Open access • Journal Article • DOI:10.1145/69558.75700

Code generation using tree matching and dynamic programming — Source link

Susan L. Graham

Institutions: Bell Labs

Published on: 01 Oct 1989 - ACM Transactions on Programming Languages and Systems (ACM)

Topics: Code generation, Compiler, Construct (python library), Instruction selection and Parsing

Code Generation Using Tree Matching and Dynamic Programming

ALFRED V. AHO
AT&T Bell Laboratories
MAHADEVAN GANAPATHI
Stanford University
and
STEVEN W. K. TJIANG
AT&T Bell Laboratories

Compiler-component generators, such as lexical analyzer generators and parser generators, have long been used to facilitate the construction of compilers. A tree-manipulation language called *twig* has been developed to help construct efficient code generators. *Twig* transforms a tree-translation scheme into a code generator that combines a fast top-down tree-pattern matching algorithm with dynamic programming. *Twig* has been used to specify and construct code generators for several experimental compilers targeted for different machines.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, optimization compiler generators*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*parallel rewriting systems*

General Terms: Algorithms

Additional Key Words and Phrases: Code generation, code generator-generator, code optimization, dynamic programming, pattern matching

1. INTRODUCTION

Research in code generation has yielded theoretical insights and practical techniques [7, 21, 37]. On the theoretical front, efficient algorithms for generating provably optimal code on broad classes of uniform-register machines have been developed for expressions with no common subexpressions [3, 40]. However, once common subexpressions are encountered or optimal code needs to be generated for machines with irregular architectures, the problem of optimal code generation

Authors' current addresses: A. V. Aho, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, N.J. 07974; M. Ganapathi and S. W. K. Tjiang, Stanford University, Department of Computer Science, Stanford, CA 94305.

has been proven to be combinatorially difficult [4, 10], and heuristic techniques for generating good code have been proposed and theoretically analyzed [4, 5].

On the experimental front, several innovative approaches to retargetable code generation have been pursued. These approaches have focused on the use of table-driven techniques to separate the machine description from the code-generation algorithm. Compilers based on some of these techniques have been easily retargeted [11, 13, 17, 25, 32, 46].

This paper presents a new language called *twig* that encapsulates some of these theoretical and experimental advances into a tree-based notation for describing and implementing code generators. The language builds on the experience of grammar-based descriptions of code generators. A compiler for *twig* has been constructed that combines an efficient tree-pattern matching algorithm along with a dynamic programming algorithm for optimal code selection. *Twig* has been used by the authors to construct several code generators, including one for the VAX that has been incorporated into the pcc2 compiler [32] and one for the MIPS-X project [12]. *Twig* has also been used by A. W. Appel to construct code generators for the VAX and the Motorola 68020 [9]. In addition to producing traditional code generators for compilers, *twig* can be used as a tool for creating tree-rewriting and tree-manipulation programs. In this vein, K. Keutzer and W. Wolf have used *twig* to construct a standard-cell synthesizer for VLSI circuits [33, 34].

2. CODE GENERATION BY TREE REWRITING

Simply speaking, a compiler consists of a front end that analyzes the source program and transforms it into an intermediate representation (IR), and a back end that transforms the IR into the target program [7]. Many factors are involved in choosing an appropriate IR, but in most cases the IR is some encoding of a graphical representation of the source program. In this paper, it is sufficient to assume the IR is a sequence of trees at the semantic level of the target machine as in [18, 23, 29].

Figure 1 shows an IR tree for an assignment statement a[i] := b in which a and i are locals, stored on the stack, whose run-time addresses are given as offsets, $\texttt{const}_a$ and $\texttt{const}_i$, from a stack pointer stored in register SP. The leaves in the tree are type attributes with subscripts; the subscript indicates the value of the attribute.

The assignment to a[i] is an indirect assignment in which the contents of the location for a[i] are set to the *r*-value of the global b. The address of the first element of the array a is found by adding the value $\texttt{const}_a$ to the contents of register SP; the value of i is in the location obtained by adding the value $\texttt{const}_i$ to the contents of register SP.

In the tree, the ind operator makes its argument a memory address. As the left child of an assignment operator, the ind node gives the location into which the *r*-value on the right side of the assignment operator is to be stored. If an argument of a + or ind operator is a memory location or a register, then the contents of that memory location or register are taken as the value.

For code generation, the target-machine instructions can be represented by tree-rewriting rules, consisting of a replacement node, a tree template, a cost,

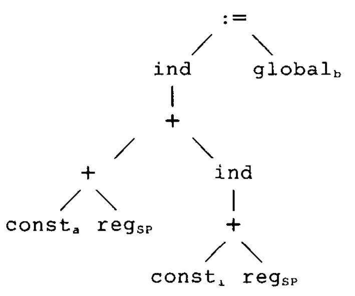


Fig. 1. Intermediate-code tree for a[i] := b.

and an action. The target code is generated by a process in which each IR tree is reduced into a single node by repeatedly finding subtrees in the IR tree that match templates and rewriting the matched subtrees by the corresponding replacement nodes. The sequence of subtrees rewritten in this process is called a *cover* of the IR tree. The target code is emitted by the actions associated with the rules used in the cover, and the total cost is the sum of the costs of the covering rules.

To be more precise, a tree-rewriting rule is a statement of the form

replacement ← *template* {*cost*} = {*action*}

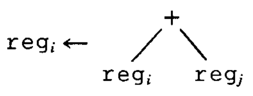
where

(1) *replacement* is a single node,
(2) *template* is a tree,
(3) *cost* is a code fragment that computes the cost associated with this template, and
(4) *action* is a code fragment.

A set of tree-rewriting rules is called a *tree-translation scheme.*

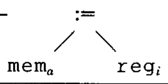
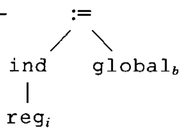
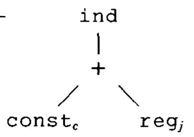
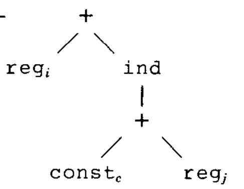
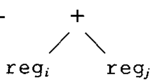
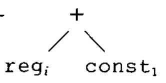
A tree-translation scheme is a convenient way to represent the instruction-selection phase of code generation. Each tree template represents a computation performed by one or more target machine instructions. The leaves of a template are attributes with subscripts, as in the IR tree. Often, certain restrictions apply to the values of the subscripts in the templates. For example, a constant may be required to fall in a certain range. These restrictions can be specified as semantic predicates in the cost function or the action, and these predicates must be satisfied before a template can match a subtree of the IR tree. Register allocation is done by the user-specified actions.

As an example of a tree-rewriting rule, consider the rule for a register-to-register add instruction, ADD Rj, Ri:



If the IR tree contains a subtree that matches this tree template, that is, a subtree whose root is labeled by the operator + and whose left and right children are quantities in registers i and j, then we might replace that subtree by a single node

Table I. Tree-Rewriting Rules for Some Target-Machine Instructions

	Rewrite rule	Cost	Instruction
(1)	$\mathtt{reg}_i \leftarrow \mathtt{const}_c$	2	MOV #*c*, R*i*
(2)	$\mathtt{reg}_i \leftarrow \mathtt{mem}_a$	2	MOV *a*, R*i*
(3)	$\lambda \leftarrow$ 	$2 + \mathtt{cost.reg}_i$	MOV R*i*, *a*
(4)	$\lambda \leftarrow$ 	$2 + \mathtt{cost.reg}_i$	MOV *b*, * R*i*
(5)	$\mathtt{reg}_i \leftarrow$ 	$2 + \mathtt{cost.reg}_j$	MOV *c*(R*j*), R*i*
(6)	$\mathtt{reg}_i \leftarrow$ 	$2 + \mathtt{cost.reg}_i + \mathtt{cost.reg}_j$	ADD *c*(R*j*), R*i*
(7)	$\mathtt{reg}_i \leftarrow$ 	$1 + \mathtt{cost.reg}_i + \mathtt{cost.reg}_j$	ADD R*j*, R*i*
(8)	$\mathtt{reg}_i \leftarrow$ 	$1 + \mathtt{cost.reg}_i$	INC R*i*

labeled $\mathtt{reg}_i$ simulating the execution of the instruction ADD R*j*, R*i*. If more than one template can match a subtree or a portion thereof, then dynamic programming is used to determine a minimum-cost cover.

Table I contains tree-rewriting rules for a few instructions for a VAX-like target machine. Instead of showing the code for the actions, we have shown the machine instruction that is generated by each rule. The first two rules correspond to load instructions, the next two to store instructions, and the remainder to indexed loads and additions. Note that rule (8) requires the value of the constant to be 1. This condition can be enforced by a semantic predicate in the cost.

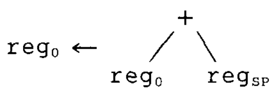
A tree-translation scheme generates code from an IR tree in the following way. All templates in the tree-rewriting rules are matched against the subtrees of the IR tree during a depth-first traversal of the tree. At each node, the costs are used to determine the best match, and the selected subtree is replaced in the IR tree by the associated replacement node. Sometimes the replacement is delayed until the cost of another larger including match can be evaluated. By this process a minimum-cost cover for the IR tree is found.

Then a second depth-first traversal of the original IR tree is made and the actions associated with the rules used in the cover are executed. If an action emits a sequence of target-machine instructions, the instructions become part of the output. The sequence of machine instructions thus generated constitutes the output of the tree-translation scheme.

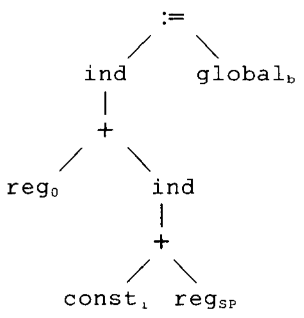
To illustrate, let us use the tree-translation scheme in Table I to process the IR tree in Figure 1. The template of the first rule

$$\texttt{reg}_0 \leftarrow \texttt{const}_a$$

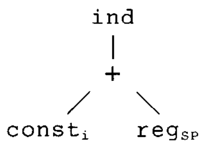
matches the leftmost leaf of the IR tree with $i = 0$ and $c = \texttt{a}$. If we use this rule, the label of the left-most leaf is changed from $\texttt{const}_a$ to $\texttt{reg}_0$, and during the second traversal the instruction `MOV #a, R0` will be generated to load the constant a into register `R0`. The template of the seventh rule with $i = 0$ and $j = \texttt{SP}$



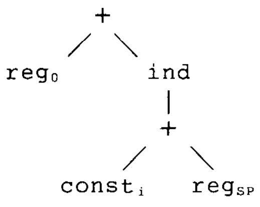
now matches the leftmost subtree with root labeled +. Using this rule, we would rewrite this subtree into a single node labeled $\texttt{reg}_0$ and later generate the instruction `ADD SP, R0`. Now the tree looks like



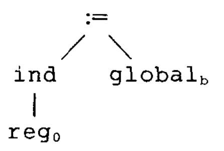
At this point, we could apply rule (5) to reduce the subtree



to a single node labeled $\texttt{reg}_1$. However, we can also use rule (6) to reduce the larger subtree



into a single node labeled $\texttt{reg}_0$ and later generate the instruction `ADD i(SP), R0`. Assuming it is more efficient to use a single instruction to compute the larger subtree rather than the smaller one, we choose the latter reduction to get



This remaining tree is matched by rule (4), which reduces the tree to a single node and later generates the instruction `MOV b, * R0`.

This sequence of tree reductions defines a covering that generates the following code sequence during the second traversal:

```
MOV #a, R0
ADD SP, R0
ADD i (SP), R0
MOV b, * R0
```

With a tree-translation scheme, specifying a code generator is similar to writing a syntax-directed specification for a translator. The tree-rewriting rules that describe the instruction set of a target machine are analogous to the productions of a context-free grammar, and the output code is generated as part of a tree-pruning process that is reminiscent of parsing. However, there are also several major differences. First, tree-pattern matching is used instead of parsing and there is no left-to-right bias in the matching algorithm as there is with some parsing algorithms. Second, a dynamic programming algorithm that runs concurrently with the tree-matching process selects an optimal covering for the IR tree using the costs associated with the tree-rewriting rules. Finally, the actions are executed after an optimal covering of the IR tree has been found.

3. PATTERN-DIRECTED CODE GENERATION

Wasilew [43] and Weingart [44] were among the first to treat code generation as a tree-rewriting process. These early approaches employed direct tree-pattern matching techniques. Fraser [15] and Cattell [11] emphasized the use of heuristic search; Fraser relied on knowledge-based rules that direct the pattern matching, whereas Cattell advocated a goal-directed heuristic search. In Cattell's approach, subgoals are created as the search continues and heuristics are used, both to order subgoal selection and to order patterns when trying to match.

Graham and Glanville pioneered the use of LR parsing techniques for code generation [22, 23]. A code generator can be constructed as syntax-directed translator in which a linearized prefix form of the IR trees is parsed by an LR parse built from a context-free grammar that describes the target machine. In this approach, the instructions of the target machine are described by a set of grammar rules. A parse of the prefix form of an IR tree corresponds to a covering of the tree with instruction templates. The target-machine instructions are generated during the reductions of the parsing process.

With the LR-parsing approach there are several practical difficulties that need to be overcome. First, an LR grammar describing a target machine such as a VAX can have over 1000 productions [26]. Machine-description grammars can produce large parser tables that may require specialized table-compression techniques [14]. Second, an LR parser does the pattern matching in a left-operand biased fashion. That is, the code for the left operand of an operator must be selected without considering the right operand. In a number of cases, the resulting code is suboptimal.

For example, consider the expression *op A B* that might appear in the prefix representation of an IR tree. Usually, machine architectures allow only certain combinations of addressing modes for the subexpressions *A* and *B*. Examples of these addressing-mode restrictions on op-code use are common in microprocessor architectures such as the iAPX-86, Z-8000, and MC-68000. If the addressing mode for *A* is selected without considering *B*, then the code generator may have to undo this selection when the time comes to select the machine instruction for *op*. Consequently, extra machine code would be needed to move *A* to an acceptable addressing mode.

Another problem in a purely syntactic approach is the difficulty of specifying target-machine architectural constraints such as register restrictions on addressing modes, of tracking expressions with results in multiple locations, and of modeling condition codes. A purely syntactic treatment requires this semantic information to be encoded syntactically as much as possible. Several tools and techniques have been developed to help cope with some of these difficulties [8, 26–29, 38].

Ganapathi and Fischer [17–20] extended the grammatical approach to code generation by using an attribute grammar to describe the instruction set of the target machine. Grammar productions specify the general form of the machine instructions, and semantic attributes and predicates specify architectural restrictions. Attributes are also used to track multiple instruction results and instruction selection is done by attributed parsing. Addressing modes are described by separate individual productions and so are operation codes. Addressing-mode selection is still left biased in the true tree-pattern matching sense, but the selection of operation codes is not biased toward any operand. Productions corresponding to operation codes usually have symmetric operand patterns. This symmetry enables the code generator to delay decisions regarding destination requirements. In effect, this decision is made on seeing the entire subtree for the operator. Thus, efficient code is produced in cases when either of the operands can be used to store the result of evaluation. Only in cases where their original results need be preserved is a call made to a register/temporary allocator.

Ganapathi and Fischer emphasize the incremental development of a code generator. Initially, productions describing the most general form of a target-language construct are listed. Later, special-case productions can be added to improve the performance of the target code [20]. To the scheme in Table I, for example, we could add rules to generate three-address instructions if desired. These special-case productions make the underlying grammar ambiguous. With ambiguous grammars, subsequent modifications can be made to a code generator with reduced effort. It is for this reason that ambiguous grammars are particularly useful in the design and specification of code generators. Deterministic parsers can be mechanically constructed from ambiguous specifications provided rules are provided to disambiguate the resulting parsing-action conflicts [6].

Attributes and semantic predicates can also be used to reduce the size of the specification of the addressing modes of the target machine. In an attribute grammar the number of grammar productions is usually smaller (a hundred or two productions instead of a thousand) than in a purely grammatical approach. Extensive grammar factoring is therefore not needed to implement a code generator based on semantic attributes and predicates, but care must now be given to the design of the attributes and predicates.

4. CODE GENERATION BY TREE MATCHING AND DYNAMIC PROGRAMMING

In this paper, we introduce a new language called *twig* for constructing code generators. A target machine is specified as a tree-translation scheme. *Twig* converts this specification into a code generator that combines a fast tree-pattern matching algorithm with an efficient dynamic programming algorithm for generating high-quality output code.

The underlying tree-matching algorithm is a generalization of Aho and Corasick's linear-time keyword-matching algorithm as suggested by Hoffman and O'Donnell [30]. The dynamic programming algorithm is a simplification of Aho and Johnson's optimal code-generation algorithm [3] that has been used in several compilers [32, 39]. This style of code generation can be readily integrated with the tree-matching process.

This approach seems to have several advantages. A *twig* machine specification is concise. With a tree-translation scheme, similar machine instructions can be factored into a common pattern, so that one syntactic match can correspond to several instructions. Rules that have the same templates but differing costs and actions can be factored into a single rule with multiple cost-action pairs. Similar factoring can be performed on rules in which only the operators differ. For example, a generic binary operator can often be defined to derive both the addition and subtraction instructions of a target machine. Since fewer patterns are needed, the description of the code generator is significantly simplified.

The dynamic programming algorithm allows the rules to be written in any order and obviates the need to deal with pattern-matching conflicts. In a parser-based approach the order of the productions is important, and parsing-action conflicts have to be carefully resolved. The dynamic programming algorithm produces code that is optimal with respect to the costs provided and eliminates the need for explicitly breaking cycles to prevent the code generator from looping, as may be necessary in a parser-based approach.

Finally, *twig* produces code generators quickly, and their size is small. For example, our *twig* specification of a VAX has 115 rules. It takes *twig* 5 seconds on a VAX 11/780 to produce the code generator from this specification, and the total size of the resulting code generator is under 50K bytes.

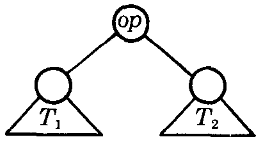
5. THE AHO-JOHNSON DYNAMIC PROGRAMMING ALGORITHM

Aho and Johnson [3] presented an algorithm based on the principle of dynamic programming to generate code for expressions on register machines. Their algorithm generates optimal code for a uniform-register machine that has r

interchangeable registers R0, R1, ..., Rr − 1 and instructions of the form Ri := E, where E is any expression containing operators, registers, and memory locations. The cost of a program is the sum of the costs of the instructions in the program.

The dynamic programming algorithm partitions the problem of generating optimal code for an expression E into subproblems of generating optimal code for the subexpressions of E. An optimal program for an expression of the form $E_1 + E_2$ is formed by combining optimal programs for the subexpressions E_1 and E_2, in one or the other order, followed by code to evaluate the operator +. The subproblems of generating optimal code for E_1 and E_2 are solved recursively. A program produced by the dynamic programming algorithm has an important property: It evaluates an expression "contiguously."

Consider the syntax tree T for the expression $E = E_1$ *op* E_2



where T_1 and T_2 are trees for E_1 and E_2, respectively. We say a machine-language program P evaluates T *contiguously* if it first evaluates those subtrees of T that need to be computed into memory and then evaluates the remainder of T, either in the order T_1, T_2, and then the root, or in the order T_2, T_1, and then the root, in either case using the previously computed values from memory whenever necessary. As an example of noncontiguous evaluation, P might first evaluate part of T_1, leaving the value in a register (instead of memory), next evaluate T_2, and then return to evaluate the rest of T_1.

For the uniform-register machine, Aho and Johnson proved that given any machine-language program P to evaluate an expression tree T, there is a program P' that computes the same expression such that

(1) P' is of no higher cost than P,
(2) P' uses no more registers than P, and
(3) P' evaluates the tree in a contiguous fashion.

This result implies that every expression tree can be evaluated optimally by a contiguous program on a uniform-register machine [3].

Some real machines have architectural features that do not always allow optimal contiguous evaluations. For example, for machines with even–odd register pairs such as the IBM System/370 machines there are examples of expression trees in which an optimal machine-language program must first evaluate into a register a portion of the left subtree of the root, then a portion of the right subtree, then another part of the left subtree, then another part of the right, and so on. This type of unbounded oscillation is unnecessary for an optimal evaluation of any expression tree using the uniform-register machine.

The contiguous evaluation property defined above says that for any expression tree T there always exists an optimal program that consists of optimal programs for subtrees of the root, followed by an instruction to evaluate the root. This property allows us to use dynamic programming to generate an optimal program for T.

The dynamic programming algorithm as presented in [3] proceeds in three phases. In the first phase, it computes bottom-up for each node n of the expression tree T an array C of costs, in which the ith component $C[i]$ is the optimal cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for $1 \leq i \leq r$. The cost includes whatever loads and stores are necessary to evaluate S in the given number of registers. It also includes the cost of computing the operator at the root of S. The zeroth component of the cost vector is the optimal cost of computing the subtree S into memory. The contiguous evaluation property ensures that an optimal program for S can be generated by considering combinations of optimal programs only for the subtrees of the root of S. This restriction sharply reduces the number of cases that need to be considered.

To compute $C[i]$ at node n, the algorithm considers each machine instruction R $:= E$ whose expression E matches the subexpression rooted at node n. By examining the cost vectors at the corresponding descendants of n, it determines the costs of evaluating the operands of E. For those operands of E that are registers, it considers all possible orders in which the corresponding subtrees of T can be evaluated into registers. In each ordering the first subtree corresponding to a register operand can be evaluated using i available registers, the second using $i - 1$ registers, and so on. To account for node n, it adds in the cost of the instruction R $:= E$ that was used to match node n. The value $C[i]$ is then the minimum cost over all possible orders.

The cost vectors for the entire tree T can be computed bottom-up in time linearly proportional to the number of nodes in T. The smallest cost in the vector for the root of T gives the minimum cost of evaluating T.

In the second phase, the algorithm traverses T, using the cost vectors to determine which subtrees of T must be computed into memory. In the third phase, the algorithm traverses each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first. These two phases can also be implemented to run in time linearly proportional to the size of the expression tree. See Aho and Johnson [3] and Aho et al. [7] for more details.

Twig uses a simplified form of this algorithm. In the compilers so far implemented with *twig*, the IR trees have been sufficiently simple that it was possible to separate register management from instruction selection. Consequently, *twig* uses an algorithm in which each subtree of the IR tree is characterized by a single scalar cost, rather than a cost vector. These scalar costs are computed using the cost expressions in the tree-translation scheme. Register assignment is done separately by a user-provided routine. These modifications have increased the speed and flexibility of *twig* without noticeably degrading the quality of the code generated for the VAX. More research is needed to determine how generally these observations apply to other machines.

6. TREE-PATTERN MATCHING

Several tree-pattern matching algorithms have been presented [30, 31, 35, 36, 41]. For code generation applications, a scheme proposed by Hoffman and

O'Donnell [30] appears promising. They suggested that template matching can be done efficiently by extending the Aho–Corasick multiple-keyword pattern-matching algorithm [1] into a top-down tree-pattern matching algorithm.

First consider the problem of finding all substrings of an input string that are contained in a given set of keywords. The essence of the Aho–Corasick algorithm is to construct a trie from the set of keywords, convert the trie into a pattern-matching automation, and then use the pattern-matching automaton to perform a parallel search for the keywords in the input string.

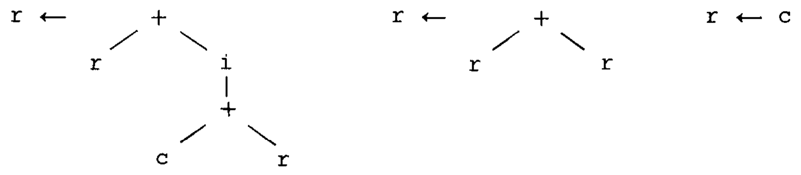
Let K be the set of keywords. The trie is built by first making a root node and then, for each keyword in K, creating a path from the root to a node whose branch labels spell out the keyword. Each node of the trie is thus uniquely characterized by the sequence of symbols on the branch labels of the path from the root to that node.

The pattern-matching automaton is constructed from the trie. The states of the automaton are the nodes of the trie; the start state is the root and the accepting states are those corresponding to complete keywords. There is a transition from state s to state t on input character c if there is a branch in the trie labeled c from node s to node t. In addition, we add a transition from the start state to itself on every input character that is not the first character of a keyword.

The pattern-matching automaton has a failure function for every state other than the start state. The failure function for a state characterized by a string u is a pointer to the state characterized by the longest prefix of some keyword in K that is also a proper suffix of u.

Both the trie and the pattern-matching automaton can be constructed in time linearly proportional to the sum of the lengths of the keywords in K. The resulting pattern-matching automaton can be run on an input string x in time linearly proportional to the length of x, independent of the size of K. Thus the entire problem of finding all substrings of x that are contained in K can be done in time $O(|K| + |x|)$ [1].

This algorithm can be directly generalized into a tree-matching algorithm by noting that a tree is characterized by the set of paths from its root to its leaves when the branches from each node are numbered 1, 2, . . . , according to the left-to-right ordering of the children [30]. For example, consider the following three tree-replacement rules



which we will refer to as t_1, t_2, and t_3, respectively. They have the following set of path strings:

```
+1r
+2i1+1c
+2i1+2r
+2r
c
```

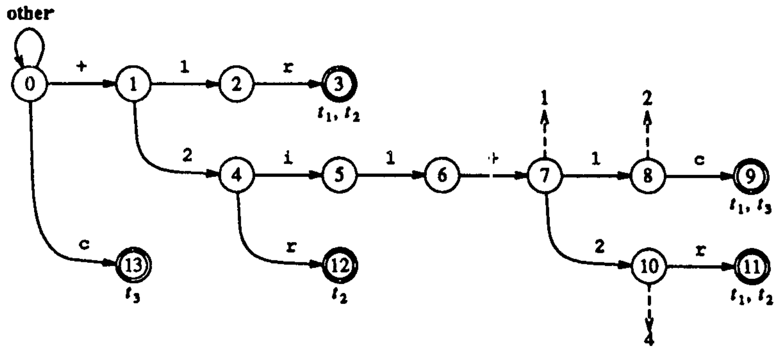


Fig. 2. Path-string-matching automaton.

The first string +1 r represents the leftmost path in the tree templates of both t_1 and t_2. Note that a path of length j in t_i is represented by a path string of length $2j + 1$. From the set of path strings, using the methods in Aho and Corasick [1], we can construct a pattern-matching automaton to match the path strings in parallel.

From the trees above, we would construct the pattern-matching automaton shown in Figure 2. State 0 is the start state and the doubly circled states are accepting. In this automaton the failure function for state 7 points to state 1, for state 8 to state 2, for state 10 to state 4, and for all the other states to state 0; the failure functions that do not point to state 0 are shown as dashed lines. At each accepting state, we also know which path string of which tree templates has been recognized. For example, at state 3, the recognized string +1 r matches the leftmost path in the tree templates of t_1 and t_2.

Let T be a set of tree-replacement rules of the form $l_i \leftarrow t_i$, where l_i is a label and t_i a tree template. We can build an automaton similar to the one above to recognize the tree templates in T in parallel in a subject tree. Let $succ(\sigma, a)$ denote the state reached from state σ on input symbol a by the automaton. The automaton creates a record of information at each node n of the subject tree: *n.parent* is the parent of n, *n.symbol* is a path-string symbol associated with n, and *n.state* is the state of the automaton after it visits n. We assume that nodes in the subject tree and the tree templates labeled by the same symbol have the same arity. The following routine, *visit*(n), will traverse the subject tree in depth-first order starting at node n, assign a state to each node, and call *post_process* to determine the matching tree templates.

```
visit(n)
{
  if n is the root then
    n.state ← succ(0, n.symbol)
      where 0 is the start state of the automaton
  else
    n.state ← succ(succ(n.parent.state, k), n.symbol)
      where n is the kth child of n.parent
  for every child c of n do
    visit(c)
  post_process(n)
}
```

A path-string match occurs at node n in the subject tree if *n.state* is accepting. It is easy to find the node in the subject tree at which this match began by tracing the path from n toward the root in the subject tree. A node of the subject tree matches a tree template if that node begins matches for all the path strings in the tree template.

In [30], Hoffman and O'Donnell propose two techniques for determining nodes beginning matches. The first technique associates a set of counters for each tree template t_i at each node in the subject tree. When a path string of t_i is recognized, its counter is incremented at the node beginning the match. The template t_i matches a subtree rooted at node n as soon as the value of its counter at node n equals the number of path strings in t_i.

Twig uses the second technique of maintaining bit strings rather than counters to keep track of partial matches. With each node n in the subject tree *twig* associates a bit string $n.b_i$ for each tree template t_i. The number of bits in b_i is equal to one plus the length of the longest path in t_i; the bits of b_i are indexed consecutively from the right starting with 0. When a path string in t_i of length $2j + 1$ is recognized at node n in the subject tree, bit j of $n.b_i$ is set. Intuitively, if bit j of $n.b_i$ is set, then node n in the subject tree matches a node at depth j in tree template t_i. (The root of a tree is at depth 0.) Bit strings allow overlapping matches to the same tree template to be recorded.

Tree recognition occurs in the call of *post_process* after all the children of a node have been visited (see the function *visit* above). At each node n, the new bit string $n.b_i$ is computed by shifting the bit strings for t_i at the children of n right 1 bit (this is equivalent to dividing by 2) and bitwise or'ing their logical product with the current bit string $n.b_i$. The intuition is that node n of the subject tree matches a node of t_i at depth j if the label of n matches the node of t_i and all of n's children match the nodes of t_i at depth $j + 1$. Under the assumption of matching arities of similarly labeled nodes, this method provides a necessary and sufficient condition for tree matching. The following routine gives the details of the tree-matching process.

```
post_process(n)
{
  $n.b_i \leftarrow 0$
  if n.state is accepting then
    set_partial(n, n.state)
  for every $t_i$ do
    $n.b_i \leftarrow n.b_i$ or $\prod_{c \in C(n)} c.b_i/2$
      where $C(n)$ is the set of all children of node n
  do_reduce(n)
}

set_partial(n, $\sigma$)
{
  for each path string of $t_i$ of length $2j + 1$ recognized at $\sigma$ do
    $n.b_i \leftarrow n.b_i$ or $2^j$
}
```

The routine *do_reduce* keeps track of reductions, which are discussed in the following paragraphs. The routine *set_partial* sets the jth bit of b_i for each recognized path string of t_i of length $2j + 1$; *set_partial* requires the length of the recognized path strings to be available at the accepting states. After *post_process*,

t_i matches the subtree rooted at node r if and only if $r.b_i$ is odd, that is, its rightmost bit is set to 1.

To find a cover, it is necessary to consider reductions. That is, once the tree part of t_i is recognized, the possible reduction of the tree part to the label of t_i must be considered in order to find covers containing this match of t_i. Since *twig* is considering many matches in parallel, the process of reduction should not change the shape of the subject tree; only some node fields are updated to reflect the reduction. The routine *do_reduce* performs the updates and implements dynamic programming.

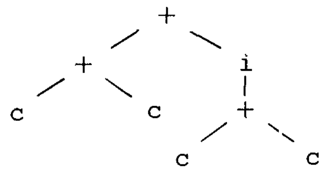
The function $cost(t_i, n)$ determines the cost of a rule t_i matching at node n. In general, the cost of a match will also depend on the costs of matches at leaves of t_i that are label symbols. The dynamic programming costs are kept in an array *n.cost*. Each element $n.cost[l]$ is the cost of the cheapest match of some rule $l \leftarrow tree$. The index of the rule that achieves $n.cost[l]$ is stored in $n.match[l]$; that is, if $n.match[l] = j$, then $cost(t_j, n) = n.cost[l]$ and $l = l_j$. Initially, before the first call of *visit*, $n.cost[l] = \infty$ and $n.match[l] = 0$ for all nodes n.

```
do_reduce(n)
{
  for every $t_i$ such that the zeroth bit of $n.b_i$ is 1 do
      if $cost(t_i, n) < n.cost[l_i]$ then
      {
        $n.cost[l_i] \leftarrow cost(t_i, n)$
        $n.match[l_i] \leftarrow i$
        if $n$ is the root then
          $\sigma \leftarrow succ(0, l_i)$
        else
          $\sigma \leftarrow succ(succ(n.parent.state, k), l_i)$
             where $n$ is the $k$th child of $n.parent$
        if $\sigma$ is an accept state then
          $set\_partial(n, \sigma)$
      }
}
```

For example, consider the rule set $T = \{t_1, t_2, t_3\}$ as given at the beginning of this section. At each node n, let the cost function be

$cost(t_1, n) = 3 +$ cost of matches at leaves labeled r
$cost(t_2, n) = 1 +$ cost of matches at leaves labeled r
$cost(t_3, n) = 1$.

Consider the subject tree S:



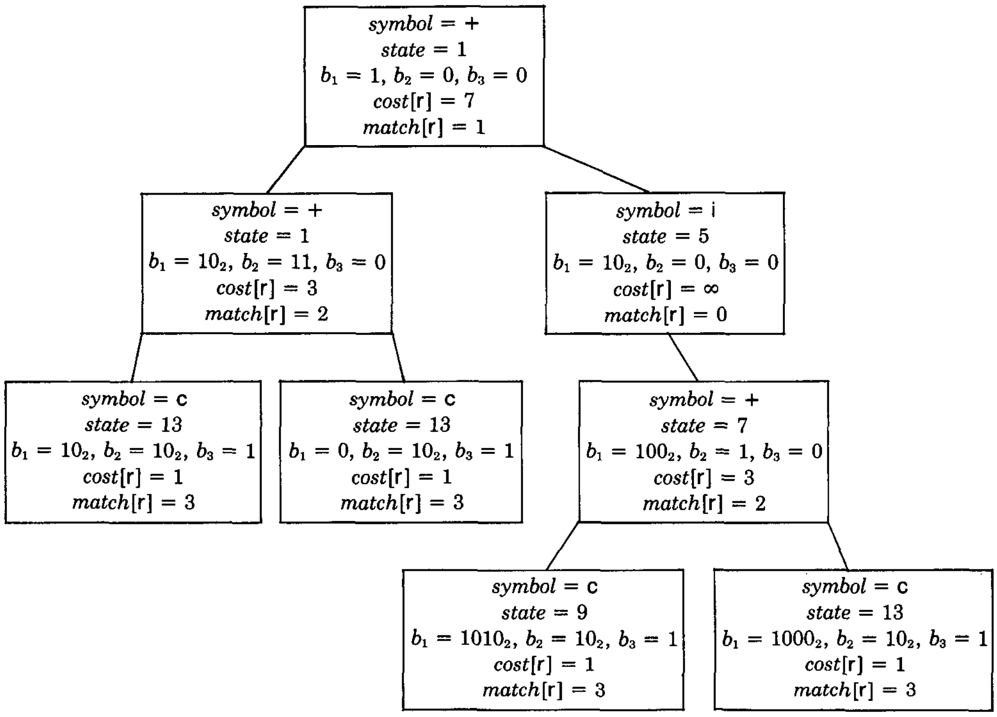
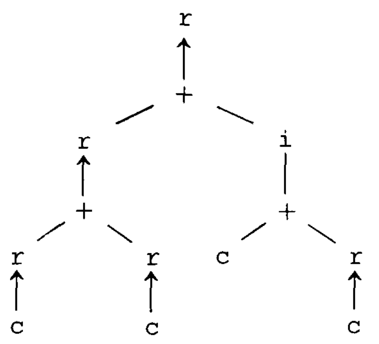


Fig. 3. An algorithm that finds matches not included in any cover.

Applying *visit* to the root of S yields the values at each node shown in Figure 3. The subscript 2 on the values of the b_i denotes "binary string."

In Figure 3, we see that the only cover starts with rule t_1 at the root and has cost seven.

The algorithm finds matches that are not included in any cover. For example, at the node with symbol + on the right branch of the root, the tree pattern t_2 matches with cost three. The cover is shown below in a form reminiscent of a parse tree with arrows indicating reductions.



7. TWIG—A LANGUAGE FOR MANIPULATING TREES

Twig is a language for processing trees that incorporates the algorithms described above. Although *twig* can be used in other tree-manipulation contexts, this section only presents its use in code generation. For other applications of *twig*, see Keutzer [33] and Keutzer and Wolf [34].

A *twig* program, hereafter called a *twig* specification, is a set of pattern-action rules together with ancillary declarative statements. To construct the *twig* portion of a code generator, we compile the *twig* specification with the *twig* compiler to create a C source file. This file will contain a subroutine that performs the matching operation. Once it has been compiled, the subroutine can be linked with the *twig* runtime library and the other parts of the code generator. To invoke the pattern matcher, we call the generated subroutine on the root of the tree to be matched. In the following sections, this tree is referred to as the subject tree.

Pattern-Action Rules

The syntax of a rule in *twig* is

label_id:*pattern*[{*cost*}] [= {*action*}]

(1) The *label_id* is an identifier that is analogous to the left-hand nonterminal symbol of a production in a context-free grammar.

(2) The *pattern* is a parenthesized prefix expression representing a tree. The pattern matcher will find all subtrees of the subject tree matching this tree pattern. When a subtree matches the pattern and the cost part does not abort, we say that the rule *matches*. Abortion is explained later.

(3) The *cost* is C source code executed by the pattern matcher when it finds a subtree matching the tree pattern. This code should return a cost to the matcher for purposes of dynamic programming. The code also determines when the action part should be called by the pattern matcher if this rule is in the minimal cost cover of the subject tree.

The cost part is optional. When omitted, the *twig* compiler and pattern matcher assume the default cost is returned. The default cost is specified elsewhere in the *twig* specification.

(4) Like the cost part, the *action* is also C source code. The code is called by the matcher once it has been determined that this rule is part of the minimal cost cover. The code may return a tree to replace the subtree matching this rule. If no return value is given, then the subtree is left unchanged. The action part may also perform other functions, such as emitting code and updating code generator data structures.

The action part is also optional. If the action is missing, the default action is to leave the subtree unchanged.

Tree Patterns

Tree patterns are written in parenthesized prefix form and can be described by the following BNF:

```
pattern ← node_id
pattern ← label_id
pattern ← node_id(subtree_list)
subtree_list ← pattern
subtree_list ← pattern, subtree_list
```

That is to say, a tree pattern is written with its root identifier, followed by an optional parenthesized list representing the subtrees of the root in order from left to right. For example,

```
identifier
op(left, right)
plus(expr, times(constant, expr))
```

are all tree patterns.

As the BNF above suggests, there are two types of identifiers in *twig*, *node_ids* and *label_ids*, corresponding to the terminals and nonterminals of tree patterns. A *node_id* denotes an internal node or leaf while a *label_id* forms the label part of a rule. Each of the *node_ids* and *label_ids* is assigned a unique integer by *twig*. The identifier-to-integer mapping is provided in a generated source file. During pattern matching, *twig* will call a user-supplied function `mtValue(`n`)` that returns the integer corresponding to the symbol of node n.

Leaves of a pattern with *label_ids* are called labeled leaves. The textually leftmost labeled leaf is first, the next second, and so on. Labeled leaves play a special role as they represent rules matching subtrees rooted at their position just as nonterminals stand for reductions of substrings in context-free grammars. For example, the subject tree given in the example at the end of Section 6 can be written as

```
plus(plus(c, c), i(plus(c, c))
```

and the rules t_1, t_2, and t_3 are written as

```
r: plus(r, i(plus(c, r));
r: plus(r, r);
r: c;
```

respectively. The symbol `plus` is used instead of + because the latter would be syntactically incorrect in *twig*. In the cover shown at the end of Section 6, the first labeled leaf `r` of t_1 represents a match of t_2 in the subject tree; the second labeled leaf is a match of t_3.

Trees, Costs, and Actions

The *twig* pattern matcher treats trees and costs as an abstract data type, so as to minimize the constraints placed on their representation. All manipulations and accesses to tree and cost values are done via a well-defined procedural interface. The details of the interface are given in [42].

All legal C constructs are permitted in the C source code of the cost and action part of a rule. In addition, the following notations are provided for access to the subject tree and internal data structures of the pattern matcher.

(1) $\$\%n\$$ denotes a pointer to the matcher data structure for the nth labeled leaf. The next section will discuss this data structure in more detail. To access the cost value associated with that leaf, the notation $\$\%n\$ \rightarrow$ `cost` may be used.

(2) \$\$ denotes a point to the root of the subject tree.

(3) $\$n_1.n_2.n_3 \ldots n_{k-1}.n_k\$$ denotes a pointer to child n_k of child n_{k-1} of child $n_{k-2} \ldots$ of child n_1 of the root of the subject tree. Each n_i is a positive integer.

(4) Cost values can be returned to the matcher by assigning to the variable cost in the cost part of a rule.

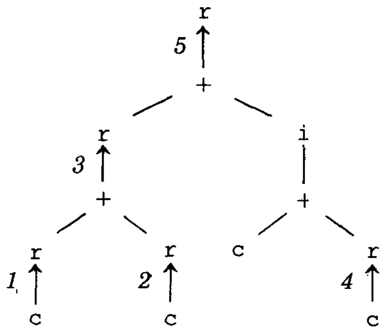
Refinements on Pattern Matching

The pattern matcher used in *twig* is similar to that described in Section 6 with a few additional modifications for improved efficiency.

The function $cost(t_i, n)$ used in the function *do_reduce* of Section 6 is implemented by calling the cost code associated with rule t_i. A match is aborted if, during the execution of the cost code, the ABORT statement is encountered; this is identical to returning an infinite cost value. The cost code also determines the mode of a match, which is described below.

Once a cover has been found, the reductions are performed during which time the action parts of the tree rules forming the cover are executed. Traditionally, one thinks of a reduction as consuming a subtree and replacing it with the label or nonterminal symbol of the rule. In *twig*, it is the execution of the action part of the rule during reduction that is important. Although the action part may modify the subject tree, *twig* does not require this to happen.

In the following, the word *reduction* will be used to mean the execution of the action part of a matching rule. The standard course of action is for *twig* to reduce matches in depth-first order. For the example of Section 6, the order of reductions is given by the numbers on the arrows in the following tree:



However, this standard order is changed if there are *top-down* mode matches. For example, if reduction 3 is from a top-down match, then it will be invoked before reductions 1 and 2. In fact, the latter two reductions will only be invoked if the built-in functions tDO($%1$) and tDO($%2$) are encountered while executing the action part of the rule causing reduction 3. In general, the function tDO($%*n*$) initiates reduction at the *n*th labeled leaf (and not at the *n*th reduction.) Explicit invocation allows the user to customize the exact ordering of execution in the cover.

A match can also be of *rewrite* mode. In that case the action code of the matching rule is executed immediately during the pattern-matching operation before any covers are computed. Matches below the rewrite-match in the tree are not reduced. To add this feature, the function *visit* is modified to call, just before returning, the routine *do_rewrite* given below.

```
do_rewrite(n)
{
  if min{n.cost} = n.cost[l] for some l and n.match[l] = m
    and $t_m$ matches with mode rewrite then
      execute the action associated with $t_m$
  visit(n)
}
```

Executing the action part of the tree-rewriting rule t_m may modify n and its subtree. Pattern matching continues after rewriting by calling *visit*(n) to reconsider the new subtree rooted at n. This can be done without modifying other parts of the tree because the bit-string technique of tracking partial matches will not propagate pattern-matching information of the replaced subtree past n toward the root. Rewrite-matches transform the subject tree during pattern matching. They are useful for canonizing subtrees for commutative operators and for performing constant folding.

Modes are determined by the cost code of a rule. While executing the cost code, if the built-in `TOPDOWN` is encountered, the match will be of top-down mode; if `REWRITE` is encountered, the mode will be rewrite; and if neither is encountered, the match is executed in the standard fashion.

Twig uses some additional techniques to improve its efficiency. If there are many tree rules, keeping a bit string for each rule at each node would consume large amounts of memory. In *twig* only the nonzero bit strings of each node are recorded. This saves memory because, on average, the number of rules that have nonzero bit strings is significantly smaller than the total number of rules.

Our version of *twig* does not keep the fields *cost*, *state*, and b_i in the actual subject tree. A separate structurally isomorphic tree is built by *twig* to hold this information.

Example 1

This *twig* specification generates VAX code for the subtract instruction:

```
prologue  { NODEPTR gettemp( ); };
node      long constant sub;
label     operand temp;

operand: long;                                      /* rule 1 */
operand: constant;                                  /* rule 2 */
operand: temp;                                      /* rule 3 */
temp:    operand;                                   /* rule 4 */
         {  cost = TEMP_COST+$%1$->cost; }
         ={ NODEPTR t = gettemp( );
            emit (''MOV'', $$, t, 0);
            return(t);
         };
operand:  sub(operand, operand)                     /* rule 5 */
         {  cost = SUB_COST+$%1-->cost+$%2$->cost; }
         ={ NODEPTR t = gettemp( );
            emit(''SUB'', $1$, $2$, t, 0);
            return(t);
         };
```

```
temp:    sub(temp, constant)                    /* rule 6 */
         { if(value($2$) ==1)
             cost = DEC_COST+$%1$→cost;
           else ABORT;
         }
         ={ emit(''DEC'', $1$, 0);
             return($1$);
         };
```

Notes

(1) The `prologue` statement provides C source code that can be referenced by the action and cost code in the rules.

(2) The `node` and `label` declarations indicate the node and label identifiers.

(3) `SUB_COST` and `DEC_COST` are the cost values for a subtract and decrement instruction, respectively. They should be provided by the user in a separate source file.

(4) Rules 3 and 4 form a potential loop, `temp`→`operand`→`temp`→`operand` ⋯, which is broken by the matcher recognizing that the cost of the second match of `temp` is less than that of the first match of `temp`.

(5) In Rule 5, the cost is the sum of the cost of the leaves plus the cost of the subtract instruction. The action clause emits code to subtract the two operands and to leave the result in a temporary location. The temporary is returned as a substitution for the subject tree.

(6) Rule 6 handles a special case where the left operand is already in a temporary and the constant is 1. In this case, the temporary is directly decremented and returned as the new tree.

(7) The routine `emit` takes a variable number of arguments, and value 0 marks the end of the argument list. The first argument is the opcode and subsequent arguments are operands of the instruction. Each operand node is converted to a representation dependent on the target machine.

Example 2

The following is a *twig* specification for the tree-rewriting rules in Figure 2:

```
node const mem assign plus ind;
label reg no_value;

reg : const                                        /* rule 1 */
  {  cost = 2; }
  ={ NODEPTR regnode = getreg( );
     emit('MOV', $1$, regnode, 0);
     return(regnode);
  };
reg : mem                                          /* rule 2 */
  {  cost = 2; }
  ={ NODEPTR regnode = getreg( );
     emit('MOV', $1$, regnode, 0);
     return(regnode);
  };
```

```
no_value: assign(mem, reg)                         /* rule 3 */
   {   cost = 2+$%1$→cost; }
    = { emit(''MOV'', $2$, $1$, 0);
        return(NULL);
   };
no_value: assign(ind(reg), mem)                    /* rule 4 */
   {   cost = 2+$%1$→cost; }
   ={  emit(''MOV'', $2$, $1$, 0);
       return(NULL);
   };
reg : ind(plus(const, reg))                        /* rule 5 */
   {  cost = 2+$%1$→cost; }
   ={ NODEPTR regnode = getreg( );
      emit(''MOV'', $$, regnode, 0);
      return(regnode);
   };
reg : plus(reg, ind(plus(const, reg)))             /* rule 6 */
   {  cost = 2+$%1$→cost+$%1$→cost; }
   ={ emit(''ADD'', $2$, $1$, 0);
      return($1$);
   };
reg : plus(reg, reg)                               /* rule 7 */
   {  cost = 1+$%1$→cost+$%2$→cost; }
   ={ emit(''ADD'', $2$, $1$, 0);
      return($1$);
   };
reg : plus(reg, constant)                          /* rule 8 */
   {  if(value($2$) ==1)
        cost = 1+$%1$→cost
      else ABORT;
   }
   ={ emitop(''INC'', $1$, 0);
      return($1$);
   };
```

Notes

(1) In rules 4 and 6, we assume that the `emit` routine will convert the tree `ind(reg)` and `ind(plus(const, reg))` into the correct target-machine addressing modes.

Additional details and applications of the *twig* language can be found in [42]. Appel [9] discusses *twig* specifications for the VAX and Motorola 68020 in detail.

8. EXPERIMENTAL RESULTS

To test these ideas, an experimental code generator for a VAX computer was built using *twig* and incorporated into the `pcc2` C compiler [32]. The modular design of the `pcc2` compiler made it easy to conduct this experiment. The compiler with the *twig* code generator is abbreviated `tcc`. Table II summarizes the overall compile times for `tcc` and the original `pcc2` compiler on 13 C programs. The first column gives the lines of code in each of these benchmark

Table II. Comparison of `tcc` and `pcc2` Compile Times

Program	Lines	tcc	pcc2	Percentage improvement
test1.c[a]	47	1.2 (0.6)	1.6 (0.6)	25.0
nm.c[b]	391	10.0 (1.3)	12.3 (1.2)	18.7
test3.c[c]	442	29.4 (2.8)	45.0 (2.8)	34.7
grep.c	458	12.1 (1.7)	15.2 (1.5)	20.4
local2.c[c]	530	8.9 (1.0)	12.3 (1.5)	27.6
local.c[c]	553	10.6 (1.3)	12.8 (1.3)	17.2
pmach.c	610	17.4 (2.3)	20.4 (2.5)	14.7
yacc	792	34.0 (3.0)	49.0 (2.9)	30.6
reader.c	1005	29.2 (2.6)	40.0 (2.4)	27.0
gencode.c[c]	1017	41.1 (4.6)	52.8 (4.2)	22.2
vmmem.c[d]	1041	19.3 (1.9)	24.4 (2.3)	20.9
cgram.c[c]	1181	30.4 (2.8)	38.6 (2.6)	21.2
ed.c	1729	42.5 (3.9)	47.0 (3.4)	9.6

[a] Tests arithmetic operators.
[b] Prints symbols from object module.
[c] Part of pcc2.
[d] part of Unix kernel.

programs. The second and third columns give the compile times in seconds on a VAX-11/780. The first number in these columns is the time spent in the compiler, the second parenthesized number is the time spent in the operating system.

As the table indicates, `tcc` is faster than `pcc2` (the average improvement was 23 percent). However, the system times for `tcc` are higher; these higher system times are caused by calls to the runtime system for dynamic storage. For a tree node, the average storage requirement is about 200 bytes; this storage is reclaimed and reused for every new IR tree. The faster compile times of `tcc` are due to the efficient tree-matching algorithm.

Creating the *twig* specification for `tcc` was straightforward. The *twig* specification for the VAX without the indexed-addressing modes was done in two weeks (while concurrently debugging the tree walker). The indexed-addressing modes were then added in a few hours once we were confident that the initial *twig* specification was correct. The final *twig* specification for the VAX code generator had 115 rules. Of these, 17 described addressing modes, another 17 were chain or transfer productions, 3 described labels, 1 reversed evaluation order, and the rest described single instructions or sequences of instructions. The specification file contained 853 lines of *twig* (about 14 pages). The figures are comparable to those of Appel's *twig* VAX specification [9].

The *twig* specification was also easy to modify. One reason is that new rules can be added to a *twig* specification independently of the other rules. The dynamic programming algorithm eliminates the possibility of looping and assures that an optimal covering of templates will always be chosen. Another reason is speed of the *twig* compiler. The *twig* compiler produced the code-generation tables from this specification very quickly, in 5.2 seconds to be precise. Thus, creating and testing the code generator could be done quickly. The code-generation tables were also very small—7.5K bytes. The entire *twig*-generated code generator for the VAX was 47.5K bytes in size.

Since pcc2 also uses dynamic programming, it was not surprising to note that the code generated by pcc2 for the sample programs was of the same quality as code generated by tcc. Only a few minor differences were noticeable because of different targeting and register-allocation strategies used by the two compilers. Occasionally tcc did better targeting. For example, tcc would generate

```
addl3  -4(fp), -8(fp), -12(fp)
```

where pcc2 would generate

```
addl3  -4(fp), -8(fp), r0
movl   r0, -12(fp)
```

However, it should be pointed out that the peephole optimizer usually used with pcc2 will perform this retargeting.

Occasionally pcc2 handled temporary registers better. For example, tcc would generate

```
cvtbl  (r0), r1
```

using two registers, where pcc2 would only use one:

```
cvtbl  (r0), r0
```

This aspect of code generation is a symptom of how temporary registers were allocated in tcc. Registers were allocated before code was emitted, and thus any registers freed during code emissions were not reused until they were explicitly freed by the action rule.

The MIPS-X Compiler

For the MIPS-X project [12], a new compiler was generated using *twig* and compared with a previously hand-generated compiler written in Pascal. The *twig*-generated code generator compiled significantly faster (about 40 percent) than the hand-generated compiler and generated slightly faster code (a few percent) even though the hand-generated compiler did some peephole optimizations. However, precise numerical comparisons are not meaningful here because the hand-generated MIPS-X compiler worked by generating MIPS code that a cross assembler transformed into MIPS-X code.

9. CONCLUSIONS

We believe the main advantages of *twig*'s approach to code generation are specification ease, compact tables, and fast generation times. We have found the *twig*-style tree-specification scheme well suited for describing the instruction-selection phase of code generation. Since *twig* automatically finds a minimum-cost covering using dynamic programming, the user does not need to worry about the order of the patterns in the tree-specification scheme. Moreover, additional patterns can be easily added subsequently to take advantage of machine idioms and peephole optimizations. As a consequence, we feel that a strong point of *twig* is its concise and expressive notation for describing efficient code generators. The fast compilation time of *twig* due to the efficient algorithm for constructing tree-pattern matchers also facilitates the incremental design of a code generator.

Another advantage of *twig* is the quality of the output code. The dynamic programming algorithm guarantees a minimum cost cover for each target tree. If the intermediate representation has been generated with care and if the costs of the target machine instructions have been faithfully represented in the *twig* rules, then our experience has shown that the output code is at least as good as can be generated by a hand-crafted code generator. However, *twig* does not do common subexpression elimination, algebraic simplification, or any other high-level optimizations, so further code improvement is still possible.

The speed of a *twig*-generated code generator is still slow compared with that of hand-crafted code generator for a specific machine, but, as we have seen, the speed of a *twig*-generated code generator is comparable to that of a retargetable compiler like `pcc2`. Of the total compile time, 30 percent was spent in code generation. However, we believe the speed of a *twig*-generated code generator can be further improved in several ways.

(1) About 80 percent of the time in a *twig*-generated code generator goes into the pattern matcher; of this, 20 percent is in simulating the automaton, 35 percent is in bookkeeping for the dynamic programming, and 6 percent is in computing costs. The current compact representation of the tree-pattern matching automaton uses a linear list to represent the transitions at each state except at the start state where an array representation is used. More efficient representations of the other states would speed up the tree-pattern matching. For example, storing the transitions of all states of the pattern-matching automaton for the *twig*-generated VAX code generator as arrays would use about 50K bytes but would provide a significant performance improvement.

(2) For every transition of the pattern-matching automaton a *twig*-generated code generator performs at least one procedure call to access a tree node. These procedure calls can be replaced by in-line macros.

(3) A *twig* specification may contain many chain rewrite rules such as

```
operand: temp
```

These rules increase the running time of *twig* since the effect of each of these rewrite rules is computed at run time. In many cases, it is possible to precompute the effect of these chain rewrite rules.

In summary, we feel *twig* is a promising tool for helping automate the construction of code generators. Integrating peephole optimization into code generation gives significant advantages [16, 20], and it would be interesting to evaluate adding peephole optimization into framework of *twig*.

ACKNOWLEDGMENTS

The authors are indebted to Charles Fischer, Steve Johnson, and Peter Weinberger for many valuable comments and stimulating discussions. The test programs in Table II were provided by Peter Weinberger.

REFERENCES

1. AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.

2. AHO, A. V., AND GANAPATHI, M. Efficient tree pattern matching: An aid to code generation. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages.* ACM, New York, 1985, pp. 334–340.
3. AHO, A. V., AND JOHNSON, S. C. Optimal code generation for expression trees, *J. ACM* 23, 3 (1976), 488–501.
4. AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. Code generation for expressions with common subexpressions. *J. ACM 24*, 1 (1977), 146–160.
5. AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. Code generation for machines with multi-register operations. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages.* ACM, New York, 1977, pp. 21–28.
6. AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. Deterministic parsing of ambiguous grammars. *Commun. ACM 18*, 8 (Aug. 1975), 441–452.
7. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
8. AIGRAIN, P., GRAHAM, S., HENRY, R., MCKUSICK, M., AND PELEGRI-LLOPART, E. Experience with a Graham-Glanville style code generator. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 19*, 6 (June 1984), 13–24.
9. APPEL, A. W. Concise specifications of locally optimal code generators. Tech. Rep. CS-TR-080-87, Dept. of Computer Science, Princeton University, Princeton, N.J., Feb. 1987.
10. BRUNO, J., AND SETHI, R. Code generation for a one-register machine. *J. ACM 23*, 3 (1976), 502–510.
11. CATTELL, R. G. G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst. 2*, 2 (April 1980), 173–190.
12. CHOW, P., AND HOROWITZ, M. The MIPS-X microprocessor. In *Proceedings of Wescon 1985* (San Francisco, Nov. 19–21, 1985). IEEE, New York, sec. 6-1, pp. 1–6.
13. DAVIDSON, J. W., AND FRASER, C. W. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst. 6*, 4 (Oct. 1984), 505–526.
14. DENCKER, P., DURRE, K., AND HEUFT, J. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst. 6*, 4 (Oct. 1984), 546–572.
15. FRASER, C. W. Automatic generation of code generators. Ph.D. dissertation, Yale University, New Haven, Conn., 1977.
16. FRASER, C. W., AND WENDT, A. Integrating code generation and optimization. In *Proceedings of ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 21*, 6 (June 1986), 242–248.
17. GANAPATHI, M., AND FISCHER, C. N. Affix grammar driven code generation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 560–599.
18. GANAPATHI, M., AND FISCHER, C. N. Attributed linear intermediate representations for retargetable code generators. *Softw. Pract. Exper. 14* (April 1984), 347–364.
19. GANAPATHI, M., AND FISCHER, C. N. Description-driven code generation using attribute grammars. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 1982, pp. 108–119.
20. GANAPATHI, M., AND FISCHER, C. N., Integrating code generation and peephole optimization. *Acta Inf. 25* (Jan. 1988), 85–109.
21. GANAPATHI, M., FISCHER, C. N., AND HENNESSY, J. L. Retargetable compiler code generation. *ACM Comput. Surv. 14*, 4 (Dec. 1982), 573–592.
22. GLANVILLE, R. S. A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, University of California, Berkeley, Dec. 1977.
23. GLANVILLE, R. S., AND GRAHAM, S. L. A new method for compiler code generation. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 1978, pp. 231–240.
24. GLASNER, I., MONCKE, U., AND WILHELM, R. OPTRAN, a language for the specification of program transformations. *Inf. Fach.*, 1980, 125–142.
25. GRAHAM, S. L. Table-driven code generation. *IEEE Comput. 13*, 8 (Aug. 1980), 25–34.
26. GRAHAM, S. L., HENRY, R. R., AND SCHULMAN, R. A. An experiment in table driven code generation. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 17*, 6 (June 1982), 32–43.

27. HATCHER, P. J., AND CHRISTOPHER, T. W. High-quality code generation via bottom-up tree pattern matching. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 1986, pp. 119–130.
28. HATCHER, P. J., KUKUCK, R. C., AND CHRISTOPHER, T W. Using dynamic programming to generate optimized code in a Graham–Glanville style code generator. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 19*, 6 (June 1984), 25–36.
29. HENRY, R. R. Graham–Glanville code generators. Ph.D. dissertation, Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, 1984.
30. HOFFMAN, C. W., AND O'DONNELL, M. J. Pattern matching in trees. *J. ACM 29*, 1 (1982), 68–95.
31. HUET, G., AND LEVY, J.-J. Call by need computations in non-ambiguous linear term rewriting systems. Tech. Rep. 359, IRIA Laboria, LeChesnay, France, 1979.
32. JOHNSON, S. C. A portable compiler: Theory and practice. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*. ACM, New York, 1978, pp. 97–104.
33. KEUTZER, K. Dagon: Technology binding and local optimization by dag matching. In *Proceedings of the 24th Design Automation Conference*. ACM/IEEE, New York, 1987, pp. 341–347.
34. KEUTZER, K., AND WOLF, W. Anatomy of a hardware compiler. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23*, 7 (July 1988), 95–104.
35. KRON, H. Tree templates and subtree transformational grammars. Ph.D. dissertation, University of California, Santa Cruz, 1975.
36. LANG, H.-W., SCHIMMLER, M., AND SCHMECK, H. Matching tree patterns sublinear on the average. Tech. Rep. Dept. of Informatik, University of Kiel, Kiel, West Germany, 1980.
37. LUNELL, H. Code Generator Writing Systems. Software Systems Research Center, S-58183, Linkoping, Sweden, 1983.
38. PELEGRI-LLOPART, E., AND GRAHAM, S. L. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 1988, pp. 294–308.
39. RIPKEN, K. Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe. Tech. Rep. TUM-INFO-7731, Institut fur Informatik, Technische Universitat Munchen, Munich, West Germany, July 1977.
40. SETHI, R. AND ULLMAN, J. D. The generation of optimal code for arithmetic expressions. *J. ACM 17*, 4 (1970), 715–728.
41. SNYDER, L. Recognition and selection of idioms for code optimization. *Acta Inf. 17* (1982), 327–348.
42. TJIANG, S. W. K. Twig reference manual. Computing Science Tech. Rep. 120, AT&T Bell Laboratories, Murray Hill, N.J., 1985.
43. WASILEW, S. G. A compiler writing system with optimization capabilities for complex order structures. Ph.D. dissertation, Northwestern University, Evanston, Ill., 1972.
44. WEINGART, S. W. An efficient and systematic method of compiler code generation. Ph.D. dissertation, Computer Sciences Dept., Yale University, New Haven, Com., 1973.
45. WULF, W. A. PQCC: A machine-relative compiler technology. In *Proceedings of the IEEE 4th International COMPSAC Conference*. IEEE, New York, 1980, pp. 24–36.
46. WULF, W., LEVERETT, B., CATTELL, R., HOBBS, S., NEWCOMER, J., REINER, A., AND SCHATZ, B. An overview of the production quality compiler-compiler project. *IEEE Comput. 13*, 8 (Aug. 1980), 38–49.

Received January 1986; revised February 1987, August 1988, and May 1989; accepted May 1989