

Extending Graham-Glanville Techniques for Optimal Code Generation

MAYA MADHAVAN, PRITI SHANKAR, SIDDHARTHA RAI, and U.RAMAKRISHNA
Indian Institute of Science

We propose a new technique for constructing code-generator generators, which combines the advantages of the Graham-Glanville parsing technique and the bottom-up tree parsing approach. Machine descriptions are similar to Yacc specifications. The construction effectively generates a pushdown automaton as the matching device. This device is able to handle ambiguous grammars, and can be used to generate locally optimal code without the use of heuristics. Cost computations are performed at preprocessing time. The class of regular tree grammars augmented with costs that can be handled by our system properly includes those that can be handled by bottom-up systems based on finite-state tree parsing automata. Parsing time is linear in the size of the subject tree. We have tested the system on specifications for some systems and report table sizes.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — *code generation, retargetable compilers, translator writing systems and compiler generators*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Optimal code generation, tree pattern matching, code-generator generator

1. INTRODUCTION

Regular tree grammars and Bottom Up Rewrite Systems (BURS) have been used extensively as specification mechanisms for retargetable code generation. The early work of Graham and Glanville [1978] opened up new directions in the area of code generation. They showed that if linearized intermediate code generated by the front end of a compiler is specified by a context-free grammar, and target machine instructions are represented as grammar productions, then an LR parser-generator can be used as a code-generator generator. Unfortunately the technique cannot be applied to the generally ambiguous grammars that define machine instructions, without using heuristics to resolve parsing conflicts. The use of such heuristics can often result in the generation of suboptimal code. Ganapathi and Fischer [1985] used affix grammars for retargetable code generation where semantic information is used to control parsing. Aho and Ganapathi [1985] showed that top-down tree-parsing combined with dynamic programming can be used for generating locally

The material in this paper was presented in part at the 18th International Conference on Foundations of Software Technology and Theoretical Computer Science, Chennai, India, December, 1998
Authors' address: Department of Computer Science and Automation, Institute of Science, Bangalore 560012, India; email: priti@csa.iisc.ernet.in.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2000 ACM 0164-0925/00/1100-0973 \$5.00

optimal code. The advantage of a top-down technique is that the tables describing the tree-parsing automaton are small. The disadvantage stems from the fact that cost computations associated with dynamic programming are performed at code generation time, thus slowing down the code generator. Dynamic programming using a top-down traversal was also used by Christopher et al. [1984]. Weisberger and Wilhelm [1988] describe top-down and bottom-up techniques to generate code. Bottom-up techniques based on the seminal work of Hoffman and O'Donnell [1982], Chase [1987], and Pelegri-Llopert and Graham [1988] were found to be more efficient than top-down techniques at code generation time. Hatcher and Christopher [1986] showed that cost analysis could be carried out statically in a bottom-up tree-parsing approach, so that code generation could be speeded up. Henry and Damron [1989] carried out empirical studies comparing the static and dynamic performance of the top-down and bottom-up tree-parsing techniques, and the Graham-Glanville technique. Balachandran et al. [1990] used an extension of the work of Chase to perform static cost analysis and produce optimal code. Fraser et al. [1992] developed a code generator with bottom-up tree parsing and dynamic programming. Proebsting [1995] used a simple and efficient algorithm for generating BURS tables, in which a new method called triangle trimming is used for state reduction. Ferdinand et al. [1994] reformulated the static bottom-up tree-parsing algorithm based on finite tree-automata. This generalized the work of Chase to work for regular tree grammars and included table compression techniques. More recently, Nymeyer and Katoen [1997] describe an implementation of an algorithm based on BURS theory, that computes all pattern matches, and which does a search that results in optimal code. Heuristics are used to cut down the search space. Shankar et al. [2000] construct a pushdown automaton for regular tree-parsing which can be used for code generation with dynamic cost computation [Gantait 1996].

In this paper, we combine the LR(0) parsing technique and the bottom-up tree-parsing strategy to develop a tool for retargetable, locally optimal code generation with static cost computations. The system is simple to use, general, and effective. Since the scheme is an extension of the LR(0) technique to process ambiguous but constrained context-free grammars, attributes can be associated with nonterminals in order to handle entities like register names, literal values, and so forth. The scheme retains the advantage of the tree-parsing approach, in that parsing decisions during the shift/reduce parsing process can be deferred arbitrarily far beyond the points where conflicts actually occur. However, this is at no extra cost, as matching is just parsing with an automaton called the *auxiliary automaton* replacing the deterministic finite automaton for canonical sets of LR(0) items. Machine specifications look like Yacc [Johnson 1975] input specifications containing rules and actions using attributes. Code generation is normally done in two passes as in the conventional case; the first pass is used to label the subject tree, and the second is a top-down pass used to generate code. One pass parsing in the spirit of Proebsting and Whaley [1996] is possible for a restricted class of grammars. The code generated is locally optimal, and labeling is performed in time linear in the size of the subject tree.

Section 2 presents some background on this technique; Section 3 introduces static cost computations and presents the algorithm. Section 4 gives a sufficient condition for termination. Section 5 evaluates our work within the context of current work in

retargetable code generation and presents the outputs of the algorithm on sample inputs. Finally, Section 6 concludes the paper.

2. EXTENSION OF THE LR(0) TECHNIQUE

Let A be a finite alphabet consisting of a set of operators OP and a set of terminals T . Each operator op in OP is associated with an *arity*, $arity(op)$. Elements of T have arity 0. The set $TREES(A)$ consists of all trees with internal nodes labeled with elements of OP , and leaves with labels from T . The number of children of a node labeled op is $arity(op)$. Special symbols called *wildcards* are assumed to have arity 0. If N is a set of wildcards, the set $TREES(A \cup N)$ is the set of all trees with wildcards also allowed as labels of leaves. We begin with a few definitions drawn from Balachandran et al. [1990].

Definition 2.1. A *regular cost-augmented tree grammar* G is a four tuple (N, A, P, S) where

- (1) N is a finite set of *nonterminal* symbols.
- (2) $A = T \cup OP$ is a *ranked alphabet*, with the ranking function denoted by *arity*. T is the set of *terminal* symbols, and OP is the set of *operators*.
- (3) P is a finite set of *production rules* of the form $X \rightarrow t [c]$, where $X \in N$ and t is an encoding of a tree in $TREES(A \cup N)$, and c is a cost, which is a nonnegative real number.
- (4) S is the *start symbol* of the grammar.

A *tree pattern* is thus represented by the right-hand side of a production of P in the grammar above. A production of P is called a *chain rule*, if it is of the form $A \rightarrow B$, where both A and B are nonterminals. For purposes of the discussion here we will be dealing with grammars in *normal form* defined below. Costs are assumed to be additive.

Definition 2.2. A production is said to be in normal form if it is in one of the three forms below.

- (1) $A \rightarrow op(B_1, B_2, \dots, B_k)[c]$ where $A, B_i, i = 1, 2, \dots, k$ are all nonterminals, and op has arity k .
- (2) $A \rightarrow B [c]$, where A and B are nonterminals. Such a production is called a *chain rule*.
- (3) $B \rightarrow b [c]$, where b is a terminal.

A grammar is in normal form if all its productions are in normal form. Any regular tree grammar can be put into normal form by the introduction of extra nonterminals. Example 2.1 is an example of a cost-augmented regular tree grammar in normal form. Arities of symbols in the alphabet are shown in parentheses next to the symbol.

Derivation sequences are defined in the usual way. However, we note that the objects being derived are trees. With each derivation tree is associated a cost, namely, the sum of the costs of all the productions used in constructing the derivation tree. We label each nonterminal in the derivation tree with the cost of the subtree below

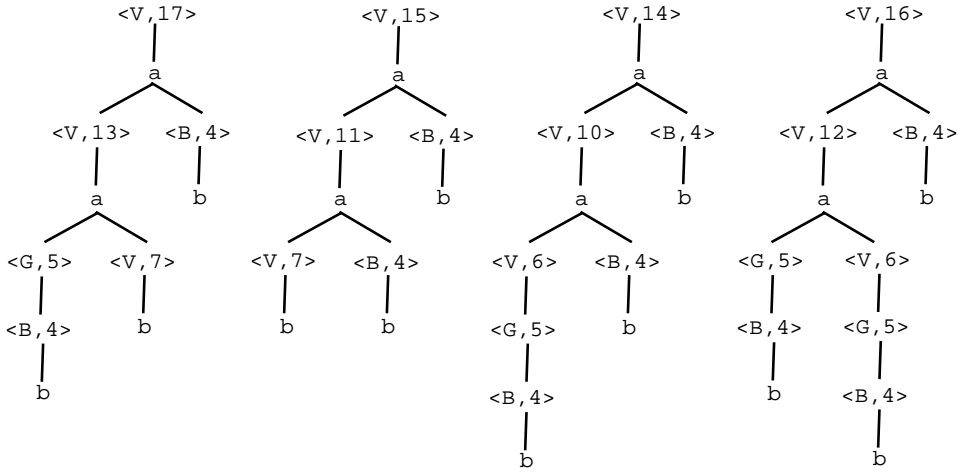


Fig. 1. Four cost-augmented derivation trees for the subject tree $a(a(b,b),b)$ in the grammar of Example 2.1.

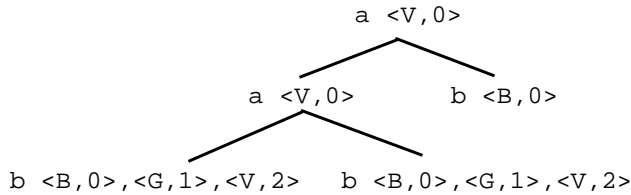


Fig. 2. Subject tree of Figure 1 shown with $\langle \text{matching nonterminal, relative cost} \rangle$ pairs.

it. Four cost-augmented derivation trees for the subject tree $a(a(b,b),b)$ in the language generated by the regular tree grammar of Example 2.1 above are displayed in Figure 1.

Example 2.1.

$G = (\{V,B,G\}, \{a(2), b(0)\}, P, V)$

P:

- $V \rightarrow a(V,B) \quad [0]$
- $V \rightarrow a(G,V) \quad [1]$
- $V \rightarrow G \quad [1]$
- $G \rightarrow B \quad [1]$
- $V \rightarrow b \quad [7]$
- $B \rightarrow b \quad [4]$

A tree pattern is said to *match* at a node in the subject tree, if a production with right-hand side corresponding to the tree pattern is used at that node in the derivation tree. Each derivation tree thus defines a set of matches, (a *set* because there may be chain rules that match) at each node in the subject tree. (Actually, as

we shall see later, conventional regular tree-pattern-matching algorithms compute, in general, a superset of the tree patterns that actually match). Typically, any algorithm that does dynamic cost computations, compares the costs of all possible derivation trees and selects one with minimal costs while computing matches. To do this it has to compute, for each nonterminal that matches at a node, the minimal cost of reducing to that nonterminal (or equivalently, deriving the portion of the subject tree rooted at that node from the nonterminal). In contrast, algorithms that perform static cost computations, precompute relative costs, and store differential costs for nonterminals. Thus, the cost associated with a nonterminal at a particular node in a subject tree is the difference between the minimal cost of deriving the subtree of the subject tree rooted at that node from the nonterminal and the minimal cost of deriving it from any other nonterminal. Figure 2 shows the relative costs and matching nonterminals at the nodes of the subject tree for which derivation trees are displayed in Figure 1.

Assuming such differences are bounded, they can be stored as part of the information in the states of a finite-state tree-parsing automaton. Thus no cost analysis need be done at matching time. Clearly, tables encoding the tree automaton with static cost analysis tend to be larger than those where dynamic analysis is performed. Both algorithms solve the following problem: *given a cost-augmented regular tree grammar G and a subject tree t , find a representation of a cheapest derivation tree for t in G .*

The regular tree-parsing problem for a grammar G can be converted into a string-parsing problem for a context-free grammar. We can construct a new grammar G' from G , by converting right-hand sides of all productions of G into postorder listings of the corresponding tree patterns. Any linear listing will work; we use a postorder listing here. Since each operator has a fixed arity, the postorder listing uniquely defines the tree. Let $post(t)$ denote the postorder listing of a tree t . We note that the grammar G' is in general ambiguous. It is shown in Shankar et al. [2000] that finding all derivation trees in the string grammar G' corresponds to finding all derivations in the tree grammar G .

We assume that the reader is familiar with the notions of right sentential forms, handles, viable prefixes, and $LR(0)$ items being valid for viable prefixes. Definitions may be found in Hopcroft and Ullman [1979]. By a viable prefix *induced* by an input string, we mean a prefix of a right sentential form from which the string can be derived. The key theorem underlying the algorithm follows:

THEOREM 2.1. *Let G' be a context-free grammar constructed from a regular tree grammar in normal form. Then for any input string w which is a prefix of a string in $L(G')$, all viable prefixes induced by w are of the same length.*

PROOF. The proof rests on the following four observations.

- (1) A shift of any symbol is always followed by a reduction.
- (2) If the symbol shifted is a terminal symbol, then the length of the viable prefix remains the same, as the handle is of length 1.

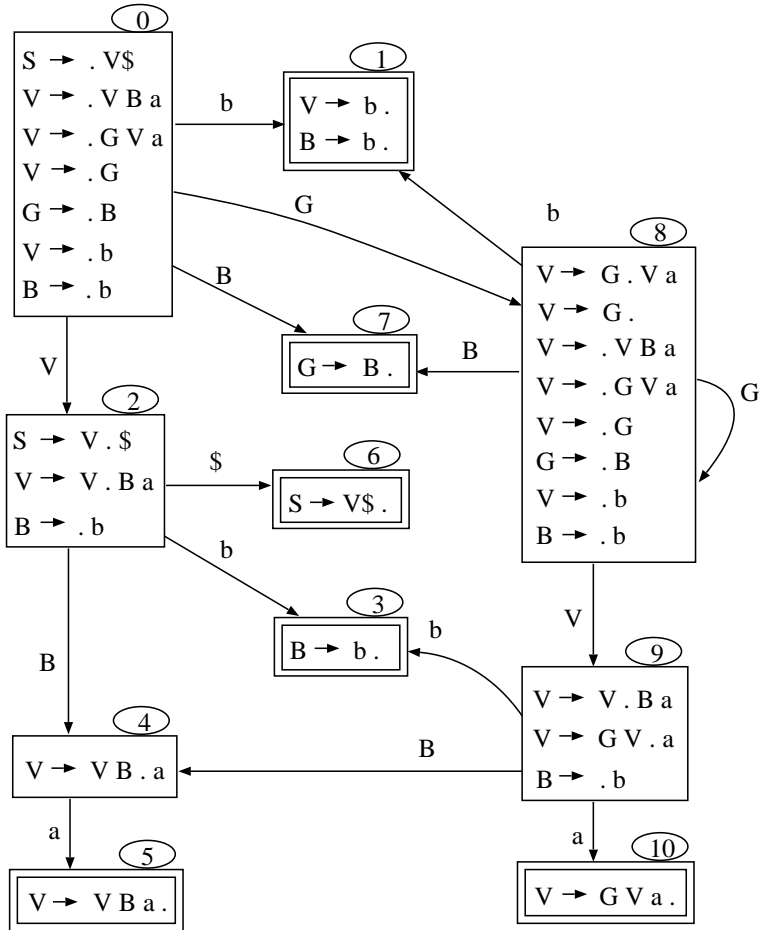


Fig. 3. DFA for sets of LR(0) items for grammar of Example 2.1.

- (3) If the symbol is an operator op , then the viable prefix reduces by length $arity(op)$.
- (4) Reduction by chain rules does not change the length of the viable prefix.

The first three observations are a consequence of the fact that the grammar is in normal form and do not depend on the rightmost derivation sequence used. Therefore all parsing sequences for an input string yield viable prefixes of the same length. \square

The fact that all viable prefixes are of the same length allows us to perform all shifts and reductions during a bottom-up parse, in synchrony. Ignoring costs for the time being, we can construct a finite-state automaton which we will henceforth call the *auxiliary automaton*, which plays the same role as the deterministic finite automaton (dfa) for canonical sets of LR(0) items does during LR parsing. Our dfa will recognize *sets* of viable prefixes. The auxiliary automaton for the context-

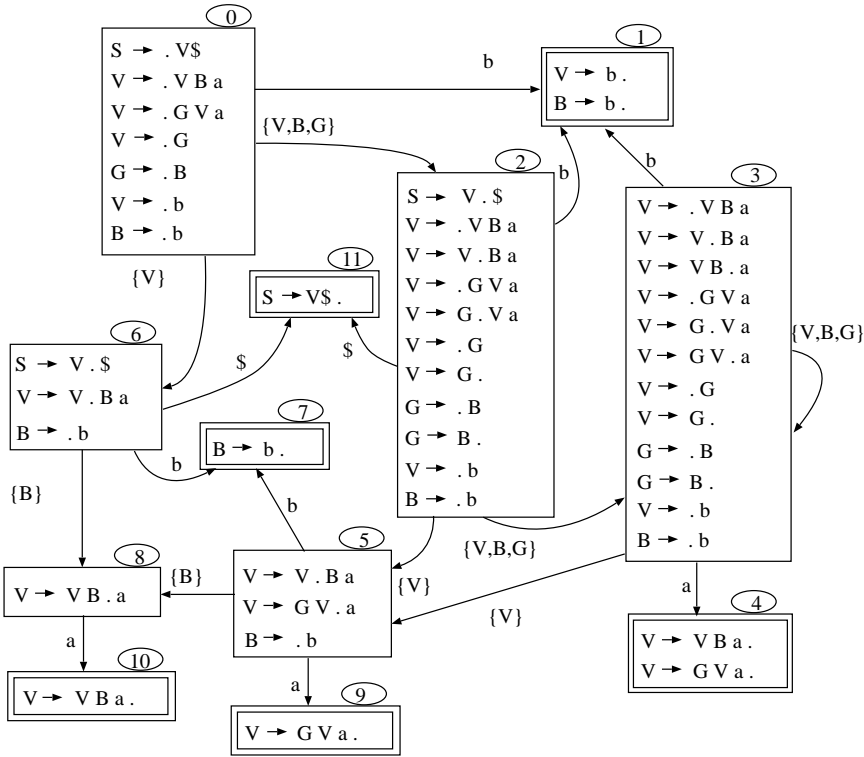


Fig. 4. The auxiliary automaton without costs for grammar of Example 2.1.

free grammar constructed from the grammar of Example 2.1 is shown in Figure 4. Thus, state 2 of the dfa is associated with the set of viable prefixes $\{V, B, G\}$. State 3 is associated with the set $\{VB, GG^*B, GG^*V, GG^*\}$. The dfa for the canonical sets of LR(0) items of the grammar is given in Figure 3. An inspection of the two figures will reveal that the states of the dfa in Figure 4 are formed by merging the appropriate states of the dfa in Figure 3. Thus state 2 of Figure 4 is formed by merging states 2, 7, 8 of Figure 3; state 3 of Figure 4 is formed by merging states 4, 7, 8, 9 of Figure 3, and so on.

The technique is formally described in Shankar et al. [2000]. We give an informal presentation here, illustrating the basic idea with the example above. The algorithm maintains two data structures, a list *list* of states generated, but not yet processed, and two sets *lcsets* and *matchsets*, containing the *shift* and *reduce* states of the new automaton respectively. Initially, the list contains only the *start* state, which is the same as that for the LR(0) dfa, and is associated with the same set of items. The sets *lcsets* and *matchsets* are initially empty. Transitions of the automaton are stored in two tables, δ_A (for transitions on input symbols) and δ_{LC} for transitions on nonterminal sets. (LC stands for “left context”.) The *goto* operation is the same as that defined for a conventional LR(0) parser. At each iteration, a state, say *p*, is picked from *list*, and for each input symbol *a* the following sequence of steps is

carried out.

- (1) Add p to *lcsets*. Generate a state, say m , to which a transition from p is made on a , creating a new set of items in the usual manner.
- (2) Check if this state m is in *matchsets*. If it is not, add it to *matchsets*. Update δ_A with this new transition.
- (3) As the grammar is in normal form, m is always a *reduce* state. This state may contain more than one complete item. However, Theorem 2.1 guarantees that all right-hand sides of complete items in a *reduce* state are of the same length, say l . Therefore find all states in *lcsets* that have paths of length l to m . These are processed states that could be exposed on stack during parsing when a reduction is attempted from m . Let q be such a state. For each such state q , perform steps 4, 5, 6, and 7 below. When all such states have been processed, go to step 8.
- (4) Set the current match state to m and the current left context state to q .
- (5) Let $A = \{A_1, A_2, \dots\}$ be the set of nonterminals on the left-hand sides of complete items in the current match state. Augment A by including all nonterminals B such that $B \Longrightarrow^* A_i, A_i \in A$ where B occurs after the dot in an item in the current left context state. Let this set be A' .
- (6) Generate a state, say v , by performing a *goto* operation from the current left context state on all nonterminals in A' , thereby creating a new set of items, and update δ_{LC} .
- (7) Check if v is already in *list* or *lcsets*. If not, add it to *list*.
- (8) For each state r already in *matchsets* check if p could be exposed when a reduction is attempted at r . If so, perform steps identical to steps 5, 6, and 7 with the current match state set to r and current left context state set to p . When all states in *matchsets* have been checked go to the next step.
- (9) Repeat step 1 with the next input symbol.

Thus, the sequence of steps above is repeated for each input symbol, every time a state is picked from *list*. Eventually *list* becomes empty, and the algorithm terminates. The functions δ_A and δ_{LC} essentially represent the transition function of the automaton.

As one may observe, the process is similar to that for the construction of the dfa for canonical sets of LR(0) items. However, in the conventional algorithm, transitions are decided completely by the set of items in the state. In our case, the transitions on sets of nonterminals can be deduced only after computing the sets of nonterminals that match in the same left context, like the A_i in step 5.

As an example, consider the start state numbered 0 in Figure 4. This is the only member in *list* to begin with, so it is picked; a transition on b takes us to state 1 which is added to *matchsets*. The transition from 0 to 1 is added to δ_A . The nonterminals that match are V and B . Adding contributions from chain rules, we get the augmented set $\{V, B, G\}$. Thus a transition from 0 to 2 is possible on the set $\{V, B, G\}$; this transition is added to δ_{LC} , and state 2 is added to *list*. Next, state 2 is picked from *list* and added to *lcsets*. A transition on b from 2 takes us to state 1 again, and this transition is added to δ_A . The matching nonterminals augmented by application of chain rules gives the set $\{V, B, G\}$. A transition on

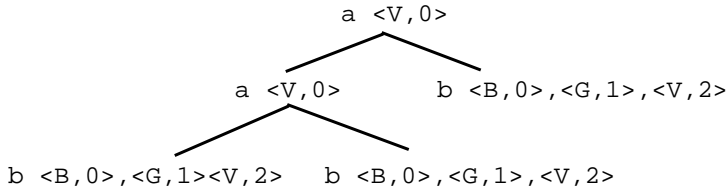


Fig. 5. <matching nonterminal, relative cost> pairs reported by the conventional algorithm.

this set from state 2 gives a new state 3, which is added to *list*, and the transition from state 2 to state 3 is added to δ_{LC} . The algorithm proceeds in this way, finally obtaining the automaton in Figure 4.

The technique for constructing the auxiliary automaton can be thought of as a kind of subset construction on the dfa for canonical sets of LR(0) items, though one does not need to explicitly begin with the LR(0) automaton. Matching is reduced to bottom-up parsing using the auxiliary automaton as the finite control. Pattern matches occur wherever reductions are called for. The following sequence of sets of right sentential forms illustrates this. Each right sentential form in a set has a double vertical bar and sometimes, in addition, a single one. The double bar separates the representation of the stack contents from the rest of the input. The single bar occurs only in states where matches occur and separates the handle from the stack contents to its left, the latter also called the *left context*. The sequence of sets of sentential forms is obtained when parsing the postorder representation of the subject tree in Figure 2 according to a context-free grammar obtained from the grammar of example 2.1. Sets which imply reductions are underlined. \$ is the bottom-of-stack marker. For each set of sentential forms, we also indicate, within parentheses, the state of the auxiliary automaton at that point in the parse.

- (1) $\{\$||bbaba\$\}$ (0)
- (2) $\{\$|b||baba\$\}$ (1)
- (3) $\{\$B||baba\$, \$V||baba\$, \$G||baba\$\}$ (2)
- (4) $\{\$G|b||aba\$, \$V|b||aba\$\}$ (1)
- (5) $\{\$GV||aba\$, \$VB||aba\$, \$GG||aba\$, \$GB||aba\$\}$ (3)
- (6) $\{\$|GVa||ba\$, \$|VBa||ba\$\}$ (4)
- (7) $\{\$V||ba\$\}$ (6)
- (8) $\{\$V|b||a\$\}$ (7)
- (9) $\{\$VB||a\$\}$ (8)
- (10) $\{\$|VBa||\$\}$ (10)
- (11) $\{\$V||\$\}$ (6)
- (12) $\{\$|V\$\}$ (11)(*accept*)

There are a few points worth noting with respect to the sequence of sets of right sentential forms above. In set 9, the handle *b* causes reduction (or equivalently is

matched) by $B \rightarrow b$ whereas in set 2 it causes reductions by both $B \rightarrow b$ and $V \rightarrow b$. This difference in the matched sets would not be present when matching using a finite-state tree automaton which would report the matches $V \rightarrow b$ and $B \rightarrow b$ for both cases. The difference here arises from the fact that the *left contexts* for the two cases are different. For set 2 it is $\$,$ and for set 9 it is $\$V$. Thus with reference to Figure 4, for set 2 in the sequence, b is matched in left context state 0, whereas for set 9 in the sequence, it is matched in left context state 6. We now formalize the notion of a pattern matching in a left context.

By an X -derivation tree we mean a subtree of a derivation tree with root labeled X . Let n be any node in a tree t . A subtree t_i is said to be to the left of node n in the tree, if the node m at which the subtree t_i is rooted occurs before n in a postorder listing of t . t_i is said to be a maximal subtree to the left of n if it is not a proper subtree of any subtree that is also to the left of n .

Definition 2.3. Let $G = (N, T, P, S)$ be a regular tree grammar in normal form, and t be a subject tree. Then pattern β represented by production $X \rightarrow \beta$ matches at node j in left context $\alpha, \alpha \in N^*$ iff conditions 1, 2, and 3 below hold.

- (1) β matches at node j or equivalently, $X \Rightarrow \beta \Rightarrow^* t'$ where t' is the subtree rooted at j .
- (2) If α is not ϵ , then the sequence of maximal complete subtrees of t to the left of j , listed from left to right, is t_1, t_2, \dots, t_k , with t_i having an X_i -derivation tree, $1 \leq i \leq k$, where $\alpha = X_1 X_2 \dots X_k$.
- (3) The string $X_1 X_2 \dots X_k X$ is a prefix of the postorder listing of some tree in $TREES(A \cup N)$ with an S -derivation.

Example 2.2. In the leftmost derivation tree in Figure 1, the pattern represented by production $B \rightarrow b$ matches in left contexts ϵ and V . In the second derivation tree from the left, both instances match in left context V .

Since our algorithm keeps track of left contextual information, it effectively performs a more refined version of regular tree-pattern-matching than the conventional bottom-up algorithm [Hoffman and O'Donnell 1982; Chase 1987; Balachandran et al. 1990]. Figure 5 shows the matchsets reported by the conventional algorithm for the subject tree of Figure 1. (The matchsets reported by our algorithm are those of Figure 2). Both algorithms, in general, report approximations to the sets that actually match when the derivation tree is complete. Our approximations are, in general, better, as demonstrated in the figures referred to above.

The conventional algorithm does not have the resources to keep track of left contexts; our algorithm can keep track of left contexts but not of right context, and therefore our sets are still, in general, approximations to the actual sets. The difference between the conventional algorithm and ours allows us to construct (pathological) examples where the size of the conventional automaton is exponential in the size of the extended $LR(0)$ automaton. One such class of examples is given in Shankar et al. [2000]. The next section describes the construction of the auxiliary automaton augmented with costs.

3. COMPUTING THE AUXILIARY AUTOMATON WITH STATIC COSTS

We begin with a theorem from Shankar et al. [2000].

THEOREM 3.1. *Let $G = (N, T, P, S)$ be a regular tree grammar, and G' the context-free grammar constructed as before. Let t a subject tree with postorder listing $a_1 \dots a_j w, a_i \in A, w \in A^*$. Then pattern β represented by production $X \rightarrow \text{post}(\beta)$ of G' matches at node j in left context α if and only if there is a rightmost derivation in the grammar G' of the form*

$$S \Longrightarrow^* \alpha X z \Longrightarrow^* \alpha \text{post}(\beta) z \Longrightarrow^* \alpha a_h \dots a_j z \Longrightarrow^* a_1 \dots a_j z, z \in A^*$$

where $a_h \dots a_j$ is the subtree rooted at node j .

Theorem 3.1 shows that finding matches of patterns in left contexts in a subject tree and obtaining derivation sequences of postorder listings of subject trees are equivalent problems. Our object here is to generalize the construction of Shankar et al. [2000] to include cost precomputations, the starting point being a context-free grammar derived from a regular tree grammar augmented with costs. We denote by $\text{rulecost}(p)$ the cost of production p . Let $G = (N, T, P, S)$ be a cost-augmented regular tree grammar in normal form.

Definition 3.1. The absolute cost of a nonterminal X matching an input symbol a in left context ϵ is represented by $\text{abscost}(\epsilon, X, a)$. For a derivation sequence d represented by $X \Longrightarrow X_1 \Longrightarrow X_2 \dots \Longrightarrow X_n \Longrightarrow a$, let $C_d = \text{rulecost}(X_n \longrightarrow a) + \sum_{i=1}^{n-1} \text{rulecost}(X_i \longrightarrow X_{i+1}) + \text{rulecost}(X \longrightarrow X_1)$; then $\text{abscost}(\epsilon, X, a) = \min_d(C_d)$.

Definition 3.2. The absolute cost of a nonterminal X matching a symbol a in left context α is defined as follows:

$$\begin{aligned} \text{abscost}(\alpha, X, a) &= \text{abscost}(\epsilon, X, a) \text{ if } X \text{ matches in left context } \alpha \\ &= \infty \text{ otherwise} \end{aligned}$$

Definition 3.3. The relative cost of a nonterminal X matching a symbol a in left context α is $\text{cost}(\alpha, X, a) = \text{abscost}(\alpha, X, a) - \min_{y \in N} \{\text{abscost}(\alpha, Y, a)\}$.

Having defined costs for trees of height one we next look at trees of height greater than one. Let t be a tree of height greater than one.

Definition 3.4. The cost $\text{abscost}(\alpha, X, t) = \infty$ if X does not match t in left context α . If X matches t in left context α , let $t = a(t_1, t_2, \dots, t_q)$ and $X \longrightarrow Y_1 Y_2 \dots Y_q a$ where Y_i matches $t_i, 1 \leq i \leq q$.

Let $abscost(\alpha, X \rightarrow Y_1 Y_2 \dots Y_q a, t) = rulecost(X \rightarrow Y_1 \dots Y_q a) + cost(\alpha, Y_1, t_1) + cost(\alpha Y_1, Y_2, t_2) + \dots + cost(\alpha Y_1 Y_2 \dots Y_{q-1}, Y_q, t_q)$. Hence define

$$abscost(\alpha, X, t) = \min_{X \Rightarrow \beta \Rightarrow *t} \{abscost(\alpha, X \rightarrow \beta, t)\}.$$

Definition 3.5. The relative cost of a nonterminal X matching a tree t in left context α is $cost(\alpha, X, t) = abscost(\alpha, X, t) - \min_{Y \Rightarrow *t} \{abscost(\alpha, Y, t)\}$.

Having defined the cost of a nonterminal matching in a left context, we next introduce the notion of an augmented rightmost derivation of a context-free grammar derived from the regular tree grammar as in the previous section. To avoid complicating notation, we assume from now on that $G = (N, T, P, S)$ is the context-free grammar

Definition 3.6. A single *augmented derivation step* is of the form $\alpha < A, c > w \Rightarrow \alpha < X_1, d_1 > \dots < X_k, d_k > < a, 0 > w$ where

- (1) α is a sequence of $< nonterminal, cost >$ pairs, such that if α' is the concatenation of first members of pairs of α then $S \Rightarrow^* \alpha' A w \Rightarrow \alpha' X_1 X_2 \dots X_k a w$
- (2) $\exists t. s. A \Rightarrow X_1 X_2 \dots X_k a \Rightarrow^* t$ with $X_i \Rightarrow^* t_i, i \leq k$ and $t = t_1 t_2 \dots t_k a$
- (3) $cost(\alpha', X_1, t_1) = d_1, cost(\alpha' X_1, X_2, t_2) = d_2, \dots, cost(\alpha' X_1 X_2 \dots X_{k-1}, X_k, t_k) = d_k$ and $c = rulecost(A \rightarrow X_1 \dots X_k) + \sum_{i=1}^k d_i - \min_B \{abscost(\alpha', B, t)\}$.

Definition 3.7. An *augmented viable prefix* is any prefix of an augmented right sentential form such that the concatenation of the first elements yields a viable prefix.

Definition 3.8. An augmented item $[A \rightarrow \alpha. \beta, c]$ is valid for an augmented viable prefix $\gamma = < X_1, c_1 > < X_2, c_2 > \dots < X_n, c_n >$ if,

- (1) $[A \rightarrow \alpha. \beta]$ is valid for viable prefix $X_1 X_2 \dots X_n$.
- (2) If $\alpha = X_i \dots X_n$ then, if $\beta \neq \epsilon$ then $c = c_i + c_{i+1} \dots + c_n$ else $c = c_i + c_{i+1} \dots + c_n + rulecost(A \rightarrow \alpha)$.

We are actually interested in a similar definition of validity for *sets* of augmented viable prefixes (which we term *set viable prefixes*), where the costs associated with the items are relative costs. Thus we are naturally led to the following definition.

Definition 3.9. An augmented item $[A \rightarrow \alpha. \beta, c]$ is valid for set viable prefix $S_1 S_2 \dots S_n$ if

- (1) There exists an augmented viable prefix $\gamma = < X_1, c_1 >, < X_2, c_2 > \dots, < X_n, c_n >$ in $S_1 S_2 \dots S_n$ with $[A \rightarrow \alpha. \beta, c']$ valid for γ .
- (2) If $\beta \neq \epsilon$ then $c = c'$ else $c = c' - \min_{\gamma' \in S_1 S_2 \dots S_n} \{c'' : [\beta \rightarrow \delta., c''] \text{ is valid for } \gamma'\}$.

3.1 Construction of the Auxiliary Automaton

The definitions above suggest the construction of a finite-state machine similar to that used for the recognition of viable prefixes of an $LR(0)$ grammar. If the grammar is $LR(0)$, each prefix of a string in the language induces exactly one viable prefix before any reduction, and there is at most one reduction possible at any step. In this case there may be several augmented viable prefixes induced by the same string. However, as these are all of the same length, shifts and reductions may be carried out using a single stack. This permits the use of a simple extension of the $LR(0)$ parsing technique. We first augment the grammar with a production of the form $Z \rightarrow S\$, [0]$ where $\$$ is a symbol not in the alphabet, in order to make the language prefix free. We next precompute sets of augmented items that are valid for all augmented viable prefixes induced by the same input string. All these are included in a single state. We can then construct the auxiliary automaton as follows:

- $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{I \mid I \text{ is a distinct set of augmented items}\}$
- $\Sigma = A \cup 2^{<N, c>}$, where $<N, c>$ is a (*non terminal, cost*) pair.
- q_0 is the state associated with the initial set of augmented items.
- F is the state containing the augmented item $[Z \rightarrow S\$, 0]$.
- δ is the transition function made up of δ_A , the set of transitions on elements of A , and δ_{LC} , those on sets of (*nonterminal, cost*) pairs. The function δ can be extended to strings of symbols in the usual way.

Assume that we can precompute the states and transitions of this auxiliary automaton. This is then the controlling automaton for a parser for the language generated by the context-free grammar G , the parser performing the additional role of minimum-cost tree selection. The auxiliary automaton plays a role analogous to the dfa for canonical sets of $LR(0)$ items in an $LR(0)$ parser.

The auxiliary automaton to be constructed will satisfy the following property:

Property 3.1. An augmented item $[A \rightarrow \alpha . \beta, c]$ is valid for an augmented viable prefix γ , if and only if, $\delta(q_0, S_1 \cdots S_n)$ contains $[A \rightarrow \alpha . \beta, c]$, where $\gamma \in S_1 S_2 \dots S_k$

As a consequence of Property 3.1, we are assured that if we use the LR parsing technique of shifts and reductions (with the difference that several reductions may be applied simultaneously), a derivation sequence in reverse which yields a state containing $[S \rightarrow \alpha . 0]$ is a cheapest (optimal) matching sequence. Although we can use a technique similar to the technique for construction of the automaton for recognizing viable prefixes of an LR grammar, there is one striking difference. The transitions out of a state of the automaton that recognizes viable prefixes of the LR grammar are determined by just the items associated with the state. Here, the transitions on sets of (*nonterminal, cost*) pairs can be deduced only *after computing sets of such pairs that match in the same left context*. There are three functions required. The first, called the *goto* function, which is similar to the corresponding function of an $LR(0)$ automaton, restricted to only terminal symbols, computes the new set of augmented items obtained by making a transition from the old set

on a terminal or operator symbol, and computes differential costs. Because our grammar is in normal form, all such transitions will always lead to complete items. Such a set of items in our terminology is called a *matchset*.

The *reduction* function has two parameters. The first is a set of complete items all of which call for reductions. The second is a set of augmented items \mathcal{I} associated with a state which will be uncovered on stack when a sequence of states corresponding to the right-hand sides of all these productions is popped off the stack. (For the time being let us not concern ourselves with how exactly such a set of items is identified.) We call such a set of items an *LCset*, and we say that a state associated with such a set is a *valid left context state* for the matchset. The function *reduction* first collects the left-hand-side (*nonterminal, cost*) pairs associated with a *matchset* into a set \mathcal{R} , and computes relative costs. It then augments \mathcal{R} with (*nonterminal, cost*) pairs that can be added using chain rules that are applicable, updating costs as it proceeds. Finally, it creates a new set of augmented items by making transitions on elements of \mathcal{R} from \mathcal{I} . We describe these functions formally below. For a given matchset m we use S_m to denote the set of left-hand-side non-terminals of complete items.

$$\text{goto}(\text{itemset}, a) = \{[A \rightarrow \alpha a ., c] / [A \rightarrow \alpha . a, c'] \in \text{itemset} \text{ and} \\ c = c' + \text{rule_cost}(A \rightarrow \alpha a) - \min\{c'' + \text{rule_cost}(B \rightarrow \beta a) / \\ [B \rightarrow \beta . a, c''] \in \text{itemset}\}\}$$

The *reduction* operation on a set of complete augmented items itemset_1 with respect to another set of augmented items, itemset_2 is encoded in the form of the following function.

```
function reduction(itemset2, itemset1)
begin
  //Costs of nonterminals in matchsets
  S = Sitemset1
  cost(X) = min{ci / [X → αi ., ci] ∈ itemset1} if X ∈ S∞ otherwise
  // Process chain rules and obtain costs of nonterminals
  temp = ∪{[A → B ., c] / ∃[A → .B, 0] ∈ itemset2 and [B → γ ., c1] ∈ itemset1
  and c = c1 + rule_cost(A → B)}
  repeat
    S = S ∪ {X / [X → Y ., c] ∈ temp}
    for X ∈ S do
      cost(X) = min(cost(X), min{ci / ∃[X → Yi ., ci] ∈ temp})
      temp = {[A → B ., c] / ∃[A → .B, 0] ∈ itemset2 and [B → Y ., c1] ∈ temp
      and c = c1 + rule_cost(A → B)}
    end for
  until no change to cost array or temp = φ
  //Compute reduction
  reduction = ∪{[A → α B . β, c] / [A → α . B β, c1] ∈ itemset2 and B ∈ S and
  c = cost(B) + c1 if β ≠ ε else
  //This is a complete item corresponding to a chain rule
```

```

c = rule_cost(A → B) − min{ci∃[X → .Y, 0] ∈ itemset2, and ci = rule_cost(X →
Y)}
end

```

The function *closure* is encoded as follows:

```

function closure(itemset)
begin
repeat
  itemset = itemset ∪ {[A → .α, 0] / [B → .Aβ, c] ∈ itemset}
until no change to itemset
closure = itemset;
end

```

Define *ClosureReduction*(*itemset₁*, *itemset₂*) as,

```

closure(reduction(itemset1, itemset2)).

```

To identify states that can be valid left context states for a given matchset, we follow a simple technique that tests a condition that is necessary but not sufficient. As a consequence some spurious states may be generated (which are unreachable at run time). The choice of this over the exact technique [Shankar et al. 2000] was dictated by considerations of efficiency, and the fact that for all the examples tested out by us, the approximate test gave the same results as the exact test, except in one case where it produced one extra state. Informally speaking we check if each item of the form [*A* → *α.*, *c*] in a matchset has a corresponding item of the form [*A* → *.α*, 0] in the state under test for the valid left context property, and that there is a tally of items with productions having the same right-hand sides. The condition is not sufficient because there may be another production of the form *B* → *β* that always matches in this left context along with the other productions, but whose corresponding complete item is not in the matchset. We omit a formal definition of this test here, and refer to it as a boolean function *validlc*(*p*, *m*) (where *p* is an *LCset* and *m* is a matchset) which returns true if *p* satisfies the condition to be eligible as a valid *LCset* for *m*.

Having defined these functions, we now present the routine for precomputation in Figure 6. We now look at an example with cost precomputation. The grammar is a transformed version of that in Example 2.1

Example 3.1. $G = (\{V, B, G\}, \{a(2), b(0)\}, P, V)$

```

P:
S → V      [0]
V → V B a  [0]
V → G V a  [1]
V → G      [1]
G → B      [1]
V → b      [7]
B → b      [4]

```

The automaton is shown in Figure 7.

Algorithm *Preprocess*

Input A cost-augmented context-free grammar $G = (N, T, P, S)$ constructed from a regular tree grammar in normal form.

Output The auxiliary automaton represented by tables δ_A, δ_{LC} , which represent transitions on elements of A and $2^{N \times C}$ respectively, where C is the set of possible costs.

```

begin
   $lcsets := \phi$ ;
   $matchsets := \phi$ ;
   $list := closure(\{[S \rightarrow \cdot \alpha, 0] \mid S \rightarrow \alpha \in P\})$ ;
  while  $list$  is not empty do
    delete next element  $q$  from  $list$  and add it to  $lcsets$ ;
    for each  $a \in A$  such that  $goto(q, a)$  is not empty do
       $m := goto(q, a)$ ;
       $\delta_A(q, a) := (match(m), S_m)$ ;
      if  $m$  is not in  $matchsets$  then
         $matchsets := matchsets \cup \{m\}$ ;
        for each state  $r$  in  $lcsets$  do
          if  $validlc(r, m)$  then
             $p := ClosureReduction(r, m)$ ;
             $\delta_{LC}(r, S_m) := (match(p), p)$ ;
            if  $p$  is not in  $list$  or  $lcsets$  then
              append  $p$  to  $list$ 
            end if
          end if
        end for
      end if
    end for
  end if
  for each state  $t$  in  $matchsets$  do
    if  $validlc(q, t)$  then
       $s := ClosureReduction(q, t)$ ;
       $\delta_{LC}(q, S_t) := (match(s), s)$ ;
      if  $s$  is not in  $list$  or  $lcsets$  then
        append  $s$  to  $list$ ;
      end if
    end if
  end for
end while
end

```

Fig. 6. Algorithm to construct the auxiliary automaton.

Let us look at a typical step in the preprocessing algorithm. Let the starting state be q_0 , i.e., the first *LCset*.

$$q_0 = \{[V \rightarrow \cdot V \$, 0], [V \rightarrow \cdot V B a, 0], [V \rightarrow \cdot G V a, 0], [V \rightarrow \cdot G, 0], [G \rightarrow \cdot B, 0], \\ [V \rightarrow \cdot b, 0], [B \rightarrow \cdot b, 0]\}$$

Using the definition of the *goto* operation, we can compute the matchset q_1 as,

$$q_1 = goto(q_0, b) = \{[V \rightarrow b \cdot, 3], [B \rightarrow b \cdot, 0]\}$$

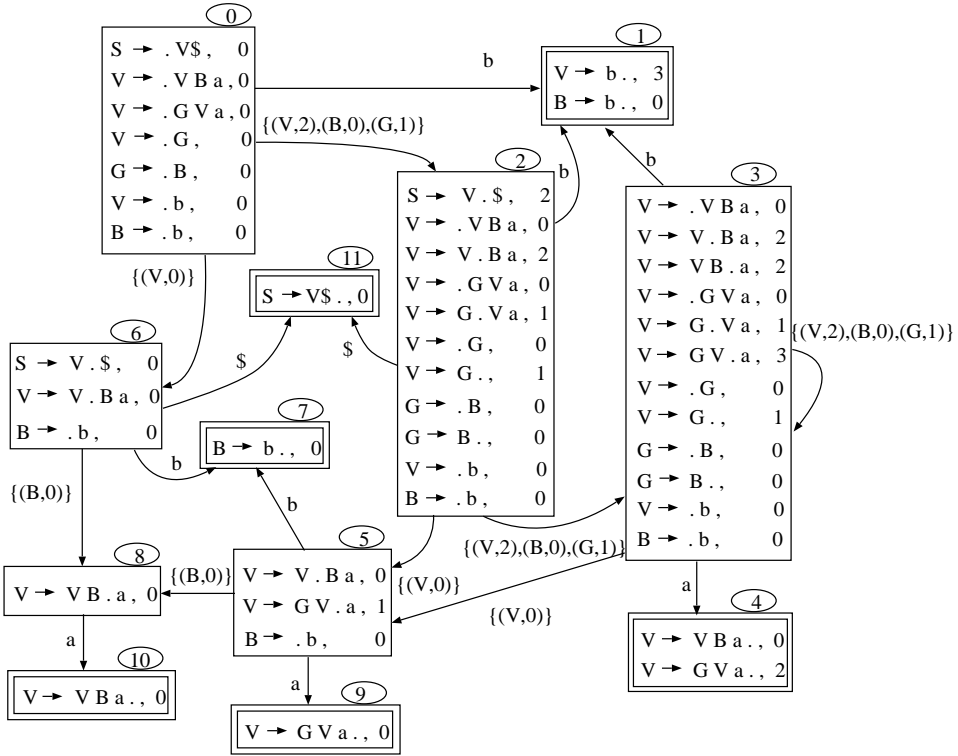


Fig. 7. Example with costs.

In the matchset, the set of matching nonterminals S_{q_1} is

$$S_{q_1} = \{V, B\}, \text{ with costs 3 and 0 respectively.}$$

Now, we can compute the set $ClosureReduction(q_0, q_1)$. First we will compute $reduction(q_0, q_1)$.

Initialization :

$$S = S_{q_1} = \{V, B\}$$

$$cost(V) = 3$$

$$cost(B) = 0$$

$$cost(G) = \infty$$

$$temp = \{[G \rightarrow B., 1]\}$$

Processing chain rules:

Iteration 1:

$$S = S \cup \{G\} = \{V, B, G\}$$

$$cost(V) = 3$$

$$cost(B) = 0$$

$$cost(G) = 1$$

$temp = \{[V \rightarrow G., 2]\}$
 Iteration 2:
 $\mathcal{S} = \mathcal{S} \cup \{V\} = \{V, B, G\}$
 $cost(V) = 2$
 $cost(B) = 0$
 $cost(G) = 1$
 $temp = \phi$

Computing reduction:

$reduction = \{[S \rightarrow V.\$, 2], [V \rightarrow V.Ba, 2], [V \rightarrow G.Va, 1], [V \rightarrow G., 1],$
 $[G \rightarrow B., 0]\}$

Once we have $reduction(q_0, q_1)$, we can use the *closure* function and compute *ClosureReduction*. Therefore,

$q_2 = ClosureReduction(q_0, q_1)$
 $= closure(reduction(q_0, q_1))$
 $= closure(\{[S \rightarrow V.\$, 2], [V \rightarrow V.Ba, 2], [V \rightarrow G.Va, 1], [V \rightarrow G., 1], [G \rightarrow$
 $B., 0], \})$
 $= \{[S \rightarrow V.\$, 2], [V \rightarrow .VBa, 0], \{[V \rightarrow V.Ba, 2], [V \rightarrow .GVa, 0], [V \rightarrow$
 $G.Va, 1],$
 $[V \rightarrow .G, 0], [V \rightarrow G., 1], [G \rightarrow .B, 0],$
 $[G \rightarrow B., 0], [V \rightarrow .b, 0], [B \rightarrow .b, 0]\}$

3.2 Table Compression

It is possible to compress the tables by defining certain equivalence relations on the set *lcsets*.

Let *arityset* be the set of arities of symbols of *A*. Define a set of equivalence relations $\{R_i \mid i \in \text{arityset}\}$ on the set *lcsets* as follows.

If *p* and *q* are in *lcsets*, then pR_iq if $\delta_A(p, a) = \delta_A(q, a)$ for all *a* with $\text{arity}(a) = i$. The table δ_A now splits into several tables δ_A^i , one for each arity. The rows of the table δ_A^i correspond to the equivalence classes of R_i . The columns correspond to columns of A_i where A_i is the set of symbols of *A* with arity *i*.

Define the set NT_i as

$$NT_i = \{B \mid B \rightarrow \alpha a \in P, \text{arity}(a) = i\}$$

Equivalence relation U_i is defined on *lcsets* as follows. For states *p* and *q*, pU_iq if for all (nonterminal, cost) pairs $\langle B, c \rangle$ in $NT_i \times C$, where *C* is the set of relative costs, $\delta_{LC}(p, \langle B, c \rangle) = \delta_{LC}(q, \langle B, c \rangle)$. The table δ_{LC} now splits into several tables δ_{LC}^i , one for each value of arity. The table δ_{LC}^i has one row for each equivalence class of U_i and a column for each distinct set of matching (nonterminal, cost) pairs that is a subset of $NT_i \times C$.

The compression of the tables can be done on line, while the tables are generated, as the equivalence classes can be computed from the sets of items associated with the states. Each equivalence relation would need an index map which maps from original indices to indices in that equivalence class, which are then used to access table entries. During matching, if the next symbol is in A_i , then table δ_A^i is first

consulted through its index map, followed by a lookup of table δ_{LC}^i through the appropriate index map.

3.3 Implementation

The optimal tree parser is made up of two modules, the preprocessor and the parser. The code generator can be built using both these modules. The input pattern syntax is specified using Yacc. The syntax-directed translation converts the input patterns into a suitable internal representation. Several structures are precomputed prior to preprocessing to speed up algorithm *Preprocess*. For instance, all possible items are precomputed and stored in an array. The set of items in each state is implemented as a bit vector. The data structures are set up so that the *goto* and *closure* operations can be performed efficiently. The states of the auxiliary automaton are stored in a hash table with open hashing. Both the δ_A and the δ_{LC} tables are represented in memory using a two-dimensional array of structures for each value of arity. In addition, there is also an index map table for each value of arity for both dimensions of the δ_A and δ_{LC} tables. The preprocessor and the matcher together require about 5000 lines of C code.

4. A SUFFICIENT CONDITION FOR TERMINATION

In some cases, the input augmented grammar may be such that the algorithm *Preprocess* does not terminate. Thus, there is no finite automaton that can store the precomputed cost information. In such cases, the only alternative is to perform dynamic cost computations.

Recall that the states of the auxiliary automaton consist of sets of $(LR(0)item, cost)$ pairs. Thus the number of states is unbounded if and only if the cost component is unbounded. This will happen whenever there is a pair of nonterminals, such that both elements of the pair match in the same left context, and derive the same sequence of infinite trees with increasing cost differences. This is illustrated in the example below.

Example 4.1.

$G = (\{S, A, B, C\}, \{d(2), b(0), a(0), e(0), f(0), c(0)\}, P, S)$,

P :

$S \rightarrow A S d$	[0]
$S \rightarrow B S d$	[0]
$A \rightarrow C A d$	[0]
$B \rightarrow C B d$	[5]
$A \rightarrow b$	[1]
$A \rightarrow e$	[1]
$B \rightarrow b$	[1]
$B \rightarrow f$	[1]
$C \rightarrow a$	[1]
$S \rightarrow c$	[1]

The nonterminals A and B match an infinite sequence of input trees in the same left context, with increasing relative costs (which increase in multiples of 5). As a result, Algorithm *Preprocess* will not terminate on this input instance.

We now give a sufficient condition for termination of the algorithm. Let us call a rule a *base* rule if it is not a chain rule. The condition is *sufficient* but not *necessary*, as it disqualifies a grammar if there is a pair of nonterminals (both occurring on the right-hand sides of base rules), that match in the same left context state, and generate the same sequence of infinite trees, without checking the condition on costs. We first motivate the development of the sufficiency condition.

Let m be a matchset. For each valid left context state p for m , one can define an enhancement of m by associating with the set of complete items in m , a set of nonterminals representing S_m , *enhanced* with the set of all nonterminals obtained by using all chain rules applicable in p . Call this set S'_m . S'_m consists of nonterminals that match in the same left context. We prune from this set all those nonterminals that are used only in chain rules, and have no further use in any base rule, to give a set \mathcal{C}_m . Included in \mathcal{C}_m then is each nonterminal A in S'_m , such that there is some item I in p with A after the dot, I corresponding to a nonchain rule. Thus, we can associate with each matchset a collection of enhanced matchsets. Each enhanced matchset is a set of complete items, together with a set of nonterminals, obtained as described above. Let \mathcal{M} be the collection of enhanced matchsets associated with the auxiliary automaton without costs. We also define an *aligned nonterminal set* below:

Definition 4.1. Let m be a matchset with complete items $[A_1 \rightarrow X_{11}X_{12} \cdots X_{1n} \text{op.}]$, $[A_2 \rightarrow X_{21}X_{22} \cdots X_{2n} \text{op.}]$, \dots , $[A_k \rightarrow X_{k1}X_{k2} \cdots X_{kn} \text{op.}]$.

The sets $\{X_{11}, X_{21}, \dots, X_{k1}\}$, $\{X_{12}, X_{22}, \dots, X_{k2}\}$, \dots , $\{X_{1n}, X_{2n}, \dots, X_{kn}\}$ are all termed aligned nonterminal sets for matchset m .

Aligned nonterminal sets are sets of nonterminals that match in the same left context state.

We next construct a directed graph $\mathcal{G} = (\mathcal{M}, \mathcal{E})$ from the automaton without costs [Shankar et al. 2000]. \mathcal{M} , the vertex set, consists of all enhanced matchsets. There is an edge from node m_1 to node m_2 if and only if some aligned nonterminal set in m_2 is equal to \mathcal{C}_{m_1} .

The graph constructed above encapsulates certain properties of *labeled* trees defined below. Consider a subject tree t , and a derivation tree for it. The derivation tree induces a labeling of the nodes of the subject tree as follows. Each node is labeled with the set of nonterminals that match at that node. (In general, there will be a *set* because of the existence of chain rules.) Suppose now that we superimpose all derivation trees for this subject tree and take the union of all the sets labeling each node. Also, for a set labeling a node, we remove the nonterminals that are used only in chain rules and retain only those that are used in base rules at the next level up in the tree. Let us call this tree with a set of nonterminals labeling each node a *labeled* tree for t .

The following relationship exists between the graph \mathcal{G} and the set of *labeled* trees for the grammar:

- (1) Each node of a labeled derivation tree corresponds to an enhanced matchset.
- (2) There is a node with label set \mathcal{N}_1 having its parent with label set \mathcal{N}_1 in some labeled tree for the grammar if and only if there is a loop at an enhanced

matchset with associated set $\mathcal{C}_1 = \mathcal{N}_1$ and with an aligned nonterminal set \mathcal{N}_1 in the graph \mathcal{G} .

- (3) There is a node with label set \mathcal{N}_1 having an ancestor with label set \mathcal{N}_1 in some labeled tree for the grammar if and only if there is a cycle containing an enhanced matchset with associated set $\mathcal{C}_1 = \mathcal{N}_1$.

A moments reflection will convince the reader that the above statements are true. Statement 3 implies that a tree rooted at a node labeled by more than one nonterminal can nest itself, which is equivalent to saying that there is an infinite sequence of trees matching a set of (more than one) nonterminals in the same left context, each nonterminal in the set being part of a base rule. Thus if the graph has no cycles including nodes with associated nonterminal set of cardinality greater than one (call these *offending* nodes), then we do not have infinite sets of trees matching in the same left context. Even if the graph has a cycle with a node with associated nonterminal set of cardinality greater than one, if for each pair of nonterminals in the set, one of them derives the other by a sequence of chain rules, then the cost differences are bounded and the algorithm is bound to terminate. Thus the only case when we are unable to guarantee that the algorithm will terminate is the case when there is a cycle with an offending node, with at least one pair of associated nonterminals, which are not derivable from one another by chain rules. We have given an informal argument in support of the following theorem:

THEOREM 4.1. *Algorithm Preprocess will terminate either if the graph \mathcal{G} constructed above has no cycles that include a node with nonterminal set of cardinality greater than one, or if it has a cycle that includes a node with nonterminal set of cardinality greater than one, each pair of nonterminals in the set is such that one of them can be derived from the other by a sequence of chain rules.*

Figure 8 shows the graph for the auxiliary automaton of Figure 7. Though the graph has three elementary cycles, none of them contains a node with nonterminal set of cardinality greater than one. Hence the algorithm *Preprocess* will terminate.

That the condition is only sufficient and not necessary can be verified by modifying the grammar of Example 4.1 so that the costs of the recursive productions for the nonterminals A and B are the same. The corresponding graph \mathcal{G} has a cycle with an offending node that does not satisfy the chain rule condition, but algorithm *Preprocess* terminates.

Some questions that naturally arise at this point are: How strong is this condition? Will it disqualify most practical grammars? The termination test was run on the following grammars: The input grammars for the X86, MIPS R3000, and SPARC architectures from Fraser and Hanson [1995]. All the grammars passed the test. However, the grammar for the M68020 used in reporting sizes in Shankar et al. [2000] and Balachandran et al. [1990] failed the test (even though static cost computations terminate on the grammar). A small fragment of the grammar that causes nontermination is displayed below:

Example 4.2.

Nonterminals: {dreglw,dregb,constant_lw,const_bw,constant_b}
 Terminals(with arity in parentheses): {const_b(0),const_w(0),not(1)}

- (1) The size of the finite-state tree-pattern-matching automaton is exponential in that of the auxiliary automaton (see Shankar et al. [2000] for an example.)
- (2) The algorithm that constructs the finite-state tree-pattern-matching automaton with static costs does not terminate, but algorithm *Preprocess* does. An example is given below.

Example 5.1.

$G = (\{S, L1, L2, A, B, C\}, \{d(2), c(1), e(0), f(0), b(0), g1(0), g2(0)\}, P, S)$

P:

S	->	L1 A d	[0]
S	->	L2 B d	[0]
A	->	A c	[0]
B	->	B c	[5]
A	->	e	[1]
B	->	f	[1]
A	->	b	[1]
B	->	b	[1]
L1	->	g1	[1]
L2	->	g2	[1]

Since the nonterminals *A* and *B* do not match in the same left context, it does not matter that these nonterminals match the same infinite sequence of trees with unbounded cost difference. The table construction for this example using our algorithm terminates. However, the computation using any bottom-up algorithm that constructs a bottom-up finite-state tree-pattern-matching automaton will not terminate.

The example above illustrates that there are some input instances of tree grammars that cannot be handled by conventional techniques but on which *Preprocess* will successfully terminate. That there are no instances on which the conventional algorithm will terminate but *Preprocess* will not is borne out by the following argument. The cost of an item is always the sum of a finite number of quantities. These quantities are differential costs of nonterminals that match in the same left context. Thus the cost of an item is unbounded if and only if the differential costs for some pair of nonterminals that match in the same left context is unbounded. Since the sets of nonterminals that match in the same left context at a given node of the subject tree are always subsets of those reported by the conventional algorithm, an unbounded differential cost arises only if there is an unbounded differential cost in the case of the conventional algorithm. Hence the claim is true.

From the practical standpoint, we believe that the advantage of this algorithm is its ability to use attributes during parsing to store entities like register names, literal values, and so on. Thus local register allocation can be done while selecting code. Example 5.2 illustrates a fragment of the input specification for the M68020 that was used for generating code with dynamic cost computation [Gantait 1996].

Example 5.2.

```
STAT    ->    ( EA1_B EA_B logop_b ) [2]
{ emitlogop (#3);
```

```

    emitffaddr ($2->s); emit (" , ");
    emitffaddr ($1->s); emit ("\n");
    freeeffaddr ($2->s); freeeffaddr ($1->s); }

```

/ Here #3 refers to the lexical value of the generic token `logop_b` which identifies the actual logical operation; `emitffaddr($ 1->s)` emits the effective address based on the field `s` of the attribute of the second nonterminal on the right-hand side of the production; `freeeffaddr` frees the register that holds the effective address.*/*

```

AREG ->CONSTANT_LW [8]
{ reg = $0->s.n = getareg (); $0->s.r = 'a';
  emit ("\tmov &%d, %%a%d\n", $1->s.offset, reg); }

```

*/*Here the first action gets a register and assigns its number to the “register” attribute of the left-hand-side nonterminal; the second action assigns an attribute that specifies the “type” of the register; the third action emits the code.*/*

One could also consider a scheme that uses top left contexts [Neumann and Seidl 1998] even while performing bottom-up parsing. This is achieved by using a linearized prefix representation of the subject tree as input to the preprocessor. The algorithm *Preprocess* has to be modified slightly to work for a prefix representation, but the general strategy remains the same, with matching time still being linear in the size of the subject tree. Parsing the prefix linearization of the input subject tree has the advantage that the ancestors as well as the left siblings of the current node have been seen already, and this may help in one-pass code generation in the spirit of Proebsting and Whaley [1996]. We illustrate with the example from Proebsting and Whaley [1996]. The grammar is displayed in Example 5.3.

Example 5.3.

```
G = ({stmt, addr, reg, con}, {ADD(2), ASGN(2), CONST(0)}, P, stmt)
```

P:

```

stmt -> ASGN addr reg
addr -> ADD reg con
addr -> reg
reg -> ADD reg con
reg -> con
con -> CONST

```

The auxiliary automaton for the grammar is displayed in Figure 9. Note that transitions on nonterminal sets have some nonterminals in parentheses. The nonterminals in parentheses are used only in chain rules and therefore do not need to be carried along in the annotation for the node during the labeling phase. If each transition on nonterminal sets of the auxiliary automaton has only one unparenthesized nonterminal, parsing can be done in one pass by knowing only the sets of nonterminals that match at the children. No further analysis on the auxiliary automaton is necessary to determine this. For the auxiliary automaton in the example it can be verified that parsing is possible in one pass.

Using a prefix linearization increases the number of states, as more information (i.e., the top context, in addition to the left context) is available for disambiguation.

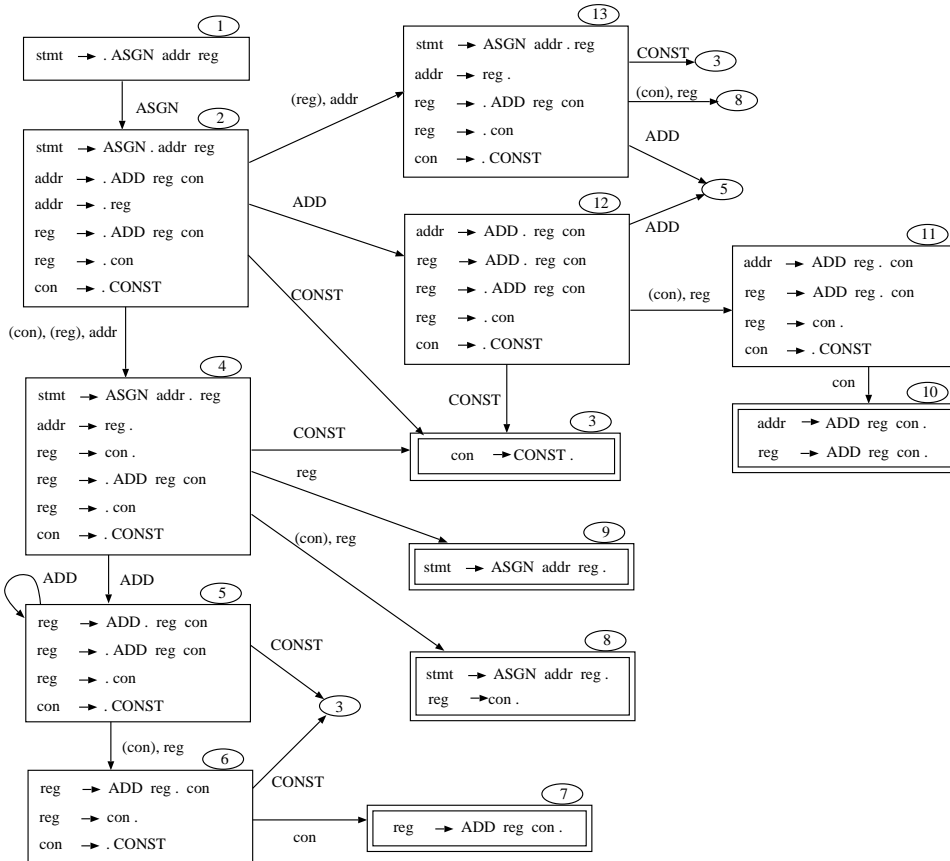


Fig. 9. The auxiliary automaton for the grammar of Example 5.3.

In Figures 10 and 11, we present tables showing the comparisons between the automata generated for postfix and prefix linearization. In addition to the size of the auxiliary automaton, we also indicate how many of the transitions of the automaton can be resolved (i.e., an unambiguous choice is possible at that point in the parse) with top left context. The figures above are for the auxiliary automaton without costs.

It is interesting to note that one-pass code generation using the auxiliary automaton does not reduce to the Graham-Glanville technique without conflicts. For instance, the grammar of the previous example has conflicts and, if used as input to a Graham-Glanville code generator, would require the use of heuristics to resolve those conflicts. However, in our case, optimal code can be generated in one pass.

Interestingly, it turns out that the sufficient condition for termination defined in the previous section is similar to the condition for parsing a grammar in one pass storing only a bounded amount of information (not necessarily corresponding only to the children). Intuitively, only a bounded amount of information need be stored, whenever parsing decisions can be unambiguously made, looking at a subtree of the

	Preorder	Postorder
matchsets	370	85
lcsets	549	103
transitions	3210	2828
resolvable transitions	2828	

Fig. 10. Comparison between auxiliary automata for parsing preorder and postorder linearizations, for the X86 instruction set.

	Preorder	Postorder
matchsets	732	144
lcsets	1070	348
transitions	4422	8346
resolvable transitions	2157	

Fig. 11. Comparison between auxiliary automata for parsing preorder and postorder linearizations, for the M68020 instruction set.

parse tree, whose size is bounded. This is guaranteed if there are no subtrees whose roots are labeled with more than one nonterminal (the nonterminal corresponding to a base rule) that nest themselves.

A question that naturally arises is related to the size of the auxiliary automaton in the worst case. The theorem below gives a loose upper bound. Q is the number of states of the auxiliary automaton. h is the maximum height of subject trees that have to be considered by the conventional algorithm in order to obtain all match sets.

THEOREM 5.1. $|Q| \leq 2^{|\langle NT, c \rangle| \times (\maxarity - 1) \times h}$, where $|\langle NT, c \rangle|$ is the size of the set of (nonterminal, cost) pairs.

PROOF. Consider a node in any derivation tree for a subject tree of height bounded by h . The number of maximal complete subtrees to the left of the node represents the length of a path in the auxiliary automaton, encoding a set of viable prefixes induced by the prefix of the input postorder string ending at the postorder predecessor of the terminal or operator associated with the node. The edges of this path are labeled by sets consisting of matching (nonterminal, cost) pairs. The maximum length of any such path is $(\maxarity - 1) \times h$. Since each edge on the path may be labeled by a set of matching (nonterminals, cost) pairs, the a bound on the total number of states of the auxiliary automaton is $2^{|\langle NT, c \rangle| \times (\maxarity - 1) \times h}$. \square

It is not clear at this point as to what kinds of input grammars will elicit worst-case behavior from our algorithm. We have run, both, the algorithm proposed here, and the conventional algorithm that builds a finite-state tree-pattern-matching automaton with static cost computations using Chase compression techniques [Balachandran et al. 1990], implemented in Kumar [1992] on some input instances that represent instruction sets for two machines. The first is the machine description for

	Intel X86		IBM R6000	
Productions	213		60	
Nonterminals	47		13	
Terminals & operators	107		30	
	With costs	Without costs	With costs	Without costs
Number of states of AA (matchsets + LCsets)	206+399	177+128	65 + 77	65 + 47
Total table size (after compression)	10459	5692	762	688
Size of index maps	2394	768	462	282
Total preprocessing time (seconds)	68.6	18.35	1.0	0.7

Fig. 12. Table sizes and execution times using our algorithm.

	Intel X86		IBM R6000	
	With costs	Without costs	With costs	Without costs
Total table size including index maps	33720	31044	2747	2747
Total preprocessing time (seconds)	4.0	3.6	0.22	0.22

Fig. 13. Table sizes and execution times for the conventional algorithm.

the Intel X86 from Fraser and Hanson [1995]. The second is the machine description for the IBM R6000. The results are displayed in Figures 12 and 13. The code was run on a Sun UltraSparc 60.

Our algorithm generates smaller tables for these instances, but requires more preprocessing time. More experimentation is necessary to study the space/time trade-offs between static and dynamic cost computations. Also, it is worth examining algorithms that perform labeling and code generation in one pass whenever possible, storing a bounded amount of information. We are currently examining such strategies.

6. CONCLUSION

We have given an algorithm that can be used for optimal code selection with static cost precomputation. The algorithm effectively solves the problem of optimal regular tree-pattern-matching by constructing a pushdown automaton that performs

matching in linear time, and precomputes costs. The automaton functions like an LR parser extended to work for ambiguous context-free grammars of a restricted kind. Though our matcher operates like an LR parser, it can, unlike the Graham-Glanville technique, defer parsing decisions to points arbitrarily far beyond the places where the parsing conflicts actually occur. Thus the advantages of tree-pattern-matching methods are preserved. Since all the machinery of LR parsers is available, attributes can be freely used during code generation to hold entities like literal values, register names, and so forth. The algorithm can be tailored to work for one pass code generation(as in Proebsting and Whaley [1996]) with virtually no change to the preprocessing algorithm. Tables generated by this method are seen to be smaller than those generated by the conventional algorithm for some example cases tested by us. Future work will concentrate on applying the tool with static cost computation for code generation, and on one-pass code generation strategies.

ACKNOWLEDGMENTS

The authors are very grateful to the referees for perceptive comments and useful suggestions, which greatly helped in improving the presentation of this paper.

REFERENCES

- AHO, A. AND GANAPATHI, M. 1985. Efficient tree pattern matching: An aid to code generation. In *Proc. 12th ACM Symp. on Principles of Programming Languages*. 334–340.
- BALACHANDRAN, A., DHAMDHERE, D., AND BISWAS, S. 1990. Efficient retargetable code generation using bottom up tree pattern matching. *Computer Languages* 3, 15, 127–140.
- CHASE, D. 1987. An improvement to bottom up tree pattern matching. In *Proc. of the 14th ACM Symp. on Principles of Programming Languages*. 168–177.
- CHRISTOPHER, T., HATCHER, P., AND KUKUK, R. 1984. Using dynamic programming to generate optimised code in a Graham-Glanville style code generator. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*. 25–36.
- FERDINAND, C., SEIDL, H., AND WILHELM, R. 1994. Tree automata for code selection. *Acta Informatica* 31, 741–760.
- FRASER, C. AND HANSON, D. 1995. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, Redwood City, California.
- FRASER, C., HANSON, D., AND PROEBSTING, T. 1992. Engineering a simple, efficient code-generator generator. *ACM Letters on Prog. Lang. and Systems* 1, 3, 213–226.
- GANAPATHI, M. AND FISCHER, C. 1985. Affix grammar driven code generation. *ACM Trans. Program. Lang. Syst.* 7, 4, 560–599.
- GANTAIT, A. 1996. Design of a bottom-up tree pattern matching algorithm and its application to code generation. M.E.Project Report, Dept of Computer Science and Automation, Indian Institute of Science, Bangalore, India.
- GRAHAM., S. AND GLANVILLE, R. 1978. A new approach to compiler code generation. In *Proc. of the 5th ACM Symp. on Principles of Programming Languages*. 231–240.
- HATCHER, P. AND CHRISTOPHER, T. 1986. High-quality code generation via bottom-up tree pattern matching. In *Proc. of the 13th ACM Symp. on Principles of Programming Languages*. 119–130.
- HENRY, R. AND DAMRON, P. 1989. Performance of table driven generators using tree pattern matching. Tech. Rep. 89-02-02, Computer Science Dept, Univ of Washington.
- HOFFMAN, C. AND O'DONNELL, M. 1982. Pattern matching in trees. *Journal of the ACM* 29, 1, 68–95.
- HOPCROFT, J. AND ULLMAN, J. 1979. *An Introduction to Automata Theory, Languages and Computation*. Addison Wesley.
- ACM Transactions on Programming Languages and Systems, Vol. 22, No. 6, November 2000.

- JOHNSON, S. 1975. Yacc - yet another compiler compiler. Tech. Rep. TR 32, AT & T, Bell Laboratories, New Jersey, USA.
- KUMAR, R. 1992. Retargetable code generation using bottom-up tree pattern matching. M.E. Project Report, Dept of Computer Science and Automation, Indian Institute of Science, Bangalore, India.
- NEUMANN, A. AND SEIDL, H. 1998. Locating matches of tree patterns in forests. In *Proceedings of 18th FSTTCS, LNCS 1530*. 134–145.
- NYMEYER, A. AND KATOEN, J. 1997. Code generation based on formal BURS theory and heuristic search. *Acta Informatica* 34, 8, 597–636.
- PELEGRI-LLOPART, E. AND GRAHAM., S. 1988. Optimal code generation for expression trees. In *Proc. of the 15th ACM Symp. on Principles of Programming Languages*. 119–129.
- PROEBSTING, T. 1995. BURS automata generation. *ACM Trans. Program. Lang. Syst.* 17, 3, 461–486.
- PROEBSTING, T. AND WHALEY, B. 1996. One-pass optimal tree parsing- with or without trees. In *International Conference on Compiler Construction*. 294–308.
- SHANKAR, P., GANTAIT, A., YUVARAJ, A., AND MADHAVAN, M. 2000. A new algorithm for linear regular tree pattern matching. *Theor. Comput. Sci.* 242, 125–142.
- WEISBERGER, B. AND WILLHELM, R. 1988. Two tree pattern matchers for code selection. In *Compiler compilers and high speed compilation, LNCS 371*. 215–229.

Received Oct 1998, Revised Feb 2000, Accepted August 2000.