DATE: July 8, 1982

TO: R D & E Personnel

FROM: David Spector, Retargetable Back End Project

SUBJECT: Experiences With Compiler Code Generation by Parsing

REFERENCE: None

KEYWORDS: LANGUAGES, COMPILERS, CODE GENERATION, PARSING, RBE

ABSTRACT

The Retargetable Back End (RBE) project of the Translator Department aims at producing a generalized table-driven code generator that will make it easy to create compilers for a variety of new and existing hardware architectures, including V-Mode, X-Mode, M68000, etc.

The feasibility of this goal has been demonstrated by the writing of a prototype code generator using the Graham-Glanville-Ganapathi method of code generation by attributed LR parsing.

This document gives an overview of the nature of the code generation problem and how the Graham-Glanville-Ganapathi method provides a solution. Results obtained using our working prototype are described.

Please direct questions to Scott Turner, ext. 4073, x.mail TURNER, MS 10B-17-3, or to any member of the RBE Group.

1 Code Generation

Code generation is the term given to the last translation step in a compiler. In this last step an Intermediate Representation (IR) of the user's program is translated into computer instructions (and stored in a Binary or other such file of executable or almost-executable code).

Code generation has traditionally been done by a detailed and ad hoc analysis of the various cases for which code is to be emitted. Such code generators have usually been programmed in the same implementation language used for the rest of the compiler.

It has become increasingly evident, however, that this traditional methodology is not adequate to produce compilers that are reliable, free from bugs, easy to extend, or easy to retarget. (To retarget means to make an existing compiler produce code that will run on some specific computer. To rehost means to make the compiler itself run on another computer.) Coding a complex transformation such as that between source language and IR or that between IR and target machine directly in the implementation language means that the entire transformation program must be rewritten whenever retargeting is necessary. In contrast, a table-driven approach separates the algorithms that are independent of the target architecture from those that are dependent on it, making the code generator much easier to comprehend and modify.

2 LR Parsing

The recognition of the constructs of any language, be it PL/I, FORTRAN, or any other, is conveniently and efficiently done in modern compiler Front Ends via table-driven parsing methodology. An example of a table-driven parser here at Prime is DEREMER (see PE-T-535), which recognizes constructs in a language by preprocessing a BNF (Backus-Naur Form) description file to produce compact tables that are used to drive an LR parser. An LR parser is a program that recognizes language constructs in a bottom-up fashion. For example, we might define a fragment of a programming language involving parenthesized expressions using the following BNF production:

```
expression ::= term | expression '+' term | '(' expression ')'
```

This means that an expression can consist either of a term, the sum of an expression and a term, or a parenthesized expression. An LR parser driven by tables constructed from this production would examine its input (from the source language file) and decide which of the three alternatives of the production apply (if none apply, either another production applies or a user syntax error has occurred). The parser recognizes the constructs described on the right-hand side of the production first, then the production as a whole is recognized. This results in a bottom-up parse because the low-level constructs are recognized before the high-level ones (a high-level construct is defined in terms of low-level ones, as we see in the sample production above).

3 The Graham-Glanville-Ganapathi Method

R. S. Glanville and S. L. Graham of the University of California - Berkeley realized that LR parsing could be applied to the Intermediate Representation of a user program (in the form of a tree of data structures) just as easily as to the user program itself. Code generation by parsing is just as fast, free of bugs, and easy to change as any other LR parsing application.

M. Ganapathi of the University of Wisconsin - Madison extended the Graham-Glanville method to make it handle more of the code generation task and to do it in a more flexible way by adding attributes, predicates, and actions. These details will be omitted here in order to simplify the presentation.

References to further information on the Graham-Glanville method and Ganapathi's extensions are provided at the end of this paper.

4 Details of the GGG Method

The Graham-Glanville-Ganapathi (GGG) method requires viewing the IR as a sequence of prefix operators and their operands. Thus a source language statement such as "a = b + c" is viewed in its prefix form as "= a + b c". LR parsing then decomposes the IR into pieces corresponding to particular machine instructions.

As an example, consider the IR statement "= a + b c" just mentioned. A typical GGG code generator would parse this into three pieces, corresponding to the desired instruction sequence

```
LDA b       Load b into a register.
ADD c       Add c to the register.
STA a       Store the register into a.
```

The three productions that would be recognized might look as follows:

```
expression ::= memory_reference
expression ::= + expression memory_reference
statement ::= '=' memory_reference expression
```

Since each production must be associated with the appropriate instruction to be emitted, productions are expanded into onductions containing the instructions, their cost (this is used to help guide the parse when alternative parses exist), and other relevant information such as Boolean expressions representing semantic restrictions (example: recognize an increment instruction only when the operand is a constant having the value 1). A simplified set of onductions for the example might look as follows:

```
expression ::= memory_reference                  : LDA $1
expression ::= + expression memory_reference     : ADD $3
statement ::= '=' memory_reference expression    : STA $2
```

The $n notation refers to a value (called an attribute) associated with the nth symbol in the production, with the left-hand side counted as symbol 0.

5 The GGG Prototype

Our experimental code generator is called DEMO.SEG and is located in directory <TRANS7>RBE>DEMO on system ENX. DEMO reads in a specified Machine Description file then accepts IR prefix strings from the terminal and displays the resulting code (code is only generated in human-readable format, since this is an experimental system).

DEMO was written to be flexible and easy to modify, at the expense of processing time. It takes a large amount of memory and a great deal of CPU time to process even a small example. This is due first to the fact that the desired Machine Description file is read in and processed every time DEMO is loaded (instead of saving the internal data structures in a file and reading them back in) and second to the fact that the parsing tables are calculated directly from the onductions at code generation time (instead of being precalculated and saved in a file). An actual implementation of the GGG method would probably run as fast as any other known table-driven code generator method.

Instructions on the usage of DEMO and a full internal description are provided in the following documents, available in the directory <TRANS7>RBE>DOC:

RBE.3.DOC	Prototype Design
RBE.4.DOC	Prototype Functionality

6 Sample Machine Description File

The following is a tiny DEMO machine description file for an imaginary computer having two memory locations (called M1 and M2) and two registers (called R1 and R2). Only the operations of loading, adding, and storing are modelled in this machine description.

```
%Types
    int    %Range -2^15...2^15-1;
    ptr    %Constants m1, m2;

%Operators
{
    Operands
          |  Result Operand
Operator  |  Type   Types          Commutes     Comments          }
+         2  int,  int,  int  %Commute;        {add}
@         1  int,  ptr;                        {dereferencing}
=         2  stmt, ptr,  int;                  {assignment}

%Registers
{   Name      Type   Number of Registers}
    r         int    2;

%Categories
{   Name      Type}
    mr        ptr;
    ref       int;

%Instructions

{Memory References}
mr   ::=  m1        : 'M1'                  %Cost 0        %Size 0;
mr   ::=  m2        : 'M2'                  %Cost 0        %Size 0;
ref  ::=  @ mr      : $2.ref_code           %Cost 2        %Size 0;
ref  ::=  r         : 'R' || STRING($1.number)
                                            %Cost 1        %Size 0;
ref  ::=  int       : '=' || STRING($1.value)
                                            %Cost 2        %Size 0;

{Instruction Set}

r    ::=  ref       :'LOAD  R' || STRING($0.number) || ','
                       || $1.ref_code || ';'
                           %Cost  1 + $1.ref_cost   %Size 1;
r    ::=  + r ref   \ EQUAL_REG($0.number, $2.number)
                    :'ADD   R' || STRING($0.number) || ','
                       || $3.ref_code || ';'
                           %Cost  2 + $3.ref_cost   %Size 1;
stmt ::=  = mr r    :'STORE R' || STRING($3.number) || ','
                       || $2.ref_code || ';'
                           %Cost 3                  %Size 1;
```

7 Experimental Results

The following subsections give examples of IR and corresponding machine code emitted for several of the machine architectures investigated. This section is meant to give an idea of the ease with which code for various computers can be generated using the GGG method; many difficulties and limitations we have discovered will not be discussed here due to their technical nature as well as their known potential for being resolvable by further work.

7.1 Prime V-Mode

Example of emitting an increment instruction as an optimization for an INTEGER*2 addition:

```
=int2 addrel SB 11 +int2 1 @int2 addrel SB 12
(Add the constant 1 to the contents of location SB+12
 and store the result in location SB+11)
---------- size = 3 ---------- cost = 940 ----------
LDA   SB%+12
A1A
STA   SB%+11
```

Example of indirection, indexing, and the use of a temporary memory location in calling a procedure and passing an argument:

```
call @ptr addrel LB 52
     arg addrel addrel SB 200 @int2 addrel SB 55
     empty
(call indirect via the contents of LB+52, with an argument
 whose address is calculated by adding to SB+200 the value
 contained in SB+55)
---------- size = 9 ---------- cost = 14680 ----------
LDX   SB%+55
EAL   SB%+200,X
STL   T_PTR4_1
PCL   LB%+52,*
AP    T_PTR4_1,*SL
```

Example of converting an INTEGER*2 to a REAL*4:

```
=real4 SB @int2 LB
(Store the INTEGER*2 value pointed to by the LB register into
 the INTEGER*4 value pointed to by the SB register)
---------- size = 5 ---------- cost = 3380 ----------
LDA   LB%
FLTA
FST   SB%
```

Example of REAL*4 arithmetic:

```
=real4 SB negr4 @real4 LB
(Move the negative of the REAL*4 value pointed to by the LB
 register to the REAL*4 variable pointed to by the SB register)
----------  size = 6  ----------  cost = 5950  ----------
FLD   LB%
FCM
FRN
FST   SB%
```

Example of INTEGER*4 arithmetic:

```
=int4 addrel SB 50 +int4 49 @int4 addrel SB 52
(Add the INTEGER*4 value contained in SB+52 to the
 constant 49 and store the result in SB+50)
----------  size = 6  ----------  cost = 1040  ----------
LDL   SB%+52
ADL   =49
STL   SB%+50
```

7.2 Prime X-Mode

The examples in this section concentrate on one aspect of X-Mode, Prime's new instruction mode; they demonstrate the ability of DEMO to easily handle the many special cases of shifting in X-Mode.

Figure 1 is an excerpt from the partial X-Mode machine description used for the examples; it contains the productions that describe all the shift instructions. The specialized instructions (e.g., those which shift by one or two) specify the conditions under which the instruction may apply, and a cost that makes it cheaper than the more general instructions. DEMO uses this information to generate the most efficient instructions by selecting, from those productions whose conditions are met, the one with the lowest cost.

The purpose of Figure 1 is to give the flavor of a machine description of a complex aspect of a machine. It contains the following terminal and non-terminal symbols that are defined in parts of the machine description that are not shown:

- gr — stands for a general register -- anytime it is used on a left side, DEMO allocates a register for it.
- int2 — is a terminal symbol into which any 16-bit integer is lexed.
- shftl — is the shift operator, which takes two int2 operands
- mrint2 — is on the left sides of productions that are not shown, but which generate reference attributes which contain the appropriate strings for two byte (half word) memory references.

mrint4 is like mrint2, but its reference attribute contains strings for four byte (full word) memory references.

```
                    {SHIFT FROM MEMORY BY 1 OR 2}
gr ::= shftl mrint2 int2  \EQUAL($3.value,1)
       : 'LHL1  R'||STRING($0.number)||',*'||$2.ref_code||';'
       %cost $2.ref_cost + 10 %size 1; {Load Halfword Left shift 1}
gr ::= shftl mrint2 int2 \EQUAL($3.value,2)
       : 'LHL2  R'||STRING($0.number)||',*'||$2.ref_code||';'
       %cost $2.ref_cost + 10 %size 1;  {Load Halfword Left shift 2}
gr ::= shftl mrint4 int2 \EQUAL($3.value,1)
       : 'LL1   R'||STRING($0.number)||',*'||$2.ref_code||';'
       %cost $2.ref_cost + 10 %size 1;  {Load word Left shift 1}
gr ::= shftl mrint4 int2 \EQUAL($3.value,2)
       : 'LL2   R'||STRING($0.number)||',*'||$2.ref_code||';'
       %cost $2.ref_cost + 10 %size 1;  {Load word Left shift 2}

                    {SHIFT FROM REGISTER BY ONE OR 2}
gr ::= shftl gr int2 \EQUAL($3.value,1)
       \EQUAL_REG($0.number,$2.number)
       : 'SLL1  R'||STRING($0.number)||';'
       %cost 5 %size 1; {Shift Logical Left 1}
gr ::= shftl gr int2
       \EQUAL($3.value,2) \EQUAL_REG($0.number,$2.number)
       : 'SLL2  R'||STRING($0.number)||';'
       %cost 5 %size 1; {Shift Logical Left 2}
gr ::= shftl gr int2
       \EQUAL($3.value,-1) \EQUAL_REG($0.number,$2.number)
       : 'SLR1  R'||STRING($0.number)||';'
       %cost 5 %size 1; {Shift Logical Right 1}
gr ::= shftl gr int2
       \EQUAL($3.value,-2) \EQUAL_REG($0.number,$2.number)
       : 'SLR2  R'||STRING($0.number)||';'
       %cost 5 %size 1; {Shift Logical Right 2}

                    {SHIFT FROM REGISTER BY CONSTANT}
gr ::= shftl gr int2
       \RANGE(-32,$3.value,32) \EQUAL_REG($0.number,$2.number)
       : 'ISHL  R'||STRING($0.number)||',='||STRING($3.value)||';'
       %cost 10 %size 1;  {Immediate SHift register Left}

             {SHIFT FROM REGISTER BY CONTENTS OF REGISTER}
gr ::= shftl gr reg \EQUAL_REG($0.number,$2.number)
       : 'SHL   R'||STRING($0.number)||',*'||$3.ref_code||';'
       %cost 100 %size 2;  {SHift Logical by contents of register}
```

Figure 1 - Shifting Excerpts from an X-Mode Grammar

7.2.1 A Very Detailed Example

The processing of the first example, which demonstrates the use of the special load-and-shift instructions (only for a left shift by one or two), will be illustrated in some detail. The IR could be from a F77 statement like:

```
I = LS(J,1)
```

where I is at SB%+0 and J is at SB%+1000. The code emitted is:

```
IR> =int4 sb shftl @int4 addrelw sb 1000 1
=INT4 SB SHFTL @INT4 ADDRELW SB INT2.1000 INT2.1
---------- size = 3 ---------- cost = 10 ----------
LL1   R1,SB%+1000      * Load word Left shift 1
ST    R1,SB%           * STore
```

Figure 2a shows the parse by which the code was emitted for this example. The rules that apply are shown in Figure 2b.

```
IR> =int4 sb shftl @int4 addrelw sb 1000 1
=INT4 SB SHFTL @INT4 ADDRELW SB INT2.1000 INT2.1 (1)
      |
      |-Ref_code: SB%
      |
=INT4 BR SHFTL @INT4 ADDRELW SB INT2.1000 INT2.1 (3)
      |
      |-Ref_code: SB%
      |
=INT4 MRNI SHFTL @INT4 ADDRELW SB INT2.1000 INT2.1 (1)
                               |
                               |-Ref_code: SB%
                               |
=INT4 MRNI SHFTL @INT4 ADDRELW BR INT2.1000 INT2.1 (4)
                         |
                         |-Ref_code: SB%+1000
                         |
=INT4 MRNI SHFTL @INT4 MRNI INT2.1 (2)
                         |
                         |-Ref_code: SB%+1000
                         |
=INT4 MRNI SHFTL @INT4_MR INT2.1 (5)
                  |
                  |-Ref_code: SB%+1000
                  |
=INT4 MRNI SHFTL_MRINT4_INT2.1 (6)
            |
            |-Code: LL1    R1,SB%+1000;
            |
=INT4 MRNI GR (7)
            |
            |-Code: LL1    R1,SB%+1000;
            |-Ref_code: R1
            |
=INT4_MRNI_REG (8)
|
|-Code: LL1    R1,SB%+1000;ST     R1,SB%;
|
STMT
LL1    R1,SB%+1000
ST     R1,SB%
```

Figure 2a - Step by Step Example of Code Generation

```
1|br        ::=  sb : 'SB%' ;
2|mr        ::=  mrni : $1.ref_code;
3|mrni      ::=  br : $1.ref_code;
4|mrni      ::=  addrelw br int2 \RANGE (0, $3.value, 2^16 - 1)
                 : $2.ref_code || '+' || STRING ($3.value);
5|mrint4    ::=  @int4 mr : $2.ref_code;
6|gr        ::=  shftl mrint4 int2 \EQUAL($3.value,1)
                : 'LL1   R'||STRING($0.number)||',R'||$2.ref_code||';' ;
7|reg       ::=  gr : 'R'||STRING($1.number);
8|stmt      ::= =int4 mr reg
              : 'ST    R'||$3.ref_code||',R'|| $2.ref_code||';' ;
```

Figure 2b - Onductions Used in Figure 2a

After the line with the IR typed by the user, in figure 2a, is a representation closer to the internal form. The only significant difference is that integers are translated to INT2.<integer value>, and are represented internally as the symbol INT2 with the attribute "value" containing the integer. The code is generated in nine reductions; the effect of each reduction is shown by underlining the part of the parse stack that is to be replaced, drawing a line to the single symbol replacing those symbols in the next picture of the parse stack, and putting next to that line the attributes that are being transferred to the new symbol. The number to the left of the onduction in Figure 2b that was used for each reduction is shown in Figure 2a in curly brackets to the right of the previous parse stack.

7.2.2 Other Examples of Shifting

Example illustrating a shift when both operands must come from memory. The IR could be from a F77 statement like:

```
I = LS(J,K)
```

where I is at SB%+0, J at LB%+0, and K at SB%+1000.

```
IR> =int4 sb shftl @int2 lb @int4 addrelw sb 1000
=INT4 SB SHFTL @INT2 LB @INT4 ADDRELW SB INT2.1000
----------  size = 8  ----------  cost = 200  ----------
L      R2,SB%+1000       * Load
LHSE   R1,LB%            * Load Halfword with Sign Extented
SHL    R1,R2             * SHift Logical
ST     R1,SB%            * STore
```

Example illustrating the use of the special case shift by one or two instructions, where the first operand is already in a register. The example could be the IR from a F77 statement like:

```
I = LS(J+5,2)
```

where I is at SB%+0 and J is at SB%+100.

```
IR> =int4 sb shftl +int4 @int4 addrelw sb 100 5 2
=INT4 SB SHFTL +INT4 @INT4 ADDRELW SB INT2.100 INT2.5 INT2.2
---------- size = 7 ---------- cost = 80 ----------
L     R1,=5              * Load
A     R1,SB%+100         * Add
SLL2  R1                 * Shift Logical Left 2
ST    R1,SB%             * STore
```

Example using special instruction that shifts during a load, and also special instruction that increments a register by 2. The IR could be from a F77 statement like:

```
I = 2 + LS(J,2)
```

where I is at SB%+0 and J is at SB%+900.

```
IR> =int4 sb +int4 2 shftl @int2 addrelw sb 900 2
=INT4 SB +INT4 INT2.2 SHFTL @INT2 ADDRELW SB INT2.900 INT2.2
---------- size = 4 ---------- cost = 15 ----------
LHL2  R1,SB%+900         * Load Hafword Left shift 2
IR2   R1                 * Increase Register by 2
ST    R1,SB%             * STore
```

7.3 8080 Microprocessor

Example of INTEGER*4 arithmetic. Note that this example uses the same IR as the last V-Mode examole above.

```
=int4 addrel SB 50 +int4 49 @int4 addrel SB 52
(Add the INTEGER*4 value contained in SB+52 to the
 constant 49 and store the result in SB+50)
----------  size = 30  ----------  cost = 103  ----------
LXI  B,=52
SPHL
DAD  B
CALL LDI4
DB   2
LXI  D,=49
CALL INTL
DB   1
CALL ADDI4
DB   2
DB   1
LXI  B,=50
SPHL
DAD  B
CALL ST4
DB   2
```

A further example of INTEGER*4 arithmetic:

```
=int4 addrel SB 11 1
(Store the constant 1 as an INTEGER*4 in the location SB+11)
----------  size = 16  ----------  cost = 38  ----------
LXI  D,=1
CALL INTL
DB   1
LXI  B,=11
SPHL
DAD  B
CALL ST4
DB   1
```

7.4 6502 Microprocessor

(These examples all use 8-bit arithmetic, for simplicity.)

Example of a large expression forcing the Accumulator to be "spilled" into a temporary memory location:

```
= 404 + + @ 400 @ 401 + @ 402 @ 403
(Store the sum of locations 400 through 403 into location 404)
LDA 403
CLC
ADC 402
STA 200
LDA 401
CLC
ADC 400
CLC
ADC 200
STA 404
```

Example of the simultaneous use of both indexing modes:

```
= +addr 100 @ +addr 400 2 3
(Store the number 3 in the location obtained by adding 2
 to the contents of location 400 to find a location s,
 then adding the contents of s to the contents of
 location 100)
LDA #3
LDX #2
LDY 400,X
STA 100,Y
```

8 Acknowledgements

The DEMO prototype code generator was written by Debby Minard, David Spector, and Scott Turner under the leadership of Scott Turner. Experiments with various machine descriptions files were conducted by Lou Gross, David Spector, and Scott Turner.

9 References

[1] M. Ganapathi, Retargetable Code Generation and Optimization Using Attribute Grammars, Ph. D. Dissertation, University of Wisconsin-Madison, 1980.

[2] R. S. Glanville, A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers, Ph. D. dissertation, University of California - Berkeley, December, 1977.

[3] R. S. Glanville and S. L. Graham, A New Method for Compiler Code Generation, Fifth ACM Symposium on the Principles of Programming Languages (POPL), January, 1978, p. 231.

[4] S. L. Graham, Table-Driven Code Generation, IEEE Computer, August, 1980, p. 25.

[5] P. K. Turner, Deterministic Parsing with Code Generation Grammars, File RBE>DOC>RBE.8.DOC, March, 1982.