An Implementation of a Code Generator Specification Language for Table Driven Code Generators

Peter L. Bird
Department of Computer and Communication Sciences
and
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Abstract

This paper discusses an implementation of Glanville's code generator generator for producing a code generator for a production Pascal compiler on an Amdahl 470.

We successfully replaced the hand written code generator of an existing compiler with one which was produced automatically from a formal specification. This paper first outlines Glanville's original scheme, then describes extensions which were necessary for generating code for a production compiler.

1. Background

Attempts to systematize the process of code emission have been ongoing since the appearance of compilers in the 1950s. There are several survey papers [4,5] devoted to this history. Lately, the techniques of formalizing code generation have concentrated on table driven methods. One research direction has used heuristic strategies for determining appropriate code sequences (see [6] for a recent contribution). A second direction uses a grammar to describe the capabilities of the intermediate form (IF) of the compiler coupled with a Syntax Directed Translation Scheme (SDTS) [7]. In this approach, the code generator parses the IF of a program and emits the machine instructions specified by the SDTS templates.

The SDTS approach has the great advantage over heuristic methods in that the operation of the parser can be proven to be correct. If the specification of the code generator is correct, then the code generator cannot emit incorrect instruction sequences. Instead it will stop and signal an error. In addition, well understood algorithms exist for constructing the code generator's tables. We are only interested in the second scheme in this paper.

The work of Glanville [1,2] forms the basis for our research. In his method, the specification of a code generator is expressed as a simple SDTS. The form of the IF is described by a context-free grammar. Associated with each production of the grammar is a sequence of templates which specify the translation from intermediate code to target code. Consider the following translation fragment for an artificial machine:

```
r.2 ::= word d.1
{   LOAD  R.2,D.1 }
r.1 ::= iadd r.1 r.2
{   ADD   R.1,R.2 }
lambda ::= store word d.1 r.2
{   STORE R.2,D.1 }
```

where word, iadd and store are operators in the IF. Given the assignment statement

```
A := A + B ;
```

the IF representation might look like

```
store (word d.a,
       iadd (word d.a, word d.b))
```

where "d.a" is the location of the variable "A". The code emitted will be:

```
Load   R1,D.A
Load   R2,D.B
Add    R1,R2
Store  R1,D.A
```

The translation templates (enclosed in curly braces above) constitute the sequences of target machine instructions corresponding to the operation found in the production. The intermediate form emitted by the front end of a compiler (the lexical analyzer, parser, tree builder and static semantic checker) is manipulated by a shaping routine which resolve variable addresses by assigning base registers and

1 This work was supported by the University of Michigan Computing Center.

displacements. After shaping, the IF serves as input to the code generator. The code generator performs a bottom-up parse of the IF, and after a reduction, emits the appropriate machine instructions.

2. Overview of CoGG

CoGG accepts a specification for a code generator, and produces a code generator consisting of

1. A skeletal parser.
2. Tables for driving the parser.
3. Special utility routines for purposes of
   - i. register allocation and
   - ii. symbol table management.

The specification for the code generator consists of a declaration section and a production section. The declaration section is divided into five subsections, each corresponding to a different type of symbol. This allows CoGG to build a symbol table which contains the type of each identifier used, enabling the table constructor to type check the use of each identifier2. The five subsections declare the following entities:

1. Nonterminals - These correspond to the types of registers managed by the register allocation routine. They are either base registers (for address computations), or the registers that can hold the results of intermediate computations.
2. Terminals - These are identifiers whose values are set by the shaping routine. They are displacements, lengths, counts, etc.
3. Operators - These are only found in productions. They include arithmetic and logical operators, data transfer operators, and indicators of different machine level data types (such as byte, halfword, fullword, etc.).
4. Opcodes - The mnemonics for the instructions of the target machine.
5. Constants - They include both numeric constants as well as semantic operators (described in section 4.).

The production section specifies the SDTS. It allows the use of template sequences (rather that single instructions) for each production. Currently up to eight machine instructions may be emitted during a single reduction. In the generated tables, the templates contain indices into the translation stack or the list of allocated registers to speed up the process of code emission.

2 Such type checking is of utmost importance when processing the description of a realistic code generator. Our code generator for the Pascal language is specified with nearly 250 productions and 600 templates.

Many of the problems of a real architecture (such as machine idioms, jump instructions, etc.) were not addressed by Glanville, as his method is merely capable of specifying straightforward string to string translations. We found it necessary to add operators enabling templates to invoke semantic operations at code generation time. These operators were needed in order to generate correct code. The operations fall into the following categories:

1. Management of symbol tables internal to the code generator.
2. Manipulations to account for machine idioms.
3. Context sensitive manipulations of the parse/translation stack.

Before discussing these extensions, we will briefly sketch the layout of a code generator produced by CoGG.

3. Code Generator Structure

The code generator consists of three portions:

1. A standard LR parser.
2. A code emission routine which is called to perform reductions and build the actual machine instructions.
3. A Loader Record Generator which resolves all label references and branch instructions, and emits standard system loader records.

The structure of the code emission routine is as follows:

```
{ Assume that a reduction has occurred. }
  begin
    remove current production from
           the parse stack.
    allocate all requested registers.
    for all associated templates do begin
      fill in required values
         { registers, displacements, etc. }
      if template requires
         semantic intervention
        then case intervention_code of
               ....
             end
        else append instruction
             to code buffer
    end
    prefix LHS to input stream.
  end
```

While parsing the IF, label locations and branch instructions are kept in a dictionary. This is necessary for reasons discussed in subsection 4.2. After all of the IF representation of a program has been processed, the loader record generator resolves the absolute addresses in a two pass traversal of the dictionary. When all label locations and branch targets have been resolved, the routine constructs the TEXT records which make up the object module.

4. Semantic Operators

The major shortcomings of Glanville's method are in the areas of machine idioms, addressing, register allocation, common subexpression handling and the typing of operands. Because a pure string to string translation is inadequate for describing the behavior of a realistic code generator, we have substantially enlarged the specification language by adding semantic operators which can deal with all of the above problems.

4.1 Register Allocation

CoGG provides the operators USING, NEED and MODIFIES to allow the user to communicate with the code generator's register allocation routine. The first two tie the register allocation routine to the translation scheme. The last is used for keeping track of common subexpressions.

The interpretation by the code generator of either the USING or the NEED directive results in a call to the register allocation routine. The operands of the directive indicate that a register of a particular type is needed to perform the computation. The NEED directive requests a specific register from some type class; USING is more general and requests any register of the class. The use of a specific register is necessary for utilizing certain machine instructions, or for system specific purposes such as subroutine calls.

The operator MODIFIES is used by the common subexpression handler. It informs the register allocation routine that the contents of a register has been changed.

For an example, consider the following templates:

```
r.2 ::= fullword dsp.1 r.1
{  USING R.2
   L     R.2,DSP.1(R.1)    }

r.1 ::= iadd r.1 fullword dsp.2 r.2
{  MODIFIES R.1
   A     R.1,DSP.2(R.2)    }
```

and the IF program segment:

```
iadd (fullword dsp.a base,
      fullword dsp.b base)
```

where base is the base register for the local data area.

For these calculations, some register is needed for temporarily holding the result of the computation. The value returned from the USING directive is inserted into the Load template, and then used as the LHS of the production. The resulting system 370 code sequence is:

```
L     R1,D.B(BASE)
A     R1,D.A(BASE)
```

The IADD template uses the MODIFIES directive to invalidate any common subexpression held in R.1.

If a specific register is requested, and that register is in use, then the current contents of that register is transferred to another register of the same type, and the translation stack is updated to reflect the change in the location of the result of that computation.

As was shown in the description of the code emission routine above, the call to the register allocator is made prior to acting upon any of the templates associated with the production; all registers required by the template sequence are allocated at one time. When a register is allocated, its use count is decremented. If a register is used as the LHS of the production, its use count is incremented when the LHS is pushed back on the parse stack3.

We use a "least recently used" register allocation strategy in an attempt to reduce operand contention in the pipeline of the machine (see [8] for a discussion including algorithms for minimizing instruction contention). Each register has a usage index associated with it. Every time a reduction occurs, a global index value is incremented. When a register is allocated for use in templates, or when it is modified, the current global index value is recorded in the register record. Thus, the register with the lowest usage index was changed at a time previous to all other registers and in terms of pipeline contention, it is "least recently used". When the register allocator is called, the free registers with the lowest index values are allocated first.

4.2 Addressing

Without knowledge of how (and where) instructions are emitted, it is impossible for any routine which only operates on the IF to specify the target location of any branch instruction. For some architectures, even if a code generator emits assembly code and an assembler is used to generate object modules, the problem of addressability of the target location remains. This is true for two reasons:

1. Routines operating only on the IF have minimal knowledge of the number of instructions it takes to implement a language construct. In our case, since the templates associated with a production may change (because the code generator is retargetable) it is inappropriate to hardwire this information into the shaper.
2. The problem of long and short instruction sizes [9,10] (and hence the absolute size of the

3 Actually every LHS is prefixed to the input stream.

object module) cannot be resolved until after all labels have been located in the generated code. Even if all instructions are generated using the long format4, the exact target location for a forward branch is unknown until after the label is encountered in the code generation process.

We have solved this problem by installing the code emiss In operator:

```
   lambda ::= label_def lbl.1
   {  LABEL_LOCATION LBL.1         }
```

The interpretation of the LABEL_LOCATION directive causes the code generator to record a relative label in the dictionary at the location of the current program counter.

A branch instruction may take the following form:

```
   lambda ::= branch_op lbl.1 cond.1 cc.1
   {  USING R.3
      BRANCH COND.1,LBL.1,R.3 }
```

As was discussed above, the binding of jump instructions to the target is resolved after all code for a module has been generated. If the target for a jump instruction resides on another page,5 then an additional load instruction (loading a page multiple value into a register) is required to establish addressability of the target location. When interpreted, the BRANCH template will allocate two instructions in the code buffer (for the case of the long instruction) and will enter into the dictionary a branch existing at the current program counter targeting this label. The allocated register (R.3 in the above templates) is to be used in the event that a long instruction is needed; this will serve as the index register.

There are many instances when it is not desirable to have all the details of a template declared in the production; unnecessary details complicate both the construction of a shaper and the form of the productions for the code generator. Consider the templates used for storing a the result of a comparison into a boolean variable:

4 In the case of certain older architectures, the long jump instruction actually consists of two machine instructions, the first of which is used to establish a base register for the second instruction.

5 On an Amdahl (or an IBM 370), all memory references are performed using base registers. The maximum range of addressability with one base register is 4096 bytes. On our machine, 1 page equals 4096 bytes.

```
   lambda ::= assign boolean dsp.1 r.1 cc.1
   {  USING R.3
      MVI    DSP.1(R.1),FALSE
      SKIP   FALSE_COND,R.3
      MVI    DSP.1(R.1),TRUE        }
```

It is undesirable to force the shaper to allocate all of the labels needed to perform the above SKIP operations. Instead, the code generator enters the branch instruction and target into its dictionary, to be resolved with the others declared in the IF representation.

4.3 Machine Idioms and Translation Stack Manipulations

As with Branch instructions above, we found it easier to handle machine idioms by semantic actions. One very important idiom concerns double register usage. The IBM 360 architecture uses an even/odd register pair when performing integer multiplication, division, or modulo arithmetic.

There are instructions which treat even/odd pairs as a single 64 bit operand, such as SLDA (shift left double arithmetic) and SRDA (shift right double arithmetic). For example, "SRDA E.1,32" will transform the 32 bit signed value in the even register into a 64 bit signed value in the even/odd pair. This is a necessary prelude to performing a division or modulo operation.

The following simple translation scheme (from [2])

```
   d.1 ::= e.1
   {   SLDA E.1,32 }
```

ignores what is happening with the odd register of the even/odd pair (aside from actually destroying the contents of E.1). This will greatly complicate the role of register allocation, possibly forcing a considerable amount of unwanted movement of register contents.

Consequently, we have included several operations in our templates which are intercepted by the code emission routine, and either cause a modified instruction to be emitted, or the translation stack to be manipulated. For an example, consider the following:

```
   r.2 ::= imult r.2 fullword dsp.1 r.1
   {  USING DBL.1
      LOAD_ODD       DBL.1,DSP.1(R.1)
      MR              DBL.1,R.2
      PUSH_ODD       DBL.1
      IGNORE_LHS                  }
```

The special LOAD operator will load the fullword value into the odd half of the double register pair. PUSH will then "push" the odd register on the top of the parse stack (it does so after performing a type conversion of the odd register into type "R.n"). IGNORE_LHS prevents the parser from pushing the LHS of the production since this has already been done.

Although this approach requires a fair amount of intervention, it seems necessary that certain contextual information be used to insure that the proper result is placed on the stack. Otherwise the scheme

```
   r.1 ::= d.1          {      }
```

may fail to retrieve the proper register. Since the results of IMULT/IDIV or IMOD operation leaves the result in a different register of the even/odd pair, the context of the operation defines the location of the result register. It is possible that the code emission routine (and the register allocation routine) could both recognize that the odd register in

```
   o.1 ::= imult ....
```

is associated with a double register through

```
   d.1 ::= <e.1,o.1>
```

but this amounts to hardwiring the result register into both the code emitter and the table constructor.

It is necessary to have the LHS of the multiplication production included in the grammar even though it is never pushed. A result is pushed when the reduction occurs (with the PUSH_ODD operator), but the LHS declared with the production is used so the table constructor recognizes that the IMULT operation is accessible to integer arithmetic computations.

4.4 Common Subexpressions (CSE)

All CSEs are detected, and their use counts established, by an IF optimizer. Two sets of productions are associated with CSEs: definition of the CSE in the code generator's symbol table, and usage of the CSE.

Establishment of a CSE requires:

1. A CSE number. Each CSE is assigned a unique identifier which is valid throughout the compilation.
2. A usage count.
3. A temporary storage location This is allocated by the shaper routine, and is used only in the event that the register value is modified.
4. A register holding the result of the computation.

The following templates are used for the declaration of a CSE:

```
   r.2 ::= make_common cse.1 cnt.1
              fullword dsp.1 r.1 r.2
   {  COMMON CSE.1,CNT.1,R.2,DSP.1,R.1 }
```

Using the CSE for a computation only requires the symbolic name of the CSE as demonstrated by the following template:

```
   r.1 ::= use_common cse.1
   {  USING R.1
      FIND_COMMON     CSE.1,R.1
      IGNORE_LHS           }
```

At code generation time, the effect of the FIND_COMMON operation is as follows: if the CSE still resides in a register, then that register value is prefixed to the input stream. If the CSE resides only in memory, however, then the address of the CSE is prefixed to the input stream. In either case, the actual current location of the CSE is needed only at the time this reduction is performed, and its management is left to semantic routines in the code generator.

4.5 Typing of Operands

Included in the operators of the IF are unary operators which give the implementer both access to and checking of different data types of the architecture. As an example, consider:

```
   r.2 ::= fullword dsp.1 r.1
   {  USING R.2
      L      R.2,DSP.1(R.1)        }
```

which forces a fullword integer be loaded, while

```
   r.2 ::= halfword dsp.1 r.1
   {  USING R.2
      LH     R.2,DSP.1(R.1)          }
```

loads a halfword.

Without an operator to indicate a variable's storage format, the code generator is constrained in its ability to generate code suitable to the machine's architecture. The code generator could ignore all but one of the machine's data types, but this would be fairly inefficient storage utilization. Alternatively, it could examine the symbol table of the front end of the compiler each time an operand was referenced. This would require a significant amount of semantic intervention as well as decreasing the modularity of the components of the compiler.

5. Comparisons

Table 1 contains information about the number of declarations used to define the tables for the code generator.

Entry (i) is a count of all identifiers used in constructing the tables. (ii) on the other hand is a count of only those symbols which can be encountered in the IF during a parse. (v) is a count of those entries which do NOT contain an error entry. (viii) is a count of the number of operators which can be encountered in the IF. These include IADD, FULLWORD, etc. (ix) is the number of operators designed to produce semantic intervention.

Table 2 contains information about the size of the object modules for the tables

i.	Number of symbols declared	247
ii.	X dimension of parse table	87
iii.	States in parsing automaton	810
iv.	Parse table entries	70470
v.	Significant Entries	30366
vi.	Productions	248
vii.	SDT templates	578
viii.	Production operators	68
ix.	Semantic operators	28

Table 1.

and the code generator, and compares this to an existing production compiler with a handwritten code generator. Entry (v) is a

i.	Template Array	8.5
ii.	Compressed Parse Table	32.7
iii.	Uncompressed Parse Table	71.5
iv.	Code generation routines	7.5
v.	PascalVS Translation routines	41.9
vi.	Full PascalVS Code generator	53.8

Table 2.
(Sizes are in pages)

measurement without support routines from the PascalVS runtime library. (vi) is a measurement using these routines.

The SDTS represented by these tables supports bitset operations with inline code generation, as well as quadruple precision (128 bit) floating point arithmetic.

Many productions have been included to take advantage of the index registers used for addressing in most instructions, as well as the various data types in this architecture. There are no less than thirteen productions associated with integer addition (IADD), where one production (add register to register) would be sufficient to generate accurate code. All of the integer operations (ISUB, IMULT, etc) have the same level of redundancy. As we see from table 2, however, the size is not significantly larger than for the translation phase of a currently used IBM program product. Additionally, the "compressed" tables are by no means minimally compressed.

6. Conclusion

The use of formal specifications of code generators for their implementation is clearly superior to the traditional approach of hand crafting them. This is especially true in the attempts to retarget a compiler to a new machine, where a hand crafted code generator would require extensive rewriting. In an SDTS approach, retargetting the code generator merely requires a rewriting of the templates associated with productions and minor modifications of the routines which actually emit the machine instructions.

The approach specified in [1,2] works in a production environment only through the introduction of substantial extensions. Code generation is a translation process which needs more information than is available to the parser on its stack. We saw this problem above with label handling and common subexpressions. The more sophisticated the utilization of machine idioms, the greater the contextual intervention required by the code generator.

The input to the code generator is actually a linearized tree structure. The process of parsing the IF by the code generator is in fact the detection and transformation of subtrees which correspond valid computations [3]. Each production in the grammar corresponds to a valid subtree which might be encountered in a computation tree. The size of the parse tables described in the last section are due to our attempt to recognize a very large number of possible tree shapes. With a larger number of recognizable patterns, the code generator can produce better code. As was stated above, a single IADD production would be enough to produce executable code, but the large number of productions allows the code generator to produce code which is as good as that produce by IBM's PascalVS [12]. See appendix one for a comparison emitted code.

This scheme should prove successful on microcomputers with limited memory. By reducing the number of productions in the grammar, the size of the parse tables is also reduced. A language implementer can therefore control the size of the compiler by changing the complexity of the grammar. This size change can be accomplished without losing the guarantee of generating correct code.

The code generator we replaced [11] produced code for a PDP-10. It encompassed 17 separate compilation units and was 5000 lines long (not including several files of type declarations). CoGG is less than 3000 lines. The code generator it produces is less than 2500 lines (not including parse tables) and is contained in 2 separate compilations. The process of adapting the original code generator to generate code for an an Amdahl is of considerable complexity when using traditional methods which require rewriting the code generator. In contrast, writing the specification for the code generator and using CoGG for its implementation was much less complicated and less error prone. It seems clear that establishing and maintaining a grammar is a much simpler task than writing and maintaining a code generator.

Acknowledgements

Professor Uwe Pleban provided much needed advice and encouragement during all phases of this project, in addition to carefully reading an earlier draft of this paper. George Schimmel helped to wade through problems encountered during construction of the table generator. Paul Pickelmann collected information about the PascalVS compiler.

References

[1] R.S. Glanville and S.L. Graham, A New Method for Compiler Code Generation, 5th ACM Symposium on Principles of Programming Languages (1978).

[2] R.S. Glanville, A Machine Independent Algorithm for Code Generation and its use in Retargetable Compilers, PhD Thesis, University of California, Berkeley, 1977.

[3] S.L. Graham, Table Driven Code Generation IEEE Computer (Aug. 1980) 25-34.

[4] M. Ganapathi and C.N. Fisher, A review of Automatic Code Generation Techniques Computer Science Tech. Report #407, Computer Science Dept. University of Wisconsin - Madison (January 1981).

[5] R.G. Cattell, A Survey and Critique of Some Models of Code Generation Department of Computer Science Report, Carnegie-Mellon University (November 1977).

[6] R.G. Cattell, Formalization and Automatic Derivation of Code Generators, PhD Thesis, Carnegie-Mellon University, 1978.

[7] W. Barrett and J. Couch, Compiler Construction: Theory and Practice SRA 1979.

[8] J.L. Hennessy and T.R. Gross, Code Generation and Reorganization in the Presence of Pipeline Constraints, 9th ACM Symposium on Principles of Programming Languages (1982).

[9] E. Robertson, Code generation and storage allocation for machines with Span dependent Instructions ACM Trans. Program. Lang. Syst. 1,1 (July 1979) 71-83.

[10] B. Leverett and T. Szymanski, Chaining Span-Dependent Jump Instructions ACM Trans. Program. Lang. Syst. 2,3 (July 1980) 274-289.

[11] R.N. Faiman, Jr. and A.A. Kortesoja, An Optimizing Pascal Compiler IEEE Transactions on Software Engineering Vol SE-6, No. 6, (November 1980).

[12] Pascal/VS Release 2.1, Program Product #5796-PNQ, IBM Corporation.

Appendix 1.

```
*  Code examples.

*  The base type of all arrays is integer.  No subscript or range checking is performed.
*  The equation compiled is:

*           x[q]:=(a[i]+b[j]*(c[k]-d[l])+(e[m] div (f[n]+g[o]))*h[p]);

                 CoGG                                           PascalVS

     l    r1,132(r12)     Load Q            |    L     03,152(,13)      Load K
     sla  r1,2            Multiply by 4.    |    SLA   03,2
     l    r2,100(r12)     Load I            |    L     04,156(,13)      Load L
     sla  r2,2                              |    SLA   04,2
     l    r3,104(r12)     Load J            |    L     05,576(03,13)    Load C(K)
     sla  r3,2                              |    S     05,776(04,13)    C(K)-D(L)
     l    r4,108(r12)     Load K            |    L     04,148(,13)      Load J
     sla  r4,2                              |    SLA   04,2
     l    r5,850(r4,r12)  Load C(K)         |    LR    07,05
     l    r6,112(r12)     Load L            |    M     06,376(04,13)    B(J) * ...
     sla  r6,2                              |    L     06,144(,13)      Load I
     s    r5,1250(r6,r12) C(K) - D(L)       |    SLA   06,2
     l    r7,450(r3,r12)  Load B(J)         |    A     07,176(06,13)    A(I)+...
     mr   r6,r5           B(J) * ...        |    L     06,164(,13)      Load N
     a    r7,150(r2,r12)  A(I) + ...        |    SLA   06,2
     l    r8,116(r12)     Load M            |    L     03,168(,13)      Load O
     sla  r8                                |    SLA   03,2
     l    r9,20(r12)      Load N            |    L     04,1176(06,13)   F(N)
     sla  r9                                |    A     04,1376(03,13)   F(N)+G(O)
     l    r2,24(r12)      Load O            |    L     05,160(,13)      Load M
     sla  r2                                |    SLA   05,2
     l    r3,2450(r2,r12) Load G(O)         |    L     08,976(05,13)    E(M)
     a    r3,2050(r9,r12) F(N) + G(O)       |    SRDA  08,32            Propogate Sign
     l    r4(r8,r12)      Load E(M)         |    DR    08,04            E(M) / ...
     srda r4,32           Propogate Sign    |    L     06,172(,13)      Load P
     dr   r4,r3           E(M) / ...        |    SLA   06,2
     l    r6,28(r12)      Load P            |    M     08,1576(06,13)   H(P) * ...
     sla  r6,2                              |    AR    07,09            Final Addition
     l    r3,2850(r6,r12) Load H(P)         |    L     06,176(,13)      Load Q
     mr   r2,r5           H(P) * ...        |    SLA   06,2
     ar   r7,r3           Final Addition    |    ST    07,1776(06,13)   Store X(Q)
     st   r7,3250(r1,r12) Store X(Q)        |

*   if  flag  then i := j - 1
*             else i := z ;
*   if p < q  then l := z ;

*  where:   i, j, k, p, q  are fullwords ,
*           b is a boolean (logical) variable,
*           z is a halfword
*                  (Note that PascalVS didn't use a halfword for the storage format).

     tm    3300(r12),1       Test B             |   TM    3780(13),1           Test B
     bc    8,Label1          Branch if false    |   BNO   @2L1                 Branch if false
     l     r1,104(r13)       Load J             |   L     03,148(,13)          Load J
     bctr  r1,r0             Decrement          |   BCTR  03,00                Decrement
     st    r1,100(r13)       Store I            |   ST    03,144(,13)          Store I
     bc    15,Label2         Branch             |   B     @2L2                 Branch
  Label:    1                                   | @2L1 DS    0H
     lh    r2,142(r13)       Load Z             |   MVC   144(4,13),168(13)    I := Z ;
     st    r2,100(r13)       Store I            | @2L2 DS    0H
  Label:    2                                   |   L     03,172(,13)          Load P
     l     r3,136(r13)       Load Q             |   C     03,176(,13)          Compare with Q
     c     r3,140(r12)       Compare with P     |   BNL   @2L3                 Branch  if >=
     bc    4,Label3          Branch if <        |   MVC   156(4,13),168(13)    L := Z
     lh    r4,142(r13)       Load Z             | @2L3 DS    0H
     st    r4,112(r13)       Store L            |
  Label:    3                                   |
```

Appendix 2.

```
  1          $options
             * symbol dump
               listing only
               punch_packed

             *  The following is the SDTS for the Amdahl 470.
             *  The format of each line is:
             *    i.  The left aligned production
             *   ii.  The Syntax Templates to be emitted when the
             *        production is used to reduce.
             *   NOTICE:  Templates MUST skip column one!

             *  Lines beginning with '*' are comments. Blank lines are
             *  ignored. All others are examined! Comments may be inserted
             *  after the instructions in the templates. Not all productions
             *  have been included.

 25          $Non-terminals
 27              r = register               A general register.
 29             cc = cond code              Condition Code register.
 31            dbl = double                 A register pair (even/odd).
 33            flt = floating               Single precision floating point.
 35           dflt = dbl float              Double precision floating point.
 37           qflt = qd float               Quadruple precision floating point.
 39          $Terminals
 41             lng = length                 Length of operand.
 43             cnt = count                  Amount of a shift (or CSE usage).
 45             lbl = lbl value              The number of a label.
 47             dsp = displacement           Displacement of operand (< 4096).
 49            cond = condition              A condition  (i.e.  LT, EQ, GE ...)
 51           error = error num              The error value for an abort.
 53            stmt = stmt num               Internal statement number.
 55           elmnt = element                Constant element of variable.
 57           value = v value                A constant to be loaded.
 59             cse = cse                    Common Sub Expression number.
 62          $Operators
         addr, fullword, hlfword, byteword, typeword realword, dblrealword, quadrealword,
         iadd, isub, imult, idiv, imod, icompare, iabs, imax, imin, iodd, assign,
         long assign, var assign, clear, decr, incr, pos constant, neg constant,
         abort op, statement, case check, uninit check, range check, subscript_check,
         boolean or, boolean and, boolean not, boolean test, test bit value,
         set bit value, store bit value, clear bit value, load bit value, radd, rsub,
         rmult, rdiv, rabs, rneg, rcompare, halve, rmin, rmax, s x cnvrt, x s cnvrt,
         d x cnvrt, x d cnvrt, s d cnvrt, d s cnvrt, l shift, r shift, branch op,
         label def, label index, case_index, procedure call, procedure_entry,
         procedure exit, name param,
         reference param, make_common, use_common
100          $Opcodes
         spm, balr, bctr, bcr, mvcl, clcl, lpr, lnr, ltr, lcr, nr, clr, o r, xr, lr, cr, ar,
         sr, mr, dr, alr, slr, lpdr, lndr, ltdr, lcdr, hdr, lrdr, mxr, mxdr, ldr, cdr, adr,
         sdr, mdr, ddr, awr, swr, lper, lner, lter, lcer, her, lrer, axr; sxr, ler, cer, aer,
         ser, mer, der, aur, sur, sth, la, stc, ic, ex, bal, bct, bc, lh, ch, ah,sh, mh, st,
         n, cl, o, x, l, c, a, s, m, d, al, sl, std, mxd, ld,cd, ad, sd, md, dd, aw, sw, ste,
         le, ce, ae, se, me, de, au, su,bxh, bxle, srl, sll, sra, sla, srdl, sldl, srda,
         slda, stm, tm,mvi, ni, cli, oi, xi, lm, clm, stcm, icm, mvc, nc, clc, oc,
         xc,tr, trt
115          $Constants
118          *  Semantic opcodes for the code generator.
         label location, label pntr, branch, branch indexed, skip, case load, abort,
         stmt record, list request, modifies, ignore lhs, IBM length, push odd,
         push even, load odd addr, load odd full, load odd half, load odd reg,
         load extended, store extended, clear extended,
134           *    Common sub expressions.
         full common, half_common, byte_common, real_common, dreal_common, find_common,
         find real common,
139           *   Plain ole' boring constants.
         false const, true const, false cond, true cond,
143           *         False cond = 8 ;  True cond = 7 ;
         zero, one, two, three, four, seven, eight, fifteen, shift32, lt, lte, eq, ne, gt,
         gte, unconditional, underflow, overflow, not initialized, array underflow,
         array overflow, case low, case high, one loc, minus one loc, bitmasks,
         save area,entry code, code base, stack base, pr base, scratch, old base

*                           Numbering conventions.
* | - Line Number
*     | - Production Number
*         | - Template Number

159          $Productions
161          ****************************************
162          *     Assignment Templates.
163          ****************************************
164   1      lambda ::= assign fullword dsp.1 r.1 r.2
165      1     st  r.2,dsp.1(zero,r.1)

173   4      lambda ::= assign hlfword r.3 dsp.1 r.1 r.2
174      4     sth r.2,dsp.1(r.3,r.1)

191  10      lambda ::= assign r.1 r.2 lng.1
192     10     IBM length   lng.1                            Need IBM length
193     11     mvc zero(lng.1,r.1),zero(r.2)
```

```
11          lambda ::= long_assign r.1 r.2 lng.1
              using dbl.1,dbl.2
        12    IBM_length   lng.1
        13    load_odd_addr   dbl.1,lng.1(zero,zero)       Load counters.
        14    load_odd_addr   dbl.2,lng.1(zero,zero)
        15    lr  dbl.1,r.1
        16    lr  dbl.2,r.2
        17    mvcl dbl.1,dbl.2

12          lambda ::= var_assign r.1 r.2 r.3
            * r.1 is the address of the target
            * r.2 is the address of the source
            * r.3 is the computed size of the move
              using dbl.1,dbl.2
        18    load_odd_reg   dbl.1,r.3
        19    load_odd_reg   dbl.2,r.3
        20    lr  dbl.1,r.1
        21    lr  dbl.2,r.2
        22    mvcl dbl.1,dbl.2

            ****************************************
            *      Loading Templates.
            ****************************************
17          r.2 ::= fullword dsp.1 r.1
              using r.2
        29    l   r.2,dsp.1(zero,r.1)

18          r.2 ::= fullword r.3 dsp.1 r.1
              using r.2
        30    l   r.2,dsp.1(r.3,r.1)

            ****************************************
            *      Addition Templates.
            ****************************************
32          r.1 ::= iadd r.1 r.2
        56    modifies r.1
        57    ar r.1,r.2

33          r.2 ::= iadd fullword dsp.1 r.1 r.2
        58    modifies r.2
        59    a  r.2,dsp.1(zero,r.1)

35          r.2 ::= iadd r.2 fullword dsp.1 r.1
        62    modifies r.2                                  Commutative template.
        63    a  r.2,dsp.1(zero,r.1)

41          r.3 ::= iadd byteword dsp.1 r.1 r.2
              using r.3
        74    xr  r.3,r.3
        75    ic  r.3,dsp.1(zero,r.1)
        76    ar  r.3,r.2

42          r.4 ::= iadd byteword r.3 dsp.1 r.1 r.2
              using r.4
        77    xr  r.4,r.4
        78    ic  r.4,dsp.1(r.3,r.1)
        79    ar  r.4,r.2

            ****************************************
            *   Division Templates
            ****************************************
63          r.1 ::= r_shift r.1 cnt.1
       144    modifies r.1
       145    sra    r.1,cnt.1

64          r.2 ::= idiv r.2 fullword dsp.1 r.1
              using dbl.1
       146    lr     dbl.1,r.2
       147    srda  dbl.1,shift32
       148    d     dbl.1,dsp.1(zero,r.1)
       149    push_odd   dbl.1      Push odd register onto stack.
       150    ignore_lhs

66          r.2 ::= idiv fullword dsp.1 r.1 r.2
              using dbl.1
       156    l     dbl.1,dsp.1(zero,r.1)
       157    srda  dbl.1,shift32
       158    dr    dbl.1,r.2
       159    push_odd   dbl.1      Push odd register onto stack.
       160    ignore_lhs

            ****************************************
            *  Standard functions
            ****************************************
74          r.1 ::= iabs r.1
       202    modifies r.1
       203    lpr  r.1,r.1

75          r.1 ::= imax r.1 r.2
       204    modifies r.1
              using r.3
       205    cr    r.1,r.2
       206    skip gte,two,r.3
       207    lr    r.1,r.2
```

```
          ****************************************
          *   Label and Branching Templates
          ****************************************
89        lambda ::= label_def lbl.1
     252    label_location lbl.1                    Label definition found.

90        lambda ::= label_index lbl.1
     253    label_pntr  lbl.1                       Set up pointer to label.

91        lambda ::= branch_op lbl.1 cond.1 cc.1
            using r.3
     254    branch   cond.1,lbl.1,r.3               Branch instruction.

92        lambda ::= branch_op lbl.1
            using r.3
     255    branch   unconditional,lbl.1,r.3        Jump instruction.

93        lambda ::= case_index lbl.1 r.1
            using      r.3
     256    sll        r.1,two
     257    case_load  r.1,lbl.1,r.3                A branch table load.
     258    bc         unconditional,zero(r.1,code_base)

          ****************************************
          *   Procedure Calls, Entry and Exit.
          ****************************************
94        lambda ::= procedure_call cnt.1 fullword dsp.1 r.1
            need r.14,r.15
     259    list_request    cnt.1
     260    l    r.15,dsp.1(zero,r.1)
     261    balr r.14,r.15

95        lambda ::= procedure_entry
            need r.14,r.15
     262    stm  r.14,stack_base,save_area(stack_base)
     263    bal  r.14,entry_code(pr_base)      Build new stack frame.

96        lambda ::= procedure_exit
            using r.14
     264    l   stack_base,old_base(stack_base)  Get old stack frame
     265    lm  r.14,stack_base,save_area(stack_base) Restore all registers.
     266    bcr unconditional,r.14

          ****************************************
          *   Integer Comparison Templates
          ****************************************
107       cc.1 ::= icompare r.1 r.2
            using cc.1
     284    cr r.1,r.2

108       cc.1 ::= icompare r.2 fullword dsp.1 r.1
            using cc.1
     285    c  r.2,dsp.1(zero,r.1)

          ****************************************
          *  Checking templates
          ****************************************
124       r.3 ::= range_check r.3 fullword dsp.1 r.1 fullword dsp.2 r.2
            need    r.14
     315    c       r.3,dsp.1(zero,r.1)
     316    bal     r.14,underflow(pr_base)
     317    c       r.3,dsp.2(zero,r.2)
     318    bal     r.14,overflow(pr_base)

125       r.3 ::= range_check r.3 r.1 r.2
            need    r.14
     319    cr      r.3,r.1
     320    bal     r.14,underflow(pr_base)
     321    cr      r.3,r.2
     322    bal     r.14,overflow(pr_base)

          ****************************************
          *  Boolean Templates.
          ****************************************
128       r.1 ::= cond.1 cc.1
          *  This puts the condition code into a register.
            using r.1,r.3
     331    la    r.1,zero(zero,zero)       Since this doesn't affect Cond code
     332    skip  cond.1,two,r.3            R.3 used for long branch
     333    la    r.1,one(zero,zero)

129       lambda ::= assign byteword dsp.1 r.1 cc.1
            using r.3
     334    mvi   dsp.1(r.1),false_const
     335    skip  false_cond,two,r.3
     336    mvi   dsp.1(r.1),true_const

131       cc.1 ::= boolean_and byteword dsp.1 r.1 byteword dsp.2 r.2
            using cc.1,r.3
     341    tm    dsp.1(r.1),one
     342    skip  false_cond,two,r.3
     343    tm    dsp.2(r.2),one
```

```
132          cc.1 ::= boolean_and r.1 byteword dsp.2 r.2
               using cc.1,r.3
       344     modifies r.1
       345     tm     dsp.2(r.2),one
       346     skip  false_cond,two,r.3
       347     n      r.1,one_loc(zero,pr_base)

             ****************************************
             *  Set manipulation templates.
             ****************************************
142          cc.1 ::= test_bit_value addr dsp.1 r.1 elmnt.1
                using cc.1
       372      tm  dsp.1(r.1),elmnt.1

143          cc.1 ::= test_bit_value r.1 elmnt.1
                using cc.1
       373      tm  zero(r.1),elmnt.1

144          cc.1 ::= test_bit_value addr dsp.1 r.1 r.2
                using cc.1,r.3
       374      modifies r.2
       375      lr    r.3,r.2
       376      srl   r.2,three                   DIV 8 -- (byte index in set).
       377      n     r.3,seven(zero,pr_base)     MOD 8 -- (bit index in byte).
       378      ic    r.2,dsp.1(r.2,r.1)          Load byte from set.
       379      sll   r.3,two                     * 4 (for fullword index)
       380      n     r.2,bitmasks(r.3,pr_base)   Test if bit set.

145          cc.1 ::= test_bit_value r.1 r.2
                using cc.1,r.3
       381      modifies  r.2
       382      lr    r.3,r.2
       383      srl   r.2,three                   DIV 8 -- (byte index in set).
       384      n     r.3,seven(zero,pr_base)     MOD 8 -- (bit index in byte).
       385      ic    r.2,zero(r.2,r.1)           Load byte from set.
       386      sll   r.3,two                     * 4 (for fullword index)
       387      n     r.2,bitmasks(r.3,pr_base)   Test if bit set.

             *  Setting a bit value in a set.
146          lambda ::= set_bit_value addr dsp.1 r.1 elmnt.1
                using cc.1
       388      oi  dsp.1(r.1),elmnt.1

147          lambda ::= set_bit_value r.1 elmnt.1
                using cc.1
       389      oi  zero(r.1),elmnt.1

148          lambda ::= set_bit_value addr dsp.1 r.1 r.2
                using cc.1,r.3,r.4
       390      modifies r.2
       391      lr    r.3,r.2
       392      srl   r.2,three                   DIV 8 -- (byte index in set).
       393      n     r.3,seven(zero,pr_base)     MOD 8 -- (bit index in byte).
       394      ic    r.4,dsp.1(r.2,r.1)          Load byte from set.
       395      sll   r.3,two                     * 4 (for fullword index)
       396      o     r.4,bitmasks(r.3,pr_base)   Or in bit setting.
       397      stc   r.4,dsp.1(r.2,r.1)          Store updated byte.

149          lambda ::= set_bit_value r.1 r.2
                using cc.1,r.3,r.4
       398      modifies r.2
       399      lr    r.3,r.2
       400      srl   r.2,three                   DIV 8 -- (byte index in set).
       401      n     r.3,seven(zero,pr_base)     MOD 8 -- (bit index in byte).
       402      ic    r.4,zero(r.2,r.1)           Load byte from set.
       403      sll   r.3,two                     * 4 (for fullword index)
       404      o     r.4,bitmasks(r.3,pr_base)   Or in bit setting.
       405      stc   r.4,zero(r.2,r.1)           Store updated byte.
```