, and a paging mechanism necessary because of the resultant phase size. The phase was developed as part of an experimental optimizing compiler.

In this paper, PL/I is used as an example source language,[2] and System/360 machine code is used because it is the particular target language for which the technique was developed. However, application of the ideas presented presuppose implementation of a language designed to be machine-independent, but do not presuppose PL/I source language or System/360 target language. The code-generation language was parameterized for different System/360 models, but not for radically different machine languages. The authors feel, however, that modifications to the language and its use might easily be made to accommodate different machine architectures.

We first describe our solution to the code-generation problem, then demonstrate the solution with a prototype compiler.

The code-generation problem

worst-case code

The semantics of PL/I are highly context-sensitive, so that worst-case code generation is a more severe problem than with simpler languages. For example, the worst-case and best-possible code that could be generated from the PL/I source statements

```
DCL (C1, C2) CHAR (10) VAR;
I=LENGTH (C1||C2);
```

are shown in Table 1. The table shows how local context-free code generation can destroy the meaning of the original source statement and then generate the only code possible. The meaning of the statement is—place in I the length of the result of concatenating C1 and C2—*not* concatenate C1 and C2 and then take the length of the result. The difference is only marginal in appearance but, as can be seen from the code generated, is significant.

source statement choices

The example illustrates another problem. A programmer is usually unaware of how a compiler processes the statements he includes in his source program. If the above statement had been written in some other manner, the code generated might have been considerably improved. For example, the statement

```
I=LENGTH (C1||C2);
```

results in poor code; the statement

```
I=LENGTH (C1) + LENGTH (C2);
```

results in the best possible code.

compiler size

Another problem is evident from studies that indicate that about fifteen times as much code generation logic is needed for unoptimized full PL/I as for optimized full FORTRAN. It was felt that, with

Table 1 Naive and context-sensitive code generation

	Naive		*Context-sensitive*	
	LA	1,WS1.1		
	L	2,DV..C1		
	LH	3,DV..C1+6	LH	14,DV..C1+6
	LTR	0,3	AH	14,DV..C2+6
	BC	8,CL.1	ST	14,I
	BCTR	3,0		
	EX	3,C..048C		
	AR	1,0		
CL.1	EQU	*		
	L	2,DV..C2		
	IC	3,DV..C2+6		
	LTR	3,3		
	BC	8,CL.2		
	AR	0,3		
	BCTR	3,0		
	EX	3,C..048C		
CL.2	EQU	*		
	STH	D,TMPDV..0480+6		
	L	14,DV..TMP..0444		
	L	15,TMPDV..0480		
	LH	8,TMPDV..0480+6		
	STH	8,DV..TMP..0444+6		
	LTR	8,8		
	BC	8,CL.3		
	BCTR	8,0		
CL.3	EQU	*		
	LH	14,DV..TMP..0444+6		
	ST	14,I		

the use of standard techniques, about fifty times as much would be needed for optimized PL/I as for optimized FORTRAN.

optimization problems

Multipass generators cause information to be lost between passes. Multipass code generators are function driven; i.e., during each pass, code is generated for a given set of functions and the passes are performed in a set order. Unless a large amount of information is retained, one phase is aware only of the data from a previous phase that it is to process; it is not aware of the use to be made later of its results. The amount of information carried around to allow communication from phase to phase is enormous for a large language.

Code in multipass generators is usually generated from the inside out. A suitable form of internal text for multipass generation is triples, or a similar structure. The triples express the relationships among the different parts of the source statement. They generally are ordered so that the innermost part of any expression occurs first. As code is generated by succeeding phases, the triples are replaced by sequences of computer instructions. The problem is

Table 2 Implications of A = B in PL/I

Precision	*A single B single*	*A single B double*	*A double B single*	*A double B double*
both aligned	L R,B ST R,A	L R,B ST R,A	SDR R,R LE R,B STD R,A	LD R,B STD R,A
both unaligned			MVC A(4),B XC A+4(4),A+4	
A unaligned			MVC A(4),B XC A+4(4),A+4	
B unaligned			MVC WKSPC(4),B SDR R,R LE R,WKSPC STD R,A	
either unaligned	MVC A(4),B	MVC A(4),B		MVC A(8),B

that unless the triples are manipulated in some way, the innermost part of an expression forces requirements on the outer parts. The consequences can be seen from the example in Table 1.

extendability

Additions to the language may require a new phase or an extensive addition to an existing phase, adding to the communication problem within the compiler. Several phases may be affected by a language change. When the compiler is first written, this is not a serious problem; however, as time passes, the introduction of new code makes it increasingly difficult to identify the areas affected by even the simplest change.

number of cases

Code generation for a large language must deal with a great many cases. The most innocent looking statement can have enormous ramifications, which, in turn, can lead to an enormous number of different strings of code. For example, consider the PL/I statement A = B, where A and B are both floating-point scalars. The possible implications are shown in Table 2.

Table 2 does not include cases for complex numbers, for extended precision, nor where there is an expression on the right hand side of the equal sign. It ignores multiple targets on the left of the assignment operator, and addressing problems are not considered. Cases specified are for time optimization but cases for space optimization are not given. The optimization is for an IBM System/360, Model 65. Other computer models require different code for best object-time performance. As this indicates, the problem is one of sheer size. The number of cases is huge in every part of the language.

the solution

The solution we saw to the problems posed by a multipass system was to have a single-pass code-generation phase. The single-pass phase allows code generation to be text-driven, which means that the text governs the order of code generation. The problem of inside-out generation was solved by the use of tree structures to represent the source text and by an outside-in order of scanning the trees.

The solution involved developing a definitional language that allows all cases to be defined. The definitions were then executed at compile time to generate the correct code.

A prototype compiler was designed and written to determine the validity of the techniques. A paging system was required to remain within the 100K-byte main storage space that was one of the design constraints for the prototype compiler.

The prototype compiler

In the prototype compiler, the functions performed in the code-generation phase are separated from the functions performed in other phases. There are five major sections in the compiler.

The front end creates trees and the required dictionary entries from the source program. It is necessary to expose addressing and subscript calculations as well as aggregate operations for code generation. A preoptimizer phase therefore expands the trees to show all such operations. Constant expressions are also evaluated. Following this phase, the trees are optimized by the cross optimizer, which performs the function of global optimization.

The code-generation phase accepts trees as input and produces pseudo-code as output. Pseudo-code differs from machine code in having symbolic registers, symbolic references to data, and symbolic references to labels in the pseudo-code.

The prototype has a skeleton back-end for storage allocation, register allocation, and final assembly. The register allocation function involves replacing symbolic registers with absolute registers, resolving register conflicts, and inserting store or load instructions as required. The prototype final assembly section produces only an object listing.

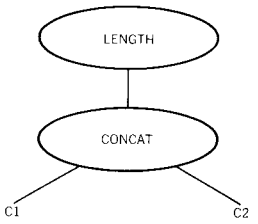
Tree text

Table 1 demonstrates a requirement for optimization at code-generation time—the context in which a program operand is to be used must be understood before that operand is evaluated.

With this ground rule in mind, several common data formats can serve as input to the code-generation phase. As an example, consider again the expression LENGTH (C1||C2) but in the following possible text forms:

1. Reverse Polish form
   C1 C2 || LENGTH
2. Standard Polish form
   LENGTH || C1 C2
3. Multi-address code with named results
   CONCAT t_1 C1 C2
   LENGTH t_2 t_1
4. Multi-address code with implicit instruction results
   1. CONCAT C1 C2
   2. LENGTH 1
5. Trees (see Figure 1)

Figure 1 Tree representation



These forms have much in common. Number 1 is derived from number 5 by a top-down, left-right tree walk; numbers 3 and 4 are derived from 1 by stacking operands left to right and then unstacking and producing code when an operator is encountered; numbers 3 and 4 are derived from 2 by stacking operators and operands separately and unstacking both, then producing code when all of an operator's operands have been encountered. In addition, each of these derivations is logically reversible. However, the issue here is not one of logical equivalence but of practical ease of processing. Bearing in mind the above ground rule, consider now a reasonable order for investigating the expression.

1. Invoke the length processor.
2. Invoke the concatenation processor. (Normally, it returns both a result and an indication of the length of that result. In this case, however, it is told to return only the length.)
3. Invoke data reference processors for C1 and then C2, telling each to return only the object time location of the length.
4. Return the object time locations of the lengths to the concatenation processor.
5. Generate the add of the lengths.
6. Return the location of the result to the length processor.
7. Return this value as the result.

In considering the above data formats for this kind of processing order, a striking implementation difficulty can be seen with both numbers 1 and 2. Whether the scanning order is forwards or backwards, it is difficult to find all of an operator's immediate operands. They are not adjacent, but separated by arbitrary distances; the location of one operand depends upon the full content of the other. Form number 3 has the same problem to a lesser degree; the scan entails investigating first operands of prior operators. Numbers 4 and 5 both give immediate access to operands. The usual hard-

ware imposition of a single-dimensional addressable store implies the same internal storage requirements for the two. They differ then only in external representation, and in the fact that using number 4 requires a bottom-up processing order to establish the context. In addition, number 5 seems to provide a more intuitive means of associating an operator with its operands diagramatically. Thus we used number 5.

interrogating trees

A large set of utility routines are provided to interrogate the trees at code-generation time. These routines are also used throughout earlier portions of the compiler to build, modify, and interrogate trees. Figure 2 gives a general idea of the complexity. It shows the complete tree for the assignment statement X=I, where both X and I are undeclared, thus acquiring the usual PL/I default characteristics.

The most important thing to note here is that all attribute information is retained in the text. During code generation, it is never necessary to interrogate the dictionary to produce code.

The code-generation mechanism

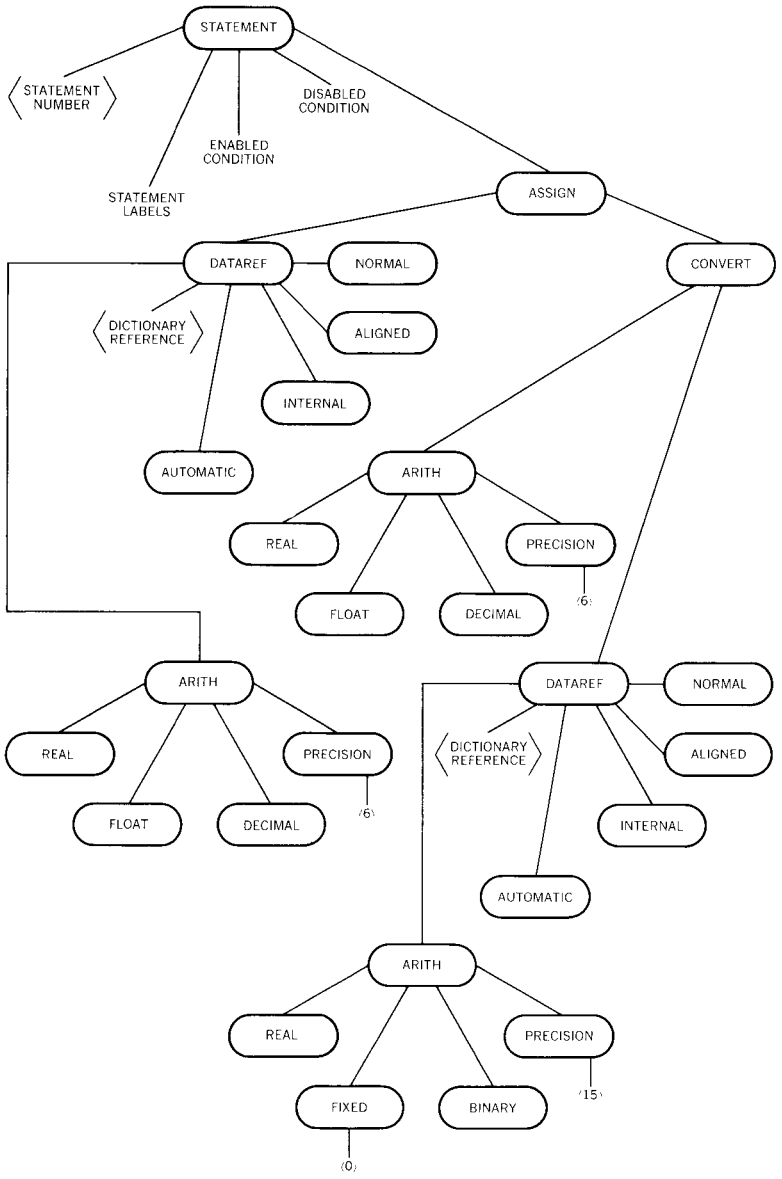
The production of pseudo-instructions at code-generation time is carried out by a set of routines called OPGEN macro definitions (OMD's). They are written in our generate coding language (GCL), pretranslated into a compressed internal form, and stored in a library as part of the compiler. They are invoked and interpreted as needed during code generation.

An OMD area is provided in main storage, and OMD's are paged into this area as required during execution. The paging mechanism is invoked when a GCL LINK command (bring in a new OMD) or an RTN command (return control to the LINKing OMD) is executed. An OMD need be in storage only while it is being executed. At any other time, it may be overwritten if the space is required. OMD's are read-only, so they need never be written out. Each (possibly recursive) invocation of an OMD involves a new allocation of dynamic workspace, which must remain active until that invocation is terminated.

The input data for the code-generation phase is the abstract tree text produced by the front end and the optimizer. The OMD's scan this text and produce from it the pseudo-instructions, which subsequently become input to the register-allocation phase.

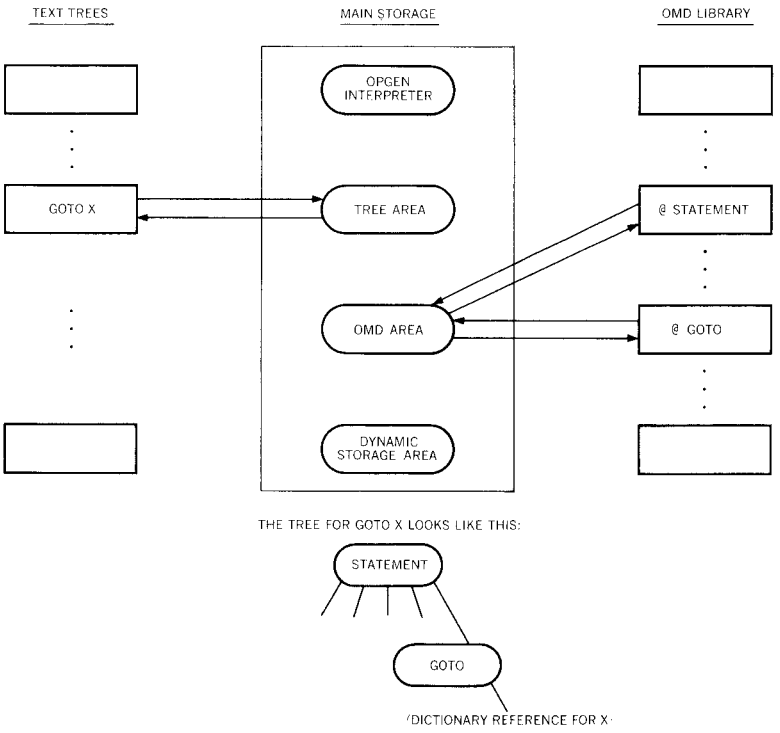
The compile-time flow of control is best illustrated by a simple example, as shown in Figure 3. The text trees are being processed one by one. Processing has just been completed for a statement, which we shall assume is GOTO X.

Figure 2 Tree for assignment statement X = 1



OPGEN, the interpreter for the OMD's, normally is a slave to the OMD's. It brings in a new OMD to be interpreted only when a LINK or RTN statement in a currently active OMD is executed. The only time that OPGEN initiates an action is when the OMD's have finished processing a statement. OPGEN then brings in the next statement tree, stores a pointer to the top node (which is always the STATEMENT node), brings in the OMD for handling this node (the @ STATEMENT OMD), and begins interpreting this OMD. (@ is a special marker indicating that the following name is an OMD name.)

Figure 3 Processing statement GOTO X



In the case shown in Figure 3, the OMD analyzes its first four arguments, possibly generating some code indicated by them. Eventually

LINK ARG(5)

is executed (the OMD representing the fifth argument of the STATEMENT node).

The OMD @ GOTO is now brought in and interpreted. It eventually generates the instruction

BC 15, X

which goes into the pseudo-instruction file, and issues an RTN statement. OPGEN stores another node pointer, checks that the @ STATEMENT is in main storage, and resumes interpretation of the @ STATEMENT from the point following the LINK. After some cleaning up, the @ STATEMENT issues a RTN, and OPGEN knows that processing for this statement has been completed. It then brings in the next tree and the cycle is repeated.

As is indicated in the above example, there is a direct relation between node names in the text trees and names of the corresponding

OMD's for processing those nodes. When a blind link is made (as in LINK ARG (5)), the indicated node is found, and the OMD invoked is that of the same name as the node name, but prefixed by the symbol @. All OMD names, in fact, begin with @, whether or not they correspond to node names.

Generate coding language

It became evident that some type of definition of the various cases to be generated was required. If this definitional language could then be executed in some way (compiled and either executed or interpreted), the definition of special cases could proceed together with the design and coding of the code-generation phase. Thus the generate coding language was developed.

language features

The following description of GCL is intended to convey the spirit of the language, which has the following characteristics:

- The language is procedural, with control passing from statement to statement unless an IF, GOTO, or subroutine call statement is encountered.
- Each OMD can reserve local storage (cells) for the duration of its execution.
- Facilities are provided for obtaining from trees the information needed to generate code.
- A subroutine call facility with argument-passing capabilities is provided.
- Complex expressions can be evaluated.
- A table look-up function allows information to be extracted from arrays of up to 256 dimensions.
- Code skeletons can be generated for all nonprivileged System/360 instructions.

procedural characteristic

In GCL, the form for the IF statement is:

IF (expression) true label, false label.

The expression is evaluated. If a true (nonzero) result is obtained, control is passed to the true label; if a false (zero) result is obtained, control is passed to the false label. The true and false labels are optional, implying that no special action is required if that condition arises.

GOTO acts as a FORTRAN GOTO, but with the extra ability to GOTO label variables (see section on local storage).

Control is passed from one OMD to another by use of the LINK and RTN statements, which are described in the section on subroutine calls.

local storage

An item with the CELL attribute is a 4-byte item that may be declared in any OMD that requires it. Cells are local to the OMD and are reallocated if the OMD is invoked recursively. A cell can hold many types of items, which can vary dynamically when the OMD is being executed.

Consider the following example of the use of a cell:

```
        DCL     C CELL
        SET     C=LABEL
        GOTO    C
        .
        .
        .
LABEL   ...
```

The GOTO passes control to whatever OMD label is held in cell C. In the sequence

```
DCL     PQ CELL
SET     PQ=1
SET     PQ=1.3584-5
```

PQ can hold both integers and floating-point numbers.

Another use of cell is shown in the sequence

```
DCL     XY CELL
SET     XY=@ PLUS
LINK    XY
```

Execution of the LINK statement passes control to the PLUS OMD.

Cells can also contain compiler-generated labels to be inserted into the pseudo-code, symbolic registers to be inserted into skeletons, 32-bit strings to hold switches, and packed decimal constants of up to 5 digits.

In addition to the cells, a long cell or string is provided. This type of storage is used when dealing with values that will not fit into four bytes, such as long floating-point constants, long decimal constants, character string constants, edit masks for conversion from numeric to character, etc. They are used exactly the same as cells, and in our implementation had a maximum length of 50 bytes. Use of the string facility is exemplified by:

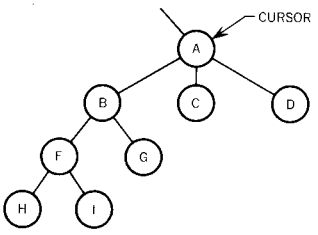
```
DCL     S STRING
SET     S=X'2021204B'
SET     S=S||X'20'
```

Another type of cell is provided for the entire code-generation phase. Such cells are thus known to all OMD's and are never dynamically reallocated.

tree analysis

To understand how a set of OMD's scans a tree and generates the correct code for that tree, it is important to understand the working of the cursor. The cursor is a pointer to the current node in the tree and can be altered by execution of certain statements. Before each tree is processed, the cursor is set to the top node.

Figure 4 Attribute expression tree

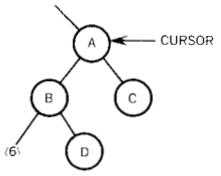


Information is extracted from the tree by use of attribute expressions, whose evaluations result in indications of presence or absence of specified nodes (Figure 4). An attribute expression is a sequence of node references (node names or argument indices) separated by any of the search specification symbols . , ; or _. Evaluation of such an expression proceeds as follows: The first node reference, which must be an argument index, is evaluated, and the cursor is pushed to this location relative to the old cursor position. If this action is impossible (the node indicated by the old cursor position had fewer arguments than the index of the one requested), then evaluation ceases and the value false (integer 0) is the expression result. Otherwise the next search symbol and the following node reference are examined. If the node reference is an argument index, the cursor is pushed as before, and the search symbol disregarded. But, if the reference is a node name, an attempt is made to push the cursor to the indicated node, if found, in a manner depending upon the search symbol. A period indicates that only immediate arguments of the current node are to be examined; a colon indicates that only the current node itself is to be examined; an underscore indicates that all descendants of the current node are to be examined. This process continues until either a search fails (expression false) or the expression is completed successfully (expression true). At completion of evaluation, the cursor is returned to its position prior to the expression evaluation.

For example, consider the tree in Figure 4, with the cursor initially as shown. Let us look at several attribute expressions.

- ARG(2):C (true)
  ARG(2), which is the node C is examined to determine whether its name is C.
- ARG(2).X or ARG(2)_X (false)
  C has no descendants, so clearly these are false.
- ARG(1):B. ARG(1).H (true)
  ARG(1) is examined and is named B. The cursor is then pushed to B's first argument and from this point (F) immediate arguments of the current node (F) are examined. One of them is H.
- ARG(1)_I (true)
  All descendants of B are checked. I is such a descendant.

Figure 5 Tree containing a value



The VALUE keyword allows values to be extracted from the tree. Consider the tree in Figure 5. The notation ⟨ ⟩ indicates a value in the tree as opposed to a node. The attribute expression

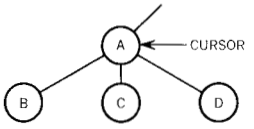
ARG(1).ARG(1).VALUE

returns the value 6. The VALUE keyword is essential to pick up scales and precisions.

subroutine calls

GCL allows OMD's to pass control from one to another by use of the LINK statement. When the RTN statement in the LINKed to OMD is encountered, control is passed to the statement following the LINK. The position of the cursor in the tree is not altered.

Figure 6 OMD-linking tree



A second type of LINK allows the tree to specify which OMD is invoked. Consider the tree shown in Figure 6.

The statement

LINK ARG(2)

causes the OMD corresponding to the node C to be invoked. The cursor is positioned at node C. When the OMD @ C has been executed and returns control to the calling OMD, the cursor is repositioned to node A. This type of blind linking is used to analyze the shape of the tree while actually generating code.

In either type of LINK statement, it is possible to pass arguments to the invoked OMD. The statement

LINK @ CONMPY (A, B, C, D, E)

passes to the CONMPY OMD the items a, b, c, d, e. The items are passed by name so that the invoked OMD can pass back results. The invoked OMD must have a similar parameter list. Thus,

```
LINK @ CONMPY (A,B,C,D,E,)
                        START @ CONMPY (P,Q,R,S,T)
                        .
                        .
                        .
                        END
```

A limited variable-length parameter list capability is provided, whereby a call to an OMD may have any number of arguments not greater than the number of parameters indicated in the OMD. The items that can be passed in an argument list include: cells, strings, registers, OMD names, and parameters. Results must be returned in cells or strings.

expressions

The language allows expressions that include all of the PL/I operators, as well as exclusive or ('?'). For example, the following expressions are allowed:

```
DCL (X,Y,Z,P,Q) CELL, S STRING
SET X=Y**2=(P>4|Q<2)*Z
SET S=S||X'20'
```

Logical expressions are evaluated to give an integer result (1 or 0). A limited amount of conversion is allowed:

```
SET X=1.0E0
SET X=X+1
```

Built-in functions include one called BIT, which tests the referenced bit in a cell and returns a true or false indication, depending on whether it is 1 or 0. The statement

```
IF (BIT(X,5))T,F
```

tests bit 5 of cell X and branches to T if bit 5 is a 1.

table look-up

The table look-up facility allows an item to be extracted from a table that can have up to 256 dimensions. The look-up is performed in response to the LOOK statement. For example, in the statement

```
LOOK error label, result cell, table name (arguments)
```

error label is the OMD label to which control is passed if the arguments do not specify a member of the table. The result cell will contain the item extracted from the table. Table name is the OMD label of the table being used in this LOOK statement.

In the simplest case of table look-up, the expressions serve as indices, so that the look-up acts as an array element reference. In more complex cases, the evaluated expressions may be tree node names, and the indexing is done by matching these names against the named table projections (rows, columns, etc.) In the simple case, a table is specified as follows:

Table Name:	TBL (dimension 1, dimension 2, etc.) type ARRAY item 1, item 2, etc.
Type:	Describes the length of each item in the array.
Dimension:	1, 2, etc., give the size of each dimension.
Item:	1, 2, etc., are the elements of the array in row major order.

The LOOK statement allows a multiple choice to be made in one statement. As an example:

```
                DCL         (X,Y,)CELL
                LOOK        ERROR,X,CONTBL(Y)
                GOTO        X
*GENERATE CODE TO MULTIPLY BY ONE
ONE
                .
                .
                .
*GENERATE CODE TO MULTIPLY BY TWO
TWO
                .
                .
                .
CONTROL TBL (6)REF
                ARRAY       ONE,TWO,THREE,FOUR,FIVE,SIX
```

Cell Y contains the constant by which a variable is to be multiplied. The LOOK extracts the OMD label from the table of the particular section of the OMD that will generate the required code. A similar effect can be achieved by a series of IF statements:

```
IF (Y=1)ONE
IF (Y=2)TWO
.
.
.
```

Many different types of items can be held in a table: floating-point constants, integers, symbolic registers, OMD labels, or OMD names.

skeletons

Code skeletons are similar in format to System/360 assembler language. When a skeleton is encountered in an OMD, it is inserted into the output file as pseudo-code. The registers used by the various skeletons must be declared. Thus the statements

```
DCL R REG(FIXED)
AR R,R
```

cause an AR skeleton to be generated with symbolic registers.

It is possible to generate code with absolute registers if these are required. The statements

```
DCL RO REG(FIXED,ABS(0)),R REG(FIXED)

LR   RO,R
```

cause an LR pseudo-instruction to be generated that loads absolute register 0 from a symbolic register.

In skeletons that require offsets and lengths, expressions can specify the required values. For example,

```
MVC     OFF*4=1(L=(L>6)*4=1,R),1(R)
```

It is possible to replace a register in a skeleton by a cell containing a register. Thus

```
DCL     R REG(FIXED),X CELL
SET     X=R
AR      X,X
```

If a skeleton refers to data in storage and the address of the storage is not known at code-generation time, the base and offset fields can be replaced by a cell containing the dictionary reference of the data. A later phase adds the addressabiity code. Thus the statement

```
DCL     X CELL, R REG(FIXED)
```

is followed by code to pick the dictionary reference from the tree and then by

```
L       R,0(X)
```

the interpreter

The execution of GCL could proceed in either of two ways—translate and interpret or compile and execute. It was decided to translate and interpret for several reasons. The translation process can be kept fairly simple. The translator takes GCL source code and compacts it in a one-for-one manner. Expressions are translated into reverse Polish notation.

To compile and execute would require a second compiler with its associated problems of housekeeping, module linkages, etc. Having an interpreter with all executable code in one place made the compiler easier to debug and more reliable. Also, because of the more compact interpreter code, it conserved main storage space.

Code-generation examples

Two examples of GCL code illustrate the code-generation process.

The first example, in Table 3, is of the complete OMD for doing floating-point assignment. It is presented to give the flavor of GCL and to indicate the relative ease of generating code for the many cases.

This example also illustrates some GCL coding conventions crucial to exploitation of the code-generation philosophy. The outside-in

Table 3 Floating-point assignment

```
                                @FLOATASSIGN OMD
    START @FLOATASSIGN
    DCL (GOPT, LLEN, RLEN, WKCELL, RATR, LATR, LO, LB, LI, LL, LR, RO, RB, RI, RR, RL,
      WKCELL) CELL, GPR REG (FIXED)
*CHECK GLOBAL CELL WHICH HAS COMPILER OPTIONS
    IF (BIT (GOPT, OPTT) = 0 | BIT(GOPT,MG5) = 0), OK
    MSG '@FLOATASSIGN OPTIMIZED ONLY FOR MOD 65, TIME OPTION'
*FIND BYTE LENGTHS OF SOURCE AND TARGET
*@FLOATLENGTH UTILITY EXPECTS CURSOR AT PARENT OF ARITH NODE
OK PUSH ARG (1)
    LINK @FLOATLENGTH (LLEN)
    POP
    PUSH ARG (2)
    LINK @FLOATLENGTH (RLEN)
    POP
    IF (LLEN = 16 | RLEN = 16), NOT16
    MSG 'DOUBLE DOUBLE LENGTH NOT SUPPORTED BY @FLOATASSIGN'
    RTN
NOT16 IF (ARG(1). ARG(1). COMPLEX | ARG (2). ARG(1). COMPLEX), NOTCPX
    MSG 'COMPLEX NOT SUPPORTED BY @FLOATASSIGN'
    RTN
NOTCPX SET LALN = 2 - ARG(1). UNALIGNED
    SET RALN = 2 - ARG(2). UNALIGNED
*NOW DO TABLE LOOKUP AND GO TO RESULT LABEL TO
*SET UP REQUIREMENTS FOR SOURCE RESULT, DEPENDING
* ON LENGTHS AND ALIGNMENTS
    LOOK ERR1, WKCELL, TBL1(LALN, LLEN/4, RALN, RLEN/4)
    GO TO WKCELL
ERR1 MSG 'ERROR IN TBL1 LOOKUP IN @FLOATASSIGN'
    RTN
*FOLLOWING ARE THE RESULT LABELS OF LOOKUP
*TARGET 4 BYTES ALIGNED, SOURCE ALIGNED. ASK FOR
*RX REFERENCE OR FLOATING REGISTER
RXFR1 SET RATR = M'F0001000'
    GOTO LRX
*8 - BYTE RESULT NEEDED IN FLOATING REGISTER, SO SOURCE
*WILL DO SDR, LE or LD or MVC(4), SDR, LE
FRFW1 SET RATR = M '30000000'
*GET ADDRESSABILITY OF TARGET AS RX or RS REFERENCE
LRX SET LATR = M 'C0000000'
    GOTO LINK
*REQUEST BOTH SOURCE AND TARGET AS RS REFERENCES
*SINCE MVC WILL BE DONE
RSI SET RATR = M '40000000'
    SET LATR = M '40000000'
*NOW LINK TO EACH ARGUMENT
*STANDARD CALLING SEQUENCE HAS BIT ATTRIBUTE CELL,
*OFFSET, BASE, INDEX, LENGTH, AND ONE EXTRA CELL
*FOR SPECIAL USE IN SOME CONTEXTS
LINK LINK ARG(1) (LATR, LO, LB, LI, LL, LR)
    LINK ARG(2) (RATR, RO, RB, RI, RL, RR)
*NOW DO LOOKUP AS BEFORE, BUT THIS TIME TO
*DECIDE WHERE TO GO TO FINISH WORK
 LOOK ERR2, WKCELL, TBL2 (LALN, LLEN/4, RALN, RLEN/4)
 GOTO WKCELL
ERR2 MSG 'ERROR IN TBL2 LOOKUP IN @FLOATASSIGN'
    RTN
*FOLLOWING ARE THE VARIOUS LABELS RESULTING
*FROM THE LOOKUP
*SOURCE IS EITHER RX (IF DATA REFERENCE) OR FLOATING
*REGISTER (IF EXPRESSION). IT HAS SET RATR TO INDICATE WHICH
RXFR2 IF (BIT (RATR, RXREF) | BIT (RATR, RSREF)) LST
```

Table 3 Floating-point assignment (cont'd)

```
*IN FLOAT REGISTER GIVEN IN RB FIELD
    STE RB, LO (LI, LB)
    RTN
*IN CORE
LST L GPR, RO (RI, RB)
    ST GPR, LO (LI, LB)
    RTN
*DOUBLE LENGTH RESULT IN REGISTER, TARGET ALIGNED
FRFUL2 STD RB, LO (LI, LB)
    RTN
*TARGET IS SHORT FLOAT, EITHER IS UNALIGNED.
*SOURCE WAS RETURNED AS RS REFERENCE
RS24 MVC LO (4, LB), RO (RB)
    RTN
*TARGET IS LONG FLOAT UNALIGNED, SOURCE LONG
*FLOAT RS REFERENCE
RS28 MVC LO (8, LB), RO (RB)
    RTN
*TARGET LONG FLOAT UNALIGNED, SOURCE SHORT FLOAT
*RS REFERENCE
RS24XC MVC LO (4, LB), RO (RB)
    XC LO + 4 (4, LB), LO + 4 (LB)
    RTN
*FOLLOWING ARE THE TWO TABLES, GIVEN IN
*ROW MAJOR ORDER
TBL1 TBL (2, 2, 2, 2) REF
    ARRY    RXFR1, RXFR1, RS1, RS1, FRFUL1, FRFUL1, FRFUL1, RS1, RS1, RS1, RS1,
            RS1, RS1, RS1, RS1, RS1
TBL2 TBL (2, 2, 2, 2) REF
    ARRY    RXFR2, RXFR2, RS4, RS4, FRFUL2, FRFUL2 FRFUL2, RS28, RS24, RS24, RS24,
            RS24, RS24XC, RS28, RS24SC, RS28
    END
```

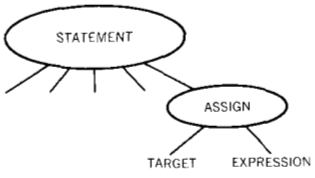
processing order automatically gives most of the context-sensitivity required. In general, it is not further required that the OMD for a node be given the identity of an argument node. If this information is needed for special cases, the OMD can of course determine it. But otherwise, it is able blindly to link to an argument node. Thus common parameter passing conventions must be used and respected within certain contexts. A common convention for all calls to expression node OMD's was used in the prototype. The first parameter is an attribute cell giving details of its requirements for location, length, alignment, etc., of the argument result. The subsequent parameters detail those specified in the first. Often the caller requests any of several alternative result conditions; in such cases, the called routine modifies the parameters to indicate which alternative has been used as most convenient.

The second example, in Table 4, shows the ease of the required context-dependent generation, which results from the outside-in processing order. This technique requires that the OMD for a certain node pass down certain requirements to the OMD's for processing its argument nodes. For example, @ ASSIGN may preallocate

Table 4 Length determination

```
 START @ LENGTHBIF (ATR, O, B, I, L, ML)
*SET BIT TO INDICATE ONLY A LENGTH REQUIRED
 SET ATR = ATR | M '00000001'
 LINK ARG (2) (ATR, O, B, I, L, ML)
 END
 START @CONCAT (ATR, O, B, I, L, ML)
    .
    .
    .
 IF (BIT (ATR, LONLY)), NORM
*SPECIAL CASE IF LENGTH ONLY REQUIRED
*ASK FIRST OPERAND TO PUT LENGTH IN REGISTER
 SET LATR = M '06004000'
 LINK ARG(2) (LATR, LO, B, LI, LL, LML)
*ASK SECOND OPERAND FR IN EITHER REGISTER OR STORAGE
 SET RATR = M 'C6000000'
 LINK ARG(3) (RATR, RO, RB, RI, RL, RML)
 IF (BIT (RATR, EGPR) | BIT (RATR, OGPR)) RR
*SECOND OPERAND IN STORAGE
 AH B, RO (RI, RB)
 RTN
*SECOND OPERAND IN REGISTER
 RR AR B, RB
 RTN
    .
    .
    .
 END
```

Figure 7 Floating-point assignment



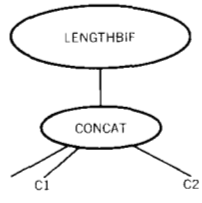
a target location for an argument's result, require a certain alignment, request RX or RS storage references, or any variety of register, etc. In this case, @ ASSIGN is asking that only a length be returned.

floating-point assignment

When @ FLOATASSIGN is invoked, the tree looks as shown in Figure 7. The sample tree in the section on tree text gives a representative tree for this case in more detail.

The GCL cursor is pointing at the ASSIGN node. Prior to this invocation, control had passed to the @ STATEMENT OMD, which then stated LINK ARG(5). This action brought in the @ ASSIGN OMD and positioned the cursor at ASSIGN. @ ASSIGN is a driver OMD that investigates the types of its arguments, then calls the appropriate assignment routine for the data-types found. In this case, it calls @ FLOATASSIGN, leaving the cursor positioned at ASSIGN.

Figure 8 Length function



The length built-in function is trivial because of the outside-in processing order followed during code generation. The @ LENGTHBIF OMD simply passes to its argument expression a preassigned bit on the first argument, indicating that only the length of the result is desired, not its value. The tree for the example is shown in Figure 8.

C1 and C2 may, in general, be string expressions. The following recursive definition of the result of the length bit for CONCAT holds for the various possible nodes under it:

@ CONCAT simply invokes its two arguments with the same bit on to indicate that only a length is required. These arguments return their result lengths as requested. @ CONCAT subsequently adds those lengths and returns this result to its caller (in this case, @ LENGTH).

The outside-in processing order ensures that no matter how complicated the argument may be, no processing is performed except that necessary to establish the length of the ultimate argument to @ LENGTH. The immediate argument OMD to @ LENGTH passes down to each argument the fact that only a length is required. Ultimately no result expressions are evaluated, only their lengths.

This example is typical of the value of the outside-in processing order. Every operand in a statement is evaluated only as required by its context.

Compile-time characteristics

Using a trace facility supplied by GCL, we obtained the following statistics: each PL/I statement results in 300 GCL statement executions; 33 GCL statement executions result in one line of pseudo-code; thus, each PL/I statement results in nine lines of pseudo-code.

Twenty OMD's were involved for each PL/I statement. Forty percent of all OMD's invoked were found to already be in main storage.

There were 211 OMD's, totaling 102,000 bytes of GCL code. Average length of an OMD was 482 bytes, the longest being 3800 bytes. Since it was estimated that this represented one-third of the code-generation phase of a PL/I compiler, a complete compiler code-generation phase would have required 600 OMD's totaling 300,000 bytes.

The compile time appeared to be slow, particularly in the code-generation phase, although no compile times are available. However, we believe that the interpreter and the OMD loader could be recoded so as to significantly reduce their size and their execution times.

Another area for improvement is the design of the prototype OMD loader. When called, the loader checks first to determine if the required OMD is already in main storage. If not, it checks for available unused storage space in the OMD area. If there is none, it simply displaces an OMD in main storage, taking up the least

space sufficient for the new OMD. More sophisticated techniques are available and could be used to advantage.

Displacement priorities should be statistically established, based on frequency of execution and size of the OMD's. Possible preloading techniques should be investigated. If a certain OMD always, or usually, calls a certain other OMD, then loading of the second OMD should accompany loading of the first.

At this time, effectiveness of these techniques cannot be quantified. The OMD structure, however, did obey the following two encouraging generalities: frequency of execution of the various OMD's varied widely, with a small number (@ STATEMENT, @ DATAREF, and several others) requiring most of the execution and most of the loader invocations—which should make a priority scheme effective. Many predictable OMD call trees were found, a result both of the OMD structuring and of the source language itself. Large high-level languages are not so modular that text-driven processing need imply totally unpredictable process sequencing. The preloading technique, therefore, should also be exploited.

Summary Comment

The authors believe that using the high-level language (GCL) for code generation has advantages in terms of extendability, flexibility, and reliability.

extendable compiler

The code-generation techniques are highly relevant to an extendable language definition system. In most such systems, definition of a new type of statement or language element involves two specifications: the information required to parse the new language element and integrate its syntax with surrounding language elements; and definition of the semantics of the new element, in terms of the compiler base language.

The second requirement means that the base language must theoretically have all power required, since all extensions are ultimately reduced to the base language. With the introduction of languages like PL/I, it becomes apparent that the base language required to extend to PL/I is very close to PL/I itself. Even if it were theoretically possible to extend a FORTRAN-type base to PL/I level, its efficiency would be doubtful.

This code-generation technique lends itself to a new kind of definition mechanism. A new language element might be defined in terms of its syntax and of the form the trees take when the element occurs in a source program. The trees might contain new nodes never before used. Where new nodes appear, new OMD's are written and entered into the system automatically, causing the semantics of

the new element to be defined precisely in a language very close to machine code (in this case, pseudo-code).

This possibility obviates the requirement of theoretical adequacy of the base language and makes it possible to define new language with efficient implementation. The authors believe that if this system of code generation is adopted by extendable language compilers, their efforts are likely to lead to considerable success.

language changes

One of the problems encountered by a compiler for a new language is that the semantics of the language elements tend to change. Keeping up with these changes can be very expensive and time consuming. Much of the compiler alteration occurs at the code-generation phase, since it is there that the new definition of the language element is finally realized. The authors believe that a code-generation scheme like the one described would allow language changes to be made easily and at small cost. OMD's can be added to a system almost ad infinitum.

reliability

When writing an optimizing compiler for a large language, two types of reliability become necessary. The writer must ensure that the compiler is bug-free and produces code that works. All cases must be covered with equal care to avoid leaving traps and pitfalls.

The experience gained during the coding of the code-generation stage of the prototype indicated that OMD's are easily debugged, requiring an average of three machine runs to check and debug. It was easy to think about the special cases involved because the problem was to identify the special cases, rather than code the phase so as to generate the required code. Since coding time was so trivial, effort could be spent on ensuring that all cases were covered and all pitfalls removed.

shared-component compilers

The technique described is a step toward providing a common component for the code-generation process. The tree text format, tree analysis, GCL (including its translator and interpreter), and the paging mechanism are all language-independent and might serve as common tools for use in other compilers. Producing the code-generation phase for a new compiler this way then would require only the writing of OMD's needed for that language. It is probable that if one has available the complete set of OMD's for a language as rich as PL/I, a large number could be lifted intact and used in new compilers for other languages.

ACKNOWLEDGMENTS

The authors wish particularly to thank Dick Sites for his contribution to our code generation design and development. Helpful suggestions were also received from the other compiler designers: Ray Larner, Tom Peters, Dave J. G. Reid, and Eric Souers.

CITED REFERENCES

1. G. A. Blaauw and F. P. Brooks, Jr., "The structure of System/360: Part I, Outline of the logical structure," *IBM Systems Journal* **3,** Nos. 2 and 3, 119–135 (1964).
2. IBM SYSTEM/360 Operating System: PL/I F Language Reference Manual, C28-8201, International Business Machines Corporation, Data Processing Division, White Plains, New York.