Concise specifications of locally optimal code generators

Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Dynamic programming allows locally optimal instruction selection for expression trees. More importantly, the algorithm allows concise and elegant specification of code generators. Aho, Ganapathi, and Tjiang have built the Twig code-generator-generator, which produces dynamic-programming code-generators from grammar-like specifications.

Encoding a complex architecture as a grammar for a dynamic-programming code-generator-generator shows the expressive power of the technique. Each instruction, addressing mode, register and class can be expressed individually in the grammar. The grammar can be factored much more readily than with the Graham-Glanville LR(1) algorithm, so it can be much more concise. Twig specifications for the VAX and MC68020 are described, and the corresponding code generators select very good (and under the right assumptions, optimal) instruction sequences.

Limitations and possible improvements to the specification language are discussed.

1. Introduction

One of the last phases of a typical compiler is the translation of an intermediate representation of a compiled program into target-machine instructions. This *instruction selection* phase has been the subject of much research, with two primary goals: to find algorithms for selecting optimally efficient instruction sequences, and to find ways to automatically generate instruction selection programs from concise specifications of the target machine's operations.

This paper describes techniques for using Aho, Ganapathi, and Tjiang's *Twig[1],* a code-generator generator that uses tree matching and dynamic programming. The instruction sets of the DEC VAX and Motorola 68020 are used as a illustrative examples. The VAX architecture can be described with only 112 rules; the 68020 is somewhat more complicated.

Some of the problems of using Twig are described, and solutions to these problems are proposed.

1.1. Optimal instruction selection

An intermediate representation can be considered as a directed graph, in which edges point from *uses* of values to *sources* of values. Most of the work in optimal instruction selection has considered only trees -- graphs in which each source of a value has only one use. (The ''trivial'' sources, such as constants and the addresses of program variables, may have several uses; each use will be a leaf of the tree.) For more general graphs, optimal instruction selection becomes NP-complete[2, 3].

For trees, it is possible to find optimal solutions (with appropriate restrictions on the scope of the problem). For example, the problem of optimal register allocation[4, 5] can be solved using a simple bottom-up labelling algorithm. Optimal instruction selection for machines with one class of general registers is solved using a bottom-up dynamic programming algorithm[6] (the algorithm uses dynamic programming in that

the optimal solution for each node is determined from the optimal solutions of its descendants). For machines with complicated addressing modes, this algorithm becomes unwieldy, with hundreds of cases. Dynamic programming can be extended to machines with several classes of registers[7] by an algorithm analogous to the CYK algorithm[8, 9] for parsing by dynamic programming.

While these algorithms are powerful and useful, it is important to note that they select optimal code sequences only for the particular trees they are given; and those trees are not necessarily optimal versions of the programs they represent.

1.2. Formal description of instruction sets

The automatic generation of instruction selectors from a specification of the target machine requires a formal specification language for target-machine semantics. The ISP language for specifying machine instruction-sets[10] has been used to automatically generate instruction selectors for intermediate representations based on trees[11] and peephole optimizers for more general graphs[12, 13]. These target-code generators rely on heuristics to match patterns (derived from the semantics of target-machine instructions) to portions of the intermediate representation tree (or graph).

The heuristic used on trees leads to a sequence of pattern-matches identical to that made by an LR parser; this led to the formalization of target-machine semantics as LR(1) grammars[14]. Instruction selection is thus reduced to LR(1) parsing, which is efficient and well-understood. Unfortunately, the LR(1) grammars tend to be very large: typical descriptions of the VAX instruction set use hundreds of productions [15, 16].

Context-free grammars have difficulty in efficiently describing certain features of real instruction sets, particularly those involving operand size constraints. For this purpose, attribute grammars are convenient[17]. An attribute grammar describing a given machine can be much shorter than the equivalent context-free grammar.

1.3. Optimal instruction selection from formal instruction set descriptions

Combining grammar-based machine descriptions with dynamic programming, as in the *Twig* system[1], leads to concise and powerful instruction selection specifications.

Dynamic programming, as a parsing technique, has three significant advantages over LR(1) parsing: it allows highly ambiguous (but much more concise) grammars, it allows attribute information to be used with no extra trouble, and it produces optimal instruction selection. (It has the disadvantage that it tends to use more time and space.)

Because there are often several different instruction sequences that accomplish the same thing, ambiguous grammars occur naturally in the description of code generators. An LR(1) grammar cannot be ambiguous, so the ambiguities must be removed at compiler-generation time. This has the effect of deciding in advance which kind of instruction sequence to use for a given class of operations — even though the best choice might depend on attributes of the specific instance. The dynamic programming algorithm allows an ambiguous grammar, with the ambiguity resolved only when the instance is parsed. This allows the best instruction sequence to be chosen for each instance.

Many natural instruction-set grammars have too much factoring to be LR(1)[15]. Consider the following fragment of a grammar:

displ → PLUS CONST reg	*displacement addressing mode*
binop → PLUS	*binary addition operator*
binop → XOR	*binary exclusive-or operator*
operand → CONST	*immediate-constant operand*
operand → reg	*register-direct operand*
reg → displ	*move-address instruction*
reg → binop operand operand	*three-address instruction*

An LR(1) parser cannot know, after it has seen PLUS (with CONST as a lookahead symbol) whether to shift, or to reduce the PLUS as a binop. This shift-reduce conflict can be solved by removing the rule

binop → PLUS

and adding rules like

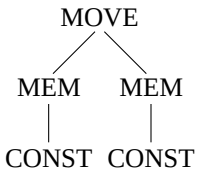
reg → PLUS operand operand

for each context in which binop appears.

Unfortunately, the grammars tend to get very large when this — and other similar necessary changes — are made. Large grammars are inefficient to process, but — more important — they are hard to get right. Graham, Henry, and Schulman state, ‘‘Most of the outstanding bugs in our code generator are caused by remaining instances of overfactoring.’’[15].

Twig handles highly factored grammars with no difficulty. For the (human) specifier of an instruction set, the ability to write concise but ambigous grammars, rather than large but unambiguous ones, is an invaluable advantage of the dynamic programming algorithm.

Another example of the limitations of LR(1) grammars for machine description is shown graphically below. There are two machine instructions that can cover different, but overlapping parts of an intermediate representation tree:



The instructions that cover the (overlapping) circled areas correspond to the register transfers:

M[constant] ← register
and
M[register] ← M[constant]

Either instruction can be used to cover the root (and some immediate descendants) of the tree. (The uncovered nodes must then be covered by other instructions.) The LR(1) parser will find a shift-reduce conflict here; if (as is typical) it resolves this by shifting, then it will always choose the first of these two instructions, even if the second might be less costly. A dynamic-programming parser, on the other hand, can evaluate the cost of both instruction sequences and choose the better one.

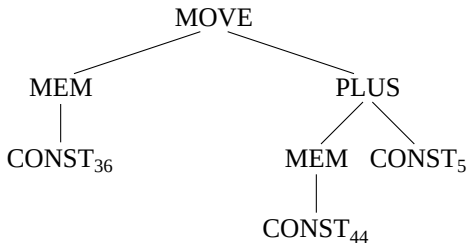
This paper demonstrates the concise and powerful nature of code generators based on dynamic-programming parsers by using the VAX and MC68020 instruction sets as illustrative examples.

2. Overview of Twig

Aho, Ganapathi, and Tjiang’s code-generator generator, *Twig*, is described in[1] and[18]. This section provides just a brief overview of the system.

The Twig system takes a specification in the form of an annotated grammar and produces a code generator. The input to the code generator is a tree, which is (presumably) a machine-independent representation of a program being compiled. The code generator translates the tree into object code (assembly language, for example) by finding tree-patterns from the grammar that match portions of the tree.

Typically, the internal nodes of the tree are operators like **MEM** or **PLUS** and the leaves are constants (often address constants). Each node in the tree is labelled by some such symbol, so that the entire tree can be represented in a functional notation. Here is an example representing the addition of 5 to the contents of address 44, and the storing of the result at address 36:



MOVE(MEM(CONST$_{36}$**),PLUS(MEM(CONST**$_{44}$**),CONST**$_5$**))**

The Twig specification is a grammar for trees that are written in this functional notation. The code generator attempts to parse its input (a tree) according to this grammar. Since the grammars are often highly ambiguous, many parses may be legal. In this case, the code generator picks the parse with the lowest cost. The cost of a parse is (typically) the sum of the costs of the reductions in the parse. The cost of a reduction is given as an annotation to the corresponding grammar rule in the specification.

Finding the lowest-cost parse can be done fairly efficiently by dynamic programming. Working from the bottom up, the cost of matching each node to each nonterminal symbol is computed. The cost of matching higher nodes in the tree can be computed just from the costs of their immediate descendents. For a fixed grammar, this takes time and space linear in the size of the input tree.

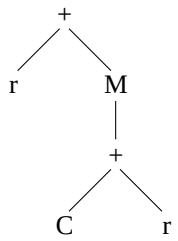
2.1. An example

To illustrate the use of instruction-set grammars, consider a simple computer with only five instructions:

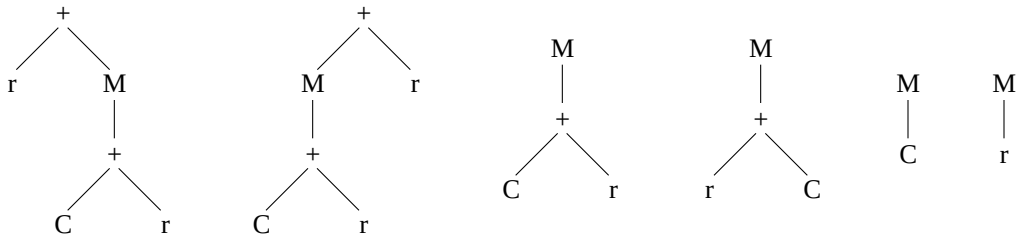
Name	**Cost**	**Effect**
ADD	2	$r_i \leftarrow r_j + r_k$
ADDI	3	$r_i \leftarrow r_j + C$
ADDM	4	$r_i \leftarrow r_j + M[C+r_k]$
STORE	4	$M[C+r_i] \leftarrow r_j$
MOVE	4	$M[r_i] \leftarrow M[r_j]$

We will assume that r_0 always contains 0.

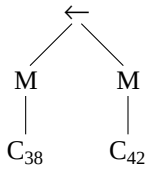
These instructions can be expressed as tree-patterns, using the operators + for addition, *M* for memory reference (fetch or store), *C* for constants, *r* for registers, and $\leftarrow$ for assignment. The ADDM instruction would be expressed as



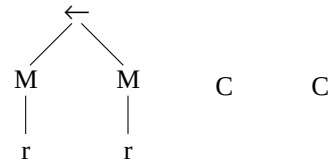
However, because addition is commutative, and because either register could be $r sub 0$, 12 patterns are needed express all the possible effects of ADDM, of which some are shown here:



A code generator is given trees representing source-language statements, and it can use these patterns to ‘‘tile’’ the tree with instructions. For example, the tree



can be tiled with the patterns



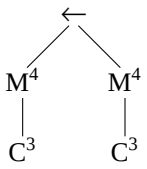
corresponding to the MOVE, ADDI, and ADDI instructions. The *r* leaves are just place-holders for sub-trees, which in this case are single-node C trees. The instruction sequence corresponding to this tiling is:

ADDI	$r_1 \leftarrow r_0 + 38$
ADDI	$r_2 \leftarrow r_0 + 42$
MOVE	$M[r_1] \leftarrow M[r_2]$

This tiling has a cost of 10 (two ADDI instructions and a MOVE). There is a cheaper way of generating instructions to cover this tree, using an ADDM and a STORE for a cost of 8. In general, the number of possible tilings can be exponential in the size of the tree.

By dynamic programming, the best tiling can be found in linear time. Using a postorder traversal of the tree, each subtree is marked with the lowest cost to tile it. At each node, all the patterns are tried, and a simple cost computation is done for any that match (an *r* leaf matches the root of some other pattern).

For example, in the traversal of the tree for M[38]←M[42], the leaf C node is matched only by an ADDI pattern, for a cost of 3. Either M node is matched by an ADDM pattern for a cost of 4. This leaves the tree marked as follows:



Now the root can be matched by a MOVE pattern; the cost is calculated by adding the 4 for the MOVE instruction to the costs of getting the *r*-leaves into registers. But these costs have already been calculated as the costs of the C nodes; the total is therefore 10. Or, the root can be matched by a STORE pattern, with one *r*-node that corresponds to the right-hand M node, for a total cost of 8.

When several different ways of matching the same node are found, only the cheapest is remembered; the running time, therefore, is proportional to the number of patterns (a constant) times the size of the tree.

In essence, this is the Aho-Johnson algorithm[6] for instruction selection. The problem with this algorithm is that a tree-pattern must represent an entire instruction; as illustrated above, some instructions require many patterns to describe all the different ways they can be used. One a machine with many addressing modes (like the VAX), a single instruction like **mov** requires thousands of patterns. It is this problem which motivated the Graham-Glanville approach[14], which uses context-free grammars to describe instruction sets.

In our example instruction set, the sum of a register and a constant (either of which can be zero) is used in several instructions; this could be abstracted as a *displacement.* This leads to a grammar with two nonterminal symbols, *reg* and *disp*:

disp → *reg*
disp → CONST
disp → +(CONST,*reg*)
disp → +(*reg*,CONST)
reg → +(*reg*,*reg*)
reg → *disp*
reg → +(*reg*, M(*disp*))
reg → +(M(*disp*),*reg*)
reg → M(*disp*)
reg → MOVE(M(*disp*),*reg*)
reg → MOVE(M(*reg*),M(*reg*))

This ‘‘factored’’ grammar is much more concise than the grammar with just the *reg* nonterminal could be (with its dozen rules for ADDM alone). The Graham-Glanville uses LR(1) parsing to do pattern-matching on such grammars. But these grammars are not naturally LR(1), which leads to parsing problems; and the instruction costs are not accounted for, which leads to suboptimal matches.

The dynamic programming algorithm can be generalized to grammars with several nonterminal symbols. The tree is traversed bottom up, as in the Aho-Johnson algorithm, but the minimum cost is kept for *each* nonterminal symbol. Then, when a pattern like *reg* → MOVE(M(*disp*),*reg*) is matched, the *reg* nonterminal of the root node is updated with a cost that is calculated from the *disp* cost of the left child’s child, and the *reg* cost of the right child.

Twig implements this dynamic programming algorithm, given a grammar annotated with cost and instruction-emission information. *Twig* also uses a clever pattern-matching algorithm[19] to avoid comparing every pattern to every node in the tree, but a description of that is beyond the scope of this paper.

2.2. Format

A Twig specification has several parts. Twig treats the tree structure as an abstract data type, so the functions that implement tree-operations (find n^{th} child, etc.) must be provided as part of the specification. The terminal symbols of the input grammar are listed in a **node** declaration; the nonterminal symbols are listed in a **label** declaration. Then, each grammar rule is presented with its annotations, which are written in the C language. The first annotation is the *cost* phrase, which consists of statements that compute the cost of applying this rule in a given situation. The second annotation is the *action* phrase, which (typically) emits the object code for a parse reduction when this rule is used.

The cost phrase of a rule will be evaluated many more times than the action phrase: the code generator must determine the costs of applying all relevant rules before it finds the minimum-cost parse. After that parse is found, all of the action rules of the constituent reductions are applied (usually) in bottom-up order.

Here is a specification of a simple instruction set that contains just three instructions: **add**, **load**, and **store**.

```
node MEM MOVE PLUS CONST;

label reg;

reg: MEM(CONST)
    {cost=2;}
    ={$$->reg = getreg();
      emit("load   r%d,%d", $$->reg, $1$->value);
     }

reg: PLUS(reg,reg)
    {cost+=1;}
    ={givereg($2$); $$->reg=$1$->reg;
      emit("add   r%d,r%d", $1$->reg, $2$->reg);
     }

reg: MOVE(MEM(CONST),reg)
    {cost+=2;}
    ={$$->reg=$2$->reg;
      emit("store  %d,r%d",  $1$->value, $2$->reg);
     }
```

In this specification, the initial part (giving the C-language interface to the concrete representation of trees) has been omitted.

The terminal symbols of the grammar — those to be found labelling the tree nodes in the input to the code generator — are listed in the first line. The nonterminals — specifying the different ‘‘modes’’ in which a subtree may be matched — are listed on the second. The remaining lines are the three productions of the grammar.

The cost phrases specify that an **add** instruction costs one unit, while **load** and **store** cost two units each. In particular, the cost of evaluating a subtree whose root matches an **add** pattern is 1 plus the cost of fetching the arguments of the **add** into registers. Thus, the cost phrase for this pattern is `{cost+=1;}` where the variable `cost` is initially equal to the sum of the costs of the two `reg` nodes that are the leaves of this pattern.

The action phrases for a specification consist of arbitrary C code, which usually accesses the matched subtree and possibly the cost structure. Register-allocation and code generation are not part of Twig, but are entirely up to the user of Twig. This specification’s action phrases include statements to allocate and

deallocate registers, and to emit assembly-language instructions.

In the cost and action phrases, $$ represents the root of the subtree matched by the instruction, and 1, 2, etc. represent the children. Other descendants are represented by giving the paths to them; so 2.1 is the first child of the second child of the root.

The costs of the subtrees are accessed in a different manner, although there are no cases of this in the example above. In general, $%n$→cost is the cost of the n^{th} leaf of the match. Leaf nodes of a match always correspond to nonterminal symbols in the grammar rule. Thus, if the third rule (in the example above) wants to refer to the cost of the **reg** child, then it uses the notation $%1$→cost.

3. Intermediate Representation Trees

The choice of an intermediate representation (IR) to serve as an interface between the parser and code-generator of a compiler has a profound effect on the simplicity and robustness of both of these components. Twig may be used with any set of terminal symbols, and does not place many constraints on the IR except that it be a tree language (i.e., not a register-transfer or DAG language).

This section describes the abstract syntax of, and interface to, a carefully chosen intermediate-representation-tree language for compilers of Algol-like languages. This tree representation is designed to have a clear (though informal) operational semantics; and, of course, to be compatible with Twig. The instruction-selection algorithm takes trees in this language and translates them into machine instructions.

Each node in a tree has an operator specifying what kind of node it is, and zero to three children; each child is another tree. In addition, each node has several attributes.

The trees are represented (concretely) in memory as records and pointers. They are represented (abstractly) in a functional notation, e.g. OP(PLUS,FETCH(NAME),CONST). They are built (by the front end of the compiler) using a specified set of tree-building interface routines.

3.1. Syntax

Figure 1 gives the syntax of IR trees.

It is tempting to confuse IR trees and parse trees. The grammar given in figure 1 will lead to parse trees, but these are not the same as IR trees. One should think of Figure 1 as a grammar for expressions written in the functional notation; these expressions correspond to IR trees.

3.2. Attributes

Each node may have one or more attributes, depending on which terminal or nonterminal symbol it corresponds to.

Symbol	Attributes possessed by nodes of this symbol.
body	**proclabel** [the assembly-language name of the procedure]
LABEL	**label** [the assembly-language label of this point in the program]
exp	**size** [the number of bytes* needed to hold the expression-value]
TEMP	**temp** [a descriptor of the register or temporary location]
NAME	**label** [the assembly-language label referred to]
CONST	**ivalue** [the integer value of this node]
CONSTF	**fvalue** [the floating-point value of this node]

Note that some nodes have more than one attribute. For example, a CONST node is also an ‘‘exp’’, so that is has both an ivalue and a size.

---

* The description here assumes a byte-addressable machine, but the model can easily be made more general.

```
body:   PROC(exp,stm)

stm:  SEQ(stm, stm)
stm:  LABEL
stm:  JUMP(exp)
stm:  CJUMP(test, exp)
stm:  exp

exp:  OP(binop, exp, exp)
exp:  UNOP(unop, exp)
exp:  UNOP(cvtop, exp)
exp:  MEM(exp)
exp:  MOVE(MEM(exp), exp)
exp:  MOVE(TEMP, exp)
exp:  ESEQ(stm, exp)
exp:  BOOL(test)
exp:  NAME
exp:  CONST
exp:  CONSTF
exp:  ALLOC(TEMP, exp)
exp:  TEMP
exp:  CALL(args, exp)

test:  CAND(test, test)
test:  NOT(test)
test:  OP(relop, exp, exp)

args:  ARG(exp, args)
args:  NOARGS

binop: FPLUS  FMINUS  FMUL  FDIV
binop: PLUS  MINUS  MUL  DIV  MOD  AND  OR  LSHIFT  RSHIFT  XOR

relop: EQ  NEQ  LT  LEQ  GT  GEQ
relop: ULT  ULEQ  UGT  UGEQ
relop: FEQ  FNEQ  FLT  FLEQ  FGT  FGEQ

unop: NEG  COMP  FNEG
cvtop: CVTSU  CVTSS  CVTSF  CVTUU  CVTUS  CVTFS  CVTFF
```

Figure 1. Syntax of IR trees

3.3. Semantics of the nodes

Each production in the grammar derives nodes with particular meanings. The semantics of each kind of node are given here.

body: PROC(exp,stm)

The ‘‘localsize’’ of a procedure is an expression (‘‘exp’’) denoting the number of bytes of local-variable space that must be allocated for each invocation of the procedure. A procedure ‘‘body’’ is simply a statement (‘‘stm’’) as the second child of a PROC node. The ‘‘proclabel’’ attribute specifies the label by which the assembly-language program can call the procedure.

stm: SEQ(stm,stm)

A statement can be a sequence of statements, with the meaning that the first is to be executed, then the second (unless of course, the first statement executes JUMP to somewhere else).

stm: LABEL

A statement may be a LABEL. This statement does nothing at execution time; it serves as the target of JUMP instructions, etc.

stm: JUMP(exp)

A statement may JUMP to any address. The address may be computed as an expression. The simple case of jumping to a fixed label is a special case: JUMP(NAME).

stm: CJUMP(test,exp)

A statement may be a conditional jump to any address. If the **test** evaluates to ''true'' at run-time, then the jump will be made to the location indicated by the expression **exp.**

stm: exp

A statement may be an expression. At run-time, the expression will be evaluated (for possible side-effects), and the result will be thrown away.

exp: OP(binop, exp, exp)

An expression may be the sum of two expressions, etc. The ''binop'' specifies which binary operator is to be used. Thus, OP(PLUS,exp1,exp2) represents the sum of the two expressions exp1 and exp2.

The *size* attribute of the OP-expression must be the same as the *size* of the children. Explicit coercions (see below) can be put in the tree to overcome this.

exp: UNOP(unop, exp)

Unary operators are just like binary operators, except that there is only one expression-child. Thus, negative 3 is **UNOP**(**NEG**,**CONST**$_3$). Of course, this particular tree could be rewritten as simply **CONST**$_{-3}$.

exp: UNOP(cvtop, exp)

Type-conversions serve two purposes: to convert from one machine data type (signed-integer, unsigned-integer, floating) to another, and to convert from one precision (size) to another. These are unusual operators in that the children may differ in ''size'' from the parent.

The distinction between long-integer, short-integer, and character is not a difference of machine data type, but simply of size. The same holds for single- and double-precision floating-point.

exp: MEM(exp)

An expression may be the result of fetching from an address of memory. The child represents the address from which to fetch, and must have a size equal to addressSize. The size of the MEM node indicates the amount of data to be fetched, starting at that address of memory. The size may be any number representable as an address.

exp: MOVE(MEM(exp),exp)

exp: MOVE(TEMP,exp)

An expression may be stored at an address in memory, or moved into a temporary location (e.g. a register). MOVE(e1,e2) is an expression whose value is that of e2, and which has the side effect of putting that value into the destination e1. The destination may be a MEM(e3) node (for a store into memory at address e3), or a **TEMP**$_i$ for a move into temporary variable i (typically a register). The number of bytes stored is the size of the MOVE node (which is equal to the size of the nodes e1 and e2).

The size may be any number representable as an address. Note, however, that one can't do much with expressions of large size other than fetch them and store them. Thus, the following represents a block move:

$$\mathrm{MOVE}(\mathrm{MEM}_{1000}(\mathrm{NAME}_{dest}), \mathrm{MEM}_{1000}(\mathrm{NAME}_{source}))$$

This tree moves 1000 bytes from label ''source'' to label ''dest,'' and is an example of an expression whose size is larger than will fit in a register or a register pair. The subscripts indicate attributes of nodes in the expression tree; for clarity in this description, most attributes are not shown.

exp: ESEQ(stm, exp)

The statement ''stm'' is evaluated, then the expression ''exp'' is evaluated. The result and size of the ESEQ are that of the expression.

exp: BOOL(test)

The test is converted into an n-byte integer, where n is ''size.'' If the test evaluates to true, the integer 1 is the result; else 0.

exp: NAME

This kind of expression yields the value associated with the label. Such a value is typically an address, so the size is usually ''addressSize.'' However, many assemblers and loaders are capable of treating 1-, 2-, or 4-byte quantities as labels.

exp: CONST

An integer constant. The size is the number of bytes used to hold the value (typically 1, 2, or 4). The value is the result of the expression.

exp: CONSTF

A floating constant. The size depends on the desired representation; 4 or 8 is typical.

exp: ALLOC(TEMP,exp)

A temporary variable ''temp'' is allocated; the ''exp'' is evaluated and its value is saved; then the temporary variable is discarded. The result of the ALLOC-expression is the result of exp. There should be no references to ''tmp'' outsize of ''exp''.

exp: TEMP

The value of a temporary variable is used. The size of the expression is the size associated with the temporary variable. The temporary variable may have been allocated with ALLOC, or may have been allocated in some other way.

A **TEMP** node represents a register that the front end of the compiler is using as an explicit temporary. Whereas an intermediate result of a tree calculation may be used only once (in the calculation of its parent), a **TEMP** register may be used in several calculations.

TEMP nodes can be used to represent specific machine registers like stack pointers and frame pointers. In this case the generation of the intermediate tree is somewhat dependent on the stack-frame layout being used. This slight machine-dependency can easily be made transparent to the front end of the compiler.

exp: CALL$_{size}$(args,exp)

A function is called with arguments. The argument list ''args'' is evaluated; these are passed to the function, whose address is given by ''exp.'' The function returns a result of size ''size,'' and this result is the value of the CALL-expression. The arguments may be evaluated left-to-right, or right-to-left, or in some other order. All arguments are call-by-value.

test: CAND(test, test)

The CAND (condition-and) operator first evaluates the test e1. If this is false, the result of the CAND is false. Otherwise, e2 is evaluated; if both e1 and e2 are true, the result is true; else false.

This is one of the few operators to guarantee the order of evaluation of the subtrees.

test: NOT(test)

The boolean value of a test is complemented. Note that tests are not expressions, and do not have sizes.

test: REL(relop,exp,exp)

A boolean test-value is created from the comparison of two equal-sized expressions. The ‘‘op’’ is a relational operator: equality, inequality, greater-than, etc.

args: ARG(exp, args)

One element of an argument-list, with two fields: ‘‘This element’’ (exp), and ‘‘rest-of-list’’ (args). Note that an ‘‘args’’ node has a size field, which is computed as the sum of the sizes of the two fields.

args: NOARGS

This is an empty argument list, or it is the ‘‘end’’ of an argument list. The size is zero.

binop: FPLUS FMINUS FMUL FDIV

These are the floating point arithmetic operators, in single- or double-precision.

binop: PLUS MINUS MUL DIV MOD AND OR LSHIFT RSHIFT XOR

These are the integer arithmetic operators, which work on 1-, 2-, or 4-byte integers.

relop: EQ NEQ LT LEQ GT GEQ

These are the signed-integer comparison operators, which work on 1-, 2-, or 4-byte integers.

relop: ULT ULEQ UGT UGEQ

These are the unsigned-integer comparison operators. Equality and inequality of unsigned integers are tested with EQ and NEQ, just as for signed integers.

relop: FEQ FNEQ FLT FLEQ FGT FGEQ

These are the floating-point comparison operators, which work on single- or double-precision numbers. (Both arguments must be of the same precision, of course.)

unop: NEG COMP FNEG

These are the unary operators. They specify the integer-negation, integer-complement, and floating-negation, respectively.

unop: CVTSU CVTSS CVTSF CVTUU CVTUS CVTFS CVTFF

These are the type-conversion operators. They convert between the three types ‘‘Signed-integer’’, ‘‘Unsigned-integer’’, and ‘‘Floating-point.’’ Thus, CVTSU converts from signed to unsigned integers.

These are among the only operators that take arguments whose size is different from their result. Thus, one may use CVTSU to convert from a 1-byte signed integer to a 4-byte unsigned integer, or from a 2-byte unsigned integer to a 1-byte signed integer, etc.

3.4. Layout of the nodes in storage

The concrete representation of the nodes in the computer is done with structs (in the C language) pointing to other structs. Here is the declaration of the type Tree:

```
typedef struct tree *Tree;

struct tree {char op; char reg, kind;
         union {int size;
            Label proclabel;
              . . . other fields
           } x;
         union {Tree child[3];
            Label label;
            int ivalue;
            double fvalue;
            struct temp *temp;
             . . .  other fields
           } u;
        };
```

The fields named x.size, x.proclabel, u.label, u.ivalue, u.fvalue, and u.temp correspond to the similarly named attributes described in a previous section. The u.child vector points at the zero, one, two, or three children of a node. Any child-bearing node with fewer than three children will have NULL's in the higher-numbered entries of the child vector. To know whether a node is child-bearing it is necessary to look at the op field, which specifies the kind of node this is.

The `reg` and `kind` fields are used by the code generation algorithm to annotate the tree, and are left blank when the trees are created by the front end of the compiler.

3.5. Constant folding

The code generators to be described in this paper are really just optimal instruction selectors, not general optimizers. To simplify the instruction selection, it is useful to do some machine-independent partial evaluation of intermediate representation trees.

Twig has a feature that is intended for this purpose (REWRITE rules), but it is difficult to use. Furthermore, the constant folding is machine-independent, and the Twig grammar describes the machine-dependent instruction selection phase. And if the constant-folding is done early, then Twig won't waste time doing cost-evaluations on trees that are to be re-written.

Therefore, it is best to do the constant-folding as the trees are constructed. For example, there is an interface function that builds an **OP** node given a size and pointers to its children:

```
Tree  buildOP(size, operator, left, right)
    int size; Tree operator, left, right;
```

All tree nodes are built using calls to such functions. When the `buildOP` function is called upon to build the tree corresponding to the sum of two **CONST** nodes, it may return a new CONST node whose value is the sum (instead of an **OP** node which formally represents the sum).

The identities listed in Figure 2 are used to simplify the structure of the trees. In general, they try to fold **CONST** nodes together, and to this end they try to move **CONST** nodes to the right in expressions. They also perform some specific rearrangements of **PLUS** and **MUL** that are useful in simplifying expressions found in array indexing. Finally, they attempt to bring **NAME** and **CONST** nodes close to each other so that the back end may take advantage of this if it is so equipped.

name: NAME
name: OP(PLUS,name,CONST)
commute: PLUS MUL XOR OR AND
assoc: PLUS MUL XOR OR AND
id0: PLUS MINUS XOR OR
id1: MUL DIV
rel: EQ NEQ GT GEQ etc.

OP(commute, CONST, exp) → OP(commute, exp, CONST)
OP(commute, name, exp) → OP(commute, exp, name)
OP(op, CONST, CONST) → CONST
OP(id0, exp, zero) → exp
OP(id1, exp, one) → exp
OP(MUL, OP(PLUS,exp, CONST), CONST) → OP(PLUS, OP(MUL, exp, CONST), CONST)
OP(MINUS, exp, CONST) → OP(PLUS, exp, CONST)
OP(assoc, OP(assoc, exp, CONST), CONST) → OP(assoc, exp, CONST)
OP(PLUS, exp, OP(PLUS, exp', exp'') → OP(PLUS, OP(PLUS, exp, exp'), exp'')
OP(PLUS, OP(PLUS, exp, name), CONST) → OP(PLUS, exp, OP(PLUS, name, CONST))
OP(PLUS, OP(PLUS, exp, name), exp') → OP(PLUS, OP(PLUS, exp, exp'), name)
UNOP(op, CONST) → CONST
OP(NEQ, BOOL(test), zero) → test
OP(rel, CONST, exp) → OP(rel', exp, CONST)

These rules are summarized here in a semi-formal notation, but some of the details have been elided. In particular, some of combinations of rules appear to have the potential to loop indefinitely, but in practice their application is restricted so that this will not happen. For example, the rule

OP(commute, CONST, exp) → OP(commute, exp, CONST)

is not applicable if **exp** is a constant. Similar restrictions on the other rules prevent looping.

These constant-folding operations are worthwhile because the code generator can be much simpler if it can assume that the **CONST** is always on the right of an addition: normally, commutative operators like **PLUS** lead to much duplication of tree-patterns, but now any pattern with **CONST** as the left operand is unnecessary.

4. The VAX instruction set

The VAX has a rather complex, though mostly consistent, instruction set[20]. The operands of VAX instructions may have several forms, known as ''addressing modes.'' The instructions themselves may be three-address (operating on two operands and placing the result in a third) or two-address (placing the result in the second source operand). There are several instructions that are just cheaper special cases of other instructions (like the increment instruction).

For example, the instruction

addl3 r1,8(r2),*(r3)+[r4]

adds the values of the first two operands — r1 and 8(r2) — and stores the result in the third — *(r3)+[r4].

The word ''add'' specifies that the result is the sum of the first two operands. The letter ''l'' indicates that the operands are to be considered as longwords (4-byte integers). The 3 appended to the opcode specifies that this is a three-address instruction (rather than a two-address instruction).

The first operand is a simple register-mode operand: the value of the operand is just the contents of register

1. The second operand is a displacement-mode operand: the sum of 8 and the contents of register 2 give the *address* of the memory cell in which the value is to be found.

The third operand is an autoincrement-deferred-indexed-mode operand, and more complicated. First, the contents of register 3 are used as an address from which to fetch a value *x*. Then, register 4 is multiplied by 4 (because this is a longword instruction) and added to *x*. Then *x* is used as the address in which to store the result of the addl3 instruction. Register 3 is then incremented (by 4).

Because the autoincrement and autodecrement modes cause side-effects as well as yield values, they are difficult, though not impossible, to use in code generation. The code-generator specification described here does not use these addressing modes. However, it uses all the other addressing modes, including the indexed modes.

4.1. Nonterminals for the VAX

The terminal symbols for the VAX's Twig specification must, of course, be exactly those of the IR trees. These were described in the previous chapter. The nonterminal symbols are free to be different, as long as the resulting grammar describes the same language of trees.

The nonterminal symbols will be:

body	the start symbol of the grammar: an entire procedure
stm	statement; i.e. code executed for side-effects only
reg	a VAX register
operand	a VAX addressing mode
location	an address in memory specifiable as an addressing mode
ioperand	a VAX indexed addressing mode
destination	an operand that denotes a storable memory location
computation	roughly, all but the last operand of a machine instruction. (This will be explained in greater detail.)
test	boolean condition: code that will branch if true
ntest	negated boolean condition: code that will branch if false
flag	an expression that will set condition codes
bigval	an expression too large to fit in registers (like an array or record)
binop	binary operator symbol (PLUS, MINUS, etc.)
unop	unary operator symbol (NEGATE etc.)
relop	relational operator (EQ, NEQ, etc.)
cvtop	a conversion operator
cvtopu	an unsigned conversion operator
name	a constant or a label
args	a list of actual parameters to a procedure
zero	a constant with value 0 (integer or floating).

4.2. Cost definitions

Each match of a nonterminal to a tree-node is associated with a given cost. These costs have several components:

space	The number of bytes occupied by the instruction
time	the amount of time taken by the instruction (measured in tenths of microseconds on a VAX-11/780). Because of pipelining variations, cache misses, etc., this must be an approximation. Newer implementations of the VAX architecture have significantly different costs; in particular, access to memory is cheaper relative to register access.
setFlags	whether the instruction sets the condition codes appropriately (e.g. to reflect the sign of the result)
sideEffect	whether the tree rooted at this node has side effects (like a jump or a store)
dontDestroy	whether the register holding the value of this node must be saved for later use
hold	the number of registers needed to hold the value of this node (in the specific way it was matched)
maxregs	the number of registers needed to compute the children of this match

These cost values will be discussed in more detail in the next section.

The COSTLESS rule, to determine which of two costs is the lesser, simply compares the sum of space and time:

```
#define COSTLESS(x,y) ( x.space+x.time < y.space+y.time )
```

By adjusting this formula, it is possible to ‘‘tune’’ the code generator to optimize for code space or for execution time, but it is not clear whether any significantly different code sequences will emerge.

The DEFAULT_COST for each node is (roughly) the sum of the costs of the children. That is, the *space* values of the children are summed, the *time* values are summed, and the *sideEffect* values are unioned.

In addition, a variant[21] of the Sethi-Ullman algorithm[5] for minimizing register-usage is used. This will require reordering the computation of the children of expression-nodes (as long as there are no side-effects among the reordered children). The cost of the optimal ordering is computed in DEFAULT_COST, yielding the cost value *maxregs*. The register allocator also takes care of register spilling, although the cost of a spill is not calculated as part of the cost of a pattern-match, which could lead to a suboptimal instruction sequence.

All other fields of the DEFAULT_COST are set to zero.

4.3. Grammar rules for the VAX

Now the production rules of the grammar will be explained. With each rule will be the cost evaluation and the code-generating action. This is the *entire* code generator specification, except for the prolog and epilog which contain the following components: The C language interface to the tree representation, the computation of default costs, the Sethi-Ullman reordering, and the listing of terminal and nonterminal symbols of the grammar.

4.3.1. The top-level rule

To match an entire procedure or function, the start symbol *body* is the left-hand-side:

body: PROC(operand,stm)

```
    { TOPDOWN; }
    ={Label masklabel = newLabel();
      emit(".align\t1\n.globl\t%s\n",$$->x.proclabel->s);
      emit("%s:\n.word\t%s\n",$$->x.proclabel->s,masklabel->s);
      tDO($%1$);
      emit("subl2\t"); emitoperand($1$); emit(",sp\n");
      tDO($%2$);
      emit("ret\n.set\t%s,0%o\n",masklabel->s,regmask&07774);
     };
```

The word **TOPDOWN** in the cost fragment specifies that the children of the **PROC** node should not be emitted until explicitly called for (by **tDO**). The implementation of the TOPDOWN macro is part of the standard Twig runtime system; it just sets a flag for the action driver.

The **emit** procedure works just like the **printf** procedure in the C language. This rule specifies that the assembly code for a procedure **abc** follows this pattern:

```
.align      1
.globl      abc
abc:
.word       L23
    computation to determine size of local-variable space
subl2       local-variable-size, sp
    body of procedure
ret
.set    L23, register-mask
```

4.3.2. Rules for statements

The next few rules give the specification of code generation for the nonterminal symbol *stm.*

stm: SEQ(stm,stm);

A statement may be a sequence of two statements. The cost is simply the sum of the two subcosts, and the emitted code is simply the concatenation of the two children. These are the default rules, so no extra specification is necessary.

stm: LABEL

```
    ={emit("%s:\n",$$->u.label->s);};
```

A statement may be a label definition. The cost is zero (the default), and the assembly language to emit is simply the label name followed by a colon.

stm: JUMP(name)

```
    { cost.space=3; cost.time+=8; cost.sideEffect=1;}
    ={emit("jbr\t"); emitname($1$); emit("\n");};
```

A statement may be an unconditional jump to a fixed address. Depending on the distance between the jump and its destination, the instruction (and its cost) will vary. Jump size resolution can't be handled easily by a one-pass algorithm, so we make an approximation for the cost and let the assembler resolve the instruction.

This instruction has a side effect, which must be noted in the cost phrase by setting the **sideEffect** field of the cost.

stm: JUMP(location)

```
{ cost.space+=1; cost.time+=8; cost.sideEffect=1; }
={emit("jmp\t"); emitlocation($1$); emit("\n");};
```

Unconditional jumps to variable addresses cannot use the ‘‘branch’’ instruction; they must use the slightly more expensive ‘‘jmp’’ instruction. The space cost is 1-byte (for the opcode) plus the space taken by the addressing mode *location.* The initial value of the `cost` variable on entry to the cost phrase is the sum of the costs of the leaves of the pattern. This pattern has only one leaf — *location* — so the initial value of `cost` is that of the space, time, etc. taken by the *location.* It thus suffices to execute `{cost.space+=1;}` to add 1 to the cost of the *location.* The time cost is about 0.8 microseconds (on a VAX-11/780) plus the cost of evaluating the *location.*

stm: CJUMP(test,NAME)

```
{ cost.sideEffect=1; TOPDOWN; }
={$1$->x.truebranch=$2$->u.label;
  EVAL;
 };
```

A conditional jump to a fixed address may be accomplished by evaluating a *test,* which branches if true to the label specified in its *truebranch* attribute. This attribute must be set before the *test* is emitted, so the match is **TOPDOWN.** Finally, EVAL calls for the evaluation of the all the subtrees (in this case, there is only one subtree, so tDO($%1$) would be equivalent).

stm: CJUMP(ntest,location)

```
{ cost.space+=1; cost.time+=4; cost.sideEffect=1; TOPDOWN;}
={$1$->x.truebranch=newLabel();
  tDO($%1$);
  tDO($%2$);
  emit("jmp\t"); emitlocation($2$);
  emit("\n%s:\n",$1$->x.truebranch->s);
 };
```

The VAX can execute conditional jumps only to fixed addresses. Therefore, there must be a conditional jump over an unconditional jump. The conditional jump must, of course, test the negated condition, so *ntest* is called for. As usual, the cost is computed as the sum of the childrens’ costs plus the number of extra bytes and microseconds called for here. In this case, there is one extra byte (the opcode of the ‘‘jmp’’ instruction). When the jump is taken, the time will be 0.8 microseconds; we assume that the jump is taken half the time, for an average cost of 0.4 microseconds. The cost of the *ntest* includes the cost of the conditional jump contained within it.

stm: operand

```
={emit("# throw away "); emitoperand($$); emit("\n");};
```

A statement may be an expression which is evaluated solely for side effects. The most general kind of expression that will fit here is an operand. It is necessary to call ‘‘emitoperand’’ to free the registers (as will be explained later), so an assembly-language comment is emitted.

stm: bigval

```
={emit("# throw away r%d\n",$$->reg); givereg($$);};
```

Since *operand* doesn’t match any expression whose value won’t fit into a register (or register-pair), this rule is added.

There are also several rules for **MOVE** statements. These will be explained later.

4.3.3. Calculating values into registers

The *reg* nonterminal represents subtrees that calculate some value (possibly with side-effects) that ends up in a register.

The nonterminal *computation* represents three-address operations, unary operations, and move instructions (two-address operations are represented by a different nonterminal). *Computations* will be described in detail in their own subsection, but here are some examples:

```
movl    8(r2),
addl3   r5,16(r4)[r2],
mnegl   varA,
```

Note that in each case the last operand — the destination — is missing. A computation specifies everything about an arithmetic operation except the destination.

reg: computation

```
        {cost.space+=1; cost.setFlags=1; HOLD1;}
        ={emitcomputation($$);
          getreg($$);
          emit("r%d\n",$$->reg);
    };
```

To generate code to evaluate a computation into a register, there are two steps. First, the instructions to compute the values of the subtrees of the computations must be emitted. Then, the instruction that computes the root of the computation from the subtrees is emitted. This final instruction will vary according to what kind of computation has been matched. It might be a (two-address) move instruction, a (three-address) subtract instruction, or a (two-address) negate instruction (et cetera).

The instructions to evaluate the subtrees are emitted by the action codes of matches that are children of the root node. (This is the default order in which Twig actions are evaluated.) The instruction at the root node — all but the last operand — is emitted by the procedure *emitcomputation.* This procedure also frees the registers that had been associated with the computation (to hold the values of its operands).

Then a register (or register-pair) is allocated to hold the result of the computation (by the call to *getreg*), and the destination operand is emitted.

The cost calculation is simple. Computations are marked with the amount of time they take, and the number of bytes required by all but the last operand. On the VAX, a simple register-mode operand takes only 1 byte, so cost.space is incremented by 1. The match is marked as one that sets the condition codes. Finally, cost.hold is set to 1 (or 2 if the size attribute of the node indicates that a register-pair will be required); this is done by the macro HOLD1.

reg: ALLOC(TEMP,reg)

```
    {cost.setFlags=$%2$->cost.setFlags;
     cost.dontDestroy=$%2$->cost.dontDestroy;
     HOLD1; cost.maxregs+=($1$->x.size/4); TOPDOWN;
    }
    ={getreg($1$); $1$->u.temp->number=$1$->reg;
      EVAL;
      givereg($1$);
      $$->reg=$2$->reg; getagain($$); givereg($2$);
     };
```

An ALLOC node provides a register to use as a TEMP node during the computation of a subtree. When control leaves the subtree, the register will be de-allocated.

The action phrase of this rule must be evaluated before the action phrase of the *reg* subtree. This is in contrast to most rules, in which the actions of the children emit assembly code, followed by the action phrase of the root of the rule. To avoid the automatic execution of the child subtree's action phrases, the rule is marked **TOPDOWN,** and the word **EVAL** is used at precisely the point where it is desired the children's actions be executed.

Afterwards, there come calls to the register allocator to indicate that the result of the ALLOC node is in the same register as the result of the second operand. The cost phrase indicates that the cost attributes are those of the second operand.

stm: ALLOC(TEMP,stm)

```
    {cost.maxregs+=($1$->x.size/4); TOPDOWN;}
    ={getreg($1$); $1$->u.temp->number=$1$->reg;
      EVAL;
      givereg($1$);
     };
```

This pattern is similar to the previous one, with some simplifications.

reg: MOVE(TEMP,computation)

```
    {cost.space+=1; cost.setFlags=cost.sideEffect=cost.dontDestroy=1;}
        /* hold 0 */
    ={emitcomputation($2$);
      $$->reg=$1$->u.temp->number;
      getagain($$);
      emit("r%d\n",$1$->u.temp->number);
     };
```

To assign a value into a **TEMP** register, the **MOVE** operation is used. The result of the assignment, as with the **MOVE** operation, is the value assigned.

The *dontDestroy* field of the cost is marked, indicating that this match can't be used in any context that requires overwriting of the register allocated to the MOVE node — since this would also overwrite the TEMP register, which must be saved for possible further use.

The number of registers required to hold the result is zero; this is because an ALLOC operator has (presumably) reserved the **TEMP** register, and no additional temporaries are needed. Because zero is the default, no HOLD specification is required.

The call to *getagain* indicates that there is one more use of the **TEMP** register, which needs to be held until *givereg* is called with the **MOVE** node as an argument.

reg: TEMP

```
    {cost.dontDestroy=1; /* hold 0 */;}
    ={$$->reg=$$->u.temp->number;
      getagain($$);
     };
```

This pattern indicates that a **TEMP** can be used in an expression as well as assigned to. Many of the remarks made in describing **MOVE** — about cost calculation and register allocation — apply here.

Certain other *reg* patterns will be described later.

4.3.4. Locations in memory

The addressing modes of the VAX can be classified into those corresponding to locations in memory and those that are constant or registers. The nonterminal *location* will be used to represent the former.

The twig rules for the different *location* modes are all similar. The space costs are computed straightforwardly as the number of bytes required to specify the operand. The time costs are computed as the amount of time that each kind of operand adds to the cost of the instruction of which it is a part.

The *hold* field of the cost must be the sum of the *hold* fields of all the children of the *location.* This is because the location match does not in itself combine its children into a value in one register; this is only done later when the instruction is executed. For example, in the instruction

addl3 4(r1)[r2],6(r3),r4

both r1 and r2 (of the first operand) must be held until the addl3 instruction is executed. (If the Sethi-

Ullman register-allocation scheme is not used, then the *hold* calculations are not necessary.)

Here are the patterns to match *locations:*

location: reg
```
{cost.space+=1; HOLDALL;}
={$$->kind=1;};
```
location: OP(PLUS,name,reg)
```
{cost.space+=1; HOLDALL;}
={$$->kind=2;};
```
location: OP(PLUS,reg,name)
```
{cost.space+=1; HOLDALL;}
={$$->kind=3;};
```
location: MEM(OP(PLUS,name,reg))
```
{cost.space+=1; cost.time+=4; HOLDALL;}
={$$->kind=4;};
```
location: MEM(OP(PLUS,reg,name))
```
{cost.space+=1; cost.time+=4; HOLDALL;}
={$$->kind=5;};
```
location: name
```
{cost.space+=1;}
={$$->kind=6;};
```

There are two separate parts to the action of an *operand* or *location.* The first part must emit instructions that bring the appropriate values into registers. The second part must emit the assembler specification of the operand. These two parts are not contiguous. Consider this program:

```
movl  abc,r1
subl3 $4,ghi,r3
addl3 (r1),8(r3),r5
```

The first operand of the last instruction was generated from the tree **MEM**(**MEM**(**NAME**$_{abc}$)). When the action phrase of the first operand was invoked by Twig, two things were done: the child of that operand was emitted (i.e. movl abc,r1); and the *kind* of match was recorded (i.e. $$→kind=1). Then the same was done for the second operand of the last instruction.

When the *computation* corresponding to the last instruction was emitted by *emitcomputation*, the function *emitlocation* was called for each operand. This function examines the *kind* of its argument, and emits the appropriate assembly-language phrase:

```
emitlocation(t) Tree t;
{
 switch(t->kind)
    {case 1:  /* reg */
        emit("(r%d)",t->reg); givereg(t); break;
     case 4: /* MEM(OP(PLUS,name,reg)) */
        emit("*"); t=t->u.child[0];  /* fall through */
     case 2: /* OP(PLUS,name,reg) */
        emitname(t->u.child[1]);
        emit("(r%d)",t->u.child[2]->reg);
        givereg(t->u.child[2]->reg); break;
     case 5: /* MEM(OP(PLUS,reg,name)) */
        emit("*"); t=t->u.child[0];  /* fall through */
     case 3: /* OP(PLUS,reg,name) */
        emitname(t->u.child[2]);
        emit("(r%d)",t->u.child[1]->reg);
        givereg(t->u.child[1]->reg); break;
     case 6: /* name */
        emitname(t); break;
    }
}
```

4.3.5. Index mode

Many of the VAX addressing modes can be augmented by an ‘‘index,’’ which is a register multiplied by a constant (1, 2, 4, or 8, depending on the operation code). This is somewhat tricky, as the size of the multiplier is specified not in the address itself, but in the instruction of which the address is a part. The solution is to match any constant, but to remember (in a field of the *cost* variable) what the constant is. Then any other pattern that contains an indexed operand *(ioperand)* must check that the constant is correct.

ioperand: OP(PLUS,location,OP(MUL,reg,CONST))

```
    {cost.isize=$3.3$->u.ivalue;
     cost.space+=1; cost.time+=6; HOLDALL;}
    ={$$->kind=11;};
```

ioperand: OP(PLUS,location,reg)

```
    {cost.isize=1;
     cost.space+=1; cost.time+=6; HOLDALL;}
    ={$$->kind=12;};
```

ioperand: OP(PLUS,OP(MUL,reg,CONST),location)

```
    {cost.isize=$2.3$->u.ivalue;
     cost.space+=1; cost.time+=6; HOLDALL;}
    ={$$->kind=13;};
```

ioperand: OP(PLUS,reg,location)

```
    {cost.isize=1;
     cost.space+=1; cost.time+=6; HOLDALL;}
    ={$$->kind=14;};
```

We assume that intermediate trees are always generated with CONST nodes on the right-hand-side of commutative operators. However, we still need four versions of *ioperands,* because the front end cannot know whether to put **location** or **OP(MUL,reg,CONST)** first. In addition, when the **CONST** is 1, the pattern may be simpler.

The *isize* field of the cost is set to the size of the multiplier; all rules that use *ioperand* check that field.

4.3.6. Destinations

The nonterminal *destination* represents VAX addressing modes that denote storable locations. These are of the form **MEM**(*x*), where *x* is an address.

destination: MEM(location)

```
{cost.time+=4; HOLDALL;}
={$$->kind=24;};
```

destination: MEM(ioperand)

```
{if ($$->x.size != $%1$->cost.isize) ABORT;
 cost.time+=4; HOLDALL;}
={$$->kind=25;};
```

The fact that all nodes include a specification of their representation size makes it possible to handle the checking of *isize* in index-mode operands very simply.

4.3.7. R-value operands

The nonterminal *operand* represents r-expressions: those that may be used as subexpressions, but that do not necessarily represent memory locations. Operands may be registers (*reg*), constants (*name* or *CONSTF*), or *destinations.*

operand: reg

```
{cost.space+=1; HOLDALL;
 cost.setFlags = $%1$->cost.setFlags;}
={$$->kind=21;};
```

operand: name

```
{cost.space+=1;}
={$$->kind=22;};
```

operand: CONST

```
{if (!($$->u.ivalue >= 0 && $$->u.ivalue < 64)) ABORT;
 cost.space=1;}
={$$->kind=22;};
```

operand: CONSTF

```
{cost.space = 1+$$->x.size; }
={$$->kind=23;};
```

operand: destination

```
{HOLDALL;}
```

The nonterminal *name* stands for integer constants, labels, and sums of constants and labels. Any such constant recognized as a *name* will have an appropriate space cost calculated. The rule for using a *name* as an *operand* requires one extra byte to specify the addressing mode, in addition to the space taken by the *name.*

The VAX has a ‘‘short constant’’ addressing mode that fits in just one byte. Such short constants will be recognized by both the *name* and the *CONST* rules, above; but the latter will have the lower cost. This is a simple example of the utility of ambiguous grammars.

As with *locations,* the evaluation pass emits subtrees and records the *kind* of the match; the operand specifier is emitted (later) by a call to the procedure *emitoperand:*

```
emitoperand(t) Tree t;
{switch(t->kind)
       {case 21: /* reg */
        emit("r%d",t->reg); givereg(t); break;
    case 22: /* name */
        emit("$"); emitname(t); break;
    case 23: /* CONSTF */
        emit("$%f", t->u.fvalue); break;
    case 24: /* MEM(location) */
        emitlocation(t->u.child[0]); break;
    case 25: /* MEM(ioperand) */
        emitioperand(t->u.child[0]); break;
    }
}
```

4.3.8. Computations

There are a variety of instructions that do some calculation (involving one or two operands) and move the result into a register or other destination. These will be called *computations.* The VAX assembly language for a *computation* looks like an arithmetic instruction (three-address), a unary arithmetic instruction (two address), or a move instruction; but with the last (destination) operand not specified. As with operands, computations must be emitted in two phases; the first phase emits the instructions that calculated values needed by the various operands of the instrution; the second phase (invoked by a call to *emitcomputation*) emits the appropriate assembler instruction (without the last operand).

computation: operand

```
    {if ($$->x.size>8 || sizecode[$$->x.size]=='.') ABORT;
     cost.setFlags = $%1$->cost.setFlags;
     cost.time += 4; cost.space+=1; HOLDALL;};
```

computation: OP(binop,operand,operand)

```
    {cost.space+=1; HOLDALL;}
        ={$$->kind=31;};
```

computation: UNOP(unop,operand)

```
    {cost.space+=1; HOLDALL;}
    ={$$->kind=32;};
```

computation: UNOP(cvtop,operand)

```
    {cost.space+=1; HOLDALL;}
    ={$$->kind=33;};
```

computation: location

```
    {cost.time+=4; cost.space+=1; HOLDALL;};
```

computation: ioperand

```
    {if ($%1$->cost.isize > 8 || sizecode[$%1$->cost.isize]=='.') ABORT;
     cost.time+=4; cost.space+=1; HOLDALL;};
```

There are several kinds of computations here. A computation can be simply an operand, in which case a move instruction is generated. In this case the *kind* field of the node will have been set by an *operand* tree-pattern, and it is left unchanged by the *computation* pattern.

A computation can be an arithmetic operator applied to two operands, in which case an arithmetic instruction (like **add** or **div** is generated. In this case, the *binop* itself contains the time component of the cost, since different operators take different amounts of time. A computation can be the application of a conversion operator to an operand, in which case a conversion instruction is emitted. Finally, a computation may correspond to the VAX *move address* instruction, for which the source operand is any location (possibly indexed).

As with locations and operands, computations can't be emitted until an entire instruction is built, so a separate procedure traverses the tree looking at the *kind* fields of the nodes:

```
emitcomputation(t) Tree t;
{switch(t->kind)
       {case 31: /* OP(binop, operand, operand) */
     {emit("%s%c3\t",vaxop[t->u.child[0]->op],
            t->u.child[0]->u.op.sizecode[t->x.size]);
      emitoperand(t->u.child[2]); emit(",");
      emitoperand(t->u.child[1]); emit(",");
     } break;
    case 32: /* UNOP(unop,operand) */
     {emit("%s%c\t",vaxop[t->u.child[0]->op],sizecode[t->x.size]);
      emitoperand(t->u.child[1]); emit(",");
     } break;
    case 33:  /* UNOP(cvtop,operand) */
     {emit("cvt%c%c\t",
        t->u.child[0]->u.cvt.sizecodefrom[t->u.child[1]->x.size],
        t->u.child[0]->u.cvt.sizecodeto[t->x.size]);
      emitoperand(t->u.child[1]); emit(",");
     } break;
    case 34: /* UNOP(cvtopu,computation) */
     emitcomputation(t->u.child[1]);
       break;
    case 35: /* OP(LSHIFT,operand,operand) */
     emit("ashl\t");
     emitoperand(t->u.child[2]); emit(",");
     emitoperand(t->u.child[1]); emit(",");
       break;
    case 36: /* OP(RSHIFT,reg,reg) */
     {struct tree q;
      getreg(&q);
      emit("subl3\tr%d,$32,r%d\nextzv\tr%d,r%d,r%d," ,
        t->u.child[2]->reg,q.reg,t->u.child[2]->reg,q.reg,
            t->u.child[1]->reg);
      givereg(&q); givereg(t->u.child[1]); givereg(t->u.child[2]);
     } break;
    case 37: /* OP(RSHIFT,reg,CONST) */
      emit("extzv\t$%d,$%d,r%d,",
        t->u.child[2]->u.ivalue,
        32-t->u.child[2]->u.ivalue,
        t->u.child[1]->reg);
      givereg(t->u.child[1]);
      break;
    case 1: case 2: case 3: case 4: case 5: case 6: case 7: case 8:
         /* location */
     emit("movab\t"); emitlocation(t); emit(",");
       break;
    case 11: case 12: case 13: case 14: case 15:
        /* ioperand */
     emit("mova%c\t",sizecode[sizeioperand(t)]);
     emitioperand(t); emit(",");
       break;
    case 21: case 22: case 23: case 24: case 25: case 26: case 27:
        /* operand */
     {emit("mov%c\t",sizecode[t->x.size]);
      emitoperand(t); emit(",");
```

```
        } break;
      default: emit("# Strange kind of computation into ");
         }
}
```

4.3.9. Boolean conditions

Many computers, including the VAX, evaluate boolean tests in two phases. The first phase, carried out by a compare (or arithmetic) instruction, sets the condition flags depending on whether the result is zero/nonzero, negative/nonnegative, et cetera. The second phase performs a conditional jump depending on the values of the condition flags.

That breakdown into phases turns out to be convenient to recognize in the design of a code generator. The nonterminal *flag* will match a comparison or arithmetic instruction, and will emit an instruction that sets the condition flags of the machine. The nonterminal *test* will match a boolean expression built from one or more *flags* using the operators **NOT** (boolean negation) and **CAND** (conditional-and, in which the second argument is evaluated *only* if the first argument is true). The conditional-or can be built from *NOT* and *CAND* using DeMorgan's law.

A *test* corresponds to a block of instructions that will branch to a given address if a given condition (built from **CAND**, **NOT**, and **flags**) is true; otherwise control will fall through to the next block of instructions. The nonterminal *ntest* (for *negated test*) stands for a block of instructions that will branch if a given condition is false.

The action phrases of *tests* and *ntests* must be evaluated **TOPDOWN** to set the *truebranch* attribute of the children of the *test* (or *ntest*) before the children's instructions — which will contain jumps to the *truebranch* label — are emitted.

```
test: CAND(ntest,test)
     {TOPDOWN;}
     ={Label l=newLabel();
       $1$->x.truebranch=l;
       $2$->x.truebranch=$$->x.truebranch;
       EVAL;
       emit("%s:\n",l->s);
          };
ntest: CAND(ntest,ntest)
     {TOPDOWN;}
     ={$1$->x.truebranch=$$->x.truebranch;
       $2$->x.truebranch=$$->x.truebranch;
       EVAL;
     };
test: NOT(ntest)
     {TOPDOWN;}
     ={$1$->x.truebranch=$$->x.truebranch;
       EVAL;
      };
ntest: NOT(test)
     {TOPDOWN;}
     ={$1$->x.truebranch=$$->x.truebranch;
       EVAL;
      };
ntest: flag
     {cost.space += 4; cost.time += 8;}
     ={emit("j%s\t%s\n",vaxop[negate($$->kind)],$$->x.truebranch->s);};
test: flag
     {cost.space += 4; cost.time += 8;}
     ={emit("j%s\t%s\n",vaxop[$$->kind],$$->x.truebranch->s);};
```

The productions above show how to build *tests* subtrees from *flag* primitives, which represent the condition code settings of the VAX. There are three ways to set the condition codes: by a *cmp* (compare) instruction, which subtracts one operand from the other and sets the condition codes according to the result; by a *tst* (test) instruction, which is like a compare against zero; or by using the condition code generated from the previous instruction. The latter is only appropriate if the previous instruction generates a useful condition code; the *setFlags* field of the *cost* is maintained to provide this information.

The *kind* field node carries the information about which condition is to be tested in the jump instruction. The rules for *flag* set this field, and the rules for *test* and *ntest* read the field.

flag: OP(relop,operand,operand)

```
{cost.space += 1; cost.time += 4;}
    ={emit("cmp%c\t",$1$->u.op.sizecode[$2$->x.size]);
  emitoperand($2$); emit(","); emitoperand($3$); emit("\n");
  $$->kind=$1$->op;
 };
```

flag: OP(relop,operand,zero)

```
{cost.space += 1; cost.time += 4;}
={emit("tst%c\t",$1$->u.op.sizecode[$2$->x.size]);
  emitoperand($2$); emit("\n");
  $$->kind=$1$->op;
 };
```

flag: OP(relop,stm,zero)

```
{if (!$%2$->cost.setFlags) ABORT;}
={$$->kind=$1$->op;};
```

Finally, a test may be turned back into a number (0 or 1) by the **BOOL** operator. One could imagine a machine with a convenient ‘‘move condition code to register’’ instruction; but the VAX does not have one, so a conditional jump is used:

reg: BOOL(test)

```
{cost.space+=9; cost.time+=12; cost.setFlags=1; HOLD1; TOPDOWN;}
={Label merge=newLabel();
  $1$->x.truebranch=newLabel();
  EVAL;
  getreg($$);
  emit("clrl\tr%d\njbr\t%s\n%s:\nmovl\t$1,r%d\n%s:\n",
    $$->reg,merge->s,$1$->x.truebranch->s,$$->reg,merge->s);
 };
```

4.3.10. Expressions too large to fit in registers

In languages where entire records or arrays may be assigned or passed by value, it is necessary to match patterns like

MOVE(MEM(location),MEM(location))

where the *size* attribute of the **MEM** nodes (equivalently, of the **MOVE** node) are arbitrarily large. The patterns previously shown all assume that fetched values will fit in registers (or register-pairs). However, when the data-value is too large, we may match the entire pattern as a block-move.

Some languages allow multiple assignments:

a := b := c := d;

where a, b, c, and d are all record variables. Thus the patterns to be matched are slightly more general than the one shown above:

bigval: MEM(reg)

```
    ={$$->reg=$1$->reg; getagain($$); givereg($1$);};
```

bigval: MOVE(MEM(location),bigval)

```
    {if ($$->x.size>65535) ABORT;
     cost.space+=2; cost.time+= 56+ 37*$$->x.size/10; cost.sideEffect=1;
    }
    ={int i;
      for(i=0;i<=5;i++) makeAvail(i,07700);
      emit("movc3\t$%d,(r%d),",$$->x.size,$2$->reg);
      emitlocation($1.1$);  emit("\n");
      $$->reg=$2$->reg; getagain($$); givereg($2$);
     };
```

A register is associated with each *bigval;* this register holds the address (not the value) of the appropriate record value. The register allocator must be told (via `makeAvail` to make register 0 through 5 available for this instruction to use.

4.3.11. Operators

The operator classes, which match tokens like **PLUS** and **NEG,** are recognized as the nonterminals *binop* (binary operator), *unop* (unary operator), *cvtop* (conversion operator), and *cvtopu* (unsigned conversion operator) by a set of trivial rules. For each operator, the cost of execution time is recorded, as well as the size code vector (to indicate operand sizes in integer format — b,w,l — or floating point format — f,d — as appropriate).

```
unop: FNEG      {cost.time = 8;} ={$$->u.op.sizecode=sizecodef;};
unop: NEG       {cost.time = 4;}={$$->u.op.sizecode=sizecodei;};
unop: COMP      {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};

binop: FPLUS    {cost.time = 8;} ={$$->u.op.sizecode=sizecodef;};
binop: FMINUS   {cost.time = 8;} ={$$->u.op.sizecode=sizecodef;};
binop: FMUL     {cost.time = 12;} ={$$->u.op.sizecode=sizecodef;};
binop: FDIV     {cost.time = 40;} ={$$->u.op.sizecode=sizecodef;};

binop: PLUS     {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};
binop: MINUS    {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};
binop: MUL      {cost.time = 16;} ={$$->u.op.sizecode=sizecodei;};
binop: DIV      {cost.time = 100;} ={$$->u.op.sizecode=sizecodei;};
binop: OR       {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};
binop: XOR      {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};
binop: ANDNOT   {cost.time = 4;} ={$$->u.op.sizecode=sizecodei;};

relop: EQ       ={$$->u.op.sizecode=sizecodei;};
relop: NEQ      ={$$->u.op.sizecode=sizecodei;};
relop: LT       ={$$->u.op.sizecode=sizecodei;};
relop: LEQ      ={$$->u.op.sizecode=sizecodei;};
relop: ULT      ={$$->u.op.sizecode=sizecodei;};
relop: ULEQ     ={$$->u.op.sizecode=sizecodei;};
relop: GT       ={$$->u.op.sizecode=sizecodei;};
relop: GEQ      ={$$->u.op.sizecode=sizecodei;};
relop: UGT      ={$$->u.op.sizecode=sizecodei;};
relop: UGEQ     ={$$->u.op.sizecode=sizecodei;};

relop: FEQ      ={$$->u.op.sizecode=sizecodef;};
relop: FNEQ     ={$$->u.op.sizecode=sizecodef;};
relop: FLT      ={$$->u.op.sizecode=sizecodef;};
relop: FLEQ     ={$$->u.op.sizecode=sizecodef;};
relop: FGT      ={$$->u.op.sizecode=sizecodef;};
relop: FGEQ     ={$$->u.op.sizecode=sizecodef;};

cvtop: CVTSS    {cost.time=4;}
          ={$$->u.cvt.sizecodefrom=sizecodei;
            $$->u.cvt.sizecodeto=sizecodei;
           };
cvtop: CVTSF    {cost.time=8;}
          ={$$->u.cvt.sizecodefrom=sizecodei;
            $$->u.cvt.sizecodeto=sizecodef;
           };
cvtop: CVTFS    {cost.time=8;}
          ={$$->u.cvt.sizecodefrom=sizecodef;
            $$->u.cvt.sizecodeto=sizecodei;
           };
cvtop: CVTFF    {cost.time=8;}
          ={$$->u.cvt.sizecodefrom=sizecodef;
            $$->u.cvt.sizecodeto=sizecodef;
           };

cvtopu: CVTSU;
cvtopu: CVTUU;
```

cvtopu: CVTUS;

4.3.12. Constants and labels

Literal constants and labels are interchangeable in some contexts and not in others. When either will do, the symbol *name* is used:

```
name: CONST
        {int i=$$->u.ivalue;
         cost.space= i>=0 ? i>32767 ? 4 : i>127 ? 2 : 1
                          : i<-32768? 4 : i<-128? 2 : 1 ;
        };

name: NAME
        {cost.space=4;};
name: OP(PLUS,NAME,CONST)
        {cost.space=4;};
```

Note that the amount of space occupied by a constant depends on the size (byte, word, or longword) of the constant. The same is true for labels, but it's harder to determine the actual space requirements before assembly, so a conservative estimate is used.

4.3.13. Storing into memory

There are two different ways to store values into memory. If a value is in a register, then it may be moved into memory; the value remains in the register and may be re-used directly. Thus, a **MOVE** of a register value may be matched as a register.

If, on the other hand, a value is computed and stored in the same instruction, so that it is never explicitly put into a register, then the resulting computation may not be matched as a register; it must be matched as a statement.

```
reg: MOVE(destination,reg)
    {cost .time+= 12; cost.space+=2; cost.setFlags=1; cost.sideEffect=1;
     HOLD1;}
    ={$$->reg=$2$->reg;
      getagain($$); givereg($2$);
      emit("mov%c\tr%d,",sizecode[$$->x.size],$2$->reg);
      emitoperand($1$);
      emit("\n");
     };
```

The **MOVE** operator stores its right operand into memory at the address specified by its left operand; it also yields the right operand as its result. Thus, a register may match a **MOVE** node if the right operand matches as a register; the register will be moved into memory and will remain unchanged as the result of the **MOVE**.

The left operand may be any destination. The time required is 1.2 microseconds over and above the time required to compute the *destination* and the *reg.* The space required is the space for the *destination* plus 2 bytes: one for the ''mov'' opcode and one for the register-mode operand. This instruction sets the condition codes appropriately, has a side effect, and requires 1 register (or register-pair) to hold its ''result.'' These facts are indicated in the cost phrase (`HOLD1` looks at the *size* of the node to determine how many registers are required).

Instead of allocating a new register for the **MOVE** node, the result will be kept in the same register as was used for the right operand. The register-number must be copied from the *reg* attribute field of the right child into the *reg* field of the root. The calls to *getagain* and *givereg* tell the register-allocator that this has been done. Then a move instruction is generated; *emitoperand* emits the destination l-operand.

The VAX actually has several different move instructions:

movb to move a byte

movw to move a word (two bytes)

movl to move a longword (four bytes)

movq to move a quadword (eight bytes)

The array *sizecode* is indexed by the size of the **MOVE** node (1, 2, 4, or 8) and has as its elements the appropriate ASCII characters for the different sizes:

```
char sizecode[] = ".bw.l...q";
```

Thus, this **MOVE** pattern takes care of storing all sizes for which the VAX has a move instruction.

If the value to be stored is never in a register, then the result of the computation is not available for computations in the parent node of the tree. That is, such a **MOVE** can be matched as a statement, but not as a register:

stm: MOVE(destination,computation)

```
{cost.time += 8; cost.setFlags=1; cost.sideEffect=1;}
={emitcomputation($2$);
  emitoperand($1$);
  emit("\n");
 };
```

There are some cases where an operand in memory is to be destructively changed by a two-address instruction. When one of the source operands is the same as the destination, this kind of pattern can be matched:

stm: MOVE(destination,OP(binop,operand,operand))

```
{if (cost.sideEffect || !equal($1$,$2.2$)) ABORT;
 cost.time -= $%3$->cost.time; cost.space -= $%3$->cost.space;
 cost.space += 1; cost.time += 8; cost.setFlags=1; cost.sideEffect=1;
 TOPDOWN;
}
={$%3$->cost.dontEval=1; EVAL;
  emit("%s%c2\t",vaxop[$2.1$->op],$2.1$->u.op.sizecode[$$->x.size]);
  emitoperand($2.3$); emit(","); emitoperand($1$); emit("\n");
 };
```

stm: MOVE(destination,OP(binop,operand,operand))

```
{if (!(nodeops[$2.1$->op].attributes&PROPcommute)) ABORT;
 if (cost.sideEffect || !equal($1$,$2.3$)) ABORT;
 cost.time -= $%4$->cost.time;  cost.space -= $%4$->cost.space;
 cost.space += 1; cost.time += 8; cost.setFlags=1; cost.sideEffect=1;
 TOPDOWN;
}
={$%4$->cost.dontEval=1; EVAL;
  emit("%s%c2\t",vaxop[$2.1$->op],$2.1$->u.op.sizecode[$$->x.size]);
  emitoperand($2.2$); emit(","); emitoperand($1$); emit("\n");
 };
```

If the *destination* is the same as the second *operand* (such a test is made by the *equal* function), and if the operand does not cause any side effects, then the two-address form of the instruction can be used. The operand will not be emitted. In this case, its cost must be subtracted from the node's cost, as it has been prematurely added in the *DEFAULT_COST*. Furthermore, not be emitted; this fact is communicated by setting *dontEval* for that leaf of the pattern.

The second version of this pattern works only for commutative operators, and does the same optimization for the first operand.

Finally, there are patterns corresponding to the VAX *increment, decrement,* and *clear* instructions.

stm: MOVE(destination,OP(PLUS,operand,CONST))

```
{if (!(equal($1$,$2.2$) &&
    ($2.3$->u.ivalue == 1  || $2.3$->u.ivalue == -1)))  ABORT;
 cost.time -= $%2$->cost.time;   cost.space -= $%2$->cost.space;
 cost.time += 8; cost.space+= 1;
 cost.setFlags=1; cost.sideEffect=1;
 TOPDOWN;
}
={$%2$->cost.dontEval=1; EVAL;
  emit("%s%c\t", $2.3$->u.ivalue>0 ? "inc":"dec",
        sizecode[$1$->x.size]);
  emitoperand($1$);
  emit("\n");
 };
```

stm: MOVE(destination,zero)

```
{cost.time += 4; cost.space += 1; cost.setFlags=cost.sideEffect=1;}
={emit("clr%c\t",sizecode[$1$->x.size]);
  emitoperand($1$);
  emit("\n");
 };
```

4.3.14. Destructive register operations

The two-address arithmetic instructions put newly computed values into the register containing the second operand. These can be matched as registers only if the *dontDestroy* field of the given register is not set. (It would be set if the register's value was used in several places; in this case it would have been specified as a **TEMP.)**

The patterns below produce VAX instructions that destroy one of their operands:

reg: OP(binop,reg,operand)

```
    {cost.space += 2; cost.setFlags=1; HOLD1;
     if ($%2$->cost.dontDestroy) ABORT;
    }
    ={emit("%s%c2\t",vaxop[$1$->op],$1$->u.op.sizecode[$$->x.size]);
      emitoperand($3$);
      emit(",r%d\n",$2$->reg);
      $$->reg=$2$->reg;
      getagain($$); givereg($2$);
     };
```

reg: OP(binop,operand,reg)

```
    {cost.space += 2; cost.setFlags=1; HOLD1;
     if ($%3$->cost.dontDestroy) ABORT;
     if (!(nodeops[$1$->op].attributes&PROPcommute)) ABORT;
    }
    ={emit("%s%c2\t",vaxop[$1$->op],$1$->u.op.sizecode[$$->x.size]);
      emitoperand($2$);
      emit(",r%d\n",$3$->reg);
      $$->reg=$3$->reg;
      getagain($$); givereg($3$);
     };
```

reg: OP(PLUS,reg,CONST)

```
    {if (!($3$->u.ivalue == 1  || $3$->u.ivalue == -1))  ABORT;
     if ($%1$->cost.dontDestroy) ABORT;
     cost.time += 4; cost.space += 2; cost.setFlags=1; HOLD1;
    }
    ={$$->reg=$2$->reg; getagain($$); givereg($2$);
      emit("%s%c\tr%d\n", $3$->u.ivalue>0 ? "inc":"dec",
            sizecode[$$->x.size], $$->reg);
     };
```

4.3.15. Exceptional instructions

Some VAX instructions do not fit conveniently into any pattern. These include the shift, modulo, and unsigned conversion operations. Individual patterns are provided for these instructions.

computation: OP(RSHIFT,reg,reg)

```
    {if ($$->x.size != 4 || $2$->x.size != 4 || $3$->x.size != 1) REWRITE;
     cost.time += 24; cost.space+=8; HOLDALL;}
    ={if ($$->x.size != 4 || $2$->x.size != 4 || $3$->x.size != 1)
        {SUBSTITUTE(tCONVERT($$->x.size,OPcvtuu,
                 tOP(4,OPrshift,
                tCONVERT(4,OPcvtuu,$2$),
                tCONVERT(1,OPcvtuu,$3$))));}
      else $$->kind=36;
     };
```

computation: OP(RSHIFT,reg,CONST)

```
    {if ($$->x.size != 4 || $2$->x.size != 4) ABORT;
     cost.time += 20; cost.space += 4; HOLDALL;}
    ={$$->kind=37;};
```

computation: OP(LSHIFT,operand,operand)

```
    {if ($$->x.size != 4 || $2$->x.size != 4 || $3$->x.size != 1) REWRITE;
     cost.time+=20; cost.space+=1; HOLDALL;}
    ={if ($$->x.size != 4 || $2$->x.size != 4 || $3$->x.size != 1)
        {SUBSTITUTE(tCONVERT($$->x.size,OPcvtuu,
                 tOP(4,OPlshift,
                tCONVERT(4,OPcvtuu,$2$),
                tCONVERT(1,OPcvtuu,$3$))));}
      else $$->kind=35;
    };
```

reg: OP(MOD,computation,reg) *(/* true modulo, not remainder */)*

```
    {cost.space+=17; cost.time+=142; cost.setFlags=1; HOLD1;}
    ={Tree t=tCONST(8,0); Label lab=newLabel();
      emitcomputation($2$);
      getreg(t);
      emit("r%d\n",t->reg+1);
      if ($2$->x.size!=4)
         emit("cvt%cl\tr%d,r%d\n",
            sizecode[$2$->x.size],t->reg+1,t->reg+1);
      emit("ashq\t$-32,r%d,r%d\n",t->reg,t->reg);
      emit("ediv\tr%d,r%d,r%d,r%d\n",$3$->reg,t->reg,t->reg+1,t->reg);
      emit("tstl\tr%d\njgeq\t%s\naddl2\tr%d,r%d\n%s:\n",
            t->reg,lab->s,$3$->reg,t->reg,lab->s);
      $$->reg=t->reg;
      getagain($$); givereg(t); givereg($3$); TreeFree(t);
     };
```

reg: UNOP(cvtopu,reg)

```
    {if ($$->x.size > $2$->x.size) ABORT;}
    ={$$->reg=$2$->reg;
      getagain($$); givereg($2$);
     };
```

reg: UNOP(CVTSU,reg)

```
    {if ($$->x.size > $2$->x.size) REWRITE; else ABORT;}
    ={mtSetNodes($$,2,tUNOP($$->x.size,OPcvtss,$2$));};
```

reg: UNOP(CVTUS,reg)
```
    {if ($$->x.size > $2$->x.size) REWRITE; else ABORT;}
    ={mtSetNodes($$,2,tUNOP($$->x.size,OPcvtuu,$2$));};
```

reg: UNOP(CVTUU,computation)
```
    {cost.time+=4; cost.space+=3;}
    ={getreg($$);
      emit("clrl   r$d0,$$->reg);
      emitcomputation($2$);
      emit("r%d0,$$->reg);
     };
```
reg: OP(AND,reg,reg)
```
    {REWRITE;}
    ={mtSetNodes($$,3,tUNOP($$->x.size,OPcomp,$3$));
      $1$->op=OPandnot;
     };
```

4.3.16. Procedure calls and parameters

A **CALL** node has two children: an argument list, and a procedure address. Each argument in the list is pushed onto the stack by the evaluation of an **ARG** node, and then a *calls* instruction is emitted.

The forms of **ARG** nodes are similar to the forms of **MOVE** nodes. That is, the argument may be a *computation*, an *operand*, or a *bigval*. In any case, the proper instruction is emitted to copy the argument onto the stack. The **NOARGS** node is used to terminate an argument list.

4.3.17. Constants with value zero

zero: CONST `{if ($$->u.ivalue != 0) ABORT;};`
zero: CONSTF `{if ($$->u.fvalue != 0.0) ABORT;};`

4.3.18. Miscellany

The **ESEQ** operator has a statement as its left child, and an expression as its right. The statement is to be evaluated, and the resulting value is that of the expression.

reg: ESEQ(stm,reg)
```
    {cost.setFlags=$%2$->cost.setFlags; HOLD1;
     cost.dontDestroy=$%2$->cost.dontDestroy;}
    ={$$->reg=$2$->reg; getagain($$); givereg($2$);};
```

This is not as general as it could be; one could imagine that these patterns would also be valid:

location: ESEQ(stm,location)
operand: ESEQ(stm,operand)
computation: ESEQ(stm,computation)
bigval: ESEQ(stm,bigval)

As an example of a suboptimal instruction sequence that results from the absence of these four rules, consider how this tree

$$\mathbf{MEM}(\mathbf{ESEQ}(\mathbf{MOVE}(\mathbf{TEMP}_{r0},\mathbf{CONST}_0),\mathbf{NAME}_A))$$

may be matched as a register. Without the rule **location: ESEQ(stm,location)** the node $\mathbf{NAME}_A$ must be matched as a register, resulting in the instructions:

```
clrl    r0
moval   A,r1
movl(r1),r1
```

But with the extra rule, **NAME**$_A$ may be matched as a location, which does not generate an extra instruction:

```
clrl    r0
movl    A,r1
```

5. The MC68020 instruction set

The Motorola MC68020 has a large, complex, and somewhat unorthogonal instruction set[22]. There are many addressing modes, each instruction takes only a particular subset of these modes, and there are several different forms of many instructions. (In particular, the MC68020 has several new addressing modes not found on its predecessor, the MC68010.)

5.1. Nonterminal symbols for the MC68020

The Twig specification of the MC68020 has 35 nonterminal symbols (and 141 grammar rules, not including floating point operations), while the specification for the VAX has only 20 symbols (and 112 rules). This is for several reasons: The addressing modes are more difficult to specify; there are two kinds of registers; and temporary values must be treated more generally.

The full specification will not be given here, as it is in many respects similar to the specification of the VAX. However, some of the differences will be summarized.

5.1.1. Registers

The MC68020 has eight ‘‘address registers’’ a0–a7 and eight ‘‘data registers’’ d0–d7. At least one of the operands of an arithmetic instruction must be a data register, and the addressing modes rely more heavily on address registers (though these rules are often broken). This means that there must be two nonterminals representing registers, not just one. We will call them **areg** and **dreg** here.

Furthermore, it is necessary to distinguish between registers holding temporary values that are used just once, and registers holding local variables used many times. The former may be the destinations of two-address arithmetic instructions, and the latter may not. That is, if the value d_3+5 is to be computed, and there is an instruction that computes $d_3 \leftarrow d_3$+5, then this instruction may be used only if there is no need to use the original value d_3 in other places. We will let **dregx** stand for a data register whose value is needed only once, and **aregx** for an address register whose value is needed only once. These nonterminals represent registers whose contents may be overwritten as new values are computed from them.

Of course, any register that is overwritable may be used in an instruction that doesn’t happen to overwrite it. This is expressed by the rules:

dreg: dregx
areg: aregx

which have no cost and which emit no instructions.

In the specification for the VAX, non-overwritability was expressed using an element of the cost vector: *dontDestroy*. This is less accurate than having two nonterminal symbols. When there are two nonterminals (like **dreg** and **dregx**) then the dynamic programming will compute the cheapest way of evaluating a given tree node into a use-once register, and the cheapest way of evaluating into a must-save register. When one nonterminal is used, only the single cheapest way to compute the node is found; then the *dontDestroy* field is looked at as an afterthought. This could lead to a more expensive match overall. On the VAX, whenever a two-address instruction pattern rejects a match because *dontDestroy* is set, there is always a corresponding three-address pattern available at only a slightly higher cost. But the MC68020 has only two-address instructions, so the one-nonterminal approximation would not work nearly as well.

Finally, there are contexts where either an address register or a data register is permitted, so the nonterminal **reg** stands for either kind of register. Thus, where the VAX specification has one nonterminal to stand for a register, the MC68020 specification requires five.

5.1.2. Addressing modes

The addressing modes of the MC68020 are not much more complicated than those of the VAX, but it is more complicated to represent them in a context-free grammar. This is because a typical addressing mode has several components, any of which are optional, which are added together to form the address. Because addition is commutative and associative, the grammar must represent any permutation of any subset of these components.

By factoring the grammar for addressing modes heavily, the number of rules is brought down substantially from *n* factorial — at the cost of introducing several nonterminal symbols. In fact, 15 symbols are used to express the MC68020 addressing modes, where 4 symbols are used for the VAX addressing modes.

5.2. Duplication of rules

Because there are several symbols that represent registers, some of the instructions (and pseudo-instructions like **ALLOC)** that can operate on any kind of register must be replicated for each class. For example, the VAX specification has a rule:

reg: MOVE(destination,reg)

This means that a register may be stored into memory, and then re-used as a value. Since the M68020 has four nonterminals for registers, there must be four copies of this rule:

dreg: MOVE(destination,dreg)
dregx: MOVE(destination,dregx)
areg: MOVE(destination,areg)
aregx: MOVE(destination,aregx)

That is, a many-use data register may be stored into memory, but remains a many-use data register, etc. This replication of rules occurs in many places throughout the specification.

5.3. Cost computations

In a highly pipelined computer like the MC68020, it is difficult to calculate the time required to execute a particular instruction, as the instructions overlap each other in time. Similarly, it is difficult to impute costs to the various components of an addressing mode. On the other hand, the dynamic programming algorithm requires that the costs of subtrees must be independent of each other.

In this situation, an approximation must be made. The cost of each instruction and each addressing-mode component must be estimated independently of context. Motorola has published detailed timing charts which give the timings of the different classes of instructions in the various addressing modes, and by comparing different entries in these tables, costs can be approximated for each rule in the specification.

(This problem is less noticeable with the VAX, but only because detailed instruction timing data is not available from the manufacturer!)

6. Performance

Three are three aspects to consider when evaluating a code-generator generator: the conciseness of the specification, the efficiency of code generation, and the efficiency of the generated code.

6.1. Conciseness

The specification described in this paper for the VAX architecture takes 112 rules. (About 40 of these are trivial — for example, the rule that names **XOR** as a binary operator.) These rules cover about 121 different instructions and about 12 different addressing modes. (The bit-field, loop-control, and character

string instructions are not specified.)

Each rule requires a few lines to write down. It is quite possible that a special-purpose language for the action (instead of the C programming language) would reduce the size of each rule's specification.

The specification described here may be compared with other work in the literature (see Table 1). The number of rules in a specification — quoted in the table — is interesting; but perhaps more important is the flexibility of the specification language, and the interactions between rules. The dynamic-programming algorithm is very forgiving, in that special-case rules don't prevent general-case rules from working; and in general each rule may be specified without considering how it will impinge on other rules.

No claim is made that this is the ''optimally concise'' specification of a machine architecture. Indeed, section 7 summarizes the redundancies, along with some possible improvements.

Authors	Rules	Remarks
Appel	112	Present work. Dynamic programming, attribute grammar. 63 terminals, 20 nonterminals, 112 productions.
Graham, Henry, Schulman[15]	458	LR(1) Grammar. Before type replication: 115 terminals, 96 nonterminals, 458 productions. After type replication: 219 terminals, 148 nonterminals, 1073 productions.
Fraser, Davidson[23]	222	Peephole-optimization based code generator. 38 instruction-descriptions, 15 address-mode descriptions each replicated up to 4 times, 29 miscellaneous, 100 translation rules. Does not include floating point or 16-bit word datatypes.
Aigrain, Graham, Henry, McKusick, Pelegri-Llopart[16]	146	Improvement of Graham, Henry, Schulman. 146 ''meta-pacs'' are automatically transformed into 662 LR(1) productions on 246 grammar symbols.
Kessler[24]	238	Architecture analysis to find ''idioms.'' 238 instruction descriptions are automatically reduced to 111 instruction families; from these, 1273 idioms are generated. These idioms can be used by code generators.

Table 1. Specifications of the VAX.

6.2. Generated code

The object code produced by the dynamic-programming algorithm is quite good. A backtracking program that solves the 8-queens problem is used here as an example. (Actually, 18 queens are used on an 18 by 18 board.) The program is shown in Appendix 1, along with its compilations into assembly code for both the VAX and the MC68020. The generated code appears quite good; the only way of improving it seems to be to keep local variables in registers, which is beyond the scope of an instruction-selection algorithm.

The compiler is organized in three phases: the front end translates the input language into machine-independent intermediate representation trees; then there is some machine-independent constant-folding in the trees; then the Twig code generator translates the trees into assembly language. The constant-folding improves the trees considerably.

6.3. Time and space for code generator

The code generators produced by Twig can be expected to run in $O(c_1 dN)$ time and $O(c_2 kN)$ space, where d is the number of nonterminals in the grammar, N is the size of the input tree, and c_i are constants depending on the implementation. The value of k is an (approximately linear) function of the number of rules in the description (and their sizes). Because (for any given Twig specification) d and k are constant, this means that the code generators will run in linear time and linear space. In practice, because d is as much as 20 or 40, the efficiency is not as great as that of some of the other linear-time and linear-space algorithms in the literature.

The following benchmarks were performed on a VAX 8700. Generating assembly code for the MC68020, from a 283-node tree (for the 18-queens backtracking procedure shown in appendix 1), used about 256000 bytes and 1.23 seconds. For the VAX, the same intermediate tree used 196000 bytes and 0.48 seconds. The better performance for the VAX is undoubtedly attributable to the fact that the VAX's description has fewer nonterminal symbols, especially for its addressing modes.

These values for space and time include all overhead: the time required to format the assembly-language output, and the costly overhead (several bytes and many microseconds per record) of the *malloc* memory-allocator. The Twig program, and the author's cost and evaluation fragments, could undoubtedly be tuned to improve the performance significantly.

The space requirements can easily be reduced. Though the dynamic programming algorithm requires space linear in the size of the tree, there is no reason to make a tree as large as an entire procedure. Instead, only one statement at a time should be generated. This will work as long as there is no nontrivial tree-pattern in the specification that crosses a statement boundary. In both descriptions, there is only a rule of the form

stm: SEQ(stm, stm)

which is a ''trivial'' rule. By doing a top-down tree traversal of **SEQ** nodes, the ''top-level'' statements could be easily found, and the code generator could be called on each one. This would save no time, but now the space required would be proportional to the size of the largest statement generated, rather than to the size of the largest procedure.

7. Problems and Improvements

The goal of this research has been to find concise specifications for the instruction sets of complicated machines. One way of making a specification concise is to use a macro preprocessor or other data compression algorithm, so that a short but semantically opaque specification can be expanded automatically into a long but usable specification. This method is unattractive because there is no guarantee that the structure of the specification will match the structure of the architecture, and it is difficult to provide axioms for the manipulation of macros.

Instead, we seek to find a specification that simultaneously is concise and has a structure matching that of the architecture it specifies. The Twig specifications given in this paper are reasonably concise, but they do contain patterns of repetition, where certain features of the instruction set must be specified several times in different contexts.

This problem could be solved if Twig were somewhat less general. The Twig system is not so much a code-generator-generator as it is a general tree-pattern matcher. By making the Twig system aware of certain identities, the size of specifications could be reduced.

The next section will describe some of the redundancies in current specifications, and some ideas that could eliminate them.

7.1. Commutative operators

One general class of problems appears in the specification of the MC68020 addressing modes. Some parts of these addressing modes take the form:

$$d + A_n + X_m \cdot S$$

where each of the three terms is optional. (The meaning of each term is not important in this discussion.) Because the plus operator is commutative and associative, any of the following patterns describes this addressing mode:

$d + A_n + X_m \cdot S$	$A_n + X_m \cdot S$	d
$d + X_m \cdot S + A_n$	$X_m \cdot S + A_n$	A_n
$A_n + d + X_m \cdot S$	$d + X_m \cdot S$	$X_m \cdot S$
$A_n + X_m \cdot S + d$	$X_m \cdot S + d$	0
$X_m \cdot S + d + A_n$	$d + A_n$	
$X_m \cdot S + A_n + d$	$A_n + d$	

The constant-folder always arranges PLUS-trees to associate to the left; otherwise, there would be even more patterns! These 16 patterns can be reduced to 10 (in ‘‘standard’’ Twig) by the introduction of an additional nonterminal symbol, and perhaps to 9 if we omit the last one (if perhaps *d* itself matches 0). This is still a large number of patterns, however, especially considering that they represent only one component of one addressing mode.

Perhaps a good solution would be to introduce a permutation operator, so that **OP(PLUS,[a,b])** — using square brackets to denote the permutable children — would behave like either of the two patterns **OP(PLUS,a,b)** or **OP(PLUS,b,a).** Unfortunately, the patterns for the addition of three terms would still be repetitive:

OP(PLUS,[OP(PLUS,[a,b]),c])
OP(PLUS,[OP(PLUS,[c,a]),b])

because the first of these rules by itself could not match **OP(PLUS,OP(PLUS,a,c),b).**

7.2. Different treatments of the same registers

Another case of duplication occurs when instructions that move values into registers must be represented in two different ways: once to produce intermediate values for tree nodes, and once to explicitly move values into temporary registers. For example, a (hypothetical) **add** instruction:

reg: OP(PLUS,reg,reg)
reg: MOVE(TEMP,OP(PLUS,reg,reg))

Since these two rules describe the same instruction, the costs will be the same, as will the emitted code.

There’s no obvious solution here. This problem may just be a symptom of the inconsistent treatment of temporary variables in a tree-based intermediate form: those that are used just once are internal nodes of the tree, while those used more than once must be handled differently.

7.3. Pseudo-operations

The third kind of duplication occurs when there are classes of rules that work on any of several nonterminal symbols, but that require the same nonterminal throughout. Consider the **ALLOC** pseudo-operation. The tree **ALLOC(TEMP,exp)** means that a new temporary register **TEMP** should be allocated, then the subtree **exp** should be generated, then the temporary should be released. This is a way of providing local scope for compiler-introduced temporaries. The symbol **exp** in this pattern is just a label of convenience, however; it appears in the abstract description of the intermediate language, but it is not a nonterminal of any particular machine-description. One way of introducing **ALLOC** into a machine description is to replace **exp** by **reg** as follows:

reg: ALLOC(TEMP,reg)

But this artificially forces the subexpression to be computed into a register, when it may not be natural to do so. For optimal results, it is necessary to have all these rules (for the VAX; even more are necessary for the MC68020!):

reg: ALLOC(TEMP,reg)
stm: ALLOC(TEMP,stm)
reg: ALLOC(TEMP,reg)
bigval: ALLOC(TEMP,bigval)
operand: ALLOC(TEMP,operand)
location: ALLOC(TEMP,location)
ioperand: ALLOC(TEMP,ioperand)
name: ALLOC(TEMP,name)
computation: ALLOC(TEMP,computation)
flags: ALLOC(TEMP,flags)
zero: ALLOC(TEMP,zero)

There is one rule here for each nonterminal that can represent **exp.** In the specification given in section 4, only a small subset of these are used, for the sake of sanity and readability. The **ESEQ** pseudo-operation (similar to the ‘‘comma’’ operator in C) behaves analogously.

This problem could be handled by making the left-hand-side nonterminal decidable by the cost function, rather than having it be fixed (as in the current system). Then, a new nonterminal **exp** could be made, with rules of the form

```
exp: reg    {cost.nonterminal = REG;}
exp: operand    {cost.nonterminal = OPERAND;}
...
```

for each nonterminal that represents an expression. These rules would store in their cost field the particular nonterminal that was matched. Finally a rule of the form

?: ALLOC(TEMP,exp) `{leftHandSide := $%1$->cost.nonterminal;}`

would describe all ‘‘versions’’ of **ALLOC.**

7.4. Parameterized nonterminals

Some nonterminals can match in more than one mode. For example, the Twig specification for the MC68020 has two different nonterminals representing a D-register; **dregx** stands for a register whose value is used only once, by the immediate parent of that node in the tree, while **dreg** stands for a register that may have more than one use. The distinction has implications for the permissibility of two-address instructions that overwrite the register as they calculate new values.

In the specification given here for the VAX, only one nonterminal for register is used, and the information about overwriteability is kept in the cost vector. In general, this scheme will not perform as well as the more general two-nonterminal scheme, because the decision about whether to compute the value with or without overwriteability is made before it is known whether overwriteability is needed. However, the VAX instruction set is rich enough in three-address instructions that the approximation produces adequate results.

A similar approximation is made (in both specifications) to designate instructions that do or do not set the condition code flags of the machine. (This is the *setFlags* field of the cost vector.) The more general scheme would be to have two nonterminals: one for calculations that set the condition codes, and one for all calculations (that may or may not set the condition codes). Actually, each nonterminal that can stand for an expression would have to be replicated.

To cope with this large but consistent set of nonterminal symbols, perhaps a way of specifying vectors of nonterminals would be appropriate. The Aho-Johnson register-allocation algorithm[6] operates on such vectors, but without the generality of context-free grammars.

This would produce specifications just as concise as the ones in this paper, but they would more accurately select optimal instruction sequences. On the other hand, the number of true nonterminals used in the dynamic programming algorithm would grow enormously, with its concomitant space and time costs.

Another possible use of parameterized nonterminals is for machines with a small set of dissimilar registers; for example, the Z80 or 80x86 family. To achieve optimality in register use, a nonterminal can be associated with each subset of registers used in a subtree.

7.5. Multi-phase actions

In the specifications presented here, the rules for operands (addressing modes) could not emit assembly code immediately, but only when the parent instruction was ready. On the other hand, it is necessary for the descendants of the addressing modes to emit assembly code, since they correspond to entire instructions. This problem was solved by having actions that just record the *kind* of match, and then traversing the tree later with procedures like *emitoperand.*

It might be nice to have more than one phase of action-code execution in Twig to handle situations like this. In particular, it should not be necessary to explicitly store the *kind* of the match, because Twig has already computed and stored the *kind* — it is just the number of the matching grammar rule.

7.6. Directed acyclic graphs

Twig is designed to operate on a tree representation. Often, a directed acyclic graph is a more natural intermediate representation. Unfortunately, Twig's dynamic programming algorithm doesn't work on graphs, so it is impractical to find the optimal match.

However, one reasonable solution is to factor the directed graph into trees. Each shared node (one with several in-edges) is made into an explicit temporary (perhaps using the ALLOC and TEMP nodes described in this paper). This tree is then matched using Twig; the match may be suboptimal, but at least a solution is generated. This technique has been used successfully[25] in a Twig-based compiler for standard-cell VLSI circuits.

8. Conclusion

Dynamic programming is a robust and efficient technique for generating machine code from expression trees. Code generator specifications written for the *Twig* system correspond closely and elegantly to the instruction sets and addressing modes of complex-instruction-set computers. The use of many nonterminal symbols to describe the different classes of instructions and operands eliminates redundancy from the specification without causing the blocking problems commonly found in LR(1) specifications.

Appendix 1. Generated code for a sample program

This appendix contains the benchmark program mentioned in section 6. It is an implementation of a backtracking solution to the 18-queens problem (as the 8-queens program runs too quickly to accurately benchmark). The program is expressed in a Pascal-like language that was chosen to produce a larger variety of intermediate trees than Pascal or C would.

```
var diag1 : array[2..36] of boolean;
    diag2 : array[-17..17] of boolean;
    row : array[1..18] of integer;
    col : array[1..18] of boolean;

procedure try(n : integer);
  var i : integer;
  begin if n>18
        then _exit(0)
        else for i := 1 to 18
         do if not (col[i]<>0
                    or else diag1[i+n]
                    or else diag2[i-n])
                then begin col[i]:=1;
                           diag1[i+n]:=1;
                           diag2[i-n]:=1;
                           row[n]:=i;
                           try(n+1);
                           col[i]:=0;
                           diag1[i+n]:=0;
                           diag2[i-n]:=0;
                     end
  end;
```

A1.1 Code generated for the MC68020

The code generator built from the MC68020 specification produced the translation below. The left-hand column is the exact code generator output; the right-hand column is the source program, as a form of annotation. The assembly-language uses the Motorola notation, rather than the Berkeley Unix notation, for the addressing modes.

```
_try:                              procedure try(n : integer);
link    a6,-#L6                      var i : integer;
moveml  #L5,-(sp)
cmpil   18,(a6,8)                    begin if n>18
jle     L3
movel   #0,-(sp)                           then _exit(0)
jsr     __exit
# throw away d0
jmp     L4
L3:
movel   #1,(a6,-4)                         else for i := 1 to 18
L2:
tstb    ([a6,-4],_col-1)                    do if not (col[i]<>0
jne     L1
movel   (a6,-4),d7                                     or else diag1[i+n]
moveal  a6,a5
addql   #8,a5
tstb    ([a5],d7,_diag1-2)
jne     L1
movel   (a6,-4),d7                                     or else diag2[i-n])
subl    (a6,8),d7
lea     _diag2,a5
tstb    (d7,a5,17)
jne     L1
```

```
moveb   #1,([a6,-4],_col-1)                    then begin col[i]:=1;
movel   (a6,-4),d7                                         diag1[i+n]:=1;
moveal  a6,a5
addql   #8,a5
moveb   #1,([a5],d7,_diag1-2)
movel   (a6,-4),d7                                         diag2[i-n]:=1;
subl    (a6,8),d7
lea     _diag2,a5
moveb   #1,(d7,a5,17)
movel   (a6,8),d7                                          row[n]:=i;
lea     _row,a5
movel   (a6,-4),(d7:l*4,a5,-4)
moveal  (a6,8),a5                                          try(n+1);
addql   #1,a5
movel   a5,-(sp)
jsr     _try
# throw away d0
clrb    ([a6,-4],_col-1)                                   col[i]:=0;
movel   (a6,-4),d7                                         diag1[i+n]:=0;
moveal  a6,a5
addql   #8,a5
clrb    ([a5],d7,_diag1-2)
movel   (a6,-4),d7                                         diag2[i-n]:=0;
subl    (a6,8),d7
lea     _diag2,a5
clrb    (d7,a5,17)
L1:                                                  end
movel   (a6,-4),d7
addql   #1,d7
movel   d7,(a6,-4)
cmpil   18,d0
jle     L2
L4:                                 end;
moveml  (a6,-16),#L5
unlk    a6
rts
L5=0x6080
```

A1.2 Code generated for the VAX

Here is the output of the Twig code generator built from the VAX specification described in this paper, when applied the the 18-queens program.

```
_try:                               procedure try(n : integer);
subl2   $4,sp                         var i : integer;
cmpl    4(r12),$18                  begin if n>18
jleq    L3
pushl   $0                                 then _exit(0)
calls   $1,__exit
# throw away r0
jmp     L4
L3:                                        else for i := 1 to 18
movl    $1,-4(r13)
L2:
movab   *-4(r13),r11                           do if not (col[i]<>0
```

```
tstb    _col+-1(r11)
jneq    L1
addl3   4(r12),-4(r13),r11                          or else diag1[i+n]
tstb    _diag1+-2(r11)
jneq    L1
subl3   4(r12),-4(r13),r11                          or else diag2[i-n])
tstb    _diag2+17(r11)
jneq    L1
movab   *-4(r13),r11                          then begin col[i]:=1;
movb    $1,_col+-1(r11)
addl3   4(r12),-4(r13),r11
movb    $1,_diag1+-2(r11)                                 diag1[i+n]:=1;
subl3   4(r12),-4(r13),r11
movb    $1,_diag2+17(r11)                                 diag2[i-n]:=1;
movab   *4(r12),r11
movab   *-4(r13),_row+-4[r11]                             row[n]:=i;
addl3   $1,4(r12),-(sp)                                   try(n+1);
calls   $1,_try
# throw away r0
movab   *-4(r13),r11                                      col[i]:=0;
clrb    _col+-1(r11)
addl3   4(r12),-4(r13),r11                                diag1[i+n]:=0;
clrb    _diag1+-2(r11)
subl3   4(r12),-4(r13),r11                                diag2[i-n]:=0;
clrb    _diag2+17(r11)                                end
L1:
addl3   $1,-4(r13),r11
movl    r11,-4(r13)
cmpl    r11,$18
jleq    L2
L4:                                 end;
ret
```

The code generator has discovered that the two instructions

```
movab   *-4(r13),r11
movl    -4(r13),r11
```

are of equal cost. Since the code generator doesn’t know that the latter instruction is the one conventionally used by humans, it blithely uses the first one that comes to hand. This non-problem could be ‘‘solved’’ by adding an ‘‘ugliness’’ field to the cost structure.

This example demonstrates that the dynamic-programming code generator makes good use of complex addressing modes. The program above could be significantly improved by holding the variables *i* and *n* in register rather than in stack-frame locations, but this is beyond the scope of the instruction-selection algorithm.

References

References

1. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, ‘‘Code generation using tree matching and dynamic programming,’’ *ACM Trans. Prog. Lang. and Systems*, vol. (to appear), 1989.
2. R. Sethi, ‘‘Complete register allocation problems,’’ *SIAM J. Computing*, vol. 4, no. 3, pp. 226-248, SIAM, 1975.
3. J. Bruno and R. Sethi, ‘‘Code generation for a one-register machine,’’ *J. ACM*, vol. 23, no. 3, pp. 502-510, ACM, 1976.
4. A. P. Ershov, ‘‘On programming of arithmetic operations,’’ *Comm. ACM*, vol. 1, no. 8, pp. 3-6, ACM, 1958.
5. R. Sethi and J. D. Ullman, ‘‘The generation of optimal code for arithmetic expressions,’’ *J. Assoc. Computing Machinery*, pp. 715-728, ACM, 1970.
6. A. V. Aho and S. C. Johnson, ‘‘Optimal code generation for expression trees,’’ *J. ACM*, vol. 23, no. 3, pp. 488-501, ACM, 1976.
7. D. E. Knuth, ‘‘A generalization of Dijkstra’s algorithm,’’ *Information Processing Letters*, vol. 6, pp. 1-5, 1977.
8. T. Kasami, ‘‘An efficient recognition and syntax algorithm for context-free languages,’’ Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, Mass., 1965.
9. D. H. Younger, ‘‘Recognition and parsing of context-free languages in time n^3,’’ *Information and Control*, vol. 10, no. 2, pp. 189-208, 1967.
10. C. Gordon Bell and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.
11. R. G. G. Cattell, ‘‘Formalization and automatic derivation of code generators,’’ Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, April 1978.
12. Christopher W. Fraser, ‘‘A compact, machine-independent peephole optimizer,’’ *Sixth ACM Symp. on Principles of Programming Languages*, pp. 1-6, ACM, 1979.
13. J. W. Davidson, ‘‘Simplifying Code Generation Through Peephole Optimization,’’ TR 81-19, Department of Computer Science, University of Arizona, Tucson, Arizona, 1981.
14. R. Steven Glanville and Susan L. Graham, ‘‘A New Method for Compiler Code Generation,’’ *Fifth ACM Symposium on Principles of Programming Languages*, pp. 231-240, ACM, 1978.
15. Susan L. Graham, Robert R. Henry, and Robert A. Schulman, ‘‘An Experiment in Table Driven Code Generation,’’ *Tenth ACM Symposium on Principles of Programming Languages*, pp. 32-43, ACM, 1983.
16. Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall K. McKusick, and Eduardo Pelegri-Llopart, ‘‘Experience with a Graham-Glanville style code generator,’’ *Proc. Sigplan ’84 Symp. on Compiler Construction (Sigplan Notices)*, vol. 19, no. 6, pp. 13-24, ACM, 1984.
17. M. Ganapathi, ‘‘Retargetable Code Generation and Optimization using Attribute Grammars,’’ PhD. Thesis, Univ. of Wisconsin, Madison, Wis., 1980.
18. Steven W. K. Tjiang, ‘‘Twig Reference Manual,’’ CSTR-120, ATT Bell Laboratories, Murray Hill, NJ, 1986.
19. C. M. Hoffmann and M. J. O’Donnell, ‘‘Pattern matching in trees,’’ *Journal of the ACM*, vol. 29, pp. 68-95, 1982.
20. *VAX Architecture Handbook*, Digital Equipment Corp., Maynard, Mass., 1979.
21. Andrew W. Appel and Kenneth J. Supowit, ‘‘Generalizations of the Sethi-Ullman algorithm for register allocation,’’ *Software — Practice/Experience*, vol. 17, no. 6, pp. 417-421, 1987.
22. *MC68020 32-Bit Microprocessor User’s Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
23. J. W. Davidson and Christopher W. Fraser, ‘‘Automatic Generation of Peephole Optimizations,’’ *Sigplan ’84 Symposium on Compiler Construction*, pp. 111-116, ACM, 1984.
24. Peter B. Kessler, ‘‘Discovering machine-specific code improvements,’’ *Proc. Sigplan ’86 Symp. on Compiler Construction (Sigplan Notices)*, vol. 21, no. 7, pp. 249-254, ACM, 1986.
25. Kurt Keutzer and Wayne Wolf, ‘‘Anatomy of a hardware compiler,’’ *Proc. ACM SIGPLAN ’88 Conf. on Prog. Lang. Design & Implementation*,, pp. 95-104, ACM, June 1988.