DATE: February 9, 1984

TO: R D & E Personnel

FROM: David Spector, Retargetable Back End Project

SUBJECT: Report on Base Level 1 of the Retargetable Back End

REFERENCE: PE-TI-1008

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 two prototype code generators. The first was known as DEMO and was described in PE-TI-1008. The second is known as RBE Base Level 1, and is described in this PE-TI. Where DEMO used LR parsing, Base Level 1 features a number of extensions including the use of tree parsing, integrated cost analysis and register and memory temporary allocation, and consequently emits better code than did DEMO, runs much faster, and handles a wider range of source programs. Although this prototype is as yet too limited in functionality to be used in actual compilers, it provides us with a base level from which we are attempting to develop a production-quality retargetable code generator. This document describes the Base Level 1 prototype code generator in detail, including brief examples of actual output. Background information on the nature of the code generation problem and how the Graham-Glanville method provides a solution may be found in PE-TI-1008, our earlier report.

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

Under no circumstances is this document, or PE-TI-1008, to be distributed to anyone other than the recipient. The subject matter includes company sensitive topics. Only R D & E Personnel are to see this document.

This is also RBE project document RBE.73.DOC.

## Table of Contents

1 Introduction....4

2 Limitations of Base Level 1....4

3 Examples....5
3.1 SPL to V-Mode....5
3.2 F77 to V-Mode....5
3.3 SPL to X-Mode....5

4 Code Selection by Parsing....6
4.1 LR Parsing....7
4.2 The Graham-Glanville Method....7
4.3 Details of the Graham-Glanville method....8
4.4 Turner's Up-Down Tree Parsing....8

5 An Overview of RBE Methods and Algorithms....9
5.1 Preprocessing....9
5.2 Code Generation....9
5.2.1 The Shaper....9
5.2.2 Decorate Pass....9
5.2.3 Select Pass....10
5.2.4 Macro Pass....10
5.3 Register Allocation....10
5.4 Temporary Allocation....10
5.5 Internal Tables....11

6 External Tables....11
6.1 Intermediate Language Definition....11
6.2 Machine Description....12

7 Development Environment....13
7.1 Coding Standards....13
7.2 Modularization....13
7.3 SDT....14
7.4 Design Reviews....14
7.5 Code Audits....14

8 Evaluation of RBE Base Level 1....14
8.1 Evaluation Areas....14
8.2 Conclusions....15

9 Acknowledgments....17

10 Bibliography.....17

## 1 Introduction

The RBE project is using a table-driven approach to code generation in which there is a separation between the data and algorithms that are independent of the target architecture from those that are dependent on it, making the code generator much easier to organize, understand, modify, and retarget.

The feasibility of this goal has been demonstrated by the writing of a prototype code generator, known as RBE Base Level 1, using the Graham-Glanville method of code generation, extended by the use of tree parsing, inherited and synthesized attributes, and an integrated mechanism for doing cost analysis, register allocation, and memory temporary allocation. This prototype provides us with a base level from which we are attempting to develop a production-quality retargetable code generator.

The Base Level 1 prototype accepts symbol table and operator files, as produced by F77 or SPL using the -LEAVE option, and displays a human-readable representation (similar to assembler language) of the machine code produced. No binary or executable output is supported.

The prototype code generator is built for a particular machine by the RBE Preprocessor, RBE_PREP, from files defining the Intermediate Language and the machine for which the code generator is to be targeted. Building a code generator can take several hours of computer time; this is reasonable. Running the resulting code generator can also take a great deal of computer time; this is not reasonable, but is a result of only one particular aspect of what the code generator is doing (very careful register allocation and cost analysis) and will be improved. Eventually, an RBE-based compiler is expected to execute in a time comparable with or better than that required for compilers using Prime's current Common Back End.

## 2 Limitations of Base Level 1

The Base Level 1 prototype is suitable only for experimental use because of these major limitations:

1. Only assignment statements are supported.
2. Only arithmetic datatypes are supported.
3. It runs very slowly.
4. Neither .BIN files nor .RUN files are produced.

These limitations have allowed us to develop the prototype more quickly.

# 3 Examples

Note that additional examples showing code for a variety of target machines may be found in PE-TI-1008.

## 3.1 SPL to V-Mode

```
a [fixed bin (15)] = a - 1;

LDA     SB%+43                          A
S1A
STA     SB%+43                          A
```

## 3.2 F77 to V-Mode

This example was compiled by F77 with -INTS, and with RBE warning messages removed.

```
D(I) = S(I)*T(K)

FLX     SB%+44                          I
FLD     SB%+244,X                       S()
FLX     SB%+45                          K
FMP     SB%+444,X                       T()
FLX     SB%+44                          I
FST     SB%+44,X                        D()
```

## 3.3 SPL to X-Mode

Here is an example of X-Mode code for a source program using pointers and arrays:

```
pstruct: proc;
    dcl (p1,p2,p3,p4) pointer;
    dcl (a,b,c,d) fixed bin (15);
    dcl (ap,bp,cp,dp) fixed bin (15) based;
    dcl ar1(100) fixed bin (15);
    dcl ar1p(100) fixed bin (15) based;
    dcl 1 bstruc based,
        2 fxb15 fixed bin (15),
        2 fxb31 fixed bin (31),
        2 flb23 float bin (23),
        2 flb47 float bin (47);
    dcl struc like bstruc;
```

```
    p1 = p3;
************************ code ************************
    ldbr    XB4,SB+152                  /* P3
    stbr    XB4,SB+146                  /* P1
*******************************************************
    p4 = addr(ar1(c));
************************ code ************************
    ldh     GR0,SB+44                   /* C
    eabr    XB4,SB+45                   /* AR1
    adbr    XB4,GR0
    stbr    XB4,SB+155                  /* P4
*******************************************************
    a = p1->ap + p2->bp;
************************ code ************************
    ldh     GR0,SB+149,@                /* P2
    adh     GR0,SB+146,@                /* P1
    sthx    GR0,SB+45                   /* A
*******************************************************
    p1->ap = ar1(40);
************************ code ************************
    ldh     GR0,SB+85                   /* AR1
    sthx    GR0,SB+146,@                /* P1
*******************************************************
    ar1(p4->cp) = p2->ar1p(d);
************************ code ************************
    ldh     GR0,SB+43                   /* D
    dcr     GR0
    ldh     GR0,SB+149,@,(GR0)          /* P2
    ldh     GR1,SB+155,@                /* P4
    sthx    GR0,SB+45,(GR1)             /* AR1
*******************************************************
    p2->bstruc.flb23 = struc.flb47 / 23;
************************ code ************************
    ld      GR0,23
    flt     FPR0,GR0
    frdvd   FPR0,SB+163                 /* STRUC.FLB47
    ldbr    XB4,SB+149                  /* P2
    fsts    FPR0,XB4+3                  /* BSTRUC.FLB23()
*******************************************************
    end;
```

## 4 Code Selection by Parsing

## 4.1 LR Parsing

The recognition of the constructs of any language, be it PL/I, FORTRAN, or any other, is conveniently and efficiently done in 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).

## 4.2 The Graham-Glanville 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.

A bibliography on the Graham-Glanville method and Ganapathi's extensions is provided at the end of this paper.

## 4.3 Details of the Graham-Glanville method

The Graham-Glanville 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 Graham-Glanville 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 memory_reference
expression ::= + expression memory_reference    : ADD memory_reference
statement ::= '=' memory_reference expression   : STA memory_reference
```

The "memory_reference" identifiers in these instruction templates refer to a value (called an attribute) associated with the symbol of that name in the production. Here the attributes are used to propagate text strings representing the memory reference portions of the instructions.

## 4.4 Turner's Up-Down Tree Parsing

In order to avoid certain situations where LR parsing commits too early to a particular production, it turns out to be more flexible to parse in a tree-oriented manner. Scott Turner has developed a method, called up-down parsing that accomplishes this. Prime is currently applying for a patent on his method, and we are also writing a paper on code generation by tree parsing that we hope will be accepted for the SIGPLAN '84 Compiler Construction Conference, to be held in Montreal.

Our implementation of up-down parsing uses a grammar that is identical in appearance to LR grammars; this is accomplished by using the fact that expression trees are equivalent to prefix expressions.

# 5 An Overview of RBE Methods and Algorithms

This section describes the methods and algorithms used by the RBE prototype. Further details can be obtained from the RBE User Manual (RBE>DOC>MANUAL.DOC, which is expected to be written soon), from the RBE detailed internal design document (RBE>DESIGN>RBE_DESIGN.DOC), or from the 80 memos containing discussions of most aspects of RBE (contained in the directory RBE>DOC and indexed in RBE>DOC>INDEX.DOC).

## 5.1 Preprocessing

The RBE Preprocessor, RBE_PREP, analyzes the Intermediate Language Definition (ILD) file and the Machine Description (MD) file and produces a number of SPL data declaration and procedure source files that become a part of RBE for the particular machine being targeted.

## 5.2 Code Generation

### 5.2.1 The Shaper

A procedure called The Shaper is used as an interface to convert the TSI intermediate representation (IR) currently produced by our front ends into the IR we have defined for RBE. Our IR is a true tree structure, and is defined by an ILD file.

### 5.2.2 Decorate Pass

The first actual pass over the IR operates bottom-up. Each node is annotated with several pieces of information calculated at this time.

The main annotation is an integer representing the state of the node with respect to up-down parsing. This state integer represents the set of grammar productions that could be recognized (selected) at this parsing point.

The other important annotation is the cost of selecting each possible production. The cost represents either the space or time requirements of the code to be emitted (this code is indicated by the macro associated with the production). This cost information is stored as a

pair of integers, representing the costs associated with each of two different extreme situations that could occur. The first is that registers can be allocated for the computation represented by the subtree rooted at this node without running out of registers. The second is that there are insufficient registers for successful allocation and that some values will have to be spilled into memory temporaries. The choice of which situation obtains (which amounts to what is usually known as register allocation) is done in the Decorate Pass and recorded in the annotation of the node.

### 5.2.3 Select Pass

The second pass over the IR operates top-down. The information left at each node by the Decorate pass is used to construct a parse tree. Selection of the cheapest production and register allocation situation is done, guided by this information. Information is stored in the parse tree concerning the order in which to emit the instructions and their components, such as operands of particular addressing modes. Registers are allocated a second time and specific register numbers recorded in the parse tree nodes.

### 5.2.4 Macro Pass

The third pass operates over the parse tree, not the IR. Each node is scanned in emission order, and macro procedures associated with its production are invoked to emit an instruction or a component of an instruction. Macros specify the construction of character strings, and can include references to inherited and synthesized attributes stored in the parse tree.

## 5.3 Register Allocation

Registers are allocated essentially by trying many combinations of allocations and choosing the best; since this process occurs during parsing, the emitted code is locally least-cost (where cost may either be with respect to time or size).

## 5.4 Temporary Allocation

Storing and loading of memory temporaries is specified in the machine grammar in the MD file; how to do the allocation is specified in a special Temp Macro also located in the MD file.

## 5.5 Internal Tables

The RBE preprocessor creates a number of SPL data declaration and procedure files that are compiled and linked with the machine-independent portion of RBE when a particular RBE code generator is built. These include files to define the grammar symbols, IR tree nodes, macros, and parse state transitions.

The sizes of the machine-specific data tables were approximately as follows (in 16-bit words):

| V-Mode | X-Mode |
|---|---|
| 71864 | 59565 |

# 6 External Tables

RBE is a set of tools that is targeted for a particular machine by means of two human-readable tables.

## 6.1 Intermediate Language Definition

The ILD file defines the set of all possible source file intermediate representations. It contains sections to define the datatypes, data representations, and operators to be represented in IR trees.

Here is a stripped-down version of our ILD file:

```
%Types
{-------------------------------------------------------------------}
{Type          Meaning                                              }
{-------------------------------------------------------------------}
{address;    {Address or pointer to anywhere in memory (predefined)}
{integer;    {Fixed binary number (predefined)}
real;        {All real numeric representations other than integer}
{stmt;       {Source statement (predefined)}

%Representations
{-------------------------------------------------------------------}
{Rep          Base Type    Size(Bits)  Meaning, Constraints
{-------------------------------------------------------------------}
int16         integer      16;         {Fixed binary number}
int32         integer      32;         {Fixed binary number}
real32VI      real         32;         {Float binary number}
real64VI      real         64;         {Float binary number}

%Alt_Representations
int16 < int32
real32VI < real64VI;

%Operators
```

```
{-------------------------------------------------------------------}
{ Format:                                                           }
{ result_type ::= operator [(internal_name)] list_of_operand_types  }
{                                            [list_of_properties]   }
{-------------------------------------------------------------------}
integer   ::= +int (addint) integer integer
              $Commutative $Associative; {Integer Addition}
real      ::= + (add) real real   $Commutative; {Addition}
real      ::= / (div) real real; {Division}
integer   ::= /int (divint) integer integer; {Integer Division}
```

## 6.2 Machine Description

The MD file describes the instruction set and machine architecture for which code is to be emitted. It contains sections to define the machine registers (what they are called, how many of each, and what they hold), the relationships between the registers (equivalence and overlap), how to allocate temporaries, and onductions (consisting of productions, which are specifications of what IR patterns to recognize, any needed context restrictions on the recognition, and any associated macros, which describe the machine instructions and components out of which complete instructions are constructed, including space and time costs and registers altered as side effects).

Here is a stripped-down version of our V-Mode MD file:

```
%Registers
{   Name    How Many    Representations  Macro            }
    A       1           int16          : nomac ();

%Categories
    goal        stmt;     {The goal nonterminal represents a
                           source statement.}
    mrint16     integer;  {A 16-bit integer operand for a memory
                           reference instruction}
    mrreal32    real;     {A 32-bit real operand for a memory
                           reference instruction}
%Instructions
{Category transformations}
mr16        ::=  mrni16;
mr16        ::=  mri16;
mr          ::=  mr16;
mrint16     ::=  @.int16 mr;

{One word stack frame and link frame references}
mrni16      ::=  stack int_const
                          \ RANGE (8, int_const.value, 255)
                          : mr_short_dir ('SB', int_const,
                                          stack.name);
mri16       ::=  addrel stack int_const X
                          \ RANGE (8, int_const.value, 255)
                          : mr_short_dir_idx ('SB', int_const,
```

```
                                                    addrel.name);

{Storing the A register into memory}
goal        ::=  = mr A                  : sta (mr);

{Calculations leaving their value in the A register}
A           ::=  +int A mrint16          : add_mac (mrint16);
A           ::=  +int A 1                : a1a ();
A           ::=  -int A mrint16          : sub_mac (mrint16.text);

%Macros
{Store the A register in memory}
sta (mem_ref):
    $Size 1;
    $Text   text_op_ = 'STA';
            text_args_ = mem_ref.text;
            $NL;
    $Time 440;
    $End;
```

# 7 Development Environment

## 7.1 Coding Standards

A detailed set of coding standards (RBE>DOC>STD.DOC) was developed early in our project to ensure that all RBE code is written in exactly the same clear style and with a uniform set of naming conventions.

## 7.2 Modularization

Large programs benefit from being modularized into sets of procedures that have common areas of functionality. We have developed a way to modularize SPL programs by defining Procedure modules, Procedure Interface modules, Data Interface modules, and Private Interface modules. All modules are registered on-line in a registration file. Further information may be found in RBE>DOC>STD.DOC.

### 7.3 SDT

We have developed a combination source control and product building system called SDT. Further information may be found in RBE>DOC>RBE.41.DOC.

### 7.4 Design Reviews

In order to ensure overall integrity and quality of our designs, we have held formal design reviews on all major RBE designs. The results of reviews are documented on forms stored in a project loose-leaf notebook. Further information may be found in RBE>DOC>RBE.31.DOC.

### 7.5 Code Audits

In order to ensure overall maintainability, understandability, and quality of our source code, we have required formal code audits for all installed RBE modules. The results of audits are documented on forms stored in a project loose-leaf notebook. Further information may be found in RBE>DOC>RBE.31.DOC.

## 8 Evaluation of RBE Base Level 1

At the conclusion of work on Base Level 1, the RBE project performed an evaluation of this base level. Our main conclusions are presented in this section.

### 8.1 Evaluation Areas

The design aspects that we intended to test in Base Level 1 were as follows:

- A working Shaper phase
- Preprocessed tables for code generation
- The basic code selection method
- The register allocation method
- The parsing algorithm
- A revised MD syntax and semantics

- The structure of the IL
- The idea of field macros
- Arithmetic and assignment operations in the IL

More fundamentally, and in view of the fact that there is not as yet wide experience with the use of the Graham-Glanville method in code generator design, we also wanted to answer the following questions:

- Can our method be made to generate high-quality local code?
- Does its understandability degrade too much when a lot of special cases are expressed in it?
- Can it be made to operate quickly enough to meet our marketing requirements?

## 8.2 Conclusions

With respect to the various design aspects listed above, we have concluded the following:

1. The Shaper concept works about as efficiently as we had anticipated. Some problems result from having to convert from the implicit addressing and datatyping of the TSI format to the explicit representation for these in the RBE format.

2. Preprocessed tables for code generation are necessary for satisfactory performance, but are difficult to implement. In addition, we found that our preprocessor did not detect certain kinds of language inadequacies in the MD file; these problems are found only at compile time and the error message reflects an obscure internal error condition that results.

3. The basic code selection method has been implemented successfully. A number of problems with it have been overcome.

4. The register allocation method has been shown to work well in many cases, although one or two fundamental bugs were found difficult to understand. Performance, however, has been a serious limitation, and we concluded it was necessary to remove register allocation from the inner loop of the Decorate phase.

5. The parsing algorithm (up-down parsing) was easily implemented and has functioned flawlessly. It is probably the greatest area of success for the project.

6. MD syntax and semantics were revised (from DEMO), providing a much greater range of functionality. We found that the MD file language was not adequate for describing all aspects of the target machine; the MD language will need to be extended. An experiment to determine how easy it is for someone outside of our project to write an MD file should be one of the first tasks of Base Level 2.

7. The structure of the IL proved adequate for the subset of IL semantics we selected.

8. The idea of field macros was proved realistic. The Macro pass is typically the fastest one. We did not, however, implement the "bit" (binary output) half of macros.

9. Arithmetic and assignment operations, including fairly complicated combinations of addressing modes, have been successfully implemented in a retargetable fashion.

10. Testing of common subexpression (CSE) support was removed from the Base Level 1 plan in order to finish in a timely manner.

With respect to the more fundamental questions listed above (our risks), we have concluded the following:

1. Our method has been demonstrated to generate high-quality local code in some limited contexts. It remains an open question whether this can also be true when the other project constraints are met.

2. There is some degradation of the understandability of the MD file when necessary special cases are expressed in it. A better answer to how retargetable RBE is will come when we have an outsider target RBE for a new machine, during Base Level 2.

3. We have not been able to answer whether RBE can be made to operate quickly enough to meet our marketing requirements; we can only say that the current register allocation method, which slows RBE by a factor of 100, is clearly infeasible. As the prototype was implemented, our feeling was that it would not be difficult to tune the Shaper, Select, and Macro passes for adequate performance. However, the actual performance of these passes has turned out to be slower than anticipated.

Our overall evaluation is mixed: We have shown the approach taken to be a reasonable one, but not all of us are satisfied with our progress in ruling out the basic risks inherent in using such experimental techniques.

## 9 Acknowledgments

The Base Level 1 prototype code generator was written by Debby Minard, David Spector, Lou Gross, and Scott Turner under the continuing leadership of Scott Turner. Our Section Manager, Ira Topping (who is unfortunately no longer with Prime), played an indispensable role in encouraging our excursions into the unknown wilds that lurk behind the use of state-of-the-art methods, balancing our resulting moments of panic, and helping us to plan our progress.

## 10 Bibliography

[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.