Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

A NEW METHOD FOR COMPILER CODE GENERATION (Extended Abstract)*

R. Steven Glanville† and Susan L. Graham
Computer Science Division
University of California
Berkeley, California 94720

Abstract

An algorithm is given to translate a relatively low-level intermediate representation of a program into assembly code or machine code for a target computer. The algorithm is table driven. A construction algorithm is used to produce the table from a functional description of the target machine. The method produces high quality code for many commercially available computers. By replacing the table, it is possible to retarget a compiler for another kind of computer. In addition techniques are given to prove the correctness of the translator.

Introduction

Since the early history of compilers, researchers have attempted to systematize and automate the production of compilers. The most succesful aspect of this attempt has been syntax analysis. It is now commonplace to use a table-driven syntax analyzer which is automatically constructed from a generalized context-free grammar specifying the syntax of the source language [see, for example, Aho-Ullman 77]. Such an analyzer is easily obtained, provably correct, and modular.

Attempts have been made to simplify the production of compilers for a specific source language by organizing compilers so as to isolate target-machine specific aspects of the translation. It is then possible to retarget the compiler by changing those portions of the compiler which concern the architecture of the machine [see, for example, Welsh-Quinn 72 or Poole 74].

In this paper we describe a method for automatically generating a table-driven coder for a compiler from a functional description of the machine instructions. The technique we use is in certain respects similar to methods used for table-driven syntax analysis and has many of the same advantages -- namely, modularity, correctness, compactness, and ease of use. The code generator produces quite good code; in particular, it is able to detect the instances in which specialized instructions can be used. It is possible for the code generator to utilize many kinds of optimization information, should it be available. In addition, our methods should prove valuable in designing retargetable compilers.

In the sections that follow, we summarize previous work on this topic, give our code generation algorithm and our table construction algorithm, explain the way in which this code is incorporated into a compiler, and summarize our implementation experience. The discussions are necessarily brief because of the length limitation on this summary.

The reader should be aware that by code generation we mean not the entire synthesis of the source program but rather the restricted task of choosing a sequence of object code instructions. Thus such issues as choices of representation, storage allocation, and machine-independent optimization are outside the scope of this work.

Background

For our purposes, previous research in code generation falls into three classes. The first class are attempts to deal with code generation mathematically, usually in order to produce optimal or near-optimal code [Aho-Johnson 76, for example]. Significant results have been obtained. However, research has been with idealized models of computers. Real computers tend not to have mathematically clean instruction sets. (There are always special instructions that implement certain computations more efficiently). Thus this approach must be extended for use in production compilers. The approach is in certain respects complementary to our techniques and one can be incorporated into the other.

The other two classes of research have tended to focus on implementation methods for real computers, often with loss in efficiency of the generated code.

The second approach to code generation is to provide information about the computer in procedural form, using special purpose code generation languages and interpreters. Examples of this approach are the UNCOL ideas [Strong 58, Steel 61], the PL/I optimizer of Elson and Rake [Elson 70], the method developed for PL/C [Wilcox 71], and the work of [Donegan 73]. These methods are an improvement over strictly ad hoc techniques, but require considerable "hand-coding" of tedious low-level details, making correctness difficult to ascertain and retargeting a chore.

*Research sponsored by the National Science Foundation under Grant MCS74-07644-A03.

†Current address: General Systems Division, Hewlett-Packard, Santa Clara, CA 95050.

The third class of methods and the one into which our work falls uses information about the target machine supplied in a descriptive form or data base. The macro approach to code generation falls into this class, as does the MIT thesis of Miller [Miller 71] and the Yale dissertation of Weingart [Weingart 73]. Our approach to code generation was initially stimulated by the work of Miller and Weingart and we have built on their ideas.

The Coding Algorithm

The output of the "front end" of the compiler is assumed to be a linearized intermediate representation (IR) of the source program. The IR consists of a sequence of parenthesis-free prefix expressions. In such a representation, operators are followed by their operands. (For simplicity, we assume in this paper that all operators are unary or binary. The techniques we describe are readily extended to n-ary operators.) The implementation decision concerning representation and storage allocation, as well as all but the low-level optimizations, are already incorporated in the IR version of the program. Each instruction of the computer is described by a prefix expression together with certain "semantic" information and an assembly -- or machine -- language template. (The examples should make these notions clearer.) The coding algorithm performs a pattern-match similar to parsing in which the IR sequence of prefix expressions is translated to a sequence of instructions. However, the situation differs from syntax analysis in the following respects. Since most operators can access their operands in a variety of ways, the target machine description is normally ambiguous. Indeed, an important factor in code generation is the way in which these ambiguities are resolved. Secondly, the 'reduce' move of the code generator is considerably more complicated than in syntax analysis, since it selects among a variety of instructions or instruction sequences on the basis of both syntactic and semantic information. Finally, error situations signalled by the code generator always signify compiler bugs.

As an example, Figure 1 describes a small set of machine instructions. Notice that in form the instruction descriptions resemble context free grammar rules. (For each instruction a corresponding assembly language instruction is also given.) The symbol to the left of '::=' designates the destination of the result of the computation; the prefix expression to the right describes the instruction computation. (The parentheses are meta-symbols used for readability.) A left hand side of lambda indicates that there is no resulting register value, i.e. the instruction is executed for its side effects.

1*	*r*.2 ::= (+ ↑ + *k*.1 *r*.1 *r*.2)	"**add**	*r*.2,*k*.1,*r*.1";
2*	*r*.1 ::= (+ *r*.1 ↑ + *k*.1 *r*.2)	"**add**	*r*.1,*k*.1,*r*.2";
3*	*r*.1 ::= (+ ↑ *k*.1 *r*.1)	"**add**	*r*.1,*k*.1";
4*	*r*.1 ::= (+ *r*.1 ↑ *k*.1)	"**add**	*r*.1,*k*.1";
5*	*r*.1 ::= (+ *r*.1 *r*.2)	"**add**	*r*.1,*r*.2";
6*	*r*.2 ::= (+ *r*.1 *r*.2)	"**add**	*r*.2,*r*.1";
7*	λ ::= (:= ↑ + *k*.1 *r*.1 *r*.2)	"**store**	*r*.2,**k*.1,*r*.1";
8*	λ ::= (:= + *k*.1 *r*.1 *r*.2)	"**store**	*r*.2,*k*.1,*r*.1";
9*	λ ::= (:= ↑ *k*.1 *r*.1)	"**store**	*r*.1,**k*.1";
10*	λ ::= (:= *k*.1 *r*.1)	"**store**	*r*.1,*k*.1";
11*	λ ::= (:= *r*.1 *r*.2)	"**store**	*r*.2,*r*.1";
12*	*r*.2 ::= (↑ + *k*.1 *r*.1)	"**load**	*r*.2,*k*.1,*r*.1";
13*	*r*.2 ::= (+ *k*.1 *r*.1)	"**load**	*r*.2,=*k*.1,*r*.1";
14*	*r*.2 ::= (+ *r*.1 *k*.1)	"**load**	*r*.2,=*k*.1,*r*.1";
15*	*r*.2 ::= (↑ *r*.1)	"**load**	*r*.2,**r*.1";
16*	*r*.1 ::= (↑ *k*.1)	"**load**	*r*.1,*k*.1";
17*	*r*.1 ::= (*k*.1)	"**load**	*r*.1,=*k*.1";

Fig. 1. Sample Instruction Set Description

By convention, 'r' designates a general purpose register and 'k' denotes a constant, typically an address offset. The store operator is := and the contents operator is ↑. The symbols following the '.' represent semantic qualifications. The qualifications on r.1 and r.2 indicate that they denote possibly distinct registers. The repetition of a qualification in an instruction indicates repetition of the same register or constant. Thus the first ADD instruction adds the value in the memory location addressed by the first constant plus the first register to the contents of the second register, leaving the sum in the second register. Observe that the commutativity of operands is indicated explicitly in instruction descriptions (rules 3 and 4, for example).

The addition of semantic restrictions to the instruction description allows a greater number of special instructions to be described. It also complicates the choice of output instructions in the shift-reduce code generation algorithm. A single syntactic instruction pattern may correspond to more than one instruction as the introduction of semantic restrictions may require the duplication of some instruction patterns by forcing some commutative operations to be represented by two identical patterns (e.g. add r.1,r.2 and add r.2,r.1) and by allowing one instruction to be specified as a special case of another (e.g. inc r.1 and add r.1 = k.1). It also may be that two distinct instructions compute the same expression but leave the result in registers of different classes (e.g. loadx x.1,α and loadr r.1,α).

The IR version of the source language consists of a sequence of prefix expressions composed of the same syntax symbols that are used in the description of the instruction set. However, in the IR, semantic qualifications carry specific information, whereas in the machine description they simply differentiate operators. The translation from source language to IR will in general depend on the semantics of the source language and the implementation decisions. For instance, in a block-structured language, the statement A := B+C might be:

:= + k.a r.7 + ↑ + k.b ↑ r.7 ↑ k.c (*)

where a, b, c designate constants and r.7 is the local base register. (Thus by inference, the base of the data segment containing B is obtained by following a static link, A is local, and C has an absolute address.)

The overall structure of the algorithm is that of an LR(1)-like deterministic shift-reduce parser. The language generated by the target machine description "productions" or "rules" with lambda replaced by X and the rule S ::= X^* added constitutes the language being analyzed and the IR is the input. Figure 2 gives the overall code generation

Algorithm 1 — The Code Generator

Input: ACTION and NEXT functions (represented in matrices) derived from machine **M**'s description, and the IR of a program, **P**, being compiled.

Output: An assembly language program for **P** on machine **M**.

Method: Perform a Shift-Reduce parse of the IR input, emitting target instructions whenever reductions are performed.

Initialization: Set the parser's state, **S**, and the stack to $\mathbf{q}_0$.

Step 1: Set the look-ahead symbol, u, to the next input symbol. If all the input has been read, set u to the end of input symbol, *$*.

Step 2: Perform the action in ACTION[**S**,u]:

shift: Push u onto the stack. Then set **S** to the value of NEXT[**S**,u], push **S** onto the stack, and advance the input one symbol. Go to step *1*.

reduce R: Output an instruction from set R or an equivalent sequence of instructions (see next section). If the rule used is '$\mathbf{r} ::= \alpha$' then pop the stack $2|\alpha|$ times and set **S** to the state on the top of the resulting stack. If $\mathbf{r} \neq \lambda$ then push **r** onto the stack, set **S** to the value of NEXT[**S**,**r**] and push **S** onto the stack. Go to step *2*†.

accept: The code generator halts. Code has been generated for the entire IR input string. This can only happen when all of the input has been read (i.e. u is *$*) and the stack is empty (i.e. **S** is $\mathbf{q}_0$).

error: Issue an error message and halt. The IR input has no parse using the underlying grammar for **M**'s instruction set.

†If $\mathbf{r} \in \mathbf{N}$, then ACTION[**S**,**r**] must be **shift**.

Fig. 2. The Code Generation Algorithm

algorithm, which is very similar to an LR(1) recognizer except for the special treatment of rules with left part λ and the code-emitting semantics of the reduce action.† Semantic information is carried along with the state information on the coder's stack. For example, when a shift is performed with the input symbol r standing for a register, the specific register represented by that r is also pushed onto the stack. Then when a reduce is done, the semantic information necessary to generate a final instruction can be read off the stack.

The instruction used when a reduce operation contains more than one instruction is picked by a simple heuristic. Instructions are ordered by the table constructor into a 'best instruction first' sequence. At code generation time, the instructions in the set for a specific reduce action are tested in that order until an instruction is found that is semantically compatible with the information on the top of the stack. Since all instructions in a reduce set have the same instruction pattern, the 'cheapest', according to some cost criteria, instructions are tested first. If the length of an object instruction in bits is used as the cost factor, a 16 bit long increment instruction will be tested before a 32 bit long add immediate instruction, since the basic patterns are identical but the 'cost' of the increment instruction is less. It is also possible, if all the instructions in the set have semantic restrictions, that more than one instruction is generated. This situation is discussed in the next section.

†For brevity, we have assumed that the reader is familiar with LR(1) parsing and its terminology.

The coding algorithm works in conjunction with a conventional register allocation routine. The machine description specifies how many and what kind of registers exist on the target computer, and which ones are available to the register allocator. Each nonterminal in the instruction set description, in general, represents a logical register or a class of logical registers. Each logical register is associated with an actual machine register or pair of registers. In this way information such as the fact that d0 is the register pair <r0,r1> is included in the machine description.

Register allocation occurs as a subtask to a reduce operation. After an instruction pattern has been semantically verified, if the result of the instruction is non-λ, and if the result register is not semantically linked to any other register in the instruction pattern, the register allocator provides a free register of the appropriate class, after setting its use count to 1. If the result register, r, is semantically linked to a register in the instruction pattern, then r must be used as the target register and its use count set to one (since it contains a newly computed value). The routine is easily generalized to incorporate information about reuse of common subexpressions.

Figure 3 contains the move table used by the code generator for the instruction set in Figure 1. In the table, the ACTION and NEXT functions are represented in the following format: Columns of the table are headed by language symbols and rows correspond to parser states. An entry in row q, column v of the table is of the form 'a:b', where a always represents ACTION(q,v) and b is NEXT(q,v) when a is shift, and the rule set by which to reduce when a is reduce. The ACTION values shift and reduce are abbreviated respectively S and R. A blank designates the error action.

Using the instruction set of Fig. 1, we show in Fig. 4 the sequence of steps taken by the code generator for the IR input expression (*). For each step we indicate the action, the resulting stack, and the intuitive current state. The instruction sequence produced by the algorithm is 15,16,1,8. The reader can observe that other code

Code Generator Move Table

	$	*r*	*k*	+	↑	:=
1*	ACCEPT					S: 2
2*		S: 3	S: 4	S: 5	S: 6	
3*		S: 7	S: 8	S: 9	S: 10	
4*		S: 11	S: 8	S: 9	S: 10	
5*		S: 12	S: 13	S: 9	S: 14	
6*		S: 15	S: 16	S: 17	S: 10	
7*	R: 11	R: 11	R: 11	R: 11	R: 11	R: 11
8*	R: 17	R: 17	R: 17	R: 17	R: 17	R: 17
9*		S: 12	S: 18	S: 9	S: 14	
10*		S: 15	S: 19	S: 20	S: 10	
11*	R: 10	R: 10	R: 10	R: 10	R: 10	R: 10
12*		S: 21	S: 22	S: 9	S: 23	
13*		S: 24	S: 8	S: 9	S: 10	
14*		S: 15	S: 25	S: 26	S: 10	
15*	R: 15	R: 15	R: 15	R: 15	R: 15	R: 15
16*		S: 27	S: 8	S: 9	S: 10	
17*		S: 12	S: 28	S: 9	S: 14	
18*		S: 29	S: 8	S: 9	S: 10	
19*	R: 16	R: 16	R: 16	R: 16	R: 16	R: 16
20*		S: 12	S: 30	S: 9	S: 14	
21*	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}
22*	R: 14	R: 14	R: 14	R: 14	R: 14	R: 14
23*		S: 15	S: 31	S: 32	S: 10	
24*		S: 33	S: 8	S: 9	S: 10	
25*		S: 34	S: 8	S: 9	S: 10	
26*		S: 12	S: 35	S: 9	S: 14	
27*	R: 9	R: 9	R: 9	R: 9	R: 9	R: 9
28*		S: 36	S: 8	S: 9	S: 10	
29*	R: 13	R: 13	R: 13	R: 13	R: 13	R: 13
30*		S: 37	S: 8	S: 9	S: 10	
31*	R: 4	R: 4	R: 4	R: 4	R: 4	R: 4
32*		S: 12	S: 38	S: 9	S: 14	
33*	R: 8	R: 8	R: 8	R: 8	R: 8	R: 8
34*	R: 3	R: 3	R: 3	R: 3	R: 3	R: 3
35*		S: 39	S: 8	S: 9	S: 10	
36*		S: 40	S: 8	S: 9	S: 10	
37*	R: 12	R: 12	R: 12	R: 12	R: 12	R: 12
38*		S: 41	S: 8	S: 9	S: 10	
39*		S: 42	S: 8	S: 9	S: 10	
40*	R: 7	R: 7	R: 7	R: 7	R: 7	R: 7
41*	R: 2	R: 2	R: 2	R: 2	R: 2	R: 2
42*	R: 1	R: 1	R: 1	R: 1	R: 1	R: 1

Fig. 3. Code Generation Table for Example

Input: := + k.a r.7 + ↑ + k.b ↑ r.7 ↑ k.c

1. shift Stack is ⊢ 1 := 2
   Have matched the first symbol of instructions 7-11.
2. shift ⊢ 1 := 2 + 5
   Matched first two symbols of rule 8, first of rules 1-6, 13, 14.
3. shift ⊢ 1 := 2 + 5 k.a 13
   Matched first three of rule 8, first two of rule 13, all of rule 17.
4. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24
   Matched first four of rule 8, all of rule 13.
5. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9
   Matched first symbol of rules 1-6, 13, 14.
6. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14
   Matched first two of rules 1 and 3, first of rules 12, 15, 16.
7. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26
   Matched first three of rule 1, first two of rule 12, first of rules 1-6, 13, 14.
8. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35
   Matched first four of rule 1, first three of rule 12, first 2 of rule 13, all of rule 17.
9. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 ↑ 10
   Matched first symbol of rules 12, 15, 16.
10. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 ↑ 10 r.7 15
    Matched all of rule 15.
11. reduce rule 15
    Assign result to r1.
    Emit "LOAD r1,*r7"
    shift left side ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 r.1 39
    Matched first five symbols of rule 1, all of rules 12, 13.
12. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 r.1 39 ↑ 10
    Matched first of rules 12, 15, 16.
13. shift ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 r.1 39 ↑ 10 k.c 19
    Matched all of rules 16 and 17.
14. reduce rule 16
    Assign result to r2.
    Emit "LOAD r2,c"
    shift left side ⊢ 1 := 2 + 5 k.a 13 r.7 24 + 9 ↑ 14 + 26 k.b 35 r.1 39 r.2
    Matched all of rule 1.
15. reduce rule 1
    Emit "ADD r2,b,r1"
    shift left side ⊢ 1 := 2 + 5 k.a 13 r.7 24 r.2 33
    Matched all of rule 8.
16. reduce rule 8
    Emit "STORE r2,a,r7"
    Stack now contains ⊢ 1. If more input remained, the code generator would read the next symbol. Since there is not, the algorithm terminates.

Fig. 4. Example of Code Generation

sequences (e.g. instructions 17,5,17,15,5,15,16,5, 11) are possible for this input.

The table construction algorithm sketched in the next section always produces a conflict-free table, hence the coding algorithm is always deterministic. The coding algorithm has the following aspects which are not obvious from the small example. A reduce action first checks the semantic restrictions (i.e. qualifications) such as specific constants, matching registers, even/odd-ness of the matched patterns (there may be more than one match), etc. When a particular instruction is selected, the register allocator is called if a result register (left hand side) is needed. In certain instances sketched in the next section, the reduce action emits more than one instruction. (Typically this happens if a long instruction is matched but its semantic constraints are not). The semantic qualifications usually arise in conjunction with specialized instructions. For example, an increment instruction is only applicable if the constant has value 1.

The correctness considerations are sketched in the next section. The reader can see that among them are the requirement that the IR vocabulary be a subset of the target machine vocabulary and that the operators have the same computational meaning in both.

The Table Construction Algorithm

The table constructor first treats the target machine descriptions as context-free grammar rules (ignoring semantics) and constructs the set of LR(0) states. (In Fig. 1, r (register) is the only nonterminal. The example generates 42 states.) There are many "inadequate states" caused by shift-reduce or reduce-reduce conflicts. Most of these conflicts represent ambiguities. For instance, in the example, rule 16 followed by rule 5 is the equivalent of rule 4. The construction algorithm seeks to resolve these conflicts in favor of short instruction sequences. We use the approach in [Aho-Johnson-Ullman 75] for the resolution of these ambiguities -- namely, shift-reduce conflicts are resolved by shifting. This heuristic usually causes the use of "more powerful" instructions. Reduce-reduce conflicts are often resolved by the semantic restrictions. If not, the longest instruction is used. These heuristics are reflected in the example in steps 3, 4, 8 and 13. The initial phase of the table constructor, in which the conflicts are resolved in this way, is given in Fig. 5. In resolving shift-reduce conflicts, SLR(1) lookahead information is used to insure that reduce actions are included where needed.

Given a general context-free grammar, the conflict-resolution rules would not necessarily yield a recognizer for the entire language generated by the grammar. However, the construction is language-preserving for uniform instruction sets. (A proof is contained in Glanville 77).

Let V be the set of vocabulary symbols for the instruction set description. An instruction set is said to be uniform if it satisfies the following condition: Any left (similarly right) operand of a binary operator b is a valid left (respectively right) operand of b in any prefix expression of V* containing b. Any operand of a unary operator u is a valid operand of u in any prefix expression of V* containing u. An instruction set is uniform if its description is uniform. The essential idea of uniformity is that operands to an operator are valid independent of context.

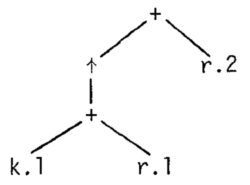
We give some examples to clarify this notion. Consider the instruction set described in Fig. 1. The operands of := are all either registers (rule 11) or special cases of registers, as evidenced by rules 12, 13, 16, and 17. The operands of the leftmost + in rules 1-6, 13, 14 are all either registers (rules 5 and 6) or prefix expressions which become registers by rules 12, 16, or 17. Since a register is an argument of ↑ (rule 15) and of + (rule 5), the arguments of the other occurrences of + and ↑ represent special cases (i.e. ambiguities). In summary, whenever :=, +, and ↑ occur, their operands are any prefix expressions corresponding to registers. Note that in this example, both + and := have the same set of left operands as right operands. Such a situation need not be true in general.

An example of an instruction set which is not uniform is:

$$r ::= (+ \uparrow k\, k)$$
$$r ::= (+ k \uparrow k)$$
$$\lambda ::= (:= r\, r)$$

In this example k is a valid first operand of + only if the second operand is '↑k' and vice-versa.

Let e1...ei...ek be a prefix expression, where $k \geq 2$ and $1 \leq i \leq k$. As is well known, the prefix expression corresponds to a (unique) binary tree with nodes labelled by the ei's, where e1 labels the root and non-leaf nodes are labelled by operators. The subtrees rooted by children of an operator node represent its operands (see, for example, [Meyers 74]). For example the right hand side of the first instruction in Fig. 1 corresponds to the tree



In order to check that an instruction set is uniform, we define the relations which are tree-equivalents of the usual string relation FOLLOW. Using these tree relations, we obtain a simple test for uniformity which is incorporated in the table constructor.

The definitions are presented with respect to a particular instruction description G. Recall that V is the set of operator and operand symbols used in the instruction description. Let $B \subseteq V$ be the binary operators; let $U \subseteq V$ be the unary operators. Define the relation LEFT on $(B \cup U) \times V$ as:

b LEFT v iff
$\exists$ an instruction $r ::= \alpha b v \beta$,
where r may be λ.

Similarly, define the relation RIGHT on $B \times V$ as:

b RIGHT v iff
$\exists$ an instruction $r ::= \alpha b \beta v \gamma$
for some prefix expression $\beta \neq \lambda$,
where r may be λ.

Clearly, b LEFT v if v is the leftmost symbol of a left operand of b in some instruction and b RIGHT v if v is the leftmost symbol of a right operand of b in some instruction. By convention, unary operators have left operands.

To extend these notions from operands in the same instruction to all operands, we need the usual grammatical leftmost and rightmost descendent

Algorithm 2 — The Initial SLR(1) Code Generator Constructor

Input: A TMDL instruction set description for machine **M**.

Output: The initial ACTION and NEXT functions (represented in matrices) for a code generator for machine **M**. An error message if the instruction set is not uniform.

Method: Construct a set of states for the context free grammar underlying the given instructions. As it is being done, fill in the NEXT and ACTION values to produce the code generator, resolving conflicts. Test each state for appropriate LEFT FIRST and RIGHT FIRST actions.

```
procedure generatestates;
begin
    $q_0 \leftarrow close(\{i \in \mathbf{I} \mid i = [\lambda \rightarrow \cdot\ \alpha]\})$;
    $k \leftarrow 0$;  $n \leftarrow 0$; /* highest state number */
    while $k \leqslant n$ do begin
        /*compute successors of $q_k$*/
        $\forall\ v \in \mathbf{V}$ do
            if $\exists\ i \in q_k$ with $i = [x \rightarrow \alpha \cdot v \beta]$ then
                ACTION[$q_k$,$v$] ← shift;
                $q' \leftarrow close(\{[x \rightarrow \alpha\ v \cdot \beta] \mid [x \rightarrow \alpha \cdot v \beta] \in q_k\})$;
                if $\exists\ q_j = q'$ then NEXT[$q_k$,$v$] = $q_j$
                else $n \leftarrow n + 1$; $q_n \leftarrow q'$; NEXT[$q_k$,$v$] ← $q_n$;
            else if $\exists\ i \in q_k$ with $i = [x \rightarrow \alpha\ \cdot]$ then
                $R \leftarrow \{[x \rightarrow \alpha\ \cdot] \in q_k \mid v \in$ FOLLOW($x$), $\nexists\ i \in q_k$, $i = [x' \rightarrow \alpha'\ \cdot]$ such that
                    both length($\alpha'$) > length($\alpha$) and $v \in$ FOLLOW($x'$)$\}$;
                ACTION[$q_k$,$v$] ← reduce $R$;
            else ACTION[$q_k$,$v$] ← error;
        /*uniformity check for $q_k$*/
        $\forall$ items of the form $[r \rightarrow \alpha \cdot v \beta] \in q_k$, $\alpha \neq \lambda$ do begin
            $x \leftarrow parent(v, \alpha v \beta)$;
            if $leftchild(v, \alpha v \beta)$ then
                $\forall\ u$ with $x$ LEFT FIRST $u$ do begin
                    if ACTION[$q_k$,$u$] = error then
                        output("Not Uniform");
                end;
            else  /* $x \in \mathbf{B}$ and $v$ begins the second operand */
                $\forall\ u$ with $x$ RIGHT FIRST $u$ do begin
                    if ACTION[$q_k$,$u$] = error then
                        else output("Not Uniform");
                end;
        end;
        /*advance to next state*/
        $k \leftarrow k + 1$;
    end;
    ACTION[$q_0$,\$] ← accept;
end generatestates;

function close(q);
begin
    repeat
        $q \leftarrow q + \{[x \rightarrow \cdot\ \alpha] \mid [y \rightarrow \beta \cdot x \gamma] \in q, x \neq \lambda\}$
    until $q$ does not change;
    return($q$);
end close;
```

Fig. 5. Initial Table Constructor

relations. Define the relation FIRST on $V \times V$ as:

u FIRST v iff
$\exists$ a derivation $u \Rightarrow^* v\alpha$
for some $\alpha \in V^*$, $u \in V$,
where derivations are defined in the usual way.

Thus u FIRST v if there is some derivation in G from u that yields v as the leftmost symbol. Similarly, define LAST on $V \times V$ as:

u LAST v iff
$\exists$ a derivation $u \Rightarrow^* \alpha v$
for some $\alpha \in V^*$, $u \in V$.

By the usual product of relations, b LEFT FIRST v iff v can appear as the first symbol in the left operand to b in a derivation in G, and b RIGHT FIRST v iff v can appear as the first symbol in the right operand to b in a derivation in G. Using these definitions, we obtain the following theorem. A more formal statement of the theorem and a proof appear in [Glanville 77].

<u>Theorem 1</u>. Let G be an instruction set description. Let the code generator tables be computed by the algorithm in Fig. 4, and let Q be the corresponding set of states. The following conditions are equivalent.

1) G is a uniform instruction set.

2a) For all u, v in V, if u LEFT FIRST v, then for every state in Q containing an item $[r \rightarrow \alpha \cdot x\beta]$, for some r, α, x, β, where u is the parent of x in the tree corresponding to $\alpha x \beta$ and x is the left child of u, it is the case that ACTION[q,v] = shift.

b) For all u, v in V, if u RIGHT FIRST v, then for every state in Q containing an item $[r \rightarrow \alpha \cdot x\beta]$ for some r, α, x, β, where u is the parent of x in the tree corresponding to $\alpha x \beta$ and x is the right child of u, it is the case that ACTION(q,v) $\in$ {shift, reduce R}.

In other words, the theorem says that if the grammar is uniform, then if a left (respectively, right) operand of an operator u is expected in a given context, then the first symbol of any possible left (respectively, right) operand of u is legal and conversely.

It is the test suggested by this theorem that is incorporated in the table constructor. In that test, parent(v,$\alpha v \beta$) returns the parent of v in the tree corresponding to the prefix expression $\alpha v \beta$. The predicate leftchild(v,$\alpha v \beta$) returns <u>true</u> if v is the left child of its parent in the tree corresponding to the prefix expression $\alpha v \beta$ and false otherwise.

Note that it suffices to check for error because only state q_0 can have an accept action. Also note that the possibility of a reduce in 2a) is ruled out by the form of the instruction descriptions (i.e. u must be an operator and must be the last symbol in α, and no right hand side of an instruction description can end in an operator).

Since it is the relations LEFT FIRST and RIGHT FIRST which characterize uniform instruction sets, the reader may wonder why the table constructor instead uses the string relation FOLLOW in computing ACTION and NEXT. The reason is that the use of FOLLOW is considerably simpler computationally and yields tables which are equivalent except for deferred error detection (see [Glanville 77] for a proof). Since errors stem only from compiler bugs, delayed error detection seems acceptable. For instruction set descriptions, we can define FOLLOW by: For $u \in V$, let

$$\text{FOLLOW1}(u) = \{v \mid \text{for some } r ::= \alpha x y \beta, \text{ where } r \text{ may be } \lambda,\ x \text{ LAST } u \text{ and } y \text{ FIRST } v\}\ .$$

Let

$$\lambda\text{-LAST} = \{u \mid \exists \text{ instruction } \lambda ::= \alpha x \text{ for some } \alpha \in V^* \text{ and } x \text{ LAST } u\}\ .$$

(It is easily shown that λ-LAST consists only of operand symbols.) Then for $u \in V$,

$$\text{FOLLOW}(u) = \begin{cases} \text{FOLLOW1}(u) & \text{if } u \notin \lambda\text{-LAST} \\ \text{FOLLOW1}(u) \cup \{\text{root-level operators}\} & \text{if } u \in \lambda\text{-LAST}\ . \end{cases}$$

The root-level operators are those which occur as leftmost operators of instructions with destination λ.

Using the notion of uniformity, it can be shown that

<u>Theorem 2</u>. Let G be an instruction description. Then Algorithm 2 declares error if and only if G is not uniform. If G is uniform, then Algorithm 1 using the tables generated by Algorithm 2 but disregarding semantic qualifications fails to reach the ACCEPT state and to generate code for an input IR only if

1) the code generator loops

2) the input IR is not syntactically within the sequence of prefix expressions described by the instruction set.

It remains to eliminate the possibility of looping, to give sufficient conditions for the IR input, and to deal with the semantic issues. We consider these topics in turn.

Since the prefix expression describing an instruction contains at least one symbol, looping could occur only because of a sequence of "chain-reductions" corresponding to register-to-register moves. The possibility of potential looping is easily detected. Each potential loop is then broken by a kind of 'state-splitting' of some of the states computed by Algorithm 2. (Details are contained in [Glanville 77]).

As previously described, the reduce action generates code by checking the semantic qualifications of the matched instruction pattern against the semantic qualifications indicated on the stack. If the set of instructions associated with a given reduce action are all semantically constrained, it is possible that none of the instructions will be compatible with the semantics on the stack. In this case, the code generator would <u>semantically block</u>. Semantic blocking can be avoided by the use of a default list of shorter instructions that contain no semantic restrictions and together compute the desired expression. Consider the memory-to-memory add instruction:

```
λ ::= (:= k.1 + ↑ k.1 ↑ k.2)   "madd k.2,k.1"
λ ::= (:= k.1 + ↑ k.2 ↑ k.1)   "madd k.2,k.1"
```

The basic instruction pattern is ':= k + ↑ k ↑ k'. If this pattern occurs in the IR, but the constants associated with each of the three k's are distinct, then the instruction cannot be used. If there are instructions with the patterns 'r ::= ↑ k', 'r ::= + ↑ k r', and 'λ ::= := k r', then they could be issued, simulating a non-restricted memory-to-memory add instruction, and the code generation could proceed from there as though the longer instruction had been issued.

Default instruction lists for all reduce R actions having no semantically unrestricted instructions are constructed by the table constructor. The lists are obtained by simulating the action of the coder using as input the right hand side of the semantically restricted instruction and using only those instructions that are shorter than the one under consideration. The construction, in effect, builds a code generator for the subset of shorter instruction patterns and generates code for the restricted instruction (cf. Algorithm 4.8 of [Glanville 77]).

In the presence of semantically restricted instruction patterns, the code generation Algorithm 1 will choose from a list of instructions in step 2 when a reduce action is performed. Assuming that this list has already been sorted by the table constructor, the instructions must be tested sequentially until one is found that is semantically compatible. In the event that no instruction is acceptable, the default list of instructions is used to implement that computation.

There are several classes of semantic restrictions that may have to be satisfied. Constants in the IR input may have to be equal to specific values, such as 1 in an increment instruction, and logical registers may have to be equivalent to specific actual registers. Multiple occurrences of a symbol in any class may have to refer to the same actual value or register. If the result is to appear in a register, then there must not be any references to the value in that register outside of that instruction pattern. Such references could exist only if some sort of common subexpression elimination has been done. Finally, any additional semantic restrictions required to properly describe a particular target computer may be added to the code generator. The proof of correctness only requires that a default instruction or list of instructions be available for each restricted instruction, so that it will always be possible to generate code.

If two instructions have the same instruction pattern but different result locations, the actual instruction used is arbitrary. Adherence to the uniformity condition insures that the coder will still accept the input regardless of the choice made. The item $[x \rightarrow \beta \cdot]$ will not be included in a state of a uniform instruction set unless a reduction using it on a valid input is still valid. For practical reasons, one may preorder the instructions by cost, by the instruction that leaves the result in a register class with more actual registers (in an attempt to avoid having to save registers in temporaries), or by any other ordering desired, and the coder will still function correctly. It is also possible to examine the operator for which the register is an operand, to determine the instruction that leaves the result in a location that is closer to a valid operand to that operator, thus possibly avoiding a subsequent register move instruction.

It is shown in [Glanville 77] that if looping and semantic blocking are eliminated by the extensions to the table constructor sketched above, and if the instruction description accurately describes the target machine, then the code generator produces correct code for all well-formed input.

The input IR is well-formed if

1) the input is a sequence of prefix expressions;

2) the operators and operands of the IR are from the same set as the operators and operands of the instructions, and have the same meaning;

3) the sequence of prefix expressions is valid, that is, it is in the language 'generated' or described by the instruction set.

Condition 2 must, of course, be checked by the implementor. The implementor either can prove that the routines generating the IR (i.e. the 'front end' of the compiler) meet specification 1), or can provide a simple routine to test the input to the code generator. Fortunately, if the instruction set is uniform, condition 3 is also relatively easy to check.

Let $LEFT_{IR}$, $RIGHT_{IR}$, and $FIRST_{IR}$ be the relations satisfied by the IR. Let $LEFT_{CG}$, $RIGHT_{CG}$, and $FIRST_{CG}$ be the relations of a uniform instruction description described previously. Then an intermediate representation (IR) is valid for a code generator (CG) provided that:

$$LEFT_{IR} \subseteq LEFT_{CG} ,$$
$$RIGHT_{IR} \subseteq RIGHT_{CG} , \text{ and}$$
$$FIRST_{IR} \subseteq FIRST_{CG} .$$

Since the compiler probably cannot generate all IR expressions generated (i.e. computed) by a uniform instruction set, the inclusion might be proper in some instances.

It is the responsibility of the implementor (or of some other part of a compiler-writing system) to specify $LEFT_{IR}$, $RIGHT_{IR}$, and $FIRST_{IR}$. However, specifying these relations is considerably simpler than characterizing the set of strings that are possible IR expressions.

How the Code Fits into a Compiler

It should be clear to the reader that the code generator is not a separate pass in a compiler but is conceptually a co-routine. The purpose of the code generator is to isolate and automate the low-level decisions of instruction selection.

As far as retargetable compilers are concerned, the coder does not encapsulate all of the machine-dependent aspects of a compiler. However, many of the remaining factors (word size, number and versatility of registers, etc.) are relatively easily specified via a checklist and a compiler designed to be retargetable can be reconfigured using the larger architectural information. For example, in a statically-named language one might use a display in registers if there were many registers and a linked list otherwise -- the choice depends on a simple consideration and, for a given source language, few possibilities exist. By a kind of conditional compilation of the compiler, one can choose the alternative for a given target machine. These issues are discussed further in [Glanville 77].

Most code optimization can also be done at a stage prior to generation of the IR input, particularly if an architectural checklist is used. For example, reordering of expression trees to evaluate more complex arguments to a binary operator first, recognition of common subexpressions, etc. can all be done earlier (hence in a portable fashion). By attaching semantic information to the operands in the IR, usage information can be used to guide the register allocator.

A slight addition to the IR and the register allocation scheme is required to utilize common subexpression (CSE) information. A binary operator, O, is added to the IR that takes an integer constant as its first operand and an arbitrary expression as its second. The meaning of this operator is that the second operand is a common subexpression that is to be used the number of times specified by its first operand. Each occurrence of the operator O, i.e. each designation of an expression as a CSE, is implicitly numbered sequentially from 1 as it is read. The first place that a CSE is used is where it is computed, as an ordinary operand. Subsequent uses are indicated by a 'use CSE' operator, ⊗, which is a unary operator taking a constant as its operand, indicating which CSE it represents. For example, the two statements:

A := B + C ;   D := B + C ;

would have an IR representation of:

:= k.aOk.2 + ↑k.b↑k.c := k.d⊗k.1

Upon encountering a define CSE operator, the code generator sets the use count of the register containing the CSE's value to the number of times that it will be used. Thus, that value will be preserved until its final use because that register will remain busy. This is equivalent to adding a special grammar rule:

r.1 ::= (O k.1 r.1)

for each logical register class to the machine description with the semantics that the use count for register r.1 is set to k.1. The mechanism does not have to actually be implemented in this manner -- it is simpler to 'hard-wire' it into the code generator. Likewise, when a use CSE operator is encountered, the actual register that contains that value is substituted into the IR stream, equivalent to a grammar rule of the form

r.1 ::= (⊗k.1)

with the associated semantics of using the actual register that contains CSE number k.1 for r.1. As usual, post-coder peephole optimization can also be done (see, for example, [Wulf et al 75]).

Easy change of the target code is possible only if the IR remains mostly unchanged. In this case, one need only run a new machine description through the table constructor and change the tables. Since the set of inputs must be a subset of the language generated by each target machine description, our approach does not work well for fundamentally different architectures. It appears that one IR can be used with most general register machines of the IBM 370, PDP-11, PDP-10, UNIVAC-1108, variety, but that a different IR might be appropriate for true stack machines. Since most of the machines in use today are general register machines, this does not seem to be a serious drawback.

It appears that this approach to code generation is useful even in a compiler that is not to be retargeted. By incorporated the decision logic in the coding algorithm and the machine description in a table, the size of the synthesis phase of the compiler should be reduced. In addition, we conjecture that a compiler organized in this fashion will be easier to modify and maintain.

Experimental Results

A code generator and table constructor were implemented in PASCAL. Several target computer descriptions, including the IBM 370 and the PDP-11 were input to the table constructor. (Our first experiments were for the PDP-11. The change to the IBM 370 required little more than a new description like that of Fig. 1 and took about one hour.) PASCAL source programs were hand-translated into IR and input to the code generator. Although our experiments to date have not been extensive, it appears that the quality of the resulting code is very good. The PDP-11 code generator produces code that is competitive with the compiler for C, the Unix systems programming language [Ritchie 77]. It does this without any kind of prior optimizations of the IR input. While its primary task is code generation, and it is not intended to be an optimizer, the code generator is particularly good at finding specialized instruction patterns, such as increment or add to memory, in the IR. Since our PDP-11 code generator produces assembly code (as do most Unix compilers), the short jump instructions, etc., of the Unix Assembler as well as the peephole optimizer used by the C compiler can also be exploited. Similar remarks apply to the code generator for the IBM 370.

References

Aho, A.V., Johnson, S.C. & Ullman, J.D., Deterministic parsing of ambiguous grammars, CACM 18:8 (August 1975), 441-452.

Aho, A.V. & Johnson, S.C., Optimal code generation for expression trees, JACM 23:3 (July 1976), 488-501.

Aho, A.V. & Ullman, J.D., Principles of Compiler Design, Addison-Wesley, Reading, MA (1977).

Donegan, M.K., An approach to the automatic generation of code generators, Ph.D. Thesis, Rice University, Houston, Texas, May 1973.

Elson, M. & Rake, S.T., Code generation techniques for large-language compilers, IBM Sys. J. 9:3 (1970), 166-188.

Glanville, R.S., A machine-independent algorithm for code generation and its use in retargetable compilers, Ph.D. Thesis, Computer Science Division, University of California, Berkeley, November 1977.

Meyers, W.J., Linear representations of tree structure: a mathematical theory of parenthesis-free notations, Technical Report STAN-CS-74-222, Computer Science Department, Stanford University, Palo Alto, CA (July 1974).

Miller, P.L., Automatic creation of a code generator from a machine description, Technical Report MAC TR-85, Project MAC, MIT, Cambridge, MA (May 1971).

Poole, P.C., Portable and adaptable compilers, in Compiler Construction. An Advanced Course, G. Goos & J. Hartmanis, eds., Lectures Notes in Computer Science, vol. 21, Springer-Verlag, New York (1974), 427-497.

Ritchie, D.M., C Reference Manual, Bell Laboratories, Murray Hill, N.J. (April 1977).

Steel, T.B. Jr., A first version of UNCOL, Proc. WJCC 19 (1961), 371-378.

Strong, J. et al, The problem of programming communication with changing machines: a proposed solution, CACM 1:8 (August 1958).

Weingart, S.W., An efficient and systematic method of compiler code generation, Ph.D. Thesis, Yale University, New Haven, CT, 1973.

Welsh, J. & Quinn, C., A PASCAL compiler for ICL 1900 series computer, Software: Practice and Experience 2:1 (Jan.-Mar. 1972), 73-77.

Wilcox, T.R., Generating machine code for high-level programming languages, Technical Report 71-103, Department of Computer Science, Cornell University, Ithaca, NY (September 1971).

Wulf, W. et al., The Design of an Optimizing Compiler, American Elsevier Publishing Co., Inc., New York (1975).