One-Pass, Optimal Tree Parsing — *With Or Without Trees*

Todd A. Proebsting Benjamin R. Whaley

TR 95–10

Abstract

This paper describes the theory behind and implementation of `wburg`, a code-generator generator that accepts tree grammars as input and produces a code generator that emits an optimal parse of an IR tree in just a single bottom-up pass. Furthermore, `wburg` eliminates the need for an explicit IR tree altogether. The generated parser emits optimal code, and can do so without retaining an entire IR tree during its single pass. The grammars that `wburg`-generated parsers can parse are a proper subset of those that the two-pass systems can handle. However, analysis indicates that `wburg` can optimally handle grammars for most major instruction sets, including the SPARC, the MIPS R3000, and the x86.

September 22, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Compilers often use intermediate representation (IR) trees to represent expressions. A compiler's front end generates an IR tree, and the back end walks the tree, emitting appropriate assembly language instructions and operands.

Several automatically generated code generators perform pattern matching on IR trees to emit optimal code. The code-generator generator consumes a cost-augmented tree grammar with associated semantic actions. The resulting code generator requires two passes over the IR tree to determine a least-cost parse and to execute the associated semantic actions. Code-generator generators based on this model include `BEG` [ESL89], `twig` [AGT89], `burg` [FHP92b, Pro92], `iburg` [FHP92a], and `lburg` [FH95].

This paper describes the theory behind and implementation of `wburg`, a code-generator generator that accepts tree grammars as input and produces a code generator that emits an optimal parse of an IR tree in just a single bottom-up pass. Furthermore, `wburg` eliminates the need for an explicit IR tree altogether. The generated parser emits optimal code, and can do so without retaining an entire IR tree during its single pass. The grammars that `wburg`-generated parsers can parse are a proper subset of those that the two-pass systems can handle. However, analysis indicates that `wburg` can optimally handle grammars for most major instruction sets, including the SPARC, the MIPS R3000, and the x86.

Our system has the advantage of not requiring a dynamically allocated IR tree, but sometimes suffers from its small amount of additional bookkeeping. Preliminary experiments indicate that `wburg`'s one-pass parsers run over 30% faster than two-pass `burg`-generated parsers that use `malloc` and `free`. Even when the overhead of dynamic allocation is completely factored out, the one-pass parsers ranged from over 50% *faster* to 3% slower than `burg`'s parsers.

2 Related Work

Many code-generator generators use cost-augmented instruction patterns to produce code generators that optimally parse IR trees. The resulting code generator uses tree pattern matching and dynamic programming to find least-cost instruction sequences for realizing the given IR tree's computations. The various code-generator generators differ in the pattern-matching technology they use, and in when they perform dynamic programming. Pattern-matching techniques vary widely in theoretical efficiency [HO82]. Previous code generators have employed many of these pattern-matching technologies, ranging from the slowest, naive [ESL89, FHP92a, FH95], to top-down [AGT89], to the fastest, bottom-up [FHP92b, Pro92, BDB90, PLG88]. Most systems perform dynamic programming at compile-time [ESL89, AGT89, FHP92a, FH95]; those based on BURS technology do all dynamic programming at compile-compile time [FHP92b, Pro92, BDB90, PLG88].

Previous tree-pattern matching systems required two passes over the IR tree: one for labeling the tree with dynamic programming information, and another for selecting the least-cost parse based on that information. To enable two tree walks, this design requires allocating and building an explicit IR tree. In contrast, `wburg`'s parsers can find an optimal parse in a single pass. Surprisingly, an explicit IR tree is not needed at all. The procedure invocations necessary to build a tree in a bottom-up fashion form a trace of the tree's structure — much like a recursive-descent parser traces out a parse tree — which is all that is necessary for our system to produce optimal code.

The Oberon compiler's code generator works in a single pass without the benefit of an explicit IR tree [WG92]. In order to generate complex addressing modes, the code generator retains a constant amount of information to guide subsequent instruction selection decisions. `wburg`'s parsers must

```
stmt: ASGN(addr, reg)     = 1 (1);
addr: ADD(reg, con)       = 2 (0);
addr: reg                 = 3 (0);
reg:  ADD(reg, con)       = 4 (1);
reg:  con                 = 5 (1);
con:  CONST               = 6 (0);
```

Figure 1: A `burg`-Style Grammar, G

also retain a constant amount of information for subsequent decisions. Oberon's code generator, however, does not generate optimal code for all expressions, and is not automatically generated.

3 `burg` Automata

`burg` is a code-generator generator that accepts a cost-augmented tree grammar as input and automatically produces a tree-parser that gives an optimal parse of an IR tree using a two-pass algorithm [FHP92b]. `burg` does all dynamic programming at compile-compile time and creates a state machine for guiding all subsequent pattern matching and instruction selection decisions. Rules in a `burg` grammar have the form

$$nonterminal: \; rule = rule\_number \; (cost);$$

Consider the `burg` grammar, G, in Figure 1. The fourth rule specifies that the result of adding an immediate value to a value held in a register can be placed in a register at the cost of one. The second rule in Figure 1 corresponds to the "register-offset" addressing mode supported by many architectures, which is free. A `burg` grammar describes the operations and addressing modes of a particular architecture. Rules in a `burg` grammar have one of two forms. A *chain rule* has only a single nonterminal on its right-hand side, while a *base rule* always has a terminal on its right-hand side. In Figure 1, rules 3 and 5 are the only chain rules.

We restrict our analysis to grammars in *normal form* [BDB90]. A grammar is in normal form if all patterns are either chain rules, or base rules of the form $n_0 \rightarrow op(n_1, \ldots, n_k)$ where n_i are all nonterminals and *op* is an operator. Normal-form grammars are no less expressive than other tree grammars, and this restriction greatly simplifies our discussion.

Parsers produced by `burg` use a two-pass algorithm to give an optimal (least-cost) parse of a tree. The first, bottom-up pass over the tree *labels* each node with a state. A state encodes, for each nonterminal in the grammar, the rule to apply at the current node to derive that nonterminal at least cost. The second, top-down pass finds the least-cost parse based on those states. This second pass *reduces* the tree by applying the appropriate rules at nodes in the tree.

Figure 2 shows the set of `burg`-generated states for grammar G (Figure 1). State 1 encodes the fact that an `ADD` node labeled with state 1 is reduced to an `addr` nonterminal using rule 2 and to a `reg` using rule 4. State 3 represents the label for all `CONST` nodes. `CONST` nodes are reduced directly to a `con` nonterminal via rule 6, but require the application of chain rules for `addr` and `reg` nonterminal reductions. Note that reducing a `CONST` node to an `addr` would require two chain rule applications (rules 3 and 5) followed by the base rule 6. All reductions at a state follow this pattern of zero or more chain rules followed by exactly one base rule. (Bottom-up reductions would reverse that order.) The application of a base rule at a node with children will cause those children

State 1 (op = ADD)	*Nonterminal*	*Rule*	#
	stmt	(none)	
	addr	addr : ADD(reg, con)	= 2
	reg	reg : ADD(reg, con)	= 4
	con	(none)	

State 2 (op = ASGN)	*Nonterminal*	*Rule*	#
	stmt	stmt: ASGN(addr, reg)	= 1
	addr	(none)	
	reg	(none)	
	con	(none)	

State 3 (op = CONST)	*Nonterminal*	*Rule*	#
	stmt	(none)	
	addr	addr: reg	= 3
	reg	reg: con	= 5
	con	con: CONST	= 6

Figure 2: The `burg` States for G

to be reduced to the nonterminals on the right-hand side of the base rule.

4 One-Pass Tree Parsers

A two-pass system defers all reduction decisions until labeling is complete. Because every node in the tree contains a label, an unbounded amount of information must be retained prior to starting the reduction process. If reductions could be done during the bottom-up labeling pass, any subtree that is reduced could be discarded since reduction is the last phase of pattern matching. The labeler would only retain unreduced subtrees until they could be reduced.

(Note: Reducers normally apply rules in a top-down order, but the semantic actions associated with those rules typically are applied in bottom-up order — the order in which code is emitted. From this point forward, we will assume that reducers reduce trees bottom-up.)

For some states, optimal reductions can be made during the bottom-up labeling pass. Because rule 1 is the only rule in state 2, it must be used to reduce any node labeled with state 2. Therefore, the labeling pass can *immediately* apply rule 1 to any node it labels with state 2. `wburg` creates an array, *reduce_now[ ]*, that is indexed by state numbers. Whenever a state contains only one base rule, `wburg` places the appropriate nonterminal (the nonterminal appearing on the left-hand side of that base rule) into the table.

Reducing a node via a base rule will cause all descendents of that node to be reduced, and thus no longer needed. Although not true for state 2, applying a base rule does *not* always completely reduce the given node. Subsequent reductions of ancestor nodes may require the application of chain rules at this node in order to derive a nonterminal other than the left-hand side of the base rule. Therefore, the node itself must be retained for the possible chain-rule applications later.

Now consider state 1, which contains two base rules, 2 and 4. The labeler cannot immediately determine the appropriate reduction when it reaches such a node. Therefore, `wburg` must mark

state	1	2	3
nonterminal	$\perp$	`stmt`	`con`

Figure 3: Array *reduce_now* for Grammar G

left_nt — Parent's State

		1	2	3
Left	1	`reg`	`addr`	$\perp$
Child's	2	$\perp$	$\perp$	$\perp$
State	3	`con`	`con`	$\perp$

right_nt — Parent's State

		1	2	3
Right	1	$\perp$	`reg`	$\perp$
Child's	2	$\perp$	$\perp$	$\perp$
State	3	`con`	`con`	$\perp$

Figure 4: Arrays *left_nt* and *right_nt* for Grammar G

state 2's entry in *reduce_now[ ]* as unknown ($\perp$). Figure 3 gives *reduce_now[ ]* for grammar G. Formally,

$$\textit{reduce\_now}[S] \equiv N \quad \text{iff state S contains only one base rule, } N \rightarrow \theta(\ldots).$$

Although state 1 contains two base rules, and a decision about which to apply depends on reductions at ancestor nodes, the state does contain enough information to guide the reductions of its children. Both base rules have the same vector of nonterminals — `(reg, con)` — on the right-hand side. Therefore, any node labeled with state 1 can immediately reduce its left child to `reg` and its right child to `con`. This fully reduces its children. The labeler must defer reductions of the state 1 node, however.

`wburg` could generate vectors like *reduce_now[ ]* to map states to the nonterminals to which their children should be reduced. Instead, we generalize this notion and create tables that guide child reductions based on the states of both the child and the parent. Two two-dimensional arrays, *left_nt[ ][ ]* and *right_nt[ ][ ]*, hold the nonterminals to which a node's children should be reduced. *left_nt[B][A]* holds the nonterminal to which a node labeled with B should be reduced if it is the left child of a node labeled with A.

An entry is put in *left_nt[B][A]* if every left-most nonterminal on the right-hand side of every base rule in A derives from the same base rule in B via zero or more chain rules. Under these conditions, the nonterminal on the left-hand side of B's base rule, N, is put in the table. This condition ensures that `wburg` knows which base rule in B ultimately will be applied when B is the left child of A. *right_nt* is similar. Formally,

$$\begin{array}{rl} \textit{left\_nt}[B][A] \equiv N & \text{iff } X \rightarrow \theta(Y, -) \in A \text{ implies } N \rightarrow \phi(\ldots) \in B \text{ and } Y \stackrel{*}{\Rightarrow} N \text{ in } B. \\ \textit{right\_nt}[B][A] \equiv N & \text{iff } X \rightarrow \theta(-, Y) \in A \text{ implies } N \rightarrow \phi(\ldots) \in B \text{ and } Y \stackrel{*}{\Rightarrow} N \text{ in } B. \end{array}$$

Figure 4 gives the tables for G. A "$\perp$" indicates that no immediate reduction is possible.

4.1 Characteristic `burg` Graph

For each cost-augmented tree grammar, `burg` generates a function `burm_state` that computes the state of a particular node given its operator and the states of its children.

$$\texttt{burm\_state} : \textit{operator} \times \textit{state}_{\textit{left}} \times \textit{state}_{\textit{right}} \rightarrow \textit{state}$$

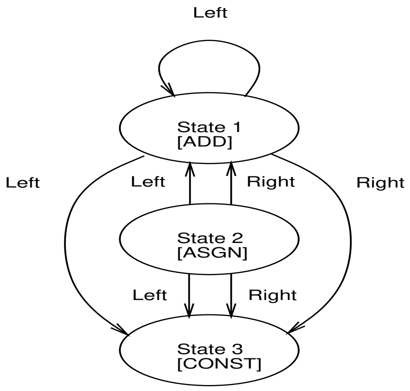

Figure 5: Transition Graph for `burg` States of Grammar G

Using `burm_state`, we can generate a *characteristic* directed graph to represent states and state transitions. Nodes represent states, and edges represent possible state transitions. Edges are annotated with either *left* or *right*, depending on whether the target node represents the left or right child of the source node. Formally,

$$A \overset{\textit{left}}{\longrightarrow} B \quad \text{iff } \exists\theta, x \text{ such that } \texttt{burm\_state}(\theta, B, x) \equiv A$$
$$A \overset{\textit{right}}{\longrightarrow} B \quad \text{iff } \exists\theta, x \text{ such that } \texttt{burm\_state}(\theta, x, B) \equiv A$$

Figure 5 shows the characteristic graph for G. This characteristic graph generates all trees described by G. In general, the characteristic graph can generate a superset of the trees that a grammar describes because not all combinations of *left* and *right*-labeled arcs leaving a particular state can actually be combined to generate that state. Only cyclic graphs generate trees of unbounded size.

A bound on the sizes of the trees generated by an acyclic graph is simply the size of the largest tree in the graph. Therefore, this bound is the maximum amount of information that must be retained between passes of a two-pass parser.

4.2 Arc Pruning

Analysis of the characteristic graph leads to a method of doing one-pass parsing that retains only a constant amount of information about the reduction of any labeled node and the subtree it roots. `wburg` will prune an arc from the graph if the arc corresponds to a state transition for which a correct base-rule reduction of the child node is known. The remaining arcs represent the reductions that the analysis cannot immediately determine and therefore must postpone until the parser can safely reduce an ancestor. If the pruned graph is acyclic, then only a bounded number of nodes must ever be retained for subsequent reduction. Because the original graph (Figure 5) is cyclic, one-pass parsing with grammar G is not possible without arc pruning.

Arcs leaving state 2 may be pruned because the parser can immediately reduce the left and right children of a state 2 node. After the parser reduces the children of a state 2 node, the parser would retain those nodes, but not their children (the node's grandchildren). The children may require subsequent chain-rule reductions, but the grandchildren will be completely reduced at this point. (The condition for pruning arcs only guarantees that one of the child's base rules can be applied immediately — application of the child's chain rules may be deferred, and so it is necessary to record the last nonterminal to which the child was reduced.)

The entries in *left_nt* and *right_nt* correspond directly to the pruned arcs. An arc $A \overset{left}{\rightarrow} B$ may be pruned if *left_nt[B][A]* $\neq \perp$ because the labeler can immediately reduce B to the given nonterminal, which forces the reduction by its base rule.

If, after applying the arc pruning rule, the resulting graph contains no cycles, then the grammar can be parsed optimally during the single, bottom-up labeling pass while retaining only a constant number of unreduced nodes at any given point. A conservative upper bound on the number of unreduced nodes is the size of the largest tree generated by the pruned graph *plus* information about deferred chain rule reductions of the nodes just off the leaves of that tree.

If the pruning rule removes all directed edges from the graph (as it will for grammar G), then the largest subtree whose reduction may have to be deferred consists of a tree node and its children. Its children will lack, at most, chain-rule applications.

5 A Code-Generator Generator

5.1 One-Pass, Optimal Parsing of IR Trees

If the application of the pruning rule creates an arc-free graph, then labeling and reducing the tree can be done in a single-pass while deferring at most three nodes worth of reductions at the most recently labeled node: the reductions at that node, and possible chain rule reductions at its children. All descendents further down the tree must be completely reduced.

`wburg` is a system for generating optimal, single-pass tree-parsers for grammars for which the characteristic graph can be completely pruned of arcs. Note that this is the same class of grammars for which `wburg` can fill all entries in *left_nt* and *right_nt* that correspond to valid state transitions.

Because the reductions induced by `wburg`'s *reduce_now*, *left_nt*, and *right_nt* tables are identical to the reductions that `burg`'s parser would apply, both systems find identical least-cost parses.

Consider the procedure *Compose* in Figure 6, which creates a new node from an operator and two child nodes. *Compose* labels a node and attaches that node to its children. Then, it initializes the node's nonterminal field to zero, indicating that the current node has not yet been reduced. The procedure checks the array *reduce_now* to see whether the node itself can be reduced immediately. If so, *Compose* will immediately reduce the entire subtree rooted at this node (except, of course, previously reduced nodes).

If *reduce_now* cannot help, the parser consults *left_nt* and *right_nt* to obtain the appropriate nonterminals to which the children should be reduced and performs those reductions. In this case, *Compose* does no reductions at *node* itself, but it does invoke reductions of its children. The node's reductions and the children's chain rule reductions must wait until this node's parent is labeled and causes subsequent reductions.

Figure 7 gives the code for *reduce*, which performs reductions. Nodes previously reduced to the desired nonterminal will terminate the reduction. *arity(rule)* gives the number of nonterminals on the right-hand side of *rule*, which represents the number of subsequent reductions this rule must invoke. For chain rules, *kids[ ]* simply holds *node*; for base rules, it is a vector of the node's children. *nts[ ][ ]* is a table of the nonterminals to which the *kids[ ]* should be reduced based on the rule number.

5.2 One-Pass, Optimal Parsing *Without* Trees

Because `burg` and other tree-parsing algorithms require two-passes over the IR tree, the IR tree must be retained until the second pass is complete. Because a tree may be of arbitrary size, the tree nodes must be dynamically allocated and deallocated. Since `wburg`'s parsers parse an IR tree

```
procedure Compose(node_ptr node, int op, node_ptr left_child, node_ptr right_child)
    node.left = left_child                                              // Assemble tree.
    node.right = right_child
    node.state = burm_state(op, left_child.state, right_child.state)    // Label tree.
    node.nt = 0                                                         // Node is not reduced.
    if (reduce_now[node.state] ≠ ⊥) then
        reduce(node, reduce_now[node.state])                            // Reduce immediately.
    else
        left_nt = left_nt[left_child.state][node.state]
        right_nt = right_nt[right_child.state][node.state]
        reduce(left_child, left_nt)                                     // Reduce children's base rules.
        reduce(right_child, right_nt)
    end if
end procedure
```

Figure 6: *Compose* Procedure

```
procedure reduce(node_ptr node, int nt)
    if nt = node.nt then return                                         // Previously reduced.
    rule = burm_rule(node.state, nt)                                    // Find rule to apply.
    for i ← 1 to arity(rule) do
        reduce(kids[i], nts[rule][i])                                   // Invoke subsequent reductions.
    end for
    // Invoke semantic action for rule here.
    node.nt = nt                                                        // Record reduction.
end procedure
```

Figure 7: Procedure *reduce*

in a single bottom-up pass it is possible for the parser to eliminate the explicit creation of the IR tree altogether. Furthermore, since the maximum number of retained nodes is three, the parser can use the run-time stack to store the retained nodes.

Consider the modified *Compose* procedure, *Compose*′, in Figure 8. *Compose*′ invokes *reduce* whenever possible. Nodes retain all necessary information for the deferred reductions of their children. The node retains the summary of its state (*node.self*) and the retained state of its children (*node.retain_left* and *node.retain_right*) — the pruning analysis guarantees that this is sufficient for one-pass parsing. Each node maintains pointers (*left* and *right*) to its children to give the reducer the illusion of an actual tree. *Compose*′ must maintain these pointers, invoke the reducer when possible, and copy the states of the children into the root node when necessary. (Note that the *left* and *right* children of *node.retain_left/right* are never accessed because the only rules they may defer are chain rules.)

If the array *reduce_now* indicates that an immediate reduction of the current node can be made, then pointers to the two children are copied into the node, and the entire subtree rooted at the current node is reduced. If the reduction of the current node must be deferred, the left and right children are reduced, and their root nodes are copied into *node.retain_left* and *node.retain_right* for

```
procedure Compose′(node_ptr node, int op, node_ptr left_child, node_ptr right_child)
    node.self.state = burm_state(op, left_child.self.state, right_child.self.state)
    node.self.nt = 0
    if (reduce_now[node.self.state] ≠ ⊥) then
        node.self.left = left_child.self                     // Create (tiny) tree for
        node.self.right = right_child.self                   //     immediate reduction of this
        reduce(node.self, reduce_now[node.self.state])       //     node (and its children).
    else
        left_nt = left_nt[left_child.self.state][node.self.state]
        right_nt = right_nt[right_child.self.state][node.self.state]
        reduce(left_child.self, left_nt)                     // Reduce children.
        reduce(right_child.self, right_nt)
        node.retain_left = left_child.self                   // Retain children for later
        node.retain_right = right_child.self                 //     chain rule applications.
        node.self.left = addressof node.retain_left          // node represents 3-node tree.
        node.self.right = addressof node.retain_right
    end if
end procedure
```

Figure 8: *Compose′* Procedure

later chain-rule reductions. The node's *left* and *right* pointers are set to point to those fields.

Compose′ constructs nodes and invokes parsing reductions. Each node retains all the information necessary for any deferred reductions at, or below, itself. Therefore, the nodes from which it was built are no longer needed. Bottom-up tree building may require maintaining multiple nodes that each represent the unreduced portion of an entire subtree. Fortunately, such nodes can often be kept on the run-time stack.

Appendix A contains a complete one-pass parsing example.

6 Implementation

`wburg` is an extension of `burg`. `wburg` and `burg`, therefore, have identical specification languages. `wburg` uses the output of `burg` to build and prune the characteristic graph. For suitable grammars, it creates *reduce_now*, *left_nt*, and *right_nt* tables.

6.1 Experimental Results

While the set of grammars that can be parsed optimally using `wburg`'s parsers is a proper subset of the grammars that can be parsed optimally using `burg`'s, analysis indicates that most useful grammars, including those describing the SPARC, the MIPS R3000, and the x86 architectures, fall within this subset.

We compared the speed of `wburg`'s one-pass parsers with `burg`'s two-pass parsers. To the best of our knowledge, `burg` produces faster two-pass parsers than any other parser-generator system. We present speeds for `burg`'s parsers both with and without dynamic memory allocation. For dynamic allocation, we use `malloc` and `free`. We also present `wburg`'s parsers both with explicit trees (using *Compose*) and without trees (using *Compose′*). `wburg`'s parsers with trees do not do

Platform	Grammar	System			
		burg w/ alloc	burg w/o alloc	wburg w/ trees	wburg w/o trees
Alpha	SPARC	19.1	13.5	13.6	13.9
	x86	22.3	17.7	16.6	17.1
SPARC	SPARC	258.6	192.5	125.8	124.9
	x86	321.5	249.9	151.3	159.7

Table 1: Speeds of Parsers (in sec.).

dynamic memory allocation. Tests utilized modified versions of `lcc` grammars for the SPARC and the x86 [FH95].

The tests consisted of labeling and reducing the 1,234 unique trees `lcc` generates when it compiles itself. Each test parsed a set of 1,234 trees (a total of 11,587 nodes) 500 times. The tests were performed on lightly loaded DEC Alpha and Sun SPARCstation workstations. The times in Table 1 were measured using the system clock.

Clearly, `wburg`'s parsers without trees are faster than `burg`'s when `burg`'s trees are allocated via `malloc` and `free`.

When both `burg` and `wburg` parsers operate on trees and do no dynamic memory allocation, `wburg`'s parsers must do bookkeeping to avoid redundant reductions that `burg`'s do not need to do. `wburg`'s parsers avoid a second tree walk, but occasionally must visit a node a second time to do deferred chain-rule reductions. When measured with `pixie` on the Alpha, the number of cycles executed for `wburg`'s parsers was only 1–2% greater than those for `burg`'s. The difference in actual running time ranged between +3% and −7%, and we believe this is due to cache-effects, which are not measured by `pixie`.

On the SPARC, `wburg`'s parsers were over 50% faster than `burg`'s even though they were saddled with additional bookkeeping to avoid redundant reductions. While very pleased by these results, we cannot currently explain them other than to speculate. Possible explanations include reduced register-window spilling from only one recursive tree walk and beneficial cache-effects.

The relative speeds of `burg`'s parsers and `wburg`'s parsers appear to be independent of grammar, which is what we expected. Because of the overhead of copying child states into parent nodes, `wburg`'s parsers without trees are slower than those with trees.

6.2 `lcc` Grammars

The `lcc` grammars are written for the `lburg` code-generator generator, and we modified them for `burg`. `lcc` grammars allow rule costs to be determined at run-time, which `burg` cannot handle, so we had to alter the grammar to use constant costs. We also put `lcc` grammars into normal form. After the changes, all three grammars met the sufficient restrictions for one-pass optimal code generation without further modification. `lcc` puts all trees in a canonical form (e.g., all constant operands appear as the right child of commutative operators), and this greatly simplifies the code generation grammars. Tests demonstrate, however, that `wburg` can handle non-canonical grammars for all possible permutations of complex addressing modes.

7 Conclusion

We have developed the theoretical basis for optimal, one-pass tree pattern matching. Using this theory, we have developed `wburg`, a code-generation system based on optimal tree-pattern matching that has two important advantages over previous systems: the code generator does labeling and reducing in a single parsing pass, and the code generator does not need to build an explicit IR tree. Both advances result in time and space advantages.

References

[AGT89] Alfred V. Aho, Mahedevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG—a generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, New York, 1989. ACM.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, California, 1995.

[FHP92a] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FHP92b] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[PLG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 294–308, New York, 1988. ACM.

[Pro92] Todd A. Proebsting. Simple and efficient BURS table generation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 331–340, New York, June 1992. ACM.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon, the Design of an Operating System and Computer.* Addison Wesley, 1992.

A Example One-Pass Parsing

We will outline the one-pass construction and reduction of the seven-node tree given in Figure 9. This example utilizes the grammar and tables described throughout this paper. `burg` states appear underneath the operators in the tree. `burg`'s left-to-right, bottom-up reduction of the example tree appears in Figure 10. Each rule in the reduction is annotated with the node at which the rule was applied.

Figure 11 gives the necessary calls for building and reducing the example tree using a `wburg`-derived one-pass parser. A–G are the tree nodes being composed, which correspond to the labels in Figure 9. The final call to *reduce* guarantees that the root node is fully reduced to the goal nonterminal, `stmt`.

Figure 12 gives the (nearly identical) calls for reducing the same tree, only *without* actually constructing the tree. Note that the nodes X–Z are reused after they serve as children in a prior composition — this is a benefit of one-pass parsing without trees.

Figure 13 outlines the actions that a one-pass parser takes during the eight function calls given in Figure 11. (The reductions induced by the calls in Figure 12 are identical.) The *Nonterminal* column indicates which of the one-pass parsing tables (*reduce_now*, *left_nt*, or *right_nt*) is determining the reductions.

The reductions at node A demonstrate the subtleties of one-pass parsing. Note that visits to A for possible reductions occur as a consequence of actions at nodes A, C, and E. When visiting A for the first time, *reduce_now* causes a reduction to `con`, which results in the application of rule "`con: CONST`." Then, A's parent node, C, uses *left_nt* to determine that A should be reduced to `con`. Since A was previously reduced to `con`, this visit by *reduce* does not cause another reduction. At this point, the parser has not applied a base rule at C. Finally, E uses *left_nt* to determine that C must be reduced to `reg`, which causes an application of rule "`reg: ADD(reg, con)`" at C. Before making that reduction, A must be revisited, and reduced via chain rule "`reg: con`" because the base rule at C requires A be reduced to a `reg`.

`wburg`-derived parsers do not produce a simple left-to-right reduction order because of the way reductions are deferred. The reductions do, however, maintain a bottom-up ordering.

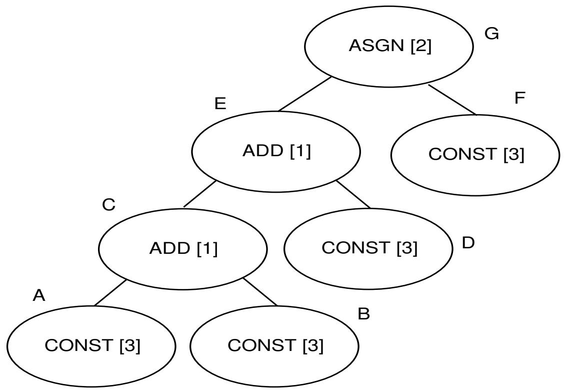

Figure 9: Example IR Tree With State Labels

Reductions	Node
`con:  CONST`	A
`reg:  con`	A
`con:  CONST`	B
`reg:  ADD(reg, con)`	C
`con:  CONST`	D
`addr: ADD(reg, con)`	E
`con:  CONST`	F
`reg:  con`	F
`stmt: ASGN(addr, reg)`	G

Figure 10: `burg`'s left-to-right, bottom-up reduction.

Compose(A, `CONST`*, —, —)*
Compose(B, `CONST`*, —, —)*
Compose(C, `ADD`*, A, B)*
Compose(D, `CONST`*, —, —)*
Compose(E, `ADD`*, C, D)*
Compose(F, `CONST`*, —, —)*
Compose(G, `ASGN`*, E, F)*
reduce(G, `stmt`*)*

Figure 11: One-Pass Parsing With a Tree.

Compose′(X, `CONST`*, —, —)*
Compose′(Y, `CONST`*, —, —)*
Compose′(Z, `ADD`*, X, Y)*
Compose′(X, `CONST`*, —, —)*
Compose′(Y, `ADD`*, Z, X)*
Compose′(X, `CONST`*, —, —)*
Compose′(Z, `ASGN`*, Y, X)*
reduce(Z, `stmt`*)*

Figure 12: Parsing *Without* a Tree.

Statement	Nonterminal		Reductions	Node
Compose(A, `CONST`*, —, —)*	*reduce_now[3]* ≡ `con`	⇒	`con:  CONST`	*A*
Compose(B, `CONST`*, —, —)*	*reduce_now[3]* ≡ `con`	⇒	`con:  CONST`	*B*
Compose(C, `ADD`*, A, B)*	*left_nt[3][1]* ≡ `con`	⇒	(Previously reduced)	*A*
	right_nt[3][1] ≡ `con`	⇒	(Previously reduced)	*B*
Compose(D, `CONST`*, —, —)*	*reduce_now[3]* ≡ `con`	⇒	`con:  CONST`	*D*
Compose(E, `ADD`*, C, D)*	*left_nt[1][1]* ≡ `reg`	⇒	`reg:  con`	*A*
			`reg:  ADD(reg, con)`	*C*
	right_nt[3][1] ≡ `con`	⇒	(Previously reduced)	*D*
Compose(F, `CONST`*, —, —)*	*reduce_now[3]* ≡ `con`	⇒	`con:  CONST`	*F*
Compose(G, `ASGN`*, E, F)*	*reduce_now[2]* ≡ `stmt`	⇒	`addr: ADD(reg, con)`	*E*
			`reg:  con`	*F*
			`stmt: ASGN(addr, reg)`	*G*
reduce(G, `stmt`*)*			(Previously reduced)	*G*

Figure 13: One-Pass Reductions