
Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

1992

Code generation using a backtracking LR parser

Laurie Anne Smith King
College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

King, Laurie Anne Smith, "Code generation using a backtracking LR parser" (1992). *Dissertations, Theses, and Masters Projects*. Paper 1539623820.

<https://dx.doi.org/doi:10.21220/s2-qkrj-qd04>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9304503

Code generation using a backtracking LR parser

King, Laurie Anne Smith, Ph.D.

The College of William and Mary, 1992

Copyright ©1992 by King, Laurie Anne Smith. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

CODE GENERATION USING A BACKTRACKING LR PARSER

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the requirements for the Degree of

Doctor of Philosophy

by

Laurie Anne Smith King

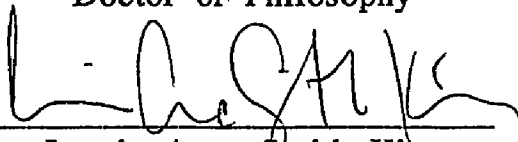
1992

Copyright © 1992 by Laurie Anne Smith King, All Rights Reserved

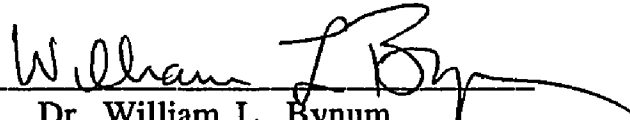
APPROVAL SHEET

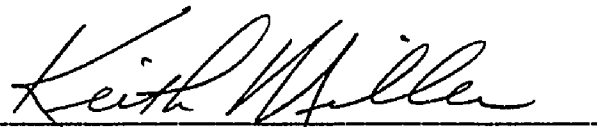
This dissertation is submitted in partial fulfillment of the requirements for the degree of

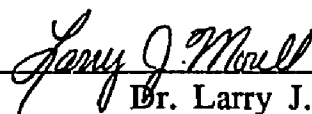
Doctor of Philosophy

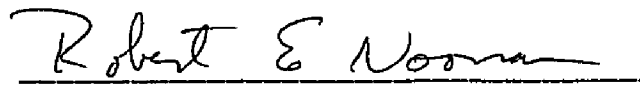

Laurie Anne Smith King


Approved, June 1992


Dr. William L. Bynum


Dr. Keith W. Miller


Dr. Larry J. Morell
Department of Computer Science, Hampton University


Dr. Robert E. Noonan


Dr. J. C. Sanwal
Department of Mathematics, The College of William and Mary

In memory of my grandmothers,

Hope Smith Shackson Focht
and
Gretchen Bergetta Smith Smith

Table of Contents

	Page
Acknowledgements	v
List of Figures	vi
Abstract	viii
Chapter 1 Introduction	2
Chapter 2 Blocking Problems in Parsing-based Code Generators	26
Chapter 3 A Backtracking LR Parser	40
Chapter 4 Backing up over Semantic Actions	62
Chapter 5 Code Generation as Tree Transformations	69
Chapter 6 An Algorithm for Converting Tree Transformations to an Affix Grammar	84
Chapter 7 Reparse Backup	95
Chapter 8 Conclusion	101
Appendix A Peephole Optimization of the Intermediate Code	106
Appendix B Computing Usage Counts	111
Appendix C Instruction Selection	114
Appendix D Selected Test Runs	117
Appendix E Reparse Backup Example Revisited	142
Appendix F TTL and a Backtracking LR Parser Produce a Recognizer for any Recursively Enumerable Language	145
Bibliography	158

ACKNOWLEDGEMENTS

I wish to express my gratitude to the many people who helped me finish (finally) this dissertation.

To Bob Noonan, my advisor, for his unending patience and of course wise guidance. To the rest of my committee, Bill Bynum for his ability to put negative results in a positive light, Keith Miller for all those late night discussions, Larry Morell for his perennial flow of ideas, and JC Sanwal for teaching me all I know about circuit design and analysis. To Bob Collins who gave me the confidence to get started and keep going.

To my friends who stuck by me when I was so very hard to get along with, Tracy Camp for that graduate student comraderie, SueAnne Kyle Johnson for those long distance phone bills, Pat Meszaros for loving my daughter, Bob and Debbie Noonan for listening to my whining and great skiing, Karen and Louis Slothouber for awesome bridge games and great beach trips, and Glenda White for girl's night out.

To my relatives who encouraged me and kept me sane, my sister Kristen Smith, her fiance Todd Barnett, my cousin Charlie Viles and his wife Emily West.

To my grandfather, Clyde T. Smith, who was very generous in his financial support.

To my parents for their love and support, David Wm. Smith who pushed me to be the best and Beverly Smith who thought I was.

To my daughter, Madison Hope Lucas, who kept everything in perspective.

And finally to my husband and best friend, Stef Lucas, a true Renaissance man, without whose help I never would have finished my PhD.

List of Figures

Figure	Page	
1.1	Tree-Rewrite Rule for an Addition Instruction	9
1.2	Twig Specification for an Addition Instruction	10
1.3	Backtracking LR Parsing-based Code Generation	24
2.1	Intermediate Code Tree for $a := b + c$	27
2.2	MD1: A Partial Code Generation Specification	28
2.3	MD1-ambiguous	29
2.4	MD2: Semantic Information Encoded Syntactically	30
2.5	MD1-ambiguous with Semantic Restrictions	30
2.6	MD3 with Semantic Restrictions	36
3.1	Example Grammar MD4	42
3.2	The Parse Table for Grammar MD4	43
3.3	The Forward Parsing Algorithm	45
3.4	GetNextInput	46
3.5	DoShift	46
3.6	DoReduce	46
3.7	Parse Trace using Grammar MD4	48
3.8	DoBacktrack Algorithm	49
3.9	Another Parse Trace using Grammar MD4	53
3.10	Grammar MD5	54
3.11	The Parse Table for Grammar MD5	54
3.12	Parse Trace using Grammar MD5	60
4.1	Grammar MD5 with Semantic Actions	64
4.2	No Semantic Information Changed	65
4.3	Register 1 is Allocated	65
4.4	Register 2 is Allocated	66
4.5	The Plus Operation is Performed	66
4.6	Trying the Shift Option	67
4.7	Register 2 is Reallocated	67
4.8	Block Trying to Apply Increment Instruction	67
5.1	Intermediate Code (IC) trees	71
5.2	A Machine Code Tree	71
5.3	Recognizing an Expression	72
5.4	A Rearrangement Tree Transformation	73
5.5	A Replacement Tree Transformation	73

5.6	A Conditional Tree Transformation	73
5.7	Backtracking LR Parsing-based Code Generation	75
5.8	Two Transformations to Machine Instructions for Addition	81
5.9	A Transformation to Generate a Register Copy	82
5.10	A Transformation to Machine Code for a Register Load	82
6.1	TTL Syntax	84
6.2	The Affix Grammar Rule Syntax for the TTL Rule of Figure 6.1	85
6.3	Tree Construction Procedures	87
6.4	Transformation Construction Algorithm	88
6.5	Simple Reduction Transformation	90
6.6	Transformations with Conditions	91
6.7	Transformations with Actions	92
6.8	Rearrangement Transformations	93
7.1	Reparse Backup Algorithm	97
7.2	IC for Assignment Statement	98
7.3	A Constant Folding Tree Transformation	98
7.4	Parse State before Constant Folding	98
7.5	Parse State after Constant Folding	99
7.6	Applying the Decr Tree Transformation	100
7.7	Parse State after Parsing Decr Rewrite	100
E.1	Parse State after Incorrect Reparse Backup over Constant Folding Transformation	143
E.2	Expression Transformations	143
E.3	Parse State after Reduce by Rule 1	144
E.4	Parse State after Reduce by Rule 2	144
F.1	Language Generators and Recognizers	145
F.2	Turing Machine Simulation	146
F.3	Turing Machine Simulation Using ID	148

ABTRACT

Although the parsing phase of the modern compiler has been automated in a machine independent fashion, the diversity of computer architectures inhibits automating the code generation phase. During code generation, some intermediate representation of a source program is transformed into actual machine instructions. The need for portable compilers has driven research towards the automatic generation of code generators.

This research investigates the use of a backtracking LR parser that treats code generation as a series of tree transformations.

Code Generation Using A Backtracking LR Parser

Chapter 1

Introduction

"A compiler's primary function is to compile, organize the compilation, and go right back to compiling. It compiles basically only those things that require [sic] to be compiled, ignoring things that should not be compiled. The main way a compiler compiles, is to compile the things to be compiled until the compilation is complete."
The definition of a compiler given by a student in an Introductory Computer Science course. (1992)

1. Motivation and Goals

For the past two decades, a considerable amount of research has been devoted towards reducing effort required to construct quality compilers. The rapid pace with which new machine architectures become available, and the desire to port compilers to take advantage of these new architectures has motivated much of the research.

Design and implementation techniques for handcrafting compilers are ill-suited to the complexity of the complex clerical task performed by compilers. Despite the careful attention to the software design principles employed by early compiler writers, even compilers that had been in production for some time often have lingering bugs. Even if better techniques for writing compilers by hand were developed, the speed with which handcrafted compilers could be produced would be insufficient. Formal methods which would automate parts of the compiler creation task are required. Consequently, much research effort has been devoted towards the

creation of *compiler-compilers*, systems that automatically generate a compiler or translator from a specification.

Applying a divide-and-conquer strategy, compilers are divided into a front-end and a back-end. The front-end recognizes and parses a source language, and ultimately produces intermediate code from a source program. The back-end translates the intermediate representation of a source program into object code specific to a given target machine.

Research of the 1960's and 1970's, while ignoring the semantic phase, has for the most part automated the front-end in a machine independent fashion. Advances in the use of grammars and parsing has made it possible for parsers to be automatically generated from a grammar-based description of the programming language. Now a substantial compiler front-end can be implemented in a one semester, undergraduate course using a compiler-compiler.

However, the diversity of computer architectures and complexity of the code generation task has inhibited automating the creation of code generators. For the most part, the back-end has been created manually; although high quality code generators can be created this way, hand generation is tedious and lacks the formalisms required to easily demonstrate the correctness of an implementation. In short, the creation of code generators suffers from the same problems that plagued the creation of compiler front-ends twenty years ago. A goal of recent research is to create code generators from machine descriptions or code generator specifications by developing a *code-generator-generator*.

Using an algorithm to create code generators from machine descriptions or specifications is easier and faster. The algorithm can be implemented and verified once, and thereafter the user's responsibility consists only in providing an adequate machine description/specification. A further research goal is the automatic generation of code generators which produce "good" code, code that not only takes advantage of special use instructions and machine idiosyncracies but also which contains no dead code, i.e., code which has no effect.

2. Background

This chapter reviews research on the automatic derivation of code generators from machine descriptions or code generation specifications. To date, the approach generally taken has been one of divide-and-conquer. Code generation has been conceptually carved into three main tasks: instruction selection, peephole optimization and register allocation/assignment. Instruction selection is the translation of intermediate code trees generated by the compiler front-end into object code, usually either assembler or machine code. Peephole optimization improves object code by correcting inefficient code sequences, selecting special case instructions, and eliminating redundant object code. Register allocation/assignment determines which values will reside in each register. Compiler back-ends are usually implemented by dividing these tasks into subphases.

Until recently, attempts to automate code generation have concentrated on a single task of code generation and so formalisms have been developed for each task separately. Even though the division between tasks is not strict, each can be automatically created as an independent phase that runs sequentially. For example, research which concentrates on instruction selection must consider the register allocation/assignment phase and vice versa because of its impact on optimal instruction selection.

In what follows, instruction selection, peephole optimization, and register allocation/assignment will be discussed in detail by reviewing relevant papers from the recent literature. Complexity is reduced by focusing on an individual phase, which poses sufficient challenges in itself. However, the phases are naturally interdependent and there are advantages to allowing all three phases to operate simultaneously in an integrated code generator. Armed with a better and more formal understanding of each individual phase, recent research has attempted to create integrated code generators from machine descriptions or code generation specifications. Consequently, we will first describe attempts to automate each phase as a relatively independent entity and then discuss the interdependence between the phases and review recent attempts to integrate the three phases.

3. Instruction Selection

Many advances have been made in automating instruction selection. The instruction selector generators described herein consist of machine independent instruction selection algorithms which operate from a machine description or code generation specification.

The research described views instruction selection as a pattern matching operation. The code generation specification is encoded in a table and pattern matching algorithms match intermediate code trees to perform instruction selection. Three implementations of pattern matched instruction selection are presented: LR parsing approaches, an A.I. approach, and a dynamic tree-matching approach.

3.1 Instruction Selection Via Parsing

The first practical automatic generation of a code generator was made by Glanville and Graham [GlGr78], who use an LR parser as the machine independent instruction selection algorithm. The grammar is the code generation specification because it describes how intermediate code trees are converted into machine instructions. Descriptive information about the target machine in the form of a grammar is used to create LR parser tables.

Productions are classified into three kinds [GaFi85]: address mode productions, instruction selection productions and transfer productions. *Address mode* productions map intermediate code addresses into machine addresses. An *instruction selection* production specifies an intermediate code tree pattern on the right hand side, with the left hand side specifying the result, typically a data type. *Transfer* productions allow the code generator to shift an operand's storage location, for example to perform data type conversions.

Grammar rules are ordered to create a table in which the cheapest/best instruction is tried first, described as a *locally greedy heuristic* [Hen84] or as a *maximal munch strategy* [Cat78]. Other approaches [SpTu87] have attempted to perform cost analysis during parsing beyond the simple ordering of the productions. Spector and Turner have implemented a dynamic programming algorithm that extends cost analysis by considering the actual time or space cost of selecting each nonterminal available in each state during parsing.

Ganapathi and Fischer [GaFi85] extended the application of parsing techniques to instruction selection. They describe the instruction set of a target architecture using affix grammars in which attributes containing semantic information influence the construction of the parse tree.

The parsing approach for the automatic derivation of code generators has several advantages. Parsing is fast and provably correct with respect to the grammar, something ad hoc pattern matching algorithms cannot boast. A code generator built with this approach can benefit from any advances in parsing research. Furthermore, the code generator can be incrementally improved by adding new productions to the code generation specification. Another advantage of this approach is that the code generator can be retargeted quickly by modifying the code generation specification grammar.

Unfortunately, several problems arise from parsing using context-free grammars. The machine language must be uniform¹ [GIGr78] for the conflict-resolution rules to recognize the whole language. The code generation specification is usually ambiguous because machines often have several ways of accomplishing the same task; naturally, code generation specification grammars reflect this ambiguity. Secondly, problems of syntactic or semantic blocking result from the locally greedy heuristic which resolves shift-reduce conflicts in favor of the shift, details of which are discussed in Chapter 2. Finally, the Achilles heel of the parsing method is a reliance on the non-trivial creation of hand-crafted grammars.

3.2 Cattell's Approach

Cattell's code generation algorithm [Cat80], operates directly off an ISP-style machine description. Heuristic searching and other A.I. techniques are used in the algorithm which is machine independent but uses machine dependent tables to select instructions.

Instruction selection is template driven; a new set of templates generates code for a different machine. The templates are pattern-matched against intermediate code trees and each template

¹In a uniform language, the operands to an operator are valid independent of context so that if an operand is valid on the left of a binary operator, it is also valid on the right of the operator.

corresponds to a sequence of machine instructions. Instruction selection consists of emitting the machine code associated with a template when a match is found. The instruction selection algorithm is straightforward; the difficulty lies in creating the template table, called the MT. At code generator generation time, Cattell's method automatically selects both the possible templates and the instruction(s) associated with each template and thus creates the MT.

The code generator generator inputs a machine description and outputs the MT. An ISP machine description is manually transformed into a tree production machine description. Instruction function (such as addition for an ADD) is separated from operand addressing details (such as whether operands are registers or memory locations) which reduces the size of the machine description. The procedures SELECT and SEARCH build the template table, (MT), from the tree production machine description. SELECT chooses the special cases, trees, to be included in the MT and SEARCH finds code sequences that represent these trees. SELECT has a double duty because it also finds the best code sequence for each tree.

SELECT ensures that every intermediate code tree can be matched by a template in the MT. First, SELECT creates templates for all subtrees which map directly into machine instructions so that the MT includes all subtrees which can be matched by a single instruction. Second, some additional templates are added to generate more efficient code. Third, templates are included for subtrees of the form $A \leftarrow B$ for every pair of distinct address modes A and B to enable data transfers. Fourth, templates for every tree production operator are included and last, templates are included for control operators (e.g. loops and branches).

SEARCH cannot return all possible code sequences given a goal subtree, so two techniques are used to reduce the size of the search space: means-ends analysis and problem reduction. Means-ends analysis is used to explore nodes which are closer to the goal node first. Problem reduction decomposes a difficult problem into a set of smaller problems. SEARCH returns an instruction if the goal tree matches an instruction assertion exactly, otherwise it applies decomposition axioms and applies itself recursively to each new goal tree. If elements in a set of instructions are semantically close¹ to

¹As a heuristic measure of similarity, Cattell uses the primary operator of two trees.

the goal tree, then transformation axioms are applied recursively so that the transformed goal tree may be used as the goal tree.

The MT is used at code generation time by a pattern matching algorithm which matches templates in the MT against the intermediate code. The templates in the MT are ordered so that those which represent the least expensive instructions will be attempted first. If a template fails to match, the code generation algorithm continues by attempting a more expensive template. The code generator handles mismatches between the IC operands and the template, moving the operands to locations compatible with the template.

As with the parsing approaches, an advantage of Cattell's scheme is that much of the work is done at code generator generation time, rather than within the resulting code generator itself. SELECT and SEARCH operate at code generator generation time, when the MT table is created, but the final code generator just uses the MT. Cattell offers experimental evidence that his heuristic search for code sequences tends to find the optimal code sequences. Parsing approaches have difficulty with semantic blocking and Cattell's approach avoids semantic blocking because his method can "back-track." Also, the size of the machine description and the number of instructions on the target machine have little effect on the speed of the resultant code generator. In comparison, parsing approaches only pattern match in the abstract sense, by wandering through the state tables, so they are inherently faster than Cattell's method, which really does pattern matching.

The translation of an ISP description into the initial tree production machine description is done manually and takes approximately one man-week according to Cattell. A drawback to this method is, SEARCH is not guaranteed to ever find an applicable code sequence for a given subtree, or even to terminate at all. Finally, Cattell acknowledges that more research is required to describe machine data types and other special architectural features like caches.

3.2 A Tree Matching Approach

Another approach [AhGaTj89] to instruction selection views the intermediate code as an actual, versus a conceptual, tree and pattern-matches using tree patterns and tree rewrite rules. Central to this approach are two algorithms: an efficient tree matching

algorithm to recognize the intermediate code and a dynamic programming algorithm to attempt different combinations of tree matches in search of the one which generates the most efficient object code. Both algorithms must be fast in order to compete with LR parsing.

A special language, Twig, was developed for writing code generators. Efficient tree matching with dynamic programming are embedded as part of Twig. The compiler writer specifies the code generator by writing a Twig specification (program). The Twig compiler creates the desired code generator.

Twig code generator specifications consist of a list of tree-rewrite rules. Each rule has the form:

$$\textit{replacement} \leftarrow \textit{template} \{ \textit{cost} \} = \{ \textit{action} \}$$

where *replacement* is a single node, *template* is a tree, *cost* is a code fragment that computes the cost associated with the rule, and *action* is a code fragment. During code generation, templates are pattern matched against the IC. When a template matches, the IC subtree is reduced to the associated replacement node and the action part emits the corresponding machine code. The cost part measures the efficiency of the emitted code. Both the cost and action parts of the rule are code fragments supplied by the compiler writer.

For example, Figure 1.1 depicts a rewrite rule for an addition instruction for a VAX-like target machine. The replacement node and IC tree template appears on the left, followed by the cost and add instruction.

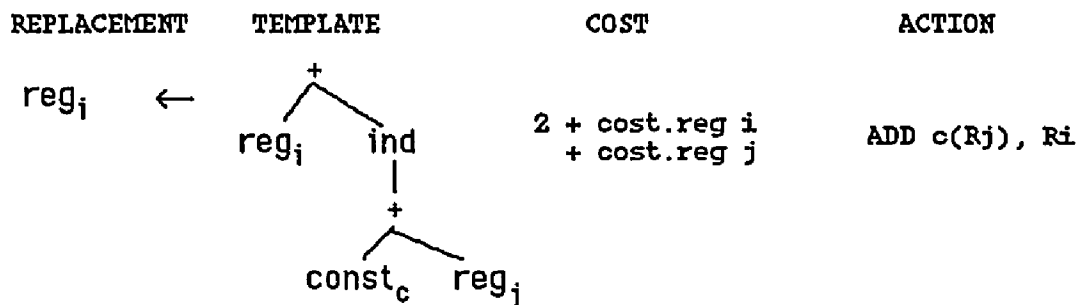


Figure 1.1 Tree-Rewrite Rule for an Addition Instruction.

The Twig specification for the rule in Figure 1.1 appears in Figure 1.2

```
reg:plus(reg, ind(plus(const, reg)))
    {cost = 2 + %1$^cost + %3$^cost;}
    { emit('ADD', %2$, %1$, 0);
      return(%1$);
    };
```

Figure 1.2 Twig Specification for an Addition Instruction.

In the Twig specification, $\%n\%$ denotes the n th labeled leaf, $\$n\%$ denotes the n th child of the root of the template and it is assumed that the emit routine will convert `ind(plus)` into the correct machine addressing mode.

Instruction selection consists of three passes over the IC. The first pass attempts to pattern match all templates in the Twig specification, computing the cost associated with each match. The cost of each template match at a node can be computed independent of the node's parents so that a minimum cost cover can be computed in a single bottom up pass. Furthermore, a vector of costs is computed in which each vector entry corresponds to the cost of a template match for a given number of available registers. Hence register availability plays a role in instruction selection. The second pass performs register allocation based on the cost vector, and the third and final pass traverses the IC performing actions associated with each node to actually emit the chosen machine instructions.

Tree-based pattern matching is less formal than LR parsing and, in the current implementation, the resultant code generators are slower. Also, Twig does not do common subexpression elimination, algebraic simplification, or other high level optimizations. But the authors claim a number of advantages of tree patterns over LR parsing for instruction selection, most notably a simplified code generator specification, freedom to write rules in any order without regard for pattern matching conflicts, and the production of code that is optimal with respect the cost information provided.

4. Peephole Optimization

Peephole optimization improves object code by correcting inefficient code sequences, selecting special-case instructions, and eliminating redundant or dead object code. Inefficient code most often arises as

compilers generate intermediate code or object code in chunks or blocks. Peephole optimization allows the code on the edges of the blocks to be optimized. The classic example is the case of replacing a jump to another jump with a single jump instruction. An example of a special-case instruction is the substitution of an increment for an add by 1. Redundant or dead code is illustrated by an if statement immediately after a while. If the two have opposite conditions, testing the condition in the if statement is redundant and the code can be pruned. Conversely, if the two have the same conditions then the if statement is dead code and should be removed.

Peephole optimizations are necessary because often the optimizations are not apparent until after instruction selection. For example, the compiler cannot anticipate a priori that a jump will be to another jump. Secondly, performing peephole optimizations after the code is created is often more efficient than attempting optimizations during code emission. Case analysis, for instance, can be used to determine whether an add can be replaced with an increment, but only at the cost of complicating and slowing the code emission.

Fraser [Fra79] initially described a peephole optimizer, PO, which performs general optimizations that are both machine-independent and are not part of global optimizations performed before instruction selection. The input to PO, is an assembly language program and a symbolic machine description. Three passes over the input are made. The first is a backward scan that determines the effect of the instructions. Instructions with no effect are removed. The next pass is a forward pass that replaces pairs of adjacent instructions by a single instruction when possible. The final pass replaces each instruction with the cheapest possible equivalent. Besides the obvious drawback of a narrow optimization window, this implementation was slow.

Davidson and Fraser [DaFr79] extended this research by further refining the peephole optimizer, PO. As in the original, PO makes three passes to optimize the program. The first pass, a backward scan, generates the effect of register transfer patterns and finds all dead cells, i.e. variables that are immediately changed without being used or condition codes that are not tested before they are reset. The list of dead cells is kept so that useless effects are ignored. When a branch is encountered, the list of dead cells is cleared because the list of dead cells depends on the instruction's lexical successor and the destination of the branch is unknown. The second pass, a

forward scan, considers the combined effect of two lexically adjacent instructions. When possible, PO replaces each variable instance in the second instruction with variable values from the first. PO can collapse branch chains by treating a branch and its target as a pair. PO also removes any unreferenced labels because any instruction pair with a label on the second instruction cannot be collapsed. The third and final pass translates register transfers back into assembly code.

This research suggested that a naive instruction selection phase can generate good code if used with PO. Sometimes a greater context than a two or three instruction window is needed, for example PO cannot collapse an otherwise reducible pair separated by a third, uncombined instruction. The optimization window is physical and a logical window is desired.

Davidson and Fraser [DaFr84] continued their research resulting in an expanded, modified version of PO. This new PO is inserted into a compiler. The entire compiler is divided into five programs. The FRONT END produces machine code for an abstract machine with a small and regular instruction set. The CODE EXPANDER converts intermediate code to register transfers and also flags cells that are obviously dead. The CACHER eliminates common subexpressions in basic blocks. The CACHER also marks the last use of each cell so that COMBINER can delete instructions which set unused cells. Each instruction is linked to the first instruction that uses one of its results, using a counting algorithm described by Frieburghouse [Fri74]. This allows COMBINER to combine logically adjacent instructions rather than physically adjacent ones, correcting a deficiency of earlier research. The ASSIGNER maps an unlimited number of pseudo registers onto hardware registers, spills¹ when necessary, and translates register transfers into assembly code.

CACHER and COMBINER consider some register allocation details to ease the burden on ASSIGNER. CACHER records and replaces references to common subexpressions with simpler register references. CACHER also calculates use-lists (links from instructions to particular expressions) for dead variable analysis. When an instruction is replaced by a cheaper one, the use-list shows if a value

¹An excess demand for registers causes the contents of one or more registers to be stored in memory (to be reloaded when the value is needed) so that the register demand can be met.

in a register is no longer needed. These use-lists also provide the information to remove redundant loads from memory. After peephole optimization, ASSIGNER performs register assignment and translates the register transfers to assembly code. If a register must be spilled, CACHER's use-lists are used once again to determine the most-distantly-used register in order to minimize the number of spills.

A shortcoming is that some expressions can be recomputed faster than they can be spilled and reloaded. Furthermore, some spills and loads could benefit from further peephole optimization. Davidson and Fraser state that this problem can be corrected by combining CACHER, COMBINER and ASSIGNER (originally developed separately to simplify development and accommodate a small address space). Once phases are combined, the COMBINER can identify common subexpressions too small for CACHER to catch, and to optimize the instructions that ASSIGNER introduces.

5. Register Allocation/Assignment

Register allocation determines which variables are stored in registers. Register assignment determines which specific registers will be used to hold a given variable's value. Using registers effectively is important for the generation of shorter, faster instructions. Retrieving values from registers is much faster than getting values from memory, so a good code generator will retain values in registers as long as possible if the value can be used in several instructions. Retaining values in registers also shortens the code somewhat because specifying a register address is shorter than specifying a memory address. Register allocation/assignment is complicated by machine idiosyncrasies. Special-case instructions often require their operands to be in a specific register, like the accumulator, so assigning a variable to the correct register will allow the use of such instructions without the overhead of a register swap. Still other instructions require operands in an even/odd register pair, which further complicates register allocation/assignment.

Two common problems arise during register allocation/assignment: the value retention problem and the register demand problem. The value retention problem occurs because the code generator tries to hold values in a register to avoid recomputing them, but must also decide when a value is no longer needed. The register demand problem occurs when there is an excess demand for registers and the

code generator must choose which register to spill and reload. Both these problems must be addressed by a good register allocator.

5.1 Register Allocation/Assignment Strategies

There are three general strategies for register allocation/assignment (RAA) which describe when RAA is performed relative to the instruction selection phase: pre-allocation, on-the-fly allocation, and post-allocation.

In a pre-allocation strategy, instruction selection is simplified because the register assignments inherently rule out the use of certain instructions, thus limiting the choices the instruction selector must make. Simplification is a double-edged sword: it also constrains instruction selection. For example, some instructions require their operands to be in an even/odd pair of registers. If the operands are not in the register pair, the instruction selector cannot use the instruction and suboptimal code may result. Furthermore, pre-allocation RAA could over-constrain instruction selection by eliminating all possible instructions, requiring the instruction selector to generate needless moves in order to emit code. Pre-allocation works best when all registers are equivalent (interchangeable within instructions) which minimizes the constraints on instruction selection imposed by the RAA.

The concept of pre-allocation is fundamentally flawed because instruction selection and RAA cannot be cleanly separated. The RAA needs knowledge of the available instructions to assign registers intelligently but the knowledge requires doing instruction selection. On the other hand, intelligent instruction selection requires knowledge of possible register allocations/assignments.

An on-the-fly RAA strategy allows the instruction selector to make choices as it generates code. An advantage over pre-allocation is that the RAA can be chosen to fit the instruction. In the case of an instruction with operands in an even/odd pair of registers, the instruction selector can insure that the operands are assigned to the register pair. When registers are exhausted, a register-spill can be generated or a different instruction chosen. For instance, a memory-to-memory instruction will avoid the spill since no registers are required. The strategy complicates instruction selection by increasing the choices for the instruction selector. Naturally, register

spills are problematic when knowledge of a register's contents will be needed again is required to make the best register spill.

A post-allocation RAA operates with an instruction selector that assumes the existence of an infinite number of registers for use during instruction selection. The RAA's responsibility is the binding of the usually infinite number of psuedo registers to a fixed number of real machine registers. Binding occurs after instruction selection and/or peephole optimization. Because RAA occurs after instruction selection, the register allocator's task is simplified. The chosen instruction determines register assignments, so the allocator makes few decisions. When a register request cannot be satisfied, the allocator has no choice but to generate a spill. The generation of unnecessary spills is a disadvantage of a post-allocation strategy.

5.2 Global Versus Local RAA

RAA can be applied with either global or local strategies. Global register allocations are fixed across one or more basic blocks. Local register allocations are fixed within basic blocks but may change from one block to the next. A basic block is a sequential set of statements which contains no jumps or labels except at the beginning and/or end of the block.

Global RAA is illustrated best within loops, when the most frequently used variables within a loop are allocated to registers. In this way, the most frequently used variables are retrieved from memory only at the start of the loop and stored in memory only at the loop's end rather than at each loop iteration. Furthermore, the same variable might be used frequently in a series of loops. Global RAA would retrieve the variable from memory at the start of the first loop and retain the register allocation across the series of basic blocks. The variable would be stored in memory only at the end of the last loop.

In a local RAA scheme, registers are typically divided into groups for global and local use. Global registers are used as described above, whereas local registers are used for expression/address computations. In a complex expression, for example, intermediate results can be stored in local registers for later use. Such values are only stored in memory if enough local registers are not available. A potential inefficiency of local RAA results because registers are arbitrarily split into global and local groups before register usage is known. Too many registers may be devoted to local use when they

would be better utilized as global registers and vice-versa. The optimal division cannot be determined a priori.

Global register allocation can be represented as a graph coloring problem, whereas local allocation can be done via usage counts. Both graph-coloring and usage counts are discussed below.

5.3 Local RAA via Usage Counts

Freiburghouse [Fre74] devised a simple and efficient method for register allocation implementing a local strategy. Glanville and Graham use this strategy to implement an on-the-fly RAA scheme within their instruction selector. Freiburghouse's algorithm provides an optimal solution to the value retention problem for each linear region. Freiburghouse shows the register demand problem is not solved by this algorithm, but is better than least recently used or least recently loaded strategies in terms of the number of loads generated in 2500 linear region test cases. An optimal solution is not feasible without lookahead.

In Freiburghouse's algorithm, a usage count is used to track the number of distinct references to a given value (variable or computation result) in a program. Algorithms that eliminate redundant computations calculate this. The usage count represents the number of times a value will be needed within the program. Every time a value is used, its usage count is decremented. Therefore a usage count of zero indicates the value will no longer be needed within the program. Usage counts are assigned before register allocation begins.

Registers are allocated to values as needed. When a value's usage count is zero, the associated register can be released since the value is no longer needed. If a register is needed and no spares are available, the register storing the value with the lowest usage count is released. If the register's value has not changed since loading spilling is unnecessary.

5.4 Global RAA via Graph Coloring

Chaitin, et al. [Cha81], discuss register allocation as a graph coloring problem. The idea of applying graph coloring to solve the register allocation problem is not new, but Chaitin describes the first attempt to actually implement this solution. This approach assumes global

register allocation. Furthermore, this implementation uses a pre-allocation RAA (i.e. register allocation is performed before the instruction selection phase).

In their formulation of RAA, colors represent registers and graph nodes represent variables (names). Since variables are never placed in the stack frame pointer, it is reserved and is not represented with a color. A graph-coloring represents a legal register allocation in which each variable is in a different register. In coloring the graph, this method tries to keep as many values in registers as possible.

The first step in this register allocation solution is building the interference graph which describes when two variables cannot be in the same register (i.e. when two graph nodes cannot have the same color). Data-flow analysis is performed to determine which names (nodes in the graph) interfere. Two names interfere if one of them is live at the definition point of the other. To construct the interference graph, K edges are added to the graph when a new node (name) is defined, where K represents the number of live names at that point in the program. The goal is to put the non-interfering names into the same register (i.e. paint these graph nodes with the same color).

Next, RAA is performed by attempting to color the graph with N colors, where N is the number of registers. Nodes with less than N edges can be removed recursively, because such nodes can always be colored. If this reduces to an empty graph then the coloring, and RAA, is trivial. Reduction to an empty graph represents the situation in which more than enough registers are available. If an empty graph does not result, the paper describes an NP-complete algorithm which attempts to color the graph. If the graph cannot be colored, a node is removed from the graph. The removal of a node corresponds to introducing spill code to free a register. The cycle of attempted coloring and removal repeats until the graph is colored.

Chaitin describes a better method [Cha82] to insert spill code and eliminates the use of the NP-complete algorithm in a later paper. The old method, a recursive algorithm, blocks when all nodes have at least N edges. In the new method, the node with the least cost estimate is removed from the graph whenever the recursive algorithm blocks. Spill code for this register must be inserted into the intermediate code. The graph is then rebuilt and a N -coloring is again attempted. To get the least cost estimate the graph is supplemented with a table of cost estimates. This estimate is

derived from the cost to spill, which is the increase in execution time to spill, divided by the degree of the node. The cost to spill is calculated by adding the number of definition points and the number of uses where each is weighted by the estimated execution frequency. Chaitin also uses local knowledge to reload and spill once for a basic block, instead of many times during the block and accounts for allowing computations to be redone to avoid spilling and reloading.

Hennessy and Chow [ChHe84] implemented a post-allocation RAA based on graph coloring. The coloring process is driven according to cost and savings estimates computed for each live range or live range part, as live ranges are sometimes split. Live ranges are similar to Chaitin's definition-use chains. Coloring the graph assigns registers to each live range until all live ranges have registers or until the register supply is exhausted. Cost/Saving estimates prioritize variables so that variables which save the most time when in registers are assigned registers first, increasing the efficiency of the generated code. Not all registers are reserved for allocation by the graph-coloring scheme; some are allocated during a local register allocation phase using usage counts. By performing a local register allocation phase, the number of global registers (N) is reduced and the computation time of the NP-complete algorithm is also reduced.

The local allocation phase determines several parameters used to estimate variable priorities. The parameters are: the cost of moving a value from register to memory and vice versa, the execution time saved by referencing a variable in a register versus a variable in memory, and the execution time saved by each re-definition of a value in a register compared with a store to memory.

Hennessy and Chow's method has several advantages. First, the cost/saving estimates enable an informed choice of which register to spill. Second, their algorithm does not degrade (in time) when an N-coloring is unavailable. Third, the loop structure of a program is taken into account. For example, consider two variables which have similar occurrence frequencies, as determined by the local phase. If one variable is used across contiguous code segments, assigning that variable to the same register (color) across code segments will minimize register loads and stores. The global allocator can recognize this situation and make the more efficient allocation. Finally, it is not always desirable to allocate a register to a variable. A cost/saving estimate in which cost is greater than savings yields such a low

priority that the variable will never be assigned a register, hence this scheme does not fall prey to overallocation. Although these cost/saving estimate formulas are good, calculating the estimates adds extra phases to make the whole algorithm suboptimal [ChHe84].

6. Integrated Approaches To Code Generation

So far instruction selection, peephole optimization, and register allocation have been discussed as independent tasks ordered sequentially as subphases. In reality, the tasks are interdependent and the division into independent phases is undesirable because of *phase ordering problems*, so named because often one phase introduces an inefficiency that can only be corrected by an earlier phase. For example, performing register allocation after instruction selection may introduce an inefficiency when creating register spills. It may be that a spill could be avoided by choosing a different instruction, or it might be cheaper to recompute a value rather than spill and reload. Neither alternative is available after instruction selection is complete. Another example occurs in the way the peephole optimizer and the register allocator are ordered. Performing peephole optimization first can reduce the need for registers by dead-variable analysis and common subexpression elimination. Allocating registers can introduce loads and stores for spilling and reloading values. This code should be further optimized by the peephole optimizer. Requiring the peephole optimizer and the register allocator to run sequentially can introduce inefficiencies no matter which goes first.

The phase ordering problem poses the dilemma that no matter how the phases are ordered, inefficiencies result. The dilemma could be solved if the tasks were integrated into a single phase so that all three tasks operated cooperatively. For example, suppose an instruction requires its operands to be in adjacent registers. If the register allocator operates after instruction selection, it can insure that the operands for this instruction are placed in adjacent registers if possible. However, if the register allocator operates before instruction selection, it could influence the instruction selected. For the instruction selector to pick the best instruction, it must have knowledge about whether an instruction will cause a spill or extra moves to make its decision. Conversely, the register allocator needs information about the instruction selected to best allocate the registers. To minimize spills and extra moves the instruction selector

and register allocator must make a mutual decision, which is not possible in phase-ordered code generation.

Other advantages are gained as a natural result of integrated code generation. An advantage of combining two phases is that one machine description or code generation specification can be used rather than two or three. Furthermore, when tasks are separate phases, each phase often duplicates the efforts of a previous phase thus making compile-time code generation 10-20% slower than if the phases were combined [HeGa86]. Peephole optimization can be performed on intermediate code to simplify instruction selection [TaVSS82]. For example, additions of one can be replaced by increments at the intermediate code level. Case analysis during instruction selection is reduced because the instruction selector no longer considers this optimization. Both integrated and sequential code generation can use this form of peephole optimization. However to take advantage of machine dependencies, some peephole optimization can only be done after instruction selection. Thus performing peephole optimization on intermediate code alone is insufficient.

Because integration of instruction selection, peephole optimization and register allocation appears promising, some recent research has attempted the derivation of code generators in which phases are integrated (i.e. two or more phases are combined). In what follows, two attempts to extend earlier research to offer more integrated approaches to code generation are described. One approach, by Fraser and Wendt, attempts to integrate from the original perspective of peephole optimization. The second approach, by Ganapathi and Fischer, attempts to integrate from the original perspective of parsing-based instruction selection. Each is described in turn.

6.1 Fraser and Wendt

Fraser and Wendt [FrWe86] describe how instruction selection and peephole optimization are performed by HOP, a single, general, rule-based system that matches and replaces register transfer patterns. One set of rules generates naive code, represented as register transfers; another set peephole optimizes these register transfers by combining juxtaposed instructions into a single instruction during generation; still other rules translate the optimized register transfers to assembly code.

Phase combination is accomplished as follows. When an instruction selection rule is executed, a register transfer is created instead of intermediate code so optimization rules can be applied. The register transfers are recycled through the rules until no more optimizations apply.

Combining instruction selection and peephole optimization is also faster because no time is wasted on I/O between phases. (The first phase no longer has to dismantle and output its structure only to have the next phase read and create a similar structure.)

Furthermore, many optimization rules are generated at code generator generation time by PO and loaded into the compile-time optimizer HOP, thus the overall time for peephole optimizations is reduced.

Phase-ordering problems are not as likely to occur when instruction selection and peephole optimization are tightly integrated. The register allocator is part of the instruction selection phase; an infinite number of psuedo-registers are **not** used. The register allocator may naively *generate* more spills than needed, but as optimizations are applied unnecessary spills are removed. The spill code is not written unless it is actually needed (i.e. cannot be optimized away.) A "spill count" is used for each register to delay spill code emission until all optimization is complete to see if the code is actually ever needed. The code produced from this new compiler is not as good as the original compiler with separate phases but the authors attribute this to the fact that common subexpression elimination has not yet been implemented.

6.2 Ganapathi and Fischer

Ganapathi and Fischer's framework for integration works with attributed parsing code generators and builds on their earlier work on instruction selection [GaFi88]. Additional grammar productions (i.e. productions that describe special purpose instructions) are added to the instruction selection productions to perform peephole optimizations. Since attributes maintain contextual information, peephole optimizations on logically adjacent instructions are also possible. The structure is largely machine independent.

A peephole optimizing production is composed of the RHS of the leading logically adjacent instruction, a non-terminal 'V', and the RHS

of the trailing logically adjacent instruction. Code that would have been emitted for the leading and trailing instructions (if they had not been logically adjacent) is buffered as an attribute of the LHS of the peephole production. When the peephole production is recognized, improved code is emitted instead of the original sequences. The code between the logically adjacent instructions is guaranteed to be emitted by the LHS attribute of the peephole production. The productions are ordered so that the peephole optimization productions come first. Adding peephole optimizing productions does not make the code generator any more likely to block. (If it did not block before these productions were added, it will not block now.) Thus if peephole productions can be applied, they will be.

Nested peephole optimizations are allowed; overlapped windows of peephole optimizations are not permitted (when two matches overlap and neither properly contains the other). The authors say that this restriction can be overcome with multiple buffers that emit code to different files that are then merged in correct sequence for final assembly but implementation of this solution may not be practical. Secondly, with a single-pass implementation it is not possible to iteratively apply peephole optimizations to improve the code quality. Here again the authors say that iterative peephole-optimization opportunities do not appear frequent enough to cause serious degradation in code quality. A multi-pass attribute evaluation scheme could solve this problem.

7. Backtracking LR-parsing for Code Generation

The goal of this thesis is to present a formal method of automatically deriving code generators from a code generation specification. We present the concept of a backtracking LR-parser, describe its use for the automatic creation of a code generator using a tree-based notation as the code generation specification (which is translated into an affix grammar), and give implementation details.

The backtracking LR parsing code generator described in subsequent chapters approaches the automatic derivation of code generators from the perspective of pattern matched instruction selection. This research is closely related to, and benefits from, earlier work by Glanville and Graham [GlGr78], Henry [Hen84], and Ganapathi and Fischer [GaFi85]. However, the introduction of backtracking gives our approach some of the flavor of Cattell's approach [Cat80], and our tree-based notation for machine descriptions has the flavor of Aho,

Ganapathi and Tjiang [AhGaTj89]. In this way, our research attempts to combine the best of our predecessors.

As compared with other parser based methods, a backtracking LR parsing code generator shares many of the same strengths. Because our code generator uses attributed grammars, it is more closely related to the affix grammar driven code generator of Ganapathi and Fischer [GaFi85] than the Glanville and Graham [GlGr78] and Henry [Hen84] code generators. Backtracking LR parsing also has the additional advantage of avoiding syntactic and semantic blocking problem via backtracking. In this sense backtracking LR parsing combines the flexibility of Cattell's approach [Cat80] with the formalism of parsing.

In comparison with the dynamic tree pattern matching approach exemplified by Twig [AhGaTj89], backtracking LR parsing code generators can perform common subexpression elimination and other high level optimizations which Twig does not. Two advantages Twig claimed over other parser-based approaches are notational ease and avoidance of semantic blocking. Neither advantage holds over using tree transformations with a backtracking LR parser. As stated previously, the backtracking LR parsing code generator avoids blocking, and we have developed a new tree-based notation for writing code generation specifications that is comparable to Twig in terms of notation.

The effects gained from combining register transfers to perform peephole optimizations (and instruction selection in the case of Fraser and Wendt [FrWe86]) a la Davidson and Fraser [DaFr84] and Fraser and Wendt [FrWe86] can also be performed with a backtracking LR parser. The reparsing allowed by the backtracking would allow trees to be combined in much the same way that registers are combined. Although in this research, instead of combining two register transfers, two trees are collapsed and the result reparsed. In this way a logical window is available as well as the ability to handle overlapped and nested optimizations.

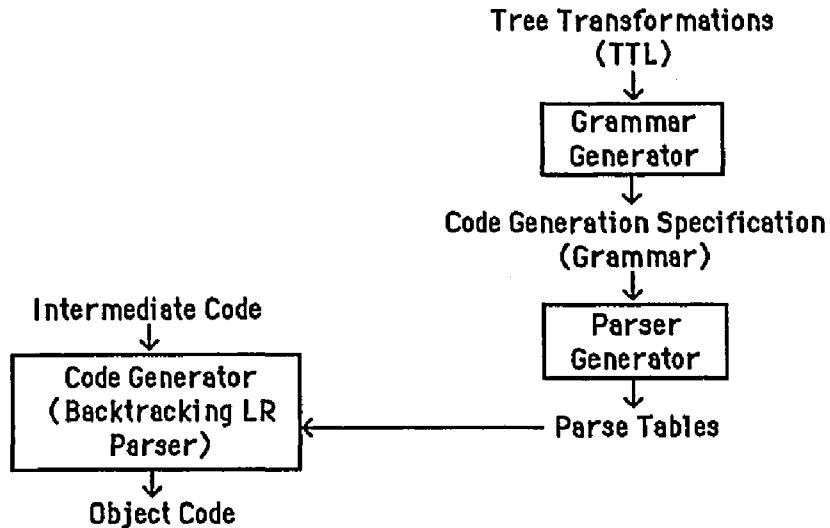


Figure 1.3 Backtracking LR Parsing-based Code Generation

Figure 1.3 depicts the use of a backtracking LR parser for the automatic derivation of a code generator from a TTL specification. There are four items this thesis discusses in detail:

- 1) The code generator produced from the code generation specification, represented by the box on the left of Figure 1.3.
- 2) The modified parse tables which enable backtracking LR parser depicted on the lower right of Figure 1.3, and to a lesser extent the parser generator shown directly above the parse tables in the figure.
- 3) The tree-based notation, written in the tree transformation language (TTL), is used for code generation specifications, labeled Tree Transformations in the upper right of Figure 1.3.
- 4) The grammar generator, shown in the box in the upper right of Figure 1.3, which transforms TTL code generation specifications into grammar-based specifications.

The remainder of this thesis is organized as follows. Chapter 2 reviews LR parser-based approaches to the automatic derivation of code generators with particular attention to the role of semantic actions for code generation and the problems of syntactic and

semantic blocking. The context is established which motivated our investigation of the use of backtracking LR parsers for the automatic derivation of code generators. In Chapter 3, the modifications to the parser generator, to the parse tables, and to the parsing algorithm which enable backtracking are discussed. The problem of backing up in the presence of semantic actions performed during parsing is deferred until Chapter 4. Chapter 4 builds on the foundation provided in chapters 2 and 3, to discuss gathering, saving and restoring semantic information during the parse and backup over semantic actions. In Chapter 5 the concept of a tree transformation is introduced and a tree transformation language (TTL) for writing code generation specifications is described. In Chapter 6 an algorithm for a grammar generator which translates from TTL machine descriptions to an affix grammar representation is presented. This algorithm was not implemented for this thesis, but was executed manually. In Chapter 7 modifications to the code generator to allow for a specialized form of backup are discussed. These modifications were primarily caused by the need to reparse only the changed or new input created by a tree transformation. Chapter 8 summarizes the research and presents conclusions.

Chapter 2

Blocking Problems in Parsing-based Code Generators

"I went over my furniture and looked at each chair in turn, wondering whether the trouble lay there for it upsets me to see even one chair not in its usual place." from White Nights by Dostoyevsky (1848)

In this chapter, we discuss the problems caused by blocking, i.e. when the parse is in a state for which no action can be performed. We outline solutions previous researchers have used to avoid both syntactic and semantic blocking.

A compiler translates a machine-independent source language into a machine-dependent target language. Most compilers consist of a *front-end*, which recognizes the source language, and a *back-end* or code generator which produces the target machine language.

The front-end translates the source language into an equivalent representation using an intermediate code or IC. The code generator translates the IC into an equivalent representation in the target language.

Use of an intermediate representation has several benefits. A carefully designed IC permits the same front-end to be used with different code generators and vice-versa. This allows compilers, for the same language on different machines or for different languages on the same machine, to be implemented more easily and with less

duplication of effort. Second, the IC is a form of separation of concerns that simplifies both the front-end and the code generator. Information specific to the target machine is hidden from the front-end, and information specific to the source language is hidden from the code generator.

Glanville and Graham [GIGr78] identified the code generator's problem of recognizing the IC as a string pattern matching problem which can be solved using LR parsing technology. In their method, the code generator is a modified LR parser which accepts the IC as input, parses it and transforms it into the target machine language. This approach is summarized by Henry as follows:

"In the Graham-Glanville approach to code generation, a code generator is a pattern matcher/replacer built from pattern-replacement pairs. In our applications, the pattern models the computation that the instruction performs, and the replacement models the effect of the computation." [Hen84, p. 3].

The IC is viewed as a language that can be described by a grammar. Target machine instructions are associated with grammar productions to describe how the IC is translated into the target machine instructions. The association of grammar productions and target machine instructions forms the pattern-replacement pairs described by Henry. A Glanville-Graham style code generator is generated from the code generation specification grammar using an LR parser generator.

As the code generator parses the IC, target patterns represented by grammar rules are recognized and associated target machine instructions are emitted. For example, consider the IC tree in Figure 2.1 for an assignment statement.

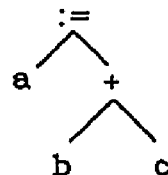


Figure 2.1 Intermediate Code Tree for `a := b + c`

Since parsers pattern match strings and this IC is represented with a tree, the IC must be *linearized* to allow parsing based pattern matching. The IC tree above can be represented in *linearized Polish prefix* form:

`:= a + b c`

In LR parsing-based code generation, the IC is described with a grammar. The following grammar will pattern match an IC of the form shown above:

- (1) `<assign>` → `:= <var> <expr>`
- (2) `<var>` → `<id>`
- (3) `<expr>` → `+ <expr> <expr>`
- (4) `<expr>` → `<var>`
- (5) `<expr>` → `<no>`

Figure 2.2 MD1: A Partial Code Generation Specification

The grammar MD1 in Figure 2.2 partially describes an assignment statement. An actual code generation specification would include many more productions to pattern match the IC. More productions are required to fully describe possible expressions (`<expr>`). When a production matches, an associated action emits machine instructions.

Machines have several possible instructions or sets of instructions to accomplish the same task, therefore there are many different translations possible for a given IC code sequence. All code generation specifications reflect this ambiguity.

An ambiguous grammar results from an inherently ambiguous code generation specification. Therefore there are a number of possible parse paths for a given linearized IC tree. For example, consider the partial code generation specification shown in Figure 2.3.

- (1) <assign> → := <id> + <id> <no>
- (2) <assign> → := <var> <expr>
- (3) <var> → <id>
- (4) <expr> → + <expr> <expr>
- (5) <expr> → <var>
- (6) <expr> → <no>

Figure 2.3 MD1-ambiguous

The first production is an optimizing production designed to generate an increment instruction. It would require both identifiers to be identical and a numeric operand with a value of one.

In Figure 2.3, MD1 is ambiguous. Both the first production:

<assign> → := <id> + <id> <no>

and the production:

<assign> → := <var> <expr>

can derive := a + a 1.

This ambiguity results in a shift-reduce conflict during parsing. Once the input := <id> is on the stack, and the + symbol appears as the next input token, there is a conflict. Reducing <id> to <var> will eventually result in use of the production (2), while shifting the + token will eventually result in use of the production (1). Since shift-reduce conflict resolution occurs before the rest of the input is known, problems can result.

Grammar productions for target machine instructions also require semantic information as well as a syntactic description of the IC. For instance the production that represents an increment, <assign> → := <id> + <id> <no>, cannot be applied on most machines unless the number, <no>, is 1 and the two identifiers in the production represent the same location in memory.

Semantic information may be encoded in the grammar syntactically [GlGr78]. MD1-ambiguous is modified as shown in Figure 2.4.

- (1) <assign> → := <id> + <id> <one>
- (2) <assign> → := <var> <expr>
- (3) <var> → <id>
- (4) <expr> → + <expr> <expr>
- (5) <expr> → <var>
- (6) <expr> → <no>
- (7) <expr> → <one>
- (8) <one> → 1

Figure 2.4 MD2: Semantic Information Encoded Syntactically

Productions (7) and (8) are added and production (1) is modified so only increments by one would be allowed. Furthermore, production (7) is needed to ensure that 1's could be used in other expressions.

An LR parser can also associate one or more *semantic attributes* [Knu68] with each terminal or nonterminal symbol. These semantic attributes may be represented in the form of conditional predicates [CoNo85]. If the semantic predicate evaluates to true then all semantic restrictions have been met. In this case production number one of the grammar MD1-ambiguous would be modified as shown in Figure 2.5.

- (1) <assign> → := <id₁> + <id₂> <no>
 (<id₁>.name = <id₂>.name) AND (<no>.value = 1)

Figure 2.5 MD1-ambiguous with Semantic Restrictions

The notation uses italics to represent semantic restrictions. Subscripts are also used to distinguish between identical nonterminal symbols in the right-hand-side (RHS) of a production. The subscripts are primarily required for semantic predicates. The semantic restrictions must distinguish between the two identifiers, <id>, in the RHS.

The parser checks semantic information during the parse and by doing so, guides the parse and affects parsing decisions. If the parser only shifts semantically legal symbols, there is no need for the parser to check the symbol again during a reduction. Alternatively, if the parser checks semantic conditions on reduce actions, there is no need

to check the symbol when shifting. Thus parsers can check semantic conditions either on shifts or reduces but need not do both.

A parser that checks semantic information at *shift* actions is called a *shift-checking parser*. If we assume we had an original input of " $:= a + a 1$ " and the parser for the grammar MD1-ambiguous was a shift-checking parser, then the parser would "shift $:=$ ", "shift a " then the parser would check that the second $\langle id \rangle$, " a ", matched the first $\langle id \rangle$, " a ", before shifting that second $\langle id \rangle$ onto the parse stack.

A parser which checks semantic conditions at *reduce* actions is called a *reduce-checking parser*. In this case the whole assignment statement " $:= a + a 1$ " would be shifted onto the parse stack before any semantic information is checked.

Regardless of where the semantic information is checked, at shift or reduces, all information is checked sooner (in the case of shift checking-parsers) or later (in the case of reduce-checking parsers). Checking semantics at both shift and reduce actions is redundant. Parsers which encode semantic information syntactically are, by nature, shift-checking parsers. For the remainder of this chapter, we assume a reduce-checking parser unless stated otherwise.

1. Blocking and Code Generation

A code generator is correct if it always generates correct code, i.e. code that produces the expected I/O behavior, for a legal, linearized IC tree. Blocking occurs when the parser reaches a state for which no action is possible for the current input symbol. If blocking occurred only for illegal IC, blocking would not be a major concern. Such *IC-induced* blocks are correct and necessary [AiGrHeMcL184, p. 17]. However, blocking is of particular concern because code generators based on LR parsing can block on legal IC. These *model-induced* blocks are incorrect and unnecessary [AiGrHeMcL184, p. 17]. Consequently, steps must be taken to detect and correct model-induced blocks whenever possible. In what follows, references to blocking imply model-induced blocking unless explicitly stated otherwise.

The potential for blocking arises because code generation specification grammars are usually ambiguous. Blocking occurs because parsing (code generation) decisions are made before all the pertinent information is known. More precisely, shift-reduce

conflicts are resolved in favor the shift. As a result, an incorrect early decision causes blocking later in the parse.

Code generation specifications generally consist of a core set of productions which cover the IC syntactically and parse all legal, linearized IC trees. This basic, code-generating core is further refined by the addition of new productions which take advantage of special-purpose machine instructions. Since the core already covers the IC syntactically, the new optimization productions make the code generation specification ambiguous and introduce shift-reduce conflicts.

Two approaches to detecting and avoiding blocks are described. Both approaches attempt to detect and avoid blocking at code generator generation time. Thus both attempt to anticipate where blocks can occur at parser generation time and generate a parser which avoids the blocks.

2. Syntactic Blocking

Syntactic blocking occurs when no shift or reduce action is possible. Traditionally syntactic blocks are identified by an error action in the parse table. The MD1-ambiguous grammar of Figure 2.3 produces a parser that will syntactically block on the input `:= a + a b`. Blocking occurs because the shift-reduce conflict is resolved in favor of the shift, which will attempt to match the first production

$$\langle \text{assign} \rangle \rightarrow := \langle \text{id} \rangle + \langle \text{id} \rangle \langle \text{no} \rangle.$$

Shift-reduce conflicts are generally resolved in favor of the shift [Hen84] under the assumption that this will result in higher quality code. A syntactic block occurs when the stack contains `:= <id> + <id>`. The parser cannot shift `b` onto the stack as an `<id>` because only a `<no>` is legal. If the parser had chosen the alternative parse path, attempting the production

$$\langle \text{assign} \rangle \rightarrow := \langle \text{var} \rangle \langle \text{expr} \rangle,$$

no syntactic block would occur. This particular syntactic block could be avoided by adding an additional production:

$$\langle \text{assign} \rangle \rightarrow := \langle \text{id} \rangle + \langle \text{id} \rangle \langle \text{id} \rangle$$

To avoid all syntactic blocks, the following production would be required:

$$\langle \text{assign} \rangle \rightarrow := \langle \text{id} \rangle + \langle \text{expr} \rangle \langle \text{expr} \rangle$$

3. Glanville, Graham, and Henry Approach to Code Generation

Glanville, Graham and Henry encode semantic information syntactically [AiGrHeMcL184], thus introducing the potential for syntactic blocking. Syntactic blocks are resolved at code generator generation time (parser generation time) [Hen84] by identifying those states in which a syntactic block can occur. For each blocking state, the code generator generator derives a new grammar production which *covers* the blocking production.

The cover production is constructed from the blocking production by creating a derivation tree for the blocking production using simpler productions. This derivation tree is then transformed into the covering production which has the same left context as the blocking production and a new right context that allows the blocking symbol to shift.

Henry calls the use of cover productions a "specialized form of limited backtracking" [Hen84, p. 22]. The parser generator essentially simulates a backtrack and reparse at parser generation time, and then hardwires the effect of the backup by adding a covering production with default action. Because the limited backtrack is simulated at parser generation time, no backup actually occurs during parsing/code generation.

The default action for the covering production is generated automatically by synthesizing the simpler productions used to derive the covering production. This synthesis simulates the code generator's action to derive the code sequences to be emitted as the default action. The default action is embedded in the parse tables at parser generation time. This method also avoids the need to undo semantic actions when avoiding syntactic blocks. The covering production has the same left-context as the "blocked" production, so any semantic actions completed are still valid.

Construction of the derivation tree is possible because semantic information is represented unambiguously with unique terminal symbols. This unambiguous representation allows the parser generator to decide which simple productions are applicable while constructing the derivation tree.

Grammars which include semantic attributes on nonterminals use a single nonterminal to represent different semantic values. The representation is ambiguous at parser generation time, although semantic attributes have unambiguous values during parsing. Thus the parser generator cannot determine which simple productions are applicable and is unable to construct the derivation tree. And without the derivation tree, a covering production cannot be created.

Thus Glanville, Graham and Henry automatically detect and correct semantically induced syntactic blocks at code generator generation time for grammars which encode semantics syntactically. More precisely, they can correct for the resolution of shift-reduce conflicts in favor of the shift. An important feature of their approach is that the code to be emitted as a default action is derived automatically from the compiler writer's code generation specification.

One disadvantage of using syntax for semantics is the size of the code generation specifications. It has been argued elsewhere [AhGaTj89] that encoding all semantic information syntactically forces the code generation specifications to be large and unwieldy, on the order of 1000 productions for a VAX-11/780. The same machine requires approximately 500 productions [GaFi85] when semantic information is encoded using attributes.

Another disadvantage is that suboptimal code can result. A dynamic programming algorithm using cost information associated with each production usually generates efficient code, but "ambiguities stemming from complicated addressing modes are incorrectly resolved by our algorithms, potentially resulting in inefficient code" [AiGrHeMcL184, p.22].

4. Semantic Blocking

Semantic qualifications are a necessary part of code generation specification grammars. The potential for *semantic blocking* arises when semantic information is encoded using semantic attributes. These semantic qualifications attached to symbols prevent the continuation of parsing.

For example, the grammar MD1-ambiguous semantically blocks on the input $:= a + a 10$. Once again, resolving the shift-reduce conflict in favor of the shift will eventually attempt to match the first production. The attributed grammar production shown in Figure 2.5

fails because ($\langle no \rangle.value \langle \rangle 1$). A similar semantic block occurs on the input $:= a + b 1$ as the semantic restriction ($\langle id_1 \rangle.name = \langle id_2 \rangle.name$) would not be met. Either of these two semantic blocks could be avoided by taking the alternative parse path.

In addition to semantic blocks which originate from shift-reduce conflicts, semantic blocks can also occur if the original code generation specification is incomplete, and all possible reductions are semantically restricted for a legal IC tree.

5. Ganapathi and Fischer's Approach to Code Generation

Like Glanville and Graham, Ganapathi and Fischer [GaFi85] apply parsing techniques to instruction selection by using affix grammars [Wat77] to evaluate semantic attributes to influence the construction of the parse tree. The affix grammar's attributes allow the semantic restrictions of the machine language to be represented adequately, so that more of the target architecture is described by the grammar and less buried in hand-coded semantics. A semantic predicate evaluates the attributes associated with each terminal or nonterminal, the attributes determine whether or not a rule is applied.

Each symbol in the grammar may have a fixed number of synthetic (up the tree) or inherited (down the tree) attributes. Inherited attributes are computed before synthetic ones. Each rule in the grammar has an associated set of attribute evaluation functions. Attributes consist of two types of symbols: predicate symbols, which control parsing, and action symbols, which compute new attribute values. Predicate symbols provide complex information flow, as well as selective rejection of inappropriate productions. For example, a predicate can enforce the restriction that an increment instruction has the accumulator as its operand.

Two restrictions are necessary to make affix grammars suitable for a one pass, left-to-right, attributed bottom-up parse. First, action symbols must appear at the extreme right end of a production. Action symbols can only inherit information from their left siblings. Second, all attributes of nonterminals must be synthetic.

Ganapathi and Fischer cannot take advantage of Glanville, Graham and Henry's approach to resolving semantic blocks because they use

affix grammars and attributed parsing [GaFi85]. Ganapathi and Fischer must resolve both syntactic and semantic blocking.

Syntactic Blocking

Potential syntactic blocks can be "automatically detected when a code generator is first created. If a State S is entered by shifting an operator OP and there exists an action for that state and the next symbol N, then every state entered by shifting OP must have an action for symbol N." [GaFi85, p. 588] The compiler writer can decide whether to add grammar productions by hand to the code generation specification to avoid the syntactic block. Some potential syntactic blocks are ignored if it can be guaranteed that the blocking cannot appear in the IC.

Semantic Blocking

Ganapathi and Fischer use a reduce-checking parser. The compiler writer must identify potential semantic blocks by hand. Semantic blocking is usually corrected by adding a default production. The default production is identical to the blocking production but has no semantic qualifications. The default production will always be applicable and "consequently, it guarantees no blocking" [GaFi85, p. 589] A single default production is added for a set of semantically restricted and syntactically identical productions.

For example the code generation specification MD1-ambiguous of Figure 2.3 has been slightly modified to produce the grammar MD1-GF shown in Figure 2.6.

- (1) <assign> → := <id₁> + <id₂> <no>
 (<id₁>.name = <id₂>.name) AND (<no>.value = 1)
- (2) <assign> → := <id₁> + <id₂> <no>
 (<id₁>.name = <id₂>.name) AND (<no>.value = 2)
- (3) <assign> → := <var> <expr>
- (4) <var> → <id>
- (5) <expr> → + <expr> <expr>
- (6) <expr> → <var>
- (7) <expr> → <no>

Figure 2.6 MD3 with Semantic Restrictions

The following semantically unrestricted production, called a *default rule*, covers the first two productions:

$$\langle \text{assign} \rangle \rightarrow := \langle \text{id} \rangle + \langle \text{id} \rangle \langle \text{no} \rangle$$

The parser will attempt to apply the first two productions. If semantic restrictions disqualify these productions, the default production is guaranteed to apply. This will certainly not handle all potential blocks, but will handle any that match (1) and (2) syntactically.

Ganapathi and Fischer also resolve semantic blocks with *bail out* productions. Bail-out productions are used to resolve semantic blocks resulting from an optimization-greedy code generation strategy which selects shifts over reduces. Bail-out productions usually have semantic restrictions; the burden rests with the compiler writer to ensure that at least one of the productions will be satisfied.

The distinction between bail-out productions and default productions is unclear. Bail-out productions and default productions are identical in function; the only difference seems to be the condition which occasioned the rule's inclusion in the code generation specification. In addition, bail-out productions are used when an attempt at optimization fails due to semantic restrictions. A bail-out production attempts to preserve as much of the optimization as possible. Thus, several bail-out productions are usually added for a single semantically restricted production whereas a single default production is added for a set of semantically restricted productions.

Ganapathi and Fischer require the compiler writer to create the actions for default and bail-out productions manually. As the action parts of most productions generate machine code sequences, they suggest that such sequences could be derived automatically by a technique similar to Glanville, Graham, and Henry. Unlike Glanville, Graham, and Henry the burden is entirely on the compiler writer to ensure that default and bail-out productions are correct and have the proper semantic restrictions.

Limitations and Drawbacks

Ganapathi and Fischer's approach has drawbacks. First, the approach relies heavily on the compiler-writer's skills as most blocking detection and correction is handled manually. In this respect, their

approach moves away from the provably correct formalisms which motivated research into LR parser based code generation in the first place. Careful attention on the part of the grammar writer is needed to ensure all the correct default rules and bail-out productions have been added. Second, the need for default rules and bail-out productions complicates the code generation specification. Usually every semantically restricted production will force the addition of similar, less restricted productions to the code generation specification. This increases the size of code generation specification which in turn makes it less manageable.

Finally, their approach can result in suboptimal code. For example, consider the affix grammar rule:

```

<instr> → := <var1> + <var2> <no>
           same (<var1>, <var2>) and
           (<no>.value <= 32)                               EMIT "incr"

```

which describes an optimization that emits an increment rather than a more costly add instruction. This production can only be applied after <var₂> and <no> have been shifted onto the stack so that the attributes *same (<var₁>, <var₂>)* and *(<no>.value <= 32)* can be evaluated. Semantic blocking will occur if either attribute fails, unless a default rule to emit an "add" is in place, because failure occurs after <var₂> and <no> have been shifted onto the stack. But <var₁> and <var₂> can be arbitrarily complex, for instance:

```

<var> → [ ] <id> <expr>

```

Suppose also that the <expr> parts for <var₁> and <var₂> are the same. Ideally, the code generator should not emit code to evaluate the <expr> twice, for <var₁> and <var₂>, but use the <expr> value just calculated for <var₁> for <var₂>. The default rule must emit suboptimal code for this and similar cases.

A possible solution might be to add more rules for increment to handle the situation as a special case, but for each such special case, a corresponding default rule must also be included to avoid semantic blocking, and the grammar grows nonlinearly quickly becoming unwieldy.

6. A Backtracking Code Generator

Both syntactic and semantic blocks result from making parsing decisions before all the information is known. An incorrect decision results in a block later in the parse when a different decision earlier would have avoided the block. Shift-reduce conflicts are resolved in favor of the shift in an attempt to generate better code. In the example of grammars MD2 and MD3, the code generator attempts to apply special purpose instructions such as an increment. Efficient use of special purpose instructions is desirable, so it is reasonable to continue favoring shifts over reduces. The issue is how to recover when such decisions prove to be incorrect later in the parse.

Since the basic cause that underlies model-induced blocking is that parsing decisions are made before all relevant information is known, a LR parsing code generator with a backup capability could avoid the syntactic and semantic blocking that results from shift-reduce conflict resolution in favor of the shift. Backup solves the problem as follows: when a syntactic or semantic block occurs, simply backup by taking symbols off the parse stack and placing each back into the input. As symbols are placed in the input, check each for earlier shift-reduce conflicts resolved in favor of the shift. At that point an alternative decision could be made. If a parse exists, it will be found by exhausting all the alternatives.

Backtracking is automatic and conceptually simple. The selection of the code sequences to be emitted when recovering from blocks is a natural result of the backtrack and reparse.

Ideally, the code generation specification for a backtracking code generator starts with a core set of productions that describe the machine completely without semantic restrictions. Such a code generation specification would produce suboptimal code. The quality of the generated code is improved by adding productions to the code generation specification to take advantage of special-purpose instructions such as increment. Optimizing productions can be incrementally added without fear of blocking, and without need of adding covering, default or bail-out productions. This is a conceptually clean approach because it makes incremental inclusion of optimizations simple. The next chapter discusses the implementation of a backtracking LR parser.

Chapter 3

A Backtracking LR Parser

"Some men a forward motion love, but I by backward steps would move." from "The Retreat", Silex Scintillans by Henry Vaughan (1650)

A reduce-checking LR parser generator, Pargen [CoNo85], was modified to generate LR parsers with backtracking. Changes in the parser generator necessitated modifying the parsing algorithm. In this chapter, we focus on the mechanisms which enable syntactic backup without regard for any semantic actions performed via reduce actions. The details of gathering semantic information during the parse and backup over semantic actions are discussed in Chapter 4.

The addition of backtracking to an LR parser was motivated by the work of Louis Slothouber [Slo89] at the College of William and Mary, who used an LR parser with limited backtracking for production system interpretation. Backtracking was simplified in Slothouber's work because the parser only backed up over shift actions. Furthermore, there was no input to Slothouber's LR parser in the traditional sense. Working memory was examined to determine which productions to fire. Thus, his research was by necessity specialized to production systems and could not be used directly for our backtracking parser. The research discussed here is not an application of Slothouber's work but some of his ideas have been incorporated.

This chapter discusses an LR parser which can backup over both shift and reduce actions. After the work described in this chapter was completed Wolfgang Keller published an article [Kel91] describing a backtracking parser used in code generation. Although the research was done independently, the resulting code generator is conceptually similar. Keller's depth-first heuristic search parser implementation cannot be compared to our backtracking scheme because it is not described. It is clear that only instruction selection is implemented.

1. Modifications to the Parser Generator

A parser generator [CoNo85] was modified to produce new action tables which encoded the information necessary for backtracking. Traditional LR parse tables [AhSeUl88] contain a single action (either shift, reduce, error, or accept) for every (state, terminal) pair. This traditional organization can be extended to resolve reduce-reduce conflicts [CoNo85]. For each (state, terminal) pair which has a reduce-reduce conflict, a list of reduce actions appears in the table rather than a single action entry. Typically, semantic information is used to choose which reduce action is taken. This extension was used by Glanville and Graham [GlGr78].

The technique used to resolve reduce-reduce conflicts can be extended to a more general scheme which allows multiple shift and/or reduce actions for any (state, terminal) pair. In practice, more than one shift action for a (state, terminal) pair cannot occur. However, table entries which contain a single shift action and one or more reduce actions¹ can be used to encode the existence of shift-reduce conflicts.

As discussed in chapter 2, shift-reduce conflicts are usually resolved by the parser generator in favor of the shift. For a (state, terminal) pair with a shift-reduce conflict, traditional parse tables would only contain the shift action. Our parser generator was modified to produce parse tables which could contain a shift action and one or more reduce actions. A similar scheme was used by Slothouber [Slo89].

¹Keller [Kel91, p.112] allows only one shift and one reduce for any (state, terminal) pair.

1.1 Parse Table Example

For example, consider the following grammar, shown in Figure 3.1, which describes programs in Polish prefix consisting of declarations and simple assignment statements.

- (1) <program> → <s_list> <eof>
- (2) <S_list> → <assign> <S_list>
- (3) <S_list> →
- (4) <assign> → := <id> + <id> <no>
 <i>id₁</i>.name = <i>id₂</i>.name AND
 <i>no</i>.value = 1
- (5) <assign> → := <id> + <id> <no>
 <i>id₁</i>.name = <i>id₂</i>.name AND
 <i>no</i>.value = 2
- (6) <assign> → := <var> <expr>
- (7) <var> → <id>
- (8) <expr> → + <expr> <expr>
- (9) <expr> → <var>
- (10) <expr> → <no>

Figure 3.1 Example Grammar MD4

The grammar rules are annotated with rule numbers, and semantic restrictions appear in italics. The example grammar is an extension of the partial machine description MD1 given in Chapter two.

States	Action					Goto			
	1 <eof>	2 <id>	3 <no>	5 +	6 :=	N7 assign	N8 expr	N10 s_list	N11 var
1	r3				s4	s3	s9	s2	s11
2	s5					s3	s9	s2	s11
3	r3				s4	s3	s9	s6	s11
4		s8				s3	s9	s2	s7
5	acc					s3	s9	s2	s11
6	r2					s3	s9	s2	s11
7		s13	s10	s12		s3	s9	s2	s11
8	r7	r7	r7	s14/r7	r7	s3	s9	s2	s11
9	r6				r6	s3	s9	s2	s11
10	r10	r10	r10	r10	r10	s3	s9	s2	s11
11	r9	r9	r9	r9	r9	s3	s9	s2	s11
12		s13	s10	s12		s3	s15	s2	s11
13	r7	r7	r7	r7	r7	s3	s9	s2	s11
14		s16				s3	s9	s2	s11
15		s13	s10	s12		s3	s17	s2	s11
16			s18			s3	s9	s2	s11
17	r8	r8	r8	r8	r8	s3	s9	s2	s11
18	r4/r5				r4/r5	s3	s9	s2	s11

Figure 3.2 The Parse Table for Grammar MD4

The grammar MD4 is ambiguous. There are two possible derivations for the input `:= a + a 1` since both rule (4) and rule (6) apply. The ambiguity is manifested in the parse tables, shown in Figure 3.2, as a shift-reduce conflict on the symbol `+` in state 8. The two action entries in the table for the `+` symbol allow the parser to either shift the `+`, which leads towards application of rule (4), or reduce by rule (7), which leads towards application of rule (6).

Associating lists of shift and reduce actions with (state, terminal) pairs via the parse table is all the information the backtracking parsing algorithm requires. The parsing algorithm always attempts the shift action first, but in the event of a block the algorithm backs up and can attempt the reduce action(s) also encoded in the table. If none of the reduce actions apply because of semantic restrictions, the algorithm attempts to backup to an earlier decision point to try a different parse path.

2. Modifications to the Parsing Algorithm

The parsing algorithm was modified to take advantage of the new tables produced. The backtracking LR parsing algorithm is split into

two parts: the *forward parser* and the *backtracker*. The forward parser is essentially a traditional LR parsing algorithm, except that additional backtracking information is saved. The backtracker actually implements the backup, using the saved information to quickly find the most recent parse state where an alternative parsing decision is possible.

2.1 The Forward Parser

The forward parser is essentially a traditional LR parsing algorithm with two modifications. The symbol on the parse stack along with an index explained below is maintained as a symbol pair. Additionally, a *save stack* which is identical in type to the parse stack is maintained because parse stack elements, whether they are states or symbol pairs, must be saved so that the "state of parse" can be restored when backtracking. The save stack maintains a complete history of the current parse. (An example illustrating the creation of the save stack follows the discussion of the forward parsing algorithm.)

The traditional parse stack is unchanged with the exception of its symbol element which contains a symbol and an index. The symbol type defines the kind of index that is included. If the symbol is a terminal, the index is a marker into the action table which keeps track of the last alternative tried for a given (state, terminal) pair. If the symbol is a nonterminal, the index is the production number of the rule that was applied to produce this nonterminal.

The state and symbol elements of the stack are used to look up entries in the parse table in the usual way. In the case of terminal symbols, each entry in the parse table is a list of one or more actions. The marker indicates which action in the list should be attempted next. In our implementation, the marker is an index into the list of actions. The marker is initialized to one, and incremented each time an action from the associated list is taken. If backtracking occurs, the same state and symbol pair will be used to index the parse table again.

Since nonterminal symbols always have a list of one action in the goto table, no marker is required. Instead the index for the symbol pair is a production rule number, so that it is known which RHS was reduced to produce this nonterminal.

The forward parsing algorithm uses the next input symbol and the current state to search the parse table (action table) for the entry associated with that (state, symbol) pair. As stated previously, the entry consists of an ordered list of one or more actions. The markers are initialized so that the first action in the list is always attempted first. The forward parsing algorithm is outlined in Figure 3.3.

Action is either accept, error, reduce or shift.
ActionMarker is an integer that determines for a given action table entry which option in the ordered list is to be tried. It is paired with a terminal, currentInput to make up a symbol pair.
CurrentInput is the terminal used to determine which column in the action table to use.
RuleReducedBy is the rule number that was applied to get the nonterminal in its symbol pair.
State is the current state on the top of the parse stack and determines which row in the parse tables, action and goto, to use..
StateOrRule is a state when the action is shift or a rule number to reduce by when action is reduce.

```

GetNextInput;
parseStack := empty;
push (InitialState, parseStack);
saveStack := empty;

while true do
  state := top(parseStack);
  (action, stateOrRule) := GetAction(state,
                                   (currentInput, ActionMarker)
                                   );
  case action of
    Shift : DoShift;
    Reduce : DoReduce;
    Error : DoBacktrack;
    Accept : Halt (Accept);
  endcase;
endwhile;

```

Figure 3.3 The Forward Parsing Algorithm

Note that the forward parsing algorithm itself only differs from the traditional LR parsing algorithm by using a marker (actionMarker) to determine the next action and by calling the backtracker on error actions. Details of each procedure called by the forward parser appear in Figures 3.4, 3.5 and 3.6. Differences between the forward

parser's procedures and the traditional LR parser procedures have been italicized for clarity.

```
currentInput := Advance (input);  
actionMarker:= 1;
```

Figure 3.4 GetNextInput

The Shift algorithm, shown in Figure 3.5, is essentially the same as the traditional shift procedure, except that the symbol pushed onto the stack is a pair.

```
push ((currentInput,actionMarker), parseStack);  
push (stateOrRule, parseStack);  
GetNextInput;
```

Figure 3.5 DoShift

When a reduce action is taken by the parser, the stack elements forming the production's RHS are popped from the parse stack and pushed onto the save stack. If the production's RHS is empty, then nothing is stored on the save stack. The stack element for the leftmost symbol is pushed onto the save stack first and the stack element for the rightmost symbol is pushed last. As in the traditional Reduce algorithm, a new state is derived from the state on the top of the parse stack and the left hand side of the production just applied. Both the new symbol pair consisting of the nonterminal from the left hand side and the applied production rule number and the new state are pushed onto the parse stack. The modified DoReduce algorithm is shown in Figure 3.6.

```
RHS := length (stateOrRule) * 2;  
for i := 1 to RHS do  
    push (pop (parseStack), tempStack);  
endfor;  
  
for i := 1 to RHS do  
    push (pop (tempStack),saveStack);  
endfor;  
  
newParseState := GetGOTOtable (LHS(stateOrRule),  
                                top(parseStack));  
push ((LHS(stateOrRule), stateOrRule), parseStack);  
push (newParseState, parseStack);
```

Figure 3.6 DoReduce

2.2 An Example Parse

The example shown in Figure 3.7 illustrates the forward parsing algorithm using the example grammar MD4 of Figure 3.1 and corresponding parse table shown earlier in Figure 3.2. In this example, no backtracking occurs. The input is the assignment statement: `:= a + a 1`.

Notice that in this example, the first attempt to resolve the shift-reduce conflict at state eight in favor of the shift succeeds. Also note that the save stack does indeed maintain a complete history of the successful parse.

Notational Conventions. The current input symbol is the leftmost symbol of the input. The Parse stack is indicated by P: and the save stack by S: and both stacks are shown top to bottom.

<p>1. Input ==> <code>:= a + a 1 <eof></code> Stacks P S s1</p> <p>Action ==> Shift to state 4</p>	<p>2. Input ==> <code>a + a 1 <eof></code> Stacks P S s4 (:=, 1) s1</p> <p>Action ==> Shift to state 8</p>
<p>3. Input ==> <code>+ a 1 <eof></code> Stacks P S s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 14</p>	<p>4. Input ==> <code>a 1 <eof></code> Stacks P S s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 16</p>
<p>5. Input ==> <code>1 <eof></code> Stacks P S s16 (<id>, 1) s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 18</p>	<p>6. Input ==> <code><eof></code> Stacks P S s18 (<no>, 1) s16 (<id>, 1) s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Reduce by rule 4</p>

<p>7. Input ==> <eof> Stacks</p> <table border="0"> <thead> <tr> <th style="text-align: left; padding-right: 20px;"><u>P</u></th> <th style="text-align: left;"><u>S</u></th> </tr> </thead> <tbody> <tr> <td>s3</td> <td>s18</td> </tr> <tr> <td>(<assign>, R4)</td> <td>(<no>, 1)</td> </tr> <tr> <td>s1</td> <td>s16</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s14</td> </tr> <tr> <td></td> <td>(+, 1)</td> </tr> <tr> <td></td> <td>s8</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s4</td> </tr> <tr> <td></td> <td>(<:=, 1)</td> </tr> </tbody> </table> <p>Action ==> Reduce by rule 3</p>	<u>P</u>	<u>S</u>	s3	s18	(<assign>, R4)	(<no>, 1)	s1	s16		(<id>, 1)		s14		(+, 1)		s8		(<id>, 1)		s4		(<:=, 1)	<p>8. Input ==> <eof> Stacks</p> <table border="0"> <thead> <tr> <th style="text-align: left; padding-right: 20px;"><u>P</u></th> <th style="text-align: left;"><u>S</u></th> </tr> </thead> <tbody> <tr> <td>s6</td> <td>s18</td> </tr> <tr> <td>(<s_list>, R3)</td> <td>(<no>, 1)</td> </tr> <tr> <td>s3</td> <td>s16</td> </tr> <tr> <td>(<assign>, R4)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s1</td> <td>s14</td> </tr> <tr> <td></td> <td>(+, 1)</td> </tr> <tr> <td></td> <td>s8</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s4</td> </tr> <tr> <td></td> <td>(<:=, 1)</td> </tr> </tbody> </table> <p>Action ==> Reduce by rule 2</p>	<u>P</u>	<u>S</u>	s6	s18	(<s_list>, R3)	(<no>, 1)	s3	s16	(<assign>, R4)	(<id>, 1)	s1	s14		(+, 1)		s8		(<id>, 1)		s4		(<:=, 1)																
<u>P</u>	<u>S</u>																																																												
s3	s18																																																												
(<assign>, R4)	(<no>, 1)																																																												
s1	s16																																																												
	(<id>, 1)																																																												
	s14																																																												
	(+, 1)																																																												
	s8																																																												
	(<id>, 1)																																																												
	s4																																																												
	(<:=, 1)																																																												
<u>P</u>	<u>S</u>																																																												
s6	s18																																																												
(<s_list>, R3)	(<no>, 1)																																																												
s3	s16																																																												
(<assign>, R4)	(<id>, 1)																																																												
s1	s14																																																												
	(+, 1)																																																												
	s8																																																												
	(<id>, 1)																																																												
	s4																																																												
	(<:=, 1)																																																												
<p>9. Input ==> <eof> Stacks</p> <table border="0"> <thead> <tr> <th style="text-align: left; padding-right: 20px;"><u>P</u></th> <th style="text-align: left;"><u>S</u></th> </tr> </thead> <tbody> <tr> <td>s2</td> <td>s6</td> </tr> <tr> <td>(<s_list, R2)</td> <td>(<s_list>, R3)</td> </tr> <tr> <td>s1</td> <td>s3</td> </tr> <tr> <td></td> <td>(<assign>, R4)</td> </tr> <tr> <td></td> <td>s18</td> </tr> <tr> <td></td> <td>(<no>, 1)</td> </tr> <tr> <td></td> <td>s16</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s14</td> </tr> <tr> <td></td> <td>(+, 1)</td> </tr> <tr> <td></td> <td>s8</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s4</td> </tr> <tr> <td></td> <td>(<:=, 1)</td> </tr> </tbody> </table> <p>Action ==>> Shift to state 5</p>	<u>P</u>	<u>S</u>	s2	s6	(<s_list, R2)	(<s_list>, R3)	s1	s3		(<assign>, R4)		s18		(<no>, 1)		s16		(<id>, 1)		s14		(+, 1)		s8		(<id>, 1)		s4		(<:=, 1)	<p>10. Input ==> <eof> Stacks</p> <table border="0"> <thead> <tr> <th style="text-align: left; padding-right: 20px;"><u>P</u></th> <th style="text-align: left;"><u>S</u></th> </tr> </thead> <tbody> <tr> <td>s5</td> <td>s6</td> </tr> <tr> <td>(<eof>, 1)</td> <td>(<s_list>, R3)</td> </tr> <tr> <td>s2</td> <td>s3</td> </tr> <tr> <td>(<s_list>, R2)</td> <td>(<assign>, R4)</td> </tr> <tr> <td>s1</td> <td>s18</td> </tr> <tr> <td></td> <td>(<no>, 1)</td> </tr> <tr> <td></td> <td>s16</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s14</td> </tr> <tr> <td></td> <td>(+, 1)</td> </tr> <tr> <td></td> <td>s8</td> </tr> <tr> <td></td> <td>(<id>, 1)</td> </tr> <tr> <td></td> <td>s4</td> </tr> <tr> <td></td> <td>(<:=, 1)</td> </tr> </tbody> </table> <p>Action ==> Accept</p>	<u>P</u>	<u>S</u>	s5	s6	(<eof>, 1)	(<s_list>, R3)	s2	s3	(<s_list>, R2)	(<assign>, R4)	s1	s18		(<no>, 1)		s16		(<id>, 1)		s14		(+, 1)		s8		(<id>, 1)		s4		(<:=, 1)
<u>P</u>	<u>S</u>																																																												
s2	s6																																																												
(<s_list, R2)	(<s_list>, R3)																																																												
s1	s3																																																												
	(<assign>, R4)																																																												
	s18																																																												
	(<no>, 1)																																																												
	s16																																																												
	(<id>, 1)																																																												
	s14																																																												
	(+, 1)																																																												
	s8																																																												
	(<id>, 1)																																																												
	s4																																																												
	(<:=, 1)																																																												
<u>P</u>	<u>S</u>																																																												
s5	s6																																																												
(<eof>, 1)	(<s_list>, R3)																																																												
s2	s3																																																												
(<s_list>, R2)	(<assign>, R4)																																																												
s1	s18																																																												
	(<no>, 1)																																																												
	s16																																																												
	(<id>, 1)																																																												
	s14																																																												
	(+, 1)																																																												
	s8																																																												
	(<id>, 1)																																																												
	s4																																																												
	(<:=, 1)																																																												

Figure 3.7 Parse Trace using Grammar MD4

2.3 The Backtracker

The backtracker uses the save stack maintained by the forward parser to attempt a backup. DoBacktrack is invoked for both syntactic and semantic blocks. In the event of a syntactic block the symbol that causes the block is never put on the parse stack and remains in the input. Semantic blocks are discovered by the forward parser when none of the semantic restrictions for a list of reduce actions are met.

The backtracking algorithm iterates until an alternative path is found or until there are no other possible paths to try, i.e. the parse stack is empty. At each iteration the parse stack pops and discards the state on top of the stack. Then a symbol pair is popped and examined.

If the symbol pair represents a terminal and the marker indicates there is an alternative path for the terminal, control is returned to the forward parser with new values for current input and its marker. If the symbol pair represents a nonterminal the right hand side of the rule is removed from the save stack and pushed onto the parse stack, so that it is as if that rule had never been reduced. Each symbol of the right hand side of the production can now be examined in turn.

The backtracker algorithm is shown in Figure 3.8.

```

while not empty (parseStack) do
begin
    pop (parseStack);

    if top(parseStack) is terminal then
        (currentInput, actionMarker) :=
            pop (parseStack);

        if there exists an action list in the parse
            table for the symbol, currentInput and the
            state, top (parseStack) and there is an
            (actionMarker+1) entry in that list then

            actionMarker := actionMarker + 1;
            Exit DoBacktrack;
        endif
    else (* top of the parse stack is a nonterminal*)
        (nonTerminal, ruleReducedBy) :=
            pop (parseStack);

        RHS := length (ruleReducedBy) * 2;
        for i := 1 to RHS do
            push (pop (saveStack), tempStack);
        endfor;
        for i := 1 to RHS do
            push (pop (tempStack), parseStack);
        endfor;
    endif
endwhile;
Halt the parse with an error.

```

Figure 3.8 DoBacktrack algorithm

2.4 Simple Backtracking Example

The example shown in Figure 3.9 illustrates the backtracking algorithm using the example grammar MD4 from Figure 3.1 and

parse table shown earlier in Figure 3.2. This is a simple example in which no reduce has occurred when DoBacktrack is called, i.e. all symbols examined are popped from the parse stack and put back into the input. The input is := a + a 5.

In diagram (3), of Figure 3.9, the parser will attempt to and eventually apply the optimized increment instruction over the more costly add instruction, so in this shift-reduce conflict the shift to state 14 is chosen over the reduce by rule 7. This leads to a block in diagram (6). The increment instruction cannot be applied because the semantic restrictions are not met. DoBacktrack is called to find an alternative path. The backtracker backs up to state 8 where the most recent choice is found. This time the parser chooses the reduce by rule 7 and the parse continues to completion.

<p>1. Input ==> := a + a 5 <eof> Stacks P S s1</p> <p>Action ==> Shift to state 4</p>	<p>2. Input ==> a + a 5 <eof> Stacks P S s4 (:=, 1) s1</p> <p>Action ==> Shift to state 8</p>
<p>3. Input ==> + a 5 <eof> Stacks P S s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 14</p>	<p>4. Input ==> a 5 <eof> Stacks P S s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 16</p>
<p>5. Input ==> 5 <eof> Stacks P S s16 (<id>, 1) s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Shift to state 18</p>	<p>6. Input ==> <eof> Stacks P S s18 (<no>, 1) s16 (<id>, 1) s14 (+, 1) s8 (<id>, 1) s4 (:=, 1) s1</p> <p>Action ==> Error, cannot reduce by rule 4 or rule 5, DoBacktrack</p>

After Backup: CurrentInput is + and actionMarker is 2

- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|-------------|----------|-----|-----|-------------|-------------|-----|-----|--------------|-----------|-----|----|---|-----------|----------|----|--|-------------|-----------|-----|--|-----------|-----------|-----|---|--------------|-------------|-----|-----|---------------|-----------|-----|-----|--------------|-------------|-----|-----|----------|-----------|----|----|-------------|-----------|----|--|----------|--|----|--|
| <p>7. Input ==> a 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s8</td> <td></td> </tr> <tr> <td>(<id>, 1)</td> <td></td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Reduce by rule 7</p> | <u>P</u> | <u>S</u> | s8 | | (<id>, 1) | | s4 | | (<:=, 1) | | s1 | | <p>8. Input ==> a 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s7</td> <td>s8</td> </tr> <tr> <td>(<var>, R7)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Shift to state 12</p> | <u>P</u> | <u>S</u> | s7 | s8 | (<var>, R7) | (<id>, 1) | s4 | | (<:=, 1) | | s1 | | | | | | | | | | | | | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <p>9. Input ==> a 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s12</td> <td>s8</td> </tr> <tr> <td>(<+, 2)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s7</td> <td></td> </tr> <tr> <td>(<var>, R7)</td> <td></td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Shift to state 13</p> | <u>P</u> | <u>S</u> | s12 | s8 | (<+, 2) | (<id>, 1) | s7 | | (<var>, R7) | | s4 | | (<:=, 1) | | s1 | | <p>10. Input ==> 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s13</td> <td>s8</td> </tr> <tr> <td>(<id>, 1)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s12</td> <td></td> </tr> <tr> <td>(<+, 2)</td> <td></td> </tr> <tr> <td>s7</td> <td></td> </tr> <tr> <td>(<var>, R7)</td> <td></td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Reduce by rule 7</p> | <u>P</u> | <u>S</u> | s13 | s8 | (<id>, 1) | (<id>, 1) | s12 | | (<+, 2) | | s7 | | (<var>, R7) | | s4 | | (<:=, 1) | | s1 | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s13 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<id>, 1) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <p>11. Input ==> 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s11</td> <td>s13</td> </tr> <tr> <td>(<var>, R7)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s12</td> <td>s8</td> </tr> <tr> <td>(<+, 2)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s7</td> <td></td> </tr> <tr> <td>(<var>, R7)</td> <td></td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Reduce by rule 9</p> | <u>P</u> | <u>S</u> | s11 | s13 | (<var>, R7) | (<id>, 1) | s12 | s8 | (<+, 2) | (<id>, 1) | s7 | | (<var>, R7) | | s4 | | (<:=, 1) | | s1 | | <p>12. Input ==> 5 <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s15</td> <td>s11</td> </tr> <tr> <td>(<expr>, R9)</td> <td>(<var>, R7)</td> </tr> <tr> <td>s12</td> <td>s13</td> </tr> <tr> <td>(<+, 2)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s7</td> <td>s8</td> </tr> <tr> <td>(<var>, R7)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Shift to state 10</p> | <u>P</u> | <u>S</u> | s15 | s11 | (<expr>, R9) | (<var>, R7) | s12 | s13 | (<+, 2) | (<id>, 1) | s7 | s8 | (<var>, R7) | (<id>, 1) | s4 | | (<:=, 1) | | s1 | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s11 | s13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s15 | s11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<expr>, R9) | (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | s13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <p>13. Input ==> <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s10</td> <td>s11</td> </tr> <tr> <td>(<no>, 1)</td> <td>(<var>, R7)</td> </tr> <tr> <td>s15</td> <td>s13</td> </tr> <tr> <td>(<expr>, R9)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s12</td> <td>s8</td> </tr> <tr> <td>(<+, 2)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s7</td> <td></td> </tr> <tr> <td>(<var>, R7)</td> <td></td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Reduce by rule 10</p> | <u>P</u> | <u>S</u> | s10 | s11 | (<no>, 1) | (<var>, R7) | s15 | s13 | (<expr>, R9) | (<id>, 1) | s12 | s8 | (<+, 2) | (<id>, 1) | s7 | | (<var>, R7) | | s4 | | (<:=, 1) | | s1 | | <p>14. Input ==> <eof>
Stacks</p> <table border="0"> <tr> <td style="text-align: center;"><u>P</u></td> <td style="text-align: center;"><u>S</u></td> </tr> <tr> <td>s17</td> <td>s10</td> </tr> <tr> <td>(<expr>, R10)</td> <td>(<no>, 1)</td> </tr> <tr> <td>s15</td> <td>s11</td> </tr> <tr> <td>(<expr>, R9)</td> <td>(<var>, R7)</td> </tr> <tr> <td>s12</td> <td>s13</td> </tr> <tr> <td>(<+, 2)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s7</td> <td>s8</td> </tr> <tr> <td>(<var>, R7)</td> <td>(<id>, 1)</td> </tr> <tr> <td>s4</td> <td></td> </tr> <tr> <td>(<:=, 1)</td> <td></td> </tr> <tr> <td>s1</td> <td></td> </tr> </table> <p>Action ==> Reduce by rule 8</p> | <u>P</u> | <u>S</u> | s17 | s10 | (<expr>, R10) | (<no>, 1) | s15 | s11 | (<expr>, R9) | (<var>, R7) | s12 | s13 | (<+, 2) | (<id>, 1) | s7 | s8 | (<var>, R7) | (<id>, 1) | s4 | | (<:=, 1) | | s1 | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s10 | s11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<no>, 1) | (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s15 | s13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<expr>, R9) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <u>P</u> | <u>S</u> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s17 | s10 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<expr>, R10) | (<no>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s15 | s11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<expr>, R9) | (<var>, R7) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s12 | s13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<+, 2) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s7 | s8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<var>, R7) | (<id>, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (<:=, 1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| s1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

15. Input ==> <eof>

Stacks

P	S
s9	s17
(<expr>, R8)	(<expr>, R10)
s7	s15
(<var>, R7)	(<expr>, R9)
s4	s12
(<:=, 1)	(<+, 2)
s1	s10
	(<no>, 1)
	s11
	(<var>, R7)
	s13
	(<id>, 1)
	s8
	(<id>, 1)

Action ==> Reduce by rule 6

16. Input ==> <eof>

Stacks

P	S
s3	s9
(<assign>, R6)	(<expr>, R8)
s1	s7
	(<var>, R7)
	s4
	(<:=, 1)
	s17
	(<expr>, R10)
	s15
	(<expr>, R9)
	s12
	(<+, 2)
	s10
	(<no>, 1)
	s11
	(<var>, R7)
	s13
	(<id>, 1)
	s8
	(<id>, 1)

Action ==> Reduce by rule 3

17. Input ==> <eof>

Stacks

P	S
s6	s9
(<s_list>, R3)	(<expr>, R8)
s3	s7
(<assign>, R6)	(<var>, R7)
s1	s4
	(<:=, 1)
	s17
	(<expr>, R10)
	s15
	(<expr>, R9)
	s12
	(<+, 2)
	s10
	(<no>, 1)
	s11
	(<var>, R7)
	s13
	(<id>, 1)
	s8
	(<id>, 1)

Action ==> Reduce by rule 2

18. Input ==> <eof>

Stacks

P	S
s2	s6
(<s_list>, R2)	(<s_list>, R3)
s1	s3
	(<assign>, R6)
	s9
	(<expr>, R8)
	s7
	(<var>, R7)
	s4
	(<:=, 1)
	s17
	(<expr>, R10)
	s15
	(<expr>, R9)
	s12
	(<+, 2)
	s10
	(<no>, 1)
	s11
	(<var>, R7)
	s13
	(<id>, 1)
	s8
	(<id>, 1)

Action ==> Shift to state 5

```

19.  Input ==>
      Stacks
      P
      s5
      (<eof>, 1)
      s2
      (<s_list>, R2)
      s1

      S
      s6
      (<s_list>, R3)
      s3
      (<assign>, R6)
      s9
      (<expr>, R8)
      s7
      (<var>, R7)
      s4
      (:=, 1)
      s17
      (<expr>, R10)
      s15
      (<expr>, R9)
      s12
      (+, 2)
      s10
      (<no>, 1)
      s11
      (<var>, R7)
      s13
      (<id>, 1)
      s8
      (<id>, 1)

      Action ==> Accept

```

Figure 3.9 Another Parse Trace using grammar MD4

2.5 Example of Backtracking Over a Reduction

The example in Figure 3.12 illustrates the backtracking algorithm after several reductions have occurred. Grammar MD5 of Figure 3.10 and the parse table shown in Figure 3.11 are slightly modified versions of the grammar MD4 and its corresponding parse table.

- (1) <program> → <s_list> <eof>
- (2) <S_list> → <assign> <S_list>
- (3) <S_list> →
- (4) <assign> → := subscript <id> <expr>
 + subscript <id> <expr> <no>
 <id₁>.name = <id₂>.name AND
 <expr₁> = <expr₂> AND <no>.value = 1
- (5) <assign> → := subscript <id> <expr>
 + subscript <id> <expr> <no>
 <id₁>.name = <id₂>.name AND
 <expr₁> = <expr₂> AND <no>.value = 2
- (6) <assign> → := <var> <expr>
- (7) <var> → subscript <id> <expr>
- (8) <expr> → + <expr> <expr>
- (9) <expr> → <var>
- (10) <expr> → <no>

Figure 3.10 Grammar MD5

States	Action						Goto			
	1 sub- script	2 <eof>	3 <id>	5 <no>	6 +	7 :=	N8 assign	N9 expr	N11 s_list	N12 var
1		r3				s4	s3	s9	s2	s11
2		s5					s3	s9	s2	s11
3		r3				s4	s3	s9	s6	s11
4	s8						s3	s9	s2	s7
5		acc					s3	s9	s2	s11
6		r2					s3	s9	s2	s11
7	s13			s10	s12		s3	s9	s2	s11
8			s14				s3	s9	s2	s11
9		r6				r6	s3	s9	s2	s11
10	r10	r10		r10	r10	r10	s3	s9	s2	s11
11	r9	r9		r9	r9	r9	s3	s9	s2	s11
12	s13			s10	s12		s3	s15	s2	s11
13			s16				s3	s9	s2	s11
14	s13			s10	s12		s3	s17	s2	s11
15	s13			s10	s12		s3	s18	s2	s11
16	s13			s10	s12		s3	s19	s2	s11
17	r7	r7		r7	s20/r7	r7	s3	s9	s2	s11
18	r8	r8		r8	r8	r8	s3	s9	s2	s11
19	r7	r7		r7	r7	r7	s3	s9	s2	s11
20	s21						s3	s9	s2	s11
21			s22				s3	s9	s2	s11
22	s13			s10	s12		s3	s23	s2	s11
23				s24			s3	s9	s2	s11
24		r4/r5				r4/r5	s3	s9	s2	s11

Figure 3.11 The Parse Table for Grammar MD5

In the previous example each action was shown in a separate diagram. In Figure 3.12, each diagram may be a result of several actions. Given the input

```
:= subscript a + 2 1 + subscript a + 2 1 5
```

the parser will attempt to apply the optimized increment instruction over the more costly add instruction (see diagram (4)). In the shift-reduce conflict of state 17, the shift to state 20 is chosen over the reduce by rule 7. This leads to a block in diagram (7). The increment instruction cannot be applied because the semantic restrictions are not met. DoBacktrack is called to find an alternative path. As the backtracker backs up to state 17 where the most recent choice is found, it must expand nonterminals on the parse stack using the save stack. This time the parser chooses the reduce by rule 7 and the parse continues to completion.

1. **Input** ==> := subscript a + 2 1 + subscript a + 2 1 5 <eof>
Stacks

R	S
s10	
(<no>, 1)	
s12	
(+, 1)	
s14	
(<id>, 1)	
s8	
(subscript, 1)	
s4	
(<:=, 1)	
s1	

Actions ==> Shift to state 4, 8, 14, 12, 10, Reduce by rule 10

2. **Input** is ==> 1 + subscript a + 2 1 5 <eof>
Stacks

R	S
s15	s10
(<expr>,R10)	(<no>, 1)
s12	
(+, 1)	
s14	
(<id>, 1)	
s8	
(subscript, 1)	
s4	
(<:=, 1)	
s1	

Actions ==> Shift to state 10, Reduce by rule 10

3. Input ==> + subscript a + 2 1 5 <eof>

Stacks

<u>P</u>	<u>S</u>
s18	s10
(<expr>,R10)	(<no>, 1)
s15	s10
(<expr>,R10)	(<no>, 1)
s12	
(+, 1)	
s14	
(<id>, 1)	
s8	
(subscript, 1)	
s4	
(<:=, 1)	
s1	

Action ==> Reduce by rule 8

4. Input ==> + subscript a + 2 1 5 <eof>

Stacks

<u>P</u>	<u>S</u>
s17	s18
(<expr>, R8)	(<expr>,R10)
s14	s15
(<id>, 1)	(<expr>,R10)
s8	s12
(subscript, 1)	(+, 1)
s4	s10
(<:=, 1)	(<no>, 1)
s1	s10
	(<no>, 1)

Actions ==> Shift to states 20, 21, 22, 12 and finally to 10

5. Input ==> 1 5 <eof>

Stacks

<u>P</u>	<u>S</u>
s10	s18
(<no>, 1)	(<expr>,R10)
s12	s15
(+, 1)	(<expr>,R10)
s22	s12
(<id>, 1)	(+, 1)
s21	s10
(subscript, 1)	(<no>, 1)
s20	s10
(+, 1)	(<no>, 1)
s17	
(<expr>, R8)	
s14	
(<id>, 1)	
s8	
(subscript, 1)	
s4	
(<:=, 1)	
s1	

Actions ==> Reduce by rule 10, Shift to state 15, Reduce by rule 10 and finally reduce by rule 8

6. Input ==> 5 <eof>

Stacks

<u>P</u>	<u>S</u>
	s18
	(<expr>,R10)
s23	s15
(<expr>, R8)	(<expr>,R10)
s22	s12
(<id>, 1)	(+, 1)
s21	s10
(subscript, 1)	(<no>, 1)
s20	s10
(+, 1)	(<no>, 1)
s17	s18
(<expr>, R8)	(<expr>,R10)
s14	s15
(<id>, 1)	(<expr>,R10)
s8	s12
(subscript, 1)	(+, 1)
s4	s10
(<:=, 1)	(<no>, 1)
s1	s10
	(<no>, 1)

Action ==> Shift to state 24

7. Input ==> <eof>

Stacks

<u>P</u>	<u>S</u>
s24	s18
(<no>, 1)	(<expr>,R10)
s23	s15
(<expr>, R8)	(<expr>,R10)
s22	s12
(<id>, 1)	(+, 1)
s21	s10
(subscript, 1)	(<no>, 1)
s20	s10
(+, 1)	(<no>, 1)
s17	s18
(<expr>, R8)	(<expr>,R10)
s14	s15
(<id>, 1)	(<expr>,R10)
s8	s12
(subscript, 1)	(+, 1)
s4	s10
(<:=, 1)	(<no>, 1)
s1	s10
	(<no>, 1)

Action ==> Error, cannot reduce by rule 4 or rule 5, DoBacktrack

After Backup: CurrentInput is + and actionMarker is 2

8. Input ==> + subscript a + 2 1 5 <eof>
 Stacks

<u>P</u>	<u>S</u>
s17	s18
(<expr>, R8)	(<expr>,R10)
s14	s15
(<id>, 1)	(<expr>,R10)
s8	s12
(subscript, 1)	(+, 1)
s4	s10
(<:=, 1)	(<no>, 1)
s1	s10
	(<no>, 1)

Action ==> Reduce by Rule 7

9. Input ==> + subscript a + 2 1 5 <eof>
 Stacks

<u>P</u>	<u>S</u>
s7	s17
(<var>,R7)	(<expr>, R8)
s4	s14
(<:=, 1)	(<id>, 1)
s1	s8
	(subscript, 1)
	s18
	(<expr>,R10)
	s15
	(<expr>,R10)
	s12
	(+, 1)
	s10
	(<no>, 1)
	s10
	(<no>, 1)

Action ==> Shift to state 12

10. **Input** ==> subscript a + 2 1 5 <eof>

Stacks

P	S
s12	s17
(+, 2)	(<expr>, R8)
s7	s14
(<var>, R7)	(<id>, 1)
s4	s8
(<:=, 1)	(subscript, 1)
s1	s18
	(<expr>, R10)
	s15
	(<expr>, R10)
	s12
	(+, 1)
	s10
	(<no>, 1)
	s10
	(<no>, 1)

Actions ==> Shift to states 13, 16, 12, 10, Reduce by Rule 10, Shift to state 10, Reduce by rule 10, Reduce by Rule 8, Reduce by Rule 7, Reduce by Rule 9, Shift to state 10, Reduce by Rule 10, Reduce by Rule 8, Reduce by Rule 6, Reduce by Rule 3, Reduce by Rule 2, Shift to state 5

11. Input ==>

Stacks

E
s5
(<eof>, 1)
s2
(<s_list>, R2)
s1

S
s6
(<s_list>,R3)
s3
(<assign>,R6)
s9
(<expr>, R8)
s7
(<var>, R7)
s4
(:=, 1)
s18
(<expr>,R10)
s15
(<expr>,R9)
s12
(+, 2)
s10
(<no>, 1)
s11
(<var>, R7)
s19
(<expr>,R8)
s16
(<id>, 1)
s13
(subscript, 1)
s18
(<expr>,R10)
s15
(<expr>,R10)
s12
(+, 1)
s10
(<no>, 1)
s10
(<no>, 1)
s17
(<expr>, R8)
s14
(<id>, 1)
s8
(subscript, 1)
s18
(<expr>,R10)
s15
(<expr>,R10)
s12
(+, 1)
s10
(<no>, 1)
s10
(<no>, 1)

Action ==> Accept

Figure 3.12 Parse Trace using Grammar MD5

3. Undoing Semantic Actions during Backtracking

This chapter has presented the algorithms which implement a backtracking LR parser which can backup over both shift and reduce actions. The discussion has been limited to syntactic issues without regard for the semantic actions performed during reduce actions. Semantic actions must be incorporated if we are to do code generation with a backtracking parser. When backing up over reduces, semantic actions must be undone so that the parser can start on an alternative parse path. The next chapter discusses how semantic actions are handled during backtracking.

Chapter 4

Backing up over Semantic Actions

"Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away." by Antoine de Saint-Exupery (1900-1944), from The Book of Unusual Quotations edited by Rudolf Flesch (1957)

The previous chapter presented the modifications required to implement a backtracking LR parser which can backup over both shift and reduce actions. To backup from a blocked state, the parser must perform two operations: restore the input to its previous state and undo any semantic actions performed between the previous state and the blocked state. The previous chapter was only concerned with the syntactic issue of restoring the input. This chapter describes how backup over semantic actions is implemented. A similar scheme has been independently proposed by Keller [Kel91], although the implementation is different from the one proposed here.

1. Backup over Semantic Actions

Semantic actions are performed during reduce actions in the parser. The effects of semantic actions are encoded in tables which are then attached to the left-hand-side symbol as attributes. Conceptually, this approach saves a snapshot of the semantic state of the parse for every reduce action. The existing backtracking mechanism for restoring the input now operates without modification to undo

semantic actions as well. The previous semantic state is saved along with the input, and restored when a backup occurs.

Although this implementation is simple, it has many advantages. It is elegant, general, and completely integrated with the parser backtracking mechanism. No assumptions are made about the content of the semantic actions, other than the requirement that the actions can be encoded as an attribute. Furthermore, the implementation is consistently integrated within the framework of attributed parsing. The obvious disadvantage is amount of memory used in saving the semantic state of the parse.

The remainder of this chapter is devoted to showing an example of how this mechanism allows backup over semantic actions.

2. Register Allocation via Semantic Attributes

On-the-fly register allocation can be implemented using a register table. Each register table entry is a record that contains information about the register type, whether or not the register is free or in use; if the register is allocated, to which expression it is allocated and possibly a usage count depending on the register allocation scheme implemented. Register allocation/deallocation takes place as a semantic action associated with a parser reduce action. When the reduce occurs, the semantic action inserts a new table entry associating the left-hand-side (LHS) symbol with a register into the register table.

To make backtracking possible, the register table is encoded as a semantic attribute associated with the symbol. A copy of the register table attribute is created when each new symbol is added to the parse stack. In the case of a shift action, a copy of the register table attribute is inherited from the symbol on top of the parse stack. In the case of a reduce action, a copy of the register table attribute is synthesized from the new symbol's RHS and if a register allocation is performed as part of a semantic action, only the LHS symbol's copy is modified. Thus if a previous symbol is restored from the save stack during a backup, so is the previous register table.

3. Register Allocation Example

The grammar shown in Figure 4.1 is a modified version of the grammar of Figure 3.10 previously presented in Chapter 3. A new

rule and some semantic actions have been added. The parse table for this modified grammar would be similar to that of Figure 3.11.

```

(1) <program> → <s_list> <eof>
(2) <S_list> → <assign> <S_list>
(3) <S_list> →
(4) <assign> → := subscript <id> <expr>
                + subscript <id> <expr> <no>
                <id1>.name = <id2>.name AND
                <expr1> = <expr2> AND <no>.value = 1
(5) <assign> → := subscript <id> <expr>
                + subscript <id> <expr> <no>
                <id1>.name = <id2>.name AND
                <expr1> = <expr2> AND <no>.value = 2
(6) <assign> → := <var> <expr>
(7) <var> → subscript <id> <expr>
(8) <expr> → + <expr1> <expr2>
                <expr>.regTable := <expr2>.regTable -
                FreeRegister (<expr2>.reg) +
                ChangeRegisterBinding
                (<expr1>.reg, <expr>.reg);
(9) <expr> → <var>
                <expr>.regTable := <var>.regTable +
                GetRegister (<expr>.reg);
(10) <expr> → <no>
                <expr>.regTable := <no>.regTable +
                GetRegister (<expr>.reg);
(11) <var> → <id>

```

Figure 4.1 Grammar MD5 with Semantic Actions

In this example, we consider a parser based on the grammar of Figure 4.1, and consider its operation for the following input:

```
:= subscript a + i 1 + subscript b 2 6
```

The figures that follow depict snapshots of the parse stack and save stack at various points during the parse. The only attribute shown is the register table.

Figure 4.2 depicts the parse and save stacks after five input tokens have been consumed. The register table attribute is indicated by a bubble on the right of the stack entry. An empty table implies that

no registers have been allocated. At every shift action a copy of the previous table is made and is attached to the new symbol.

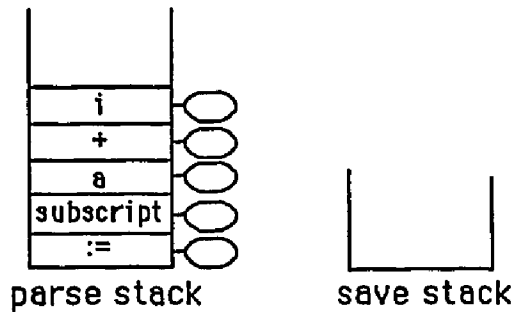


Figure 4.2 No Semantic Information Changed

In Figure 4.2, all five items have been shifted onto the parse stack, and there are five copies of the empty register table showing that no registers have been allocated. A method of eliminating unnecessary duplication of the register table is discussed in section 4 of this chapter.

The *i* is reduced to $\langle var_1 \rangle$ by rule 11 which pushes the *i* symbol onto the save stack, and then the $\langle var_1 \rangle$ is reduced to $\langle expr_1 \rangle$ by rule 9 causing the first change to the register table. In a semantic action associated with rule 9, register 1 is allocated to $\langle expr_1 \rangle$. The result is shown in Figure 4.3.

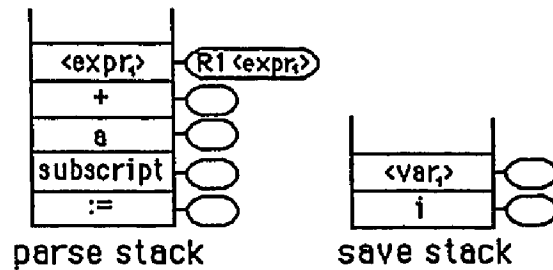


Figure 4.3 Register 1 is Allocated

The number 1 is shifted onto the parse stack, synthesizes a copy of the modified register table and then is reduced by rule 10 to $\langle expr_2 \rangle$. The reduction in rule 10 causes the semantic action GetRegister to allocate register 2 to $\langle expr_2 \rangle$. The resulting parse and save stacks are shown in Figure 4.4.

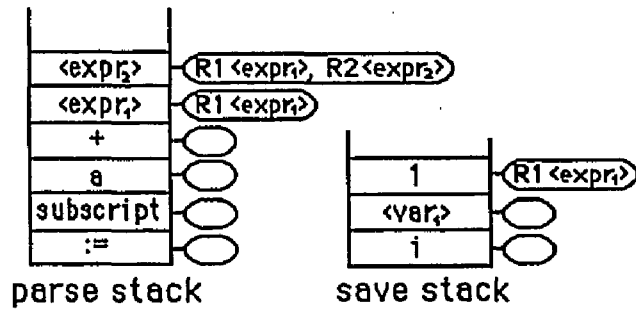


Figure 4.4 Register 2 is Allocated

Next the parser reduces by rule 8 which pops the parse stack to replace the right hand side + <expr₁><expr₂> with <expr₃>. Register 2 is freed; register 1 is transferred to hold the result of the plus operation. The resulting parse and save stacks are shown in Figure 4.5

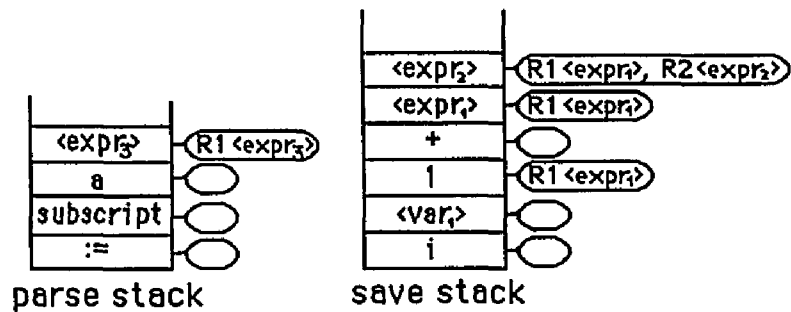


Figure 4.5 The Plus Operation is Performed

At this point, the parser encounters a shift-reduce conflict on the next input symbol, +, since the parser has the option to reduce by rule 7 or shift. The locally greedy heuristic resolves the conflict in favor of the shift, and + is shifted onto the parse stack. Figure 4.6 shows the stacks after 3 more symbols are shifted onto the parse stack.

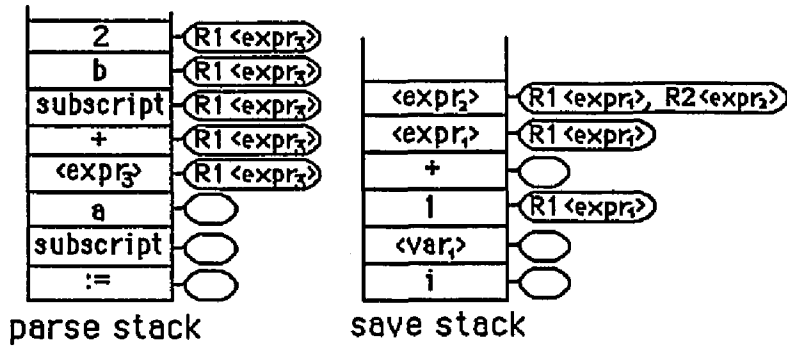


Figure 4.6 Trying the Shift Option

Figure 4.7 shows rule 10 applied, reducing the 2, representing an array index, to <expr4>, which is allocated to register 2.

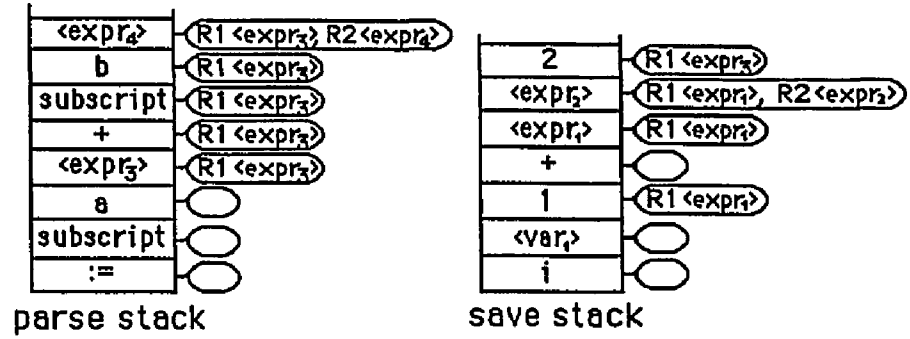


Figure 4.7 Register 2 is Reallocated

A semantic block occurs when the next input symbol, 6, is shifted onto the stack. The situation is shown in Figure 4.8.

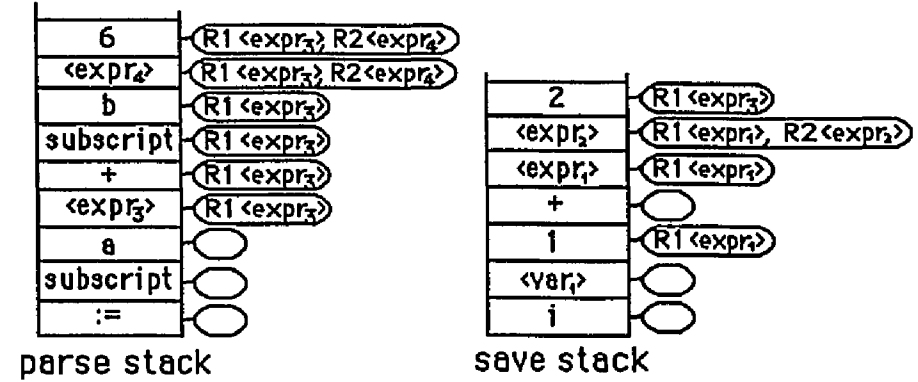


Figure 4.8 Block Trying to Apply Increment Instruction

The parser attempts to reduce by rule 4 or rule 5, but the semantic conditions are not met. The parser backups up to the point of the previous shift/reduce conflict. As symbols are popped off the save stack to restore the parse state, as explained previously in Chapter 3, the state of the register table is also restored. The resulting parse state after backup was shown previously in Figure 4.5. However, after backup the parse will proceed by attempting to reduce by rule 7, leading to the eventual reduce by rule 6, a general assignment statement.

4. Efficiency Considerations

One concern with the approach outlined above is the amount of space required to store a copy of the register table, and any other semantic attributes every time a new symbol is pushed on the parse stack. The usual space efficient implementation which only creates a new copy of the register table when the table is modified by a semantic action was employed.

Instead of storing a copy of the register table with each new symbol, each new symbol stores a pointer to a register table. If a semantic action will change the table, then a copy of the table is created, the modification is made to the copy, and the new symbol has a pointer to the modified copy. This scheme is well-known, simple and avoids wasting memory by copying identical tables unnecessarily. It should be noted that more space efficient schemes are available, one such scheme records only the changes made to the table.

5 Summary

This chapter has demonstrated that a simple method which allows backtracking over shift and reduce actions with semantic actions by storing the effects of semantic actions as attributes. In the next chapter, we turn our attention away from the mechanism which enables backtracking to discuss the code generator specification.

Chapter 5

Code Generation as Tree Transformations

"Wolfe, who had moved around the desk and into his chair, put up a palm at him: 'Please, Mr. Hombert. I think it is always advisable to take a short-cut when it is feasible.'" from The Rubber Band by Rex Stout (1936)

In this chapter, we introduce the concept of a tree transformation as a notation for writing code generation specifications and claim that tree transformations offer the same benefits as context-free grammars but have some additional advantages. Since an automatically generated code generator is only as good as the specification, improving the quality of the specification, with respect to the code generated, is desirable.

Code generation specifications for traditional LR parsing code generators are context free grammars which allow the specification to be incrementally enhanced with new grammar productions. Unfortunately, CFG code generation specifications tend to increase dramatically in size as optimizations that take advantage of efficient target machine instructions are incorporated into the machine description [Hen84]. As discussed previously in Chapter 2, adding an optimization rule often necessitates the addition of default rules to avoid semantic or syntactic blocking. The need for default rules results in large, unwieldy specifications and also calls into question the correctness of the code generator because it introduces the possibility of parse failure.

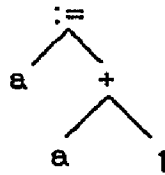
A backtracking LR parsing code generator eliminates the need for default rules (and thereby simplifying the code generation specification). When the parser "blocks", it backs up and restarts the parse.

Context free grammars offer an excellent mechanism for describing instruction selection but are less well suited for describing other phases of code generation such as common subexpression elimination and register allocation. The main difficulty is that such subphases cannot be expressed easily as grammar rules but must be embedded in the semantics associated with the rules.

Embedding information in the semantics associated with grammar rules is a symptom of the limitations of context free grammars for writing code generation specifications. The semantics portion of the grammar rule becomes a catch all for anything which cannot be easily expressed in the rule itself, which in turn reduces the legibility and formality of the machine description.

1. Tree Transformations

As an alternative to context-free grammars, tree transformations can be used for writing code generation specifications. A tree transformation describes how one tree (the source tree) is mapped or transformed into another tree (the rewrite tree). The use of tree-based notation arises naturally in the context of code generation since intermediate code is often conceptually represented as a tree, as shown in Figure 5.1.



(a) An Add by 1 IC Tree



(b) An Increment by 1 IC Tree

Figure 5.1 Intermediate Code (IC) trees

When viewed from the perspective of tree transformations, instruction selection consists of transforming intermediate code trees into machine code trees. For example, the intermediate code trees of Figure 5.1 above could be translated in the machine code tree shown in Figure 5.2 .

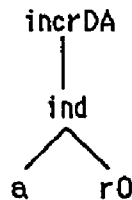


Figure 5.2 A Machine Code Tree

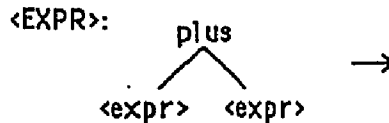
Rather than embedding the result of the transformation in the grammar rule's semantics, the tree transformation rule explicitly describes transformations in the rewrite tree. The backtracking mechanism allows the rewrite to be reparsed. For example, when intermediate code trees are expanded into machine code during instruction selection, traditional LR parser-based code generators do not make the expanded machine code tree available for further parsing. The machine code tree is emitted as object code. Backtracking provides a mechanism which makes expanded machine code trees available for further parsing, since the backtracking LR parser-based code generator can backup and reparse the expanded machine code.

Tree transformations can be classified by their effect on the source tree: reduction transformations, rearrangement transformations, and replacement transformations. Each of these is described in turn.

The *reduction transformation* rules are just grammar rules with a different syntax. Reduction transformations are captured in a grammar rule in which the RHS matches the linearized form of the source tree and the LHS represents a single node. In the context of a LR parser-based code generator, reduction transformations are required as part of the process of recognizing legal input. An example is shown in Figure 5.3.

`<EXPR> ::= plus <expr> <expr>`

(a) A Grammar Rule



(b) A Reduction Tree Transformation

Figure 5.3 Recognizing an Expression

The transformation shows the reduction of a tree representing an addition operation to an expression. The source tree appears on the left and the rewrite tree, in this case null, on the right. This convention will be used in subsequent examples of tree transformations.

Tree rearrangement and replacement transformations cannot be expressed explicitly in a context-free grammar rule itself because the left hand side is a single nonterminal. Greater expressiveness coupled with a more intuitive representation makes tree transformations an attractive way to write specifications for code generation.

A *rearrangement transformation* reorders the children of a node in the source tree, but leaves the rest of the source tree unchanged, as shown in Figure 5.4.

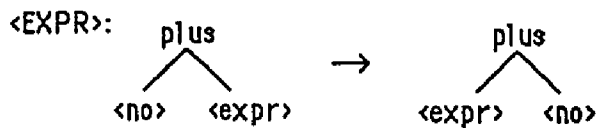


Figure 5.4 A Rearrangement Tree Transformation

The transformation shown reorders the children of the `plus` tree, so that `<no>` becomes the right child and `<expr>` becomes the left.

Replacement transformations are all other mappings of source tree to rewrite tree which are neither reduction transformations nor rearrangement transformations. The source tree is replaced by the rewrite tree. An example is shown in Figure 5.5.

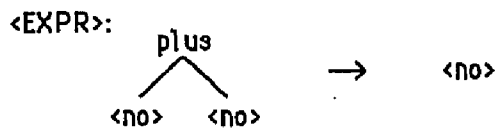


Figure 5.5 A Replacement Tree Transformation

In the transformation above, an addition operation is collapsed to a single number thereby performing the computation at code generation time, a common compile time optimization.

Each class of tree transformations can also be conditioned on semantic information contained in the source tree. If the semantic condition does not hold, the tree transformation is not applied. An example appears in Figure 5.6. Side effects known as actions can also be included in the mapping, but are not shown in this example.

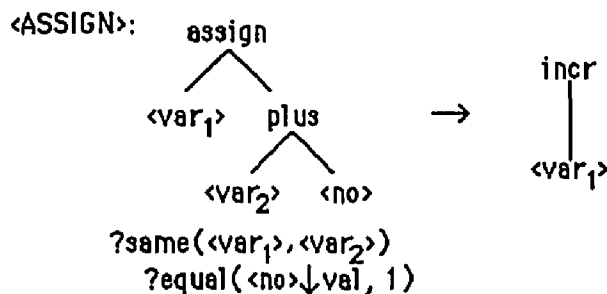


Figure 5.6 A Conditional Tree Transformation

The transformation of Figure 5.6 is a replacement transformation for instruction optimization in which an assignment is transformed into an increment instruction. The source tree on the left represents an assignment statement and the rewrite tree on the right represents an increment statement. The transformation is conditional because it should only be applied if the two variables $\langle var_1 \rangle$ and $\langle var_2 \rangle$ refer to the same variable and the value of the $\langle no \rangle$ is equal to one.

Machine instruction sets for different architectures may have different requirements for $\langle no \rangle$; whereas some have the requirement shown in the example, others may require $\langle no \rangle$ to be within a range such as [1..16]. Such restrictions are easily included as tree transformation conditions.

Tree transformations appear to be a more intuitive means of describing code generation for the following reasons. The intermediate code is already conceptually represented as a tree. Tree transformations can be used to map intermediate code trees to other intermediate code trees, intermediate code trees to machine code trees, or even machine code trees to other machine code trees. In this way, the same specification can describe many subphases of code generation including intermediate code tree optimization, instruction selection, register assignment and machine code optimization. These phases require all three kinds of tree transformations: reduction, rearrangement, and replacement. For example, replacement transformations are useful in the instruction selection phase because intermediate code trees typically expand to larger trees representing machine code as shown previously in part (b) of Figure 5.1 and Figure 5.2. Replacement transformations are also useful in collapsing intermediate code trees, a typical operation needed for intermediate code optimization as shown in Figures 5.5 and 5.6. Rearrangement transformations are used for intermediate code optimization so all other transformations can assume the location of a literal in the IC tree. Reduction transformations are necessary to recognize legal source trees.

The combination of a tree transformer with backtracking offers the advantage of a unified mechanism for code generation. Furthermore, the advantage of a unified mechanism is amplified by the convenient expression of tree transformations.

2. A Tree Transformation Language for Code Generation Specification

A Tree Transformation Language (TTL) has been developed for writing code generation specifications. A specification written in TTL is input to a grammar generator. The grammar generator produces an equivalent specification in the form of an affix grammar with tree reduction, rearrangement and replacement transformations encoded as operations associated with a reduce action by the parser. The specification grammar is passed as input to a parser generator which produces the parse tables required for a backtracking LR parser. These parse tables are used by the backtracking LR parser which inputs intermediate code (IC) and produces object code for the target machine as output. A complete description of the parse tables and backtracking parsing algorithm appears in Chapter 3.

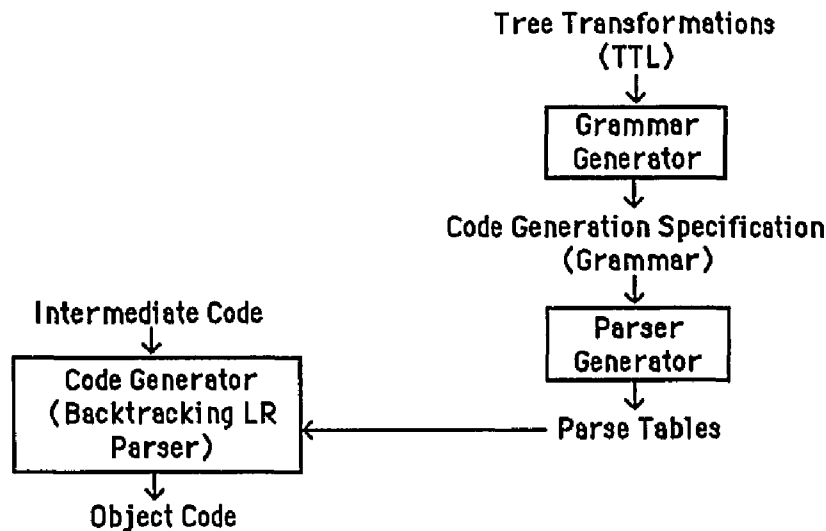


Figure 5.7 Backtracking LR Parsing-based Code Generation

Each tree transformation takes the form:

LHS : RHS {?condition} {@action} ⇒ rewrite tree

where {} represents 0 or more

The RHS is the source tree. The ?condition specifies semantic restriction(s) that must be met before the transformation can be applied, as illustrated in Figure 5.6. The @action contains side effect(s) of the application of the tree transformation. For example,

an addressing transformation which moves an operand from memory to a register decrements the number of available registers as a side effect.

For example, the TTL description for the example of Figure 5.6 appears below:

```

<ASSIGN> : assign <var1> plus <var2> <no>↑val
           ?same(<var1>, <var2>) ?equal(↓val, 1)
           ⇒ incr <var1>

```

There are two semantic conditions: the ?same condition is met if the two variables are the same, and the ?equal condition ensures that the value of the val attribute of <no> is 1. Attributes are denoted by vertical arrows as either synthesized or out parameters (↑) or inherited or in parameters (↓). The value of the val attribute of <no> is an output (synthesized) parameter as indicated by the arrow (↑) and then it is used as an in or inherited attribute to the ?equal condition.

The algorithm for translating a tree transformation specification to a Pargen affix grammar production is discussed in Chapter 6. Appendix F shows that TTL and a backtracking LR parser produce a recognizer for any recursively enumerable language. The rest of this chapter includes example applications of the tree transformation techniques for code generation.

3. Applying these ideas to Code Generation

There are three different tasks in code generation, i.e. instruction selection, optimizations, register allocation, etc. Previous researchers have developed methodologies for implementing those tasks. One goal of this thesis was to express a few of these standard algorithms to verify they could be easily implemented using the tree transformation mechanism. Three were chosen because we thought they were representative of typical applications. The first, peephole optimization of the intermediate code, was chosen because it was well suited to TTL specifications. The second, common subexpression elimination using usage counts, illustrates how an algorithm that did not appear suited to TTL specifications was easily implemented. The last, instruction selection, is a necessary part of any code generator.

A complete listing of each TTL specification described below appears in Appendices A, B and C respectively.

4. Peephole Optimization of Intermediate Code

The first methodology discussed below is described in the paper "Using Peephole Optimization on Intermediate Code" by Tanenbaum, van Staveren and Stevenson [TaStSt82]. Tannenbaum et. al., argue that the efficient compilation of simple, commonly occurring source statements is the key to producing good code. They have divided optimizations into ten subcategories. These subcategories are constant folding, operator strength reduction, null sequences, combined moves, commutative law, indirect moves, comparison, special instructions, DUP instructions, and reordering. Optimizations in seven of the subcategories have been implemented. Three categories were not implemented because our target machine could not take advantage of these classes of optimizations. These included combined moves, indirect moves and DUP instructions. Combined moves took advantage of multiword moves. Indirect moves tried to replace indirect moves with more efficient direct moves. DUP instructions tries to avoid refetching an operand that is already on the stack.

4.1 Example

The source code expression

$(3 + ((-(- X)) * 2)) + 4$

appears as the linearized code tree

plus plus 3 mult Uminus Uminus X 2 4

except numbers actually appear as the terminal symbol <no>.

The subtree Uminus Uminus X is matched against the tree transformation rule:

<EXPR> : Uminus Uminus <expr> ⇒ <expr>

The null sequence, Uminus Uminus, would be eliminated from the intermediate code producing the input

plus plus 3 mult X 2 4

The subtree mult X 2 is matched against the tree transformation rule:

<EXPR> : mult <expr><no>↑val
 ?equal(2,↓val) ⇒ shiftL <expr>

The operator strength reduction on the intermediate code would cause the elimination of the `mult` and `2` symbols and an addition of a new symbol, `shiftL`, producing the input

```
plus plus 3 shiftL X 4
```

The subtree `plus 3 shiftL X` is reordered by applying the tree transformation rule:

```
<EXPR> : <binop> <no> <expr>
        => <binop> <expr> <no>
```

so the reordering using the commutative law produces the input

```
plus plus shiftL X 3 4
```

The final tree transformation rule applied

```
<EXPR> : <binop1> <binop2> <expr> <no1>↑val1 <no2>↑val2
        ?same(<binop1>, <binop2>) ?assoc(<binop1>)
        @compute(<binop1>, ↓val1, ↓val2)↑val3
        => <binop> <expr> <no3>↓val3
```

which after constant folding produces the final input

```
plus shiftL X 7
```

Another example shows the application of a special instruction optimization. The source code in the assignment statement

```
x := x + 1
```

is represented by the linearized tree below.

```
assign X plus X 1
```

This entire subtree is matched by the tree transformation rule:

```
<ASSIGN> : assign <var1> plus <var2> <no>↑val
          ?same(<var1>, <var2>) ?equal(1, ↓val)
          => incr <var1>
```

The application of the transformation produces the final input

```
incr X
```

The last optimization example shows the class of comparison optimizations. The source code expression

```
not (X = Y)
```

would appear as the linearized intermediate code tree below.

```
not EQ X Y
```

This entire subtree is matched by the tree transformation rule:

$$\begin{aligned} \langle \text{EXPR} \rangle &: \text{not } \langle \text{relop} \rangle \uparrow \text{relop}_1 \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \\ &\quad @\text{flip}(\downarrow \text{relop}_1) \uparrow \text{relop}_2 \\ &\Rightarrow \langle \text{relop} \rangle \downarrow \text{relop}_2 \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \end{aligned}$$

The flip action returns the negation of the relational operator it is passed, i.e. a GT (greater than) passed to flip would return the operator LE (less than or equal). The not terminal symbol would be eliminated by negating the relational operator, EQ, making it NE, producing the final input

NE X Y

As is true with all effective optimizations, the resulting tree requires less machine code calculation than the original tree.

5. Common Subexpression Elimination

Common subexpression elimination was implemented using an algorithm described in the paper "Register Allocation via Usage Counts" by Freiburghouse [Fre74]. The purpose of this subphase is to recognize duplicates whether they are duplicate variables, constants or whole expressions. Common subexpressions are found by first replacing operand references, then possibly the whole expression. Usage counts tell how many references there are to a particular expression after all the references are fixed. The input, a forest of trees, is converted into a directed acyclic graph. When a common variable/number is detected, the duplicated node reference is changed to point/refer to an original node. This in turn leads to the detection of common subexpressions in a bottom-up approach: if an operation is identical to an earlier operation and their operands are the same, the second operation can be replaced with a reference to the original operation. Each time an expression is referred to by another pointer, its usage count is incremented.

This algorithm was originally implemented in our system as a preprocessor. We did not think the implementation would be a simple matter because the output of the algorithm is not a tree. The implementation using tree transformations turned out to be almost trivial. The grammar for this subphase is simple and short because of the bottom-up nature of LR parsers. Only two types of production rules are needed: one that finds duplicate operands and one that finds duplicate operations after the operands have been checked. Identifiers (variable or constant) and literals are "replaced" as

duplicates are found. It is then a trivial matter to test whether or not an operation is identical to a previous operation because the operands have already been "replaced".

A replacement is simply an adjustment of the pointer to a operand or operation to refer to an operand/operation in a previous tree or in a lefthand sibling of the current tree. In effect this changes the forest of intermediate code trees into a directed acyclic graph. In addition to changing the reference to a duplicated operand/operation, the original operand/operation's usage count must be incremented by one. In this way the number of references to an expression is known by its usage count.

5.1 Example

The assignments

```
x := a + b; w++; y := a + b;
```

form a forest of trees. A preorder walk of the trees yields the following input in linearized polish prefix notation.

```
assign x plus a b   incr w   assign y plus a b
```

The subtree a in the tree assign y plus a b is matched by the tree transformation rule:

```
<EXPR> : <var>
        @findSame(<var>)↑link   ⇒ dupl↓link
```

The `findSame` action checks all previous trees/left siblings within a linear region [Fr74] when trying to find a duplicate operand/operation. It searches back through the forest of trees to find an identical expression. It stops searching when it finds a reference to the contained subexpression, or when it finds a tree that affects an element of the expression.

If `findSame` is successful in its search, a reference to the expression is returned so that the new symbol, `dupl`, has an attribute identifying which subtree it is referring to, i.e., what it is a duplicate of. `Dupl` is used as a temporary place holder so that it can in turn make the appropriate reference.

The same rule would be applied a second time on the subtree b in the same tree assign y plus a b, producing:

```
assign x plus a b   incr w   assign y plus dupl dupl
```

The subtree `plus dupl dupl` matches the tree transformation rule:

$$\begin{aligned} \langle \text{EXPR} \rangle : \langle \text{binop} \rangle \uparrow \text{root dupl dupl} \\ \quad \text{@findsame}(\downarrow \text{root}) \uparrow \text{link} \quad \Rightarrow \text{dupl} \downarrow \text{link} \end{aligned}$$

Intermediate code would be eliminated and a new symbol added producing the input

```
assign x plus a b   incr w   assign y dupl
```

6. Instruction Selection

As our last example we show instruction selection (IS) was performed on the intermediate code tree. This involved straightforward transformations of traditional IC to machine code [GlGr78] [GaFi85].

During the instruction selection phase, tree transformations now provides an elegant solution to a problem created when common subexpressions have been eliminated in a two address machine architecture. In an architecture of this type the contents of the destination operand are destroyed when a binary operation is performed. This causes problems because some values need to be reused and therefore should not be destroyed by the operation.

If this two address restriction applies, the tree transformation rules shown below is appropriate.

$$\begin{aligned} \langle \text{EXPR} \rangle : \text{plus } \langle \text{expr}_1 \rangle \uparrow \text{useCount } \langle \text{var} \rangle \text{ ?EqOne}(\downarrow \text{useCount}) \Rightarrow \\ \quad \text{addDA } \langle \text{expr}_1 \rangle \langle \text{var} \rangle \\ \\ \langle \text{EXPR} \rangle : \text{plus } \langle \text{expr}_1 \rangle \uparrow \text{useCount } \langle \text{expr}_2 \rangle \text{ ?EqOne}(\downarrow \text{useCount}) \\ \quad \Rightarrow \text{addR } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \end{aligned}$$

Figure 5.8 Two transformations to machine instructions for addition

Figure 5.8 shows two possible addressing modes for addition in a code generation specification, in which a machine tree representing the addition operation can only be created if the destination register's, `<expr1>`, usage count is one. It can only be destroyed if the value in the register is not needed again. If the condition fails, backup occurs to the point where the transformation shown in Figure 5.9 can be applied, i.e. the source operand is put back into the input.

$$\begin{aligned} \langle \text{EXPR} \rangle &: \langle \text{binop} \rangle \langle \text{expr}_1 \rangle \uparrow \text{useCount} \text{ ?GtOne} (\downarrow \text{useCount}) \\ &\Rightarrow \langle \text{binop} \rangle \text{ copy } \langle \text{expr}_1 \rangle \end{aligned}$$

Figure 5.9 A transformation to generate a register copy

The transformation in Figure 5.9 creates a copy expression. A copy tree is handled differently than other rewrite trees that are treated as an in place substitution. All references to $\langle \text{expr}_1 \rangle$ remain intact, instead of referring to the copy $\langle \text{expr}_1 \rangle$ tree. Only the reference of the most recent parent, $\langle \text{binop} \rangle$, of the $\langle \text{expr}_1 \rangle$ tree is modified to point to the copy operation.

$$\begin{aligned} \langle \text{EXPR} \rangle &: \text{ copy } \langle \text{expr} \rangle \text{ @getRegister} (\uparrow r) \\ &\Rightarrow \text{ loadR reg} \downarrow r \langle \text{expr} \rangle \end{aligned}$$

Figure 5.10 A transformation to machine code for a register load

The transformation in Figure 5.10 transforms a copy expression into a register transfer operation. The action, `getRegister`, gets a free register that is an attribute of the register symbol. The register's usage count is automatically set to one. Another transformation reduces the register symbol to the nonterminal $\langle \text{expr} \rangle$ so that one of the transformations in Figure 5.8 can be applied. Although the register transfer costs a move, it is a worthwhile move if the $\langle \text{expr} \rangle$ represents a large expression that would otherwise have to be recalculated.

7. Summary

In this chapter, we introduced a tree transformation language, TTL, as an alternative to context free grammars for writing code generator specifications. Tree transformations were used to specify peephole optimizations on intermediate code, common subexpression elimination, and instruction selection, demonstrating the applicability of tree transformations for code generation. Tree transformations offer a unified mechanism in which a variety of code generation operations can be expressed in a single code generator specification.

In our scheme, a grammar generator translates a TTL code generator specification into an equivalent context free grammar specification

for use by the backtracking LR parsing code generator. Chapter 6 describes the translation algorithm in greater detail.

Chapter 6

Algorithm for Converting Tree Transformations to an Affix Grammar

"The word 'algorithm' itself is quite interesting; at first glance it may look as though someone intended to write 'logarithm' but jumbled up the first four letters" from The Art of Computer Programming, V. I., by D.E. Knuth (1968)

This chapter describes the algorithm used to translate machine descriptions written in TTL into an equivalent affix grammar representation suitable for a parser generator. The algorithm is used by the *grammar generator*, shown previously in Figure 5.8. The algorithm was not automated for this thesis, so the translations were done manually. As shown below, the automation will be straightforward.

1. The Tree Transformation to Affix Grammar Algorithm

As described in Chapter 5, the language for expressing tree transformations takes the general form:

LHS : RHS {?condition} {@action} ⇒ rewrite tree

Figure 6.1 TTL Syntax

A machine description consists of a series of these tree transformation rules. The grammar generator translates each tree transformation rule into an affix grammar rule shown in Figure 6.2.

```

LHS ::= RHS
/apply := (condition1) and ....
begin
    action; ....
    construct the rewriteTree.
    reparse := true;
    NewInput := rewriteTree
end;

```

Figure 6.2 The Affix Grammar Rule Syntax for the TTL Rule of Figure 6.1

The affix grammar rule of Figure 6.2 is used as a template by the grammar generator. The translation algorithm examines each TTL rule and generates a corresponding affix grammar rule by instantiating the template shown in Figure 6.2 with specific values from the TTL rule. The exact syntax of the affix grammar rule template was chosen to match the parser generator system, Pargen. [CoNo85]

The translation from the TTL Figure of 6.1 to the affix grammar rule of Figure 6.2 is straightforward. The RHS and LHS are copied exactly into the template of Figure 6.2 to form the grammar rule itself.

If there are no ?condition's in the TTL rule then the corresponding grammar rule is unconditional, i.e. semantic conditions are not considered before the rule is applied. Otherwise if ?condition's exist in the TTL rule, they are used to form the condition set in the affix grammar rule template of Figure 6.2. The list of ?condition's is changed into a sequence of Pascal boolean function calls, one function per ?condition. The function calls are joined with the Pascal boolean and construct which allows for the conditional application of grammar rules. The boolean functions themselves are supplied by the user as part of the code generator. Before a grammar rule is applied in Pargen, i.e. at a reduce action in the parser, the boolean condition is evaluated. The condition is true if the semantic attributes associated with the grammar symbols contain certain values described by the boolean function. If any condition in the conjunction does not hold, the reduce action is prevented and backup occurs.

The @action's in the TTL rule are Pascal procedure calls which are copied into the affix grammar rule template as a sequence of semicolon-separated procedure calls. The Pascal action procedures

are part of the semantics associated with the grammar, and are executed when the grammar rule is applied at a reduce action in the parser. Many of the TTL rule's have no @action's, therefore no Pascal action procedure calls are copied into the grammar semantics. The Pascal action procedures themselves are supplied by the user as part of the code generator.

The only real conversion of ?condition's and @action's to Pascal subprograms occurs in the parameters. Since the parameters from either the ?condition or the @action of the TTL specification also correspond to the attributes for the grammar symbols, the parameters are replaced by the appropriate attribute references. The ↓ indicates information passed to the subprogram and ↑ indicates information returned from the subprogram.

The remaining part of the template affix grammar rule is instantiated to perform the tree transformation. There are two cases to consider, a non-null rewrite tree and a null rewrite tree.

If the `rewriteTree` in the TTL is null, the TTL rule describes a tree reduction transformation which is modeled exactly by an affix grammar rule. The following portions of the template

```
    construct the rewriteTree;
    reparse := true;
    NewInput := rewriteTree;
are omitted by the grammar generator.
```

If the `rewriteTree` in the TTL is non-null, the rule describes a tree rearrangement or tree replacement transformation which requires the transformation operation, the "construct the rewriteTree" portion of the template, to be encoded in the semantics of the grammar rule. The grammar generator also inserts the Pascal statements

```
    reparse := true;
    NewInput := rewriteTree;
```

Setting `reparse` forces a backup in the LR parser which is limited to only backing up one symbol before the the current grammar rule. The input is then reparsed, but the input has changed because the input has been rewritten as specified by the `rewriteTree`. The replacement of the source tree by the rewrite tree is effected by inserting the statement: "NewInput := rewriteTree." Any and all references to the root of the RHS now reference the root of the `rewriteTree`, and all other references remain intact.

The grammar generator creates Pascal code to perform the rearrangement or replacement transformation by using the set of Pascal procedures shown in Figure 6.3. These tree construction procedures are included as part of the library routines for the parser and are assumed to be available during code generation.

```
MakeNewNode (newTerminal, nodePtr);
             Creates a terminal symbol.

MakeStackNode (oldSymbol, nodePtr);
             Grabs a node out of the old tree (terminal or
             nonterminal.

MakeUnary (child, parent);
MakeBinary (lChild, rChild, parent);
            Attaches node(s) to a root node forming a
            tree.
```

Figure 6.3 Tree Construction Procedures

The algorithm in Figure 6.4 specifies in general terms how tree transformations are constructed with implementation level details omitted for clarity. Also we stress that the purpose of this algorithm is to create Pascal statements which will actually perform the transformation during code generation. In Figure 6.4, the generation of a Pascal statement is indicated by a call to `EmitSemantics`.

The tree transformation construction algorithm scans the linearized TTL rewrite tree from right to left examining each symbol in turn. Conceptually, the algorithm proceeds in two stages. First, Pascal code is generated to create a tree node for the current symbol. Second, Pascal code is generated to merge the newly created tree node into the partially constructed tree. The two stages are indicated by comments in Figure 6.4 and explained below.

```

S ← emptyStack

for each sym in reverse(rewriteTree) do
begin
(* 1) Generate Make Tree Node *)

    if TTLleftmost(sym) then
        symName ← "rewriteTree"
    else
        symName ← GenName(sym)

    if (sym in RHS) then
        EmitSemantics("makeStackNode (sym, symName)")
    else
        EmitSemantics("makeNewNode (sym, symName)")

    S.push (symName)

(* 2 ) Generate Tree Construction Code *)

    if Binop (sym) then
    begin
        root ← S.pop
        lchild ← S.pop
        rchild ← S.pop

        EmitSemantics("MakeBinary(lchild, rchild, root)")
        S.Push( root)
    end (* have binary operator *)

    else if Unaryop (sym) then
    begin
        root ← S.pop
        child ← S.pop

        EmitSemantics("MakeUnary(child, root)")
        S.push(root)
    end (* have unary operator *)

end (* for each symbol *)

```

Figure 6.4 Transformation Construction Algorithm

Creating Tree Nodes

Pascal procedures to create tree nodes are predefined as part of the code generator. The tree construction algorithm simply emits Pascal procedure calls to these predefined routines. There are two types of tree nodes: tree nodes for symbols which already appear in the RHS and tree nodes for other symbols. If a tree node for a RHS symbol is to be reused, code to generate a new node is not necessarily needed and the construction algorithm could save the symbol reference on its stack. This would work in all instances.

There was a time when the implementation required a new node to be created for an old symbol, but this is no longer true. The algorithm shows the emission of code to generate new references for old symbols by inserting the call to the `MakeStackNode` procedure even though it is currently unnecessary. Otherwise if the symbol does not appear in the RHS, a tree node for the symbol is created by inserting a call to `MakeNewNode`.

Both `MakeStackNode` and `MakeNewNode` are generated to create a tree node or create a new reference to an existing tree node respectively. A generated variable name, `symName`, for this reference is required. Variable names are created by the `GenName` function in Figure 6.4. By default, the variable name for the root (leftmost symbol) of the TTL rewrite tree is `rewriteTree`. The construction algorithm saves the variable name associated with each tree node on a stack for later use to generate Pascal code which will, when executed, merge the tree nodes to assemble the rewrite tree.

Making Trees from Existing Nodes

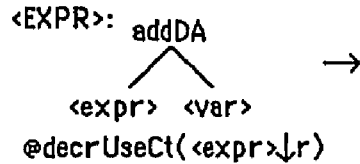
Pascal code is generated to create trees whenever an operator symbol is encountered in the right to left scan of the TTL rewrite tree. The grammar generator simply emits a procedure call to the appropriate Pascal routine: `makeUnary` or `makeBinary`, each of which has child node(s) and parent node parameters. The generated variable names for the parent node references are saved on the stack maintained by the construction algorithm and are popped as needed.

2. TTL to Affix Grammar Examples

The remainder of this chapter gives several examples of TTL to affix grammar conversions.

Reduction Transformation Example

Reduction transformations are the simplest case, since no `rewriteTree` is needed. Figure 6.5 shows the TTL rule for a reduction transformation, and its corresponding grammar rule.



(a) A picture of the transformation specification.

`<EXPR> : addDA <expr>↑r <var> @decrUseCt (↓r) ⇒`

(b) TTL specification

```
<EXPR> ::= addDA <expr> <var>
begin
    decrUseCt (<expr>.r);
end;
```

(c) Corresponding affix grammar rule

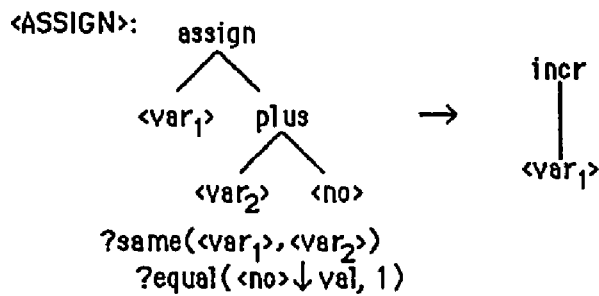
Figure 6.5 Simple Reduction Transformation

The attribute, `r`, is the register associated with the expression, `<expr>`. The parser generator system used in this thesis, Pargen, accepts the notation `<nonterminal>.attribute` to access attributes, as shown by the use of `<expr>.r` in part (c) of Figure 5.6. This notation is merely for convenience of the user, and Pargen automatically converts this notation to reference the appropriate attribute record on the parse stack.

The transformation of part (b) of Figure 6.5 contains one action, `@decrUseCt` that is converted to a Pascal procedure call. The Pascal procedure is written by the user and supplied as part of the code generator.

Conditional Transformation Example

The pictorial representation of an optimization transformation appears in part (a) of Figure 6.6.



(a) A picture of the transformation specification.

```

<ASSIGN> : assign <var1> plus <var2> <no> ↑val
           ?same(<var1>, <var2>) ?equal(↓val, 1)
           ⇒ incr <var1>
  
```

(b) TTL specification.

```

<ASSIGN> ::= assign <var1> plus <var2> <no>
/apply := same(<var1>, <var2>) and (equal(<no>.val, 1));
begin
    makeStackNode (<var1>, APtr);
    makeNewNode (incr, rewriteTree);
    MakeUnary (APtr, rewriteTree);
    reparse := true;
    NewInput := rewriteTree;
end;
  
```

(c) Corresponding affix grammar rule

Figure 6.6 Transformations with Conditions

The transformation of Figure 6.6 shows the replacement of an assignment statement by an increment instruction. The selection of an increment instruction is conditional. The `?same` and `?equal` conditions have the obvious meaning. The TTL rule for this transformation is shown in part (b) of Figure 6.6.

The TTL rule in part (b) of 6.6 is converted into the affix grammar rule shown in part (c) of Figure 6.6. The LHS and RHS, are copied directly to form the grammar rule. The condition conjunction is formed by Pascal functions which test each condition.

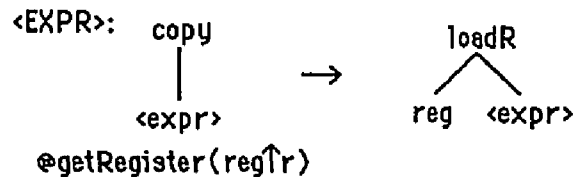
The construction algorithm of Figure 6.4 is used to construct the rewrite tree, which generates the Pascal code:

```
makeStackNode (<var1>, APtr);
makeNewNode (incr, rewriteTree);
MakeUnary (APtr, rewriteTree);
```

in part (a) of Figure 6.6. The generated variable names `APtr` and `rewriteTree` are generated by the construction algorithm. In this case, the construction algorithm operates as follows. The first symbol encountered in a right-left scan of the TTL rewrite tree is `<var1>`. As `<var1>` appears in the RHS, a call to `makeStackNode` is emitted and `APtr` is pushed on the tree translator stack. Next, the `incr` symbol is scanned. A call to `makeNewNode` is emitted and `rewriteTree` is pushed on the stack. Since `incr` is an unary operator, the stack is popped to get the parameters for the emitted call to `MakeUnary`.

Action Transformation Example

The example in Figure 6.7 shows a tree transformation which transforms a copy expression into a register transfer operation. This transformation was discussed previously in Chapter 5.



(a) A picture of the transformation specification.

```

<EXPR> : copy <expr> @getRegister(↑r) ⇒
                                             loadR reg↓r <expr>
  
```

(b) TTL specification

```

<EXPR> ::= copy <expr>
begin
    getRegister (reg.r);
    makeStackNode (<expr>, APtr);
    makeNewNode(reg, BPtr);
    makeNewNode (loadR, rewriteTree);
    MakeBinary (BPtr, APtr, rewriteTree);
    reparse := true;
    NewInput := rewriteTree;
end;
  
```

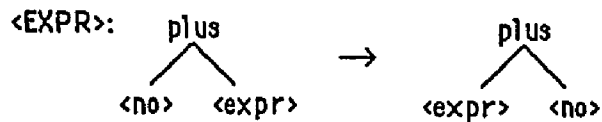
(c) Corresponding affix grammar rule

Figure 6.7 Transformations with Actions

This conversion is analogous to the last one, except this transformation includes an action, namely a call to the register allocator for a new register. The grammar generator converts the parameters of the action and copies the Pascal procedure call (with a slight syntax change): `getRegister`, into the grammar semantics. The in/out parameters (designated by \uparrow and \downarrow) in the TTL rule are used by the grammar generator to determine the appropriate attribute reference. In this case, the returned register is the "r" attribute of the `reg` symbol.

Rearrangement Transformation Example

Figure 6.8 specifies a reordering of the children of the RHS.



(a) A picture of the transformation specification.

`<EXPR> : plus <no> <expr> \Rightarrow plus <expr> <no>`

(b) TTL specification

```

<EXPR> ::= plus <no> <expr>
begin
    makeStackNode (<no>, APtr);
    makeStackNode (<expr>, BPtr);
    makeStackNode (plus, rewriteTree);
    MakeBinary (APtr, Bptr, rewriteTree);
    reparse := true;
    NewInput := rewriteTree;
end;
```

(c) Corresponding affix grammar rule

Figure 6.8 Rearrangement Transformation

The transformation is unconditional and involves no actions. Since the transformation just reorders the RHS, no calls the `makeNewNode` are generated by the grammar generator, and all attributes stored with the RHS are preserved.

3. Summary

This chapter presents a transformation construction algorithm which converts machine descriptions written in TTL into an equivalent affix grammar representation suitable for a parser generator. Four examples illustrate the algorithm for the following kinds of transformations: a reduction transformation, a replacement transformation with semantic conditions, a replacement transformation with semantic actions, and a rearrangement transformation.

Chapter 7

Reparse Backup

"And now I see with eye serene, the very pulse of the machine"
from "She was a phantom of delight", Poems in Two Volumes by
William Wordsworth (1807)

A tree transformation is more than a pattern recognizer because the transformation can also produce a rewrite tree. In the context of code generation, it is desirable to have the new pattern produced by the tree transformation available for further transformations. For example, the results of IC peephole optimizations should be made available for instruction selection transformations. In this way, tree transformations can be viewed as creating new input for the code generator in the form of a rewrite tree.

We have implemented tree transformations using LR parsing to perform the pattern recognition, semantic actions to perform any tree transformation, and backtracking to make the new input available for further transformations, i.e. reparsing. When the new input is generated, the parse state must be adjusted to allow parsing to continue with the new input. A similar situation arises in the context of syntactic error recovery [BuFi87]: when a syntax error occurs it is not sufficient to simply add, delete, or replace a token; the parse state must also be adjusted.

In Chapter 5 we described the use of tree transformations for code generator specifications, and in Chapter 6 we showed how tree transformation rules are translated to an affix grammar

representation. In this chapter, we describe the implementation used to support reparsing as the result of a tree transformation. First the differences between backups for semantic blocking and backups for tree transformations are described, then the implementation details are given, and finally the chapter concludes with an example illustrating "reparse backup".

1. Reparse Backups

Previously, backup only occurred in the case of blocking. Another backup allows for reparsing the rewrite tree produced by a tree transformation. These transformation induced backups are different from backups induced by blocking in several respects. First, transformation induced backup is much more limited as we need only backup over the source tree, represented by the RHS of the current grammar rule. Second, transformation induced backup differs in the recovery of the input for the backup. Some symbols in the source tree are used in rewrite tree, and can be retrieved from either the parse stack or the save stack and placed back in the input, just as in blocking. But new symbols that appear in the rewrite tree are not on the save stack. However since new symbols in the rewrite tree are always terminals, these can be put directly into the input.

It is not obvious that the scope of a transformation induced backup is limited; how far must we backup to restore the parse state such that the parser can continue? The answer comes from Burke and Fischer who have shown that backing up one nonterminal past the root of the source tree is sufficient [BuFi87]. This result arises in the context of syntactic error recovery, but is equally applicable here. The source tree is considered as the syntactic error, and the rewrite tree as the correct syntax.

2. Reparse Backup Implementation

The algorithm used to implement a reparse backup appears in Figure 7.1. The algorithm has three parts, denoted by comments in the algorithm. First, generate the new input from the rewrite tree and place it into the input; next, backup by adjusting the parse and save stacks; and finally, if the symbol directly to the left of the source tree root is a nonterminal, backup one more nonterminal past the root of the source tree.

```

(* Generate new input from rewrite tree *)
for sym in RightLeftRootTraversal(rewrite tree)
  if Terminal(sym) THEN
    concat(sym, input)
  else
    unwind(sym);

(* Backup by adjusting parse and save stacks *)
parseTop := adjustTop(ParseStack, sourceTree.root);
saveTop := adjustTop(SaveStack, sourceTree.root);

(* Backup one NonTerminal past root of source tree, *)
(* concatenating terminals into the input *)
if nonTerminal(topSym(parseStack)) then
begin
  sym := pop(parseStack);
  unwind(sym);
  saveTop := adjustTop(SaveStack, sym.root);
end;

Procedure Unwind (sym);
begin
  for i := sym.Rindex downto sym.Lindex
    if terminal(saveStack[i].sym) then
      concat(sym, input)
    else
      unwind(saveStack[i].sym);
end;  (* unwind *)

```

Figure 7.1: Reparse Backup Algorithm

To generate the new input, the rewrite tree is traversed and each symbol is concatenated to the front of the input buffer. Terminal symbols are placed directly into the input, but nonterminals which are part of the source tree can also appear in the rewrite tree. The terminal symbols which derived a nonterminal are recovered from the save stack and placed into the input by procedure `unwind`.

To backup, the `adjustTop` procedure shifts the parse and save stacks to remove the source tree. Since state as well as symbol information is stored on the stacks, adjusting the stacks reorients the parser. Finally, we backup one non-terminal past the root of the source tree, calling `unwind` to recover terminals into the input buffer.

3. Reparse Backup Example

The following example illustrates the algorithm for reparse backups. The input considered is intermediate code representing an assignment statement, shown in Figure 7.2.

```
assign x minus x plus 1 0
```

Figure 7.2: IC for Assignment Statement

First, a tree transformation which performs constant folding is applied to fold `plus 1 0` which causes a reparse backup. Next, an IC optimization transformation is applied which produces the IC `decr x`. The relevant tree transformation for constant folding is shown in Figure 7.3.

```
<expr> : plus <no1>↑val1 <no2>↑val2
        @compute(minus, ↓val1, ↓val2)↑val3
        ⇒ <no3>↓val3
```

Figure 7.3: A Constant Folding Tree Transformation

The affix grammar rule representing the constant folding transformation is applied when the `plus 1 0` intermediate code tree is recognized by the parser. The state of the parse and save stacks at this point are shown in Figure 7.4.

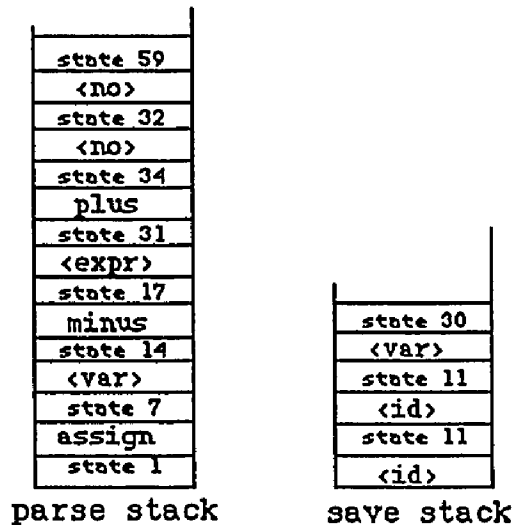


Figure 7.4: Parse State before Constant Folding

The effect of applying the constant fold transformation should be the same as if the original input were:

```
assign x minus x 1
```

The compute action determines the value of the constant fold, 1, which becomes the new input, replacing plus 1 0. The reparse backup pops the parse stack and corresponding symbols on the save stack, one nonterminal past the source tree. The resulting parse state is shown in Figure 7.5, and parsing continues with the remaining input as: x 1, the x being placed in the input by popping <expr> off the parse stack and retrieving the <id> from the save stack.

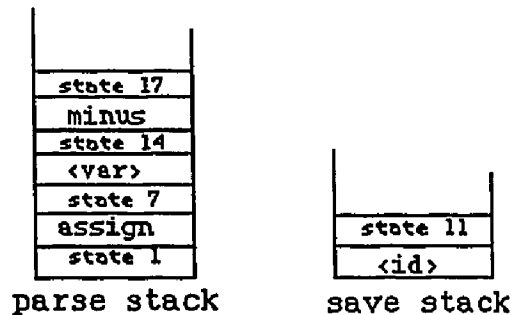


Figure 7.5: Parse State after Constant Folding

Parsing continues until the parse state shown in part (a) of Figure 7.6 is reached which allows the instruction selection transformation of part (b) of Figure 7.6 to be applied, generating the machine instruction `decr x`.

Chapter 8

Conclusions

"I don't want to achieve immortality through my work, I want to achieve it through not dying." by Woody Allen (1935-) from Woody Allen, A Biography by Eric Lax (1991)

This chapter summarizes the research results presented in the previous seven chapters of this thesis. In addition, directions for future research are identified.

1. Summary

The creation of a code generator is a substantial undertaking. Past research has aimed towards formal methods of automatically deriving code generators from target machine descriptions. This thesis makes two contributions towards the goal of targeting a code generator automatically from a specification:

- 1) We developed a backtracking LR parser and described its use for automatically deriving code generators from a code generation specification. The backtracking LR parsing code generator described was developed independently but is similar to the approach taken by Keller [Kel91].
- 2) We developed a tree-based notation, the Tree Transformation Language (TTL), for code generator specifications, and demonstrate TTL by implementing a number of standard code generation algorithms from the literature.

One advantage of a backtracking LR parser for code generation over previous methods is the simplification of blocking avoidance. Blocking which arises from locally greedy conflict resolution is avoided automatically by backtracking, eliminating the need for default rules.

The primary advantage of TTL for code generator specifications is that TTL offers a unified mechanism for describing the many different kinds of tasks performed by the code generator. To demonstrate its versatility, TTL was used to describe peephole optimizations on intermediate code, common subexpression elimination, and instruction selection. The strength of TTL is that the same notation was used to specify all of these code generator tasks.

TTL code generator specifications are translated into affix grammars for input into the parser generator which produces a backtracking LR parser. In this way, the convenience of a tree-based notation is combined with the formality and power of parsing. In addition, backtracking makes the results of a transformation available for further parsing.

In this thesis, we implemented three standard code generation algorithms in TTL: IC peephole optimizations, instruction selection and common subexpression elimination via usage counts. We expected TTL to be well suited for IC peephole optimization and instruction selection. We chose to implement common subexpression elimination via usage counts because TTL did not seem to be suited to the task, and we thought it was a good test of TTL's capabilities. Furthermore, common subexpression elimination often gets buried in the semantic actions which are difficult to trace and debug. TTL expressed common subexpression elimination at a higher conceptual level, as part of the code generator specification, which simplified the implementation.

The decision to use three separate code generator specifications instead of one was dictated by hardware limitations of the development system. The Apple Macintosh Plus computer used lacked the memory required for large code generator specifications. On a larger memory machine, one specification would have been possible.

The appendices contain samples of TTL code generator specifications, and test runs of the resulting code generator. Appendix A contains the IC peephole optimization specification, Appendix B contains the common subexpression elimination specification, and Appendix C contains the instruction selection specification. Appendix D contains a selection of test runs documenting the operation of the code generator. Appendix E illustrates what happens if "reparse backups" do not backup up one nonterminal past the source tree. Appendix F shows that TTL and a backtracking LR parser produce a recognizer for any recursively enumerable language.

2. Directions for Future Work

There are a number of straightforward extensions to the current system which would increase its utility.

1. Currently, the translation from TTL to the affix grammar representation is done manually following the algorithm in Chapter 6. This algorithm can be implemented.
2. We implemented three standard code generation algorithms in TTL: IC peephole optimizations, common subexpression elimination via usage counts and instruction selection. Consequently, we would like to implement some more challenging algorithms, some of which are discussed below.

A number of architectures implement conditional jumps based on a set of condition codes which indicate whether the last value computed or compared is negative, zero, or positive. On such machines, a common optimization is to avoid unnecessary comparisons if the condition code is already set. For example¹ in the code:

```
x := y + z
if x < 0 goto z
```

The comparison $x < 0$ can be avoided since the statement $x := y + z$ sets the condition code for the value of x as a side effect. If implemented TTL would have to maintain the required condition code information.

Data flow analysis collects information, such as live/dead variable information, which is required for a number of

¹This example is due to [AhScUI88], p. 541.

common optimizations, such as dead code elimination and register allocation via graph coloring. It would be interesting to use TTL to specify data flow analysis algorithms. After completing a TTL implementation, the next step would be to implement some algorithms which require data flow analysis information.

The dynamic programming code generation algorithm [AhSeU188] can be used to generate code for expressions on any machine with R interchangeable registers. The dynamic programming algorithm progresses in three phases. In the first phase a cost vector, $C[1..R]$, is computed bottom-up for each node, N , in the expression tree where $C[i]$ represents the optimal cost of computing the subtree rooted at N when i registers are available. In the second phase, the expression tree is traversed again using the cost vector to determine which subtrees must be computed into memory. The third phase traverses the annotated expression tree and actually generates machine code.

Another possibility would be to attempt an implementation of the dynamic programming code generation algorithm using TTL and backtracking, using the following method. LR parsing operates bottom-up, and the cost vectors can be maintained as attributes. The three phases could be implemented using three types of TTL transformation rules such that phase 1 rules produce rewrite trees attributed with cost vectors, phase 2 rules produce rewrite trees in which operands are designated as residing in memory or in registers, and phase 3 rules perform instruction selection. Backtracking allows the output of phase 1 rules to be available for phase 2, and phase 2 output to be made available for phase 3.

3. We have implemented the backend of a complete compiler, and we would like to match this backend with a front-end for C or Pascal.

In addition, work remains to better evaluate the capabilities of the current system:

1. Code generator specifications should be done for different target architectures, such as a one address instructions set or three address instruction set.

2. A number of performance evaluations should be conducted to yield more empirical measure of a backtracking LR parser for code generation including:

i. A comparison of the size of the code generator specifications can be compared to machine descriptions required for other methods.

ii. The execution time required by the resulting code generator to generate machine code can be compared to other schemes for the same target machine.

iii. The efficiency of the machine code generated by the code generator can be compared to other schemes.

Appendix A

Peephole Optimization of the Intermediate Code

The following tree transformations describe peephole optimizations on the intermediate code, a method which was described by Tannenbaum [TaStSt82]. Of the ten classes of IC peephole optimizations described by Tannenbaum, seven are represented in the tree transformations below namely: constant folding, operator strength reduction, null sequences, commutative law, comparison, special instructions and reordering. The classes of combined moves, indirect moves, and DUP instructions were not represented because the target machine code could not take advantage of these classes of optimizations.

`<program> : <slis> <eof>`

\Rightarrow

`<slis> : <assign> <slis>`

\Rightarrow

`<slis> :`

\Rightarrow

`<expr> : plus <no1> \uparrow val1 <no2> \uparrow val2
@compute(plus, \downarrow val1, \downarrow val2) \uparrow val3
 \Rightarrow <no3> \downarrow val3`

`<expr> : minus <no1> \uparrow val1 <no2> \uparrow val2
@compute(minus, \downarrow val1, \downarrow val2) \uparrow val3
 \Rightarrow <no3> \downarrow val3`

`<expr> : plus <expr> <no> \uparrow val
?equal(\downarrow val, 0)
 \Rightarrow <expr>`

$\langle \text{expr} \rangle : \text{multiply } \langle \text{expr} \rangle \langle \text{no}_1 \rangle \uparrow \text{val}_1$
 $\quad ?\text{equal}(\downarrow \text{val}_1, 0)$
 $\quad \Rightarrow \langle \text{no}_2 \rangle \downarrow 0$

$\langle \text{expr} \rangle : \text{multiply } \langle \text{expr} \rangle \langle \text{no}_1 \rangle \uparrow \text{val}_1$
 $\quad ?\text{equal}(\downarrow \text{val}_1, 1)$
 $\quad \Rightarrow \langle \text{expr} \rangle$

$\langle \text{expr} \rangle : \text{Uminus } \langle \text{no}_1 \rangle \uparrow \text{val}_1$
 $\quad @\text{compute}(\text{minus}, 0, \downarrow \text{val}_1) \uparrow \text{val}_2$
 $\quad \Rightarrow \langle \text{no}_2 \rangle \downarrow \text{val}_2$

$\langle \text{expr} \rangle : \text{Uminus Uminus } \langle \text{expr} \rangle$
 $\quad \Rightarrow \langle \text{expr} \rangle$

$\langle \text{expr} \rangle : \text{not } \langle \text{lit} \rangle \uparrow \text{true}$
 $\quad \Rightarrow \langle \text{lit} \rangle \downarrow \text{false}$

$\langle \text{expr} \rangle : \text{not } \langle \text{lit} \rangle \uparrow \text{false}$
 $\quad \Rightarrow \langle \text{lit} \rangle \downarrow \text{true}$

$\langle \text{expr} \rangle : \text{plus plus } \langle \text{expr} \rangle \langle \text{no}_1 \rangle \uparrow \text{val}_1 \langle \text{no}_2 \rangle \uparrow \text{val}_2$
 $\quad @\text{compute}(\text{plus}, \downarrow \text{val}_1, \downarrow \text{val}_2) \uparrow \text{val}_3$
 $\quad \Rightarrow \text{plus } \langle \text{expr} \rangle \langle \text{no}_3 \rangle \downarrow \text{val}_3$

$\langle \text{expr} \rangle : \text{minus } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$
 $\quad ?\text{same}(\langle \text{expr}_1 \rangle, \langle \text{expr}_2 \rangle)$
 $\quad \Rightarrow \langle \text{no} \rangle \downarrow 0$

$\langle \text{expr} \rangle : \text{multiply } \langle \text{expr} \rangle \langle \text{no} \rangle \uparrow \text{val}$
 $\quad ?\text{equal}(\downarrow \text{val}, 2)$
 $\quad \Rightarrow \text{shiftL } \langle \text{expr} \rangle$

$\langle \text{expr} \rangle : \text{plus } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$
 $\quad ?\text{same}(\langle \text{expr}_1 \rangle, \langle \text{expr}_2 \rangle)$
 $\quad \Rightarrow \text{shiftL } \langle \text{expr}_1 \rangle$

$\langle \text{expr} \rangle : \text{eq } \langle \text{expr} \rangle \langle \text{lit} \rangle \uparrow \text{val}$
 $\quad \text{?equal}(\downarrow \text{val}, \text{false})$
 $\quad \Rightarrow \text{not } \langle \text{expr} \rangle$

$\langle \text{expr} \rangle : \text{eq } \langle \text{expr} \rangle \langle \text{lit} \rangle \uparrow \text{val}$
 $\quad \text{?equal}(\downarrow \text{val}, \text{true})$
 $\quad \Rightarrow \langle \text{expr} \rangle$

$\langle \text{expr} \rangle : \text{ge } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$
 $\quad \Rightarrow \text{le } \langle \text{expr}_2 \rangle \langle \text{expr}_1 \rangle$

$\langle \text{expr} \rangle : \text{gt } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$
 $\quad \Rightarrow \text{lt } \langle \text{expr}_2 \rangle \langle \text{expr}_1 \rangle$

$\langle \text{expr} \rangle : \text{not } \langle \text{relop} \rangle \uparrow \text{relop}_1 \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$
 $\quad \text{@flip}(\downarrow \text{relop}_1) \uparrow \text{relop}_2$
 $\quad \Rightarrow \langle \text{relop} \rangle \downarrow \text{relop}_2 \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle$

$\langle \text{assign} \rangle : \text{assign } \langle \text{var}_1 \rangle \text{ plus } \langle \text{var}_2 \rangle \langle \text{no} \rangle \uparrow \text{val}$
 $\quad \text{?(same}(\langle \text{var}_1 \rangle, \langle \text{var}_2 \rangle)) \text{?equal}(\downarrow \text{val}, 1)$
 $\quad \Rightarrow \text{incr } \langle \text{var}_1 \rangle$

$\langle \text{assign} \rangle : \text{assign } \langle \text{var}_1 \rangle \text{ minus } \langle \text{var}_2 \rangle \langle \text{no} \rangle \uparrow \text{val}$
 $\quad \text{?(same}(\langle \text{var}_1 \rangle, \langle \text{var}_2 \rangle)) \text{?equal}(\downarrow \text{val}, 1)$
 $\quad \Rightarrow \text{decr } \langle \text{var}_1 \rangle$

$\langle \text{assign} \rangle : \langle \text{assign} \rangle \langle \text{var}_1 \rangle \langle \text{no} \rangle \uparrow \text{val}$
 $\quad \text{?equal}(\downarrow \text{val}, 0)$
 $\quad \Rightarrow \text{zero } \langle \text{var}_1 \rangle$

$\langle \text{expr} \rangle : \text{plus } \langle \text{no} \rangle \langle \text{expr} \rangle$
 $\quad \Rightarrow \text{plus } \langle \text{expr} \rangle \langle \text{no} \rangle$

$\langle \text{expr} \rangle : \text{multiply } \langle \text{no} \rangle \langle \text{expr} \rangle$
 $\quad \Rightarrow \text{multiply } \langle \text{expr} \rangle \langle \text{no} \rangle$

$\langle \text{expr} \rangle : \text{eq } \langle \text{lit} \rangle \langle \text{expr} \rangle$
 $\quad \Rightarrow \text{eq } \langle \text{expr} \rangle \langle \text{lit} \rangle$

<expr> : ne <lit> <expr>
⇒ ne <expr> <lit>

<assign> : assign <var> <expr>
⇒

<assign> : incr <var>
⇒

<assign> : decr <var>
⇒

<assign> : zero <var>
⇒

<expr> : Uminus <expr>
⇒

<expr> : not <expr>
⇒

<expr> : Lshift <expr>
⇒

<expr> : plus <expr> <expr>
⇒

<expr> : minus <expr> <expr>
⇒

<expr> : multiply <expr> <expr>
⇒

<expr> : <relop> <expr> <expr>
⇒

<relop> : eq
⇒

<relop> : le
⇒

$\langle \text{relop} \rangle : \text{lt}$
 \Rightarrow
 $\langle \text{relop} \rangle : \text{ne}$
 \Rightarrow
 $\langle \text{expr} \rangle : \langle \text{lit} \rangle$
 \Rightarrow
 $\langle \text{expr} \rangle : \langle \text{no} \rangle$
 \Rightarrow
 $\langle \text{expr} \rangle : \langle \text{var} \rangle$
 \Rightarrow
 $\langle \text{var} \rangle : \langle \text{id} \rangle$
 \Rightarrow

Appendix B

Computing Usage Counts

The tree transformations below implement common subexpression elimination using the usage count algorithm developed by Freiburghouse [Fre74]. There is only one action used, `findSame`, which checks all previous trees/left siblings within a linear region when trying to find a duplicate occurrence in which all the operands are still live. If `findSame` is successful in its search a link to the expression is returned so that the new symbol, `dupl`, has an attribute identifying which subtree it is referring to, i.e., what it is a duplicate of. `Dupl` is used as a place holder so that links to it can in turn trace where they refer to.

```
<program> : <slis> <eof>
           =>

<slis> : <assign> <slis>
        =>

<slis> :
        =>

<expr> : <no>
        ?findSame (<no>)↑link
        => dupl↓link

<expr> : <lit>
        ?findSame (<lit>)↑link
        => dupl↓link

<expr> : <var>
        ?findSame (<var>)↑link
        => dupl↓link
```

<expr> : <expr>
 ?findSame (<expr>)[↑]link
 ⇒ dupl[↓]link

<expr> : <binop>[↑]root dupl dupl
 ?findSame ([↓]root)[↑]link
 ⇒ dupl[↓]link

<expr> : <unaryop>[↑]root dupl
 ?findSame ([↓]root)[↑]link
 ⇒ dupl[↓]link

<assign> : assign <var> <expr>
 ⇒

<assign> : incr <var>
 ⇒

<assign> : decr <var>
 ⇒

<assign> : zero <var>
 ⇒

<unaryop> : Uminus
 ⇒

<unaryop> : not
 ⇒

<unaryop> : Lshift
 ⇒

<binop> : plus
 ⇒

<binop> : minus
 ⇒

<binop> : multiply
⇒

<binop> : eq
⇒

<binop> : le
⇒

<binop> : lt
⇒

<binop> : ne
⇒

<expr> : <unaryop> <expr>
⇒

<expr> : <binop> <expr> <expr>
⇒

<expr> : <lit>
⇒

<expr> : <no>
⇒

<expr> : <var>
⇒

<var> : <id>
⇒

<expr> : dupl
⇒

Appendix C

Instruction Selection

The tree transformations below describe instruction selection transformations which convert intermediate code into target machine code, similar to Ganapathi and Fischer [GaFi85]. The two address target machine language used is similar to the Zilog Z8000, which has a regular register set. The actions, `insertIdSymTab` and `insertArraySymTab`, insert the declared variables into the symbol table. If the intermediate code had been created by a compiler front end, the symbol table would have already existed. The `getRegister` action allocates a free pseudo register from the register table. The `decrUseCt` action decrements the usage count associated with an expression, i.e. a register, and frees the register if the usage count is zero.

```
<program> : <dlist> <slist> <eof>
           =>

<dlist> : <decl> <dlist>
         =>

<dlist> :
         =>

<decl> : decl <no> <id>
         @insertIdSymTab(<id>)
         =>

<decl> : decl <no> <id> <no>
         @insertArraySymTab(<id>)
         =>

<slist> : <assign> <slist>
         =>

<slist> :
         =>
```

<assign> : assign <var> <expr>
 ⇒ store <var> <expr>

<expr> : copy <expr>
 @getRegister(↑r)
 ⇒ loadR <reg>↓r <expr>

<expr> : plus <expr>↑useCount
 ?gtOne(↓useCount)
 ⇒ plus copy <expr>

<expr> : plus <expr>↑useCount <var>
 ?eqOne(↓useCount)
 ⇒ addDA <expr> <var>

<expr> : plus <expr₁>↑useCount <expr₂>
 ?eqOne(↓useCount)
 ⇒ addR <expr₁> <expr₂>

<expr> : <no>
 @getRegister(↑r)
 ⇒ loadIM reg↓r <no>

<expr> : <var>
 @getRegister(↑r)
 ⇒ loadDA reg↓r <var>

<var> : <id>
 ⇒ ind <id> <reg>↓r0

<var> : subscript <id> <expr>
 ⇒

<assign> : store <var> <expr>↑r
 @decrUseCt(↓r, <expr>↑r)
 ⇒

<expr> : addDA <expr>↑r <var>
 @decrUseCt(↓r, <expr>↑r)
 ⇒

$\langle \text{expr} \rangle : \text{addR } \langle \text{expr}_1 \rangle \uparrow r_1 \langle \text{expr}_2 \rangle \uparrow r_2$
 $\quad @\text{decrUseCt}(\downarrow r_1, \uparrow r_1) @\text{decrUseCt}(\downarrow r_2, \uparrow r_2)$
 \Rightarrow

$\langle \text{expr} \rangle : \text{loadDA } \langle \text{expr} \rangle \langle \text{var} \rangle$
 \Rightarrow

$\langle \text{expr} \rangle : \text{loadIM } \langle \text{expr} \rangle \langle \text{var} \rangle$
 \Rightarrow

$\langle \text{expr} \rangle : \text{loadR } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \uparrow r_2$
 $\quad @\text{decrUseCt}(\downarrow r_2, \uparrow r_2)$
 \Rightarrow

$\langle \text{var} \rangle : \text{ind } \langle \text{id} \rangle \text{ reg}$
 \Rightarrow

$\langle \text{expr} \rangle : \text{reg}$
 \Rightarrow

Appendix D

Selected Test Runs

In the examples shown below, the format is essentially the same. Each example starts with a prompt line asking for the name of the test case, followed by the user's response. The content of the test file is echoed along with the number of triples read. Each time the input, i.e. the triple table, is altered the triples are output after the "After Rewrite" message. Each row of output contains the triple number followed by either an operand and its value or an operation with two triple numbers representing its left and right operands respectively. Notice the format for echoing the input is different from displaying the new input after each rewrite. This is due to the fact that different print routines were called.

1. An Intermediate Code Optimization

The following example shows three classes of transformations, constant folding, comparison, and reordering performed on the input. If the source code expression was originally

```
v := not false = b;
```

then the following transformations, one per line, would be performed:

```
v := true = b;
```

```
v := b = true;
```

until the final input is of the form:

```
v := b;
```

The triple table is shown after each rewrite.

```
Enter the source filename: M4RPtrip.src
```

```
1 : id b
2 : lit false
3 : not 2
4 : eq 3, 1
5 : id v
6 : assign 5, 4
```

```
6 triples read.
```

After Rewrite

```
1      id      b
2
3      lit     TRUE
4      eq      3  1
5      id      v
6  assign    5  4
```

After Rewrite

```
1      id      b
2
3      lit     TRUE
4      eq      1  3
5      id      v
6  assign    5  4
```

After Rewrite

```
1
2
3
4      id      b
5      id      v
6  assign    5  4
```

2. Another Intermediate Code Optimization

This example performs constant folding, null sequence elimination and the use of specialized instructions. If the source code expression was originally

```
z := 5 + (a - a);    y := c - b;    x := x - (1 + (d * 0));
```

then the following transformations, one per line, would be performed:

```
z := 5 + 0;    y := c - b;    x := x - (1 + (d * 0));
z := 5;        y := c - b;    x := x - (1 + (d * 0));
z := 5;        y := c - b;    x := x - (1 + 0);
z := 5;        y := c - b;    x := x - 1;
```

until the final input is of the form:

```
z := 5;        y := c - b;    decr x;
```

The output on the next page shows the execution of the above example.

Enter the source filename: BRPtrip.src

```
1 : id a
2 : minus 1, 1
3 : no 5
4 : plus 3, 2
5 : id z
6 : assign 5, 4
7 : id b
8 : id c
9 : minus 8, 7
10 : id y
11 : assign 10, 9
12 : no 0
13 : id d
14 : multiply 13, 12
15 : no 1
16 : plus 15, 14
17 : id x
18 : minus 17, 16
19 : assign 17, 18
```

19 triples read.

After Rewrite

```
1
2      no      0
3      no      5
4      plus    3  2
5      id      z
6      assign  5  4
7      id      b
8      id      c
9      minus   8  7
10     id      y
11     assign  10 9
12     no      0
13     id      d
14 multiply  13 12
15     no      1
16     plus    15 14
17     id      x
18     minus   17 16
19     assign  17 18
```

After Rewrite

1			
2			
3			
4	no		5
5	id		z
6	assign	5	4
7	id		b
8	id		c
9	minus	8	7
10	id		y
11	assign	10	9
12	no		0
13	id		d
14	multiply	13	12
15	no		1
16	plus	15	14
17	id		x
18	minus	17	16
19	assign	17	18

After Rewrite

1			
2			
3			
4	no		5
5	id		z
6	assign	5	4
7	id		b
8	id		c
9	minus	8	7
10	id		y
11	assign	10	9
12			
13			
14	no		0
15	no		1
16	plus	15	14
17	id		x
18	minus	17	16
19	assign	17	18

After Rewrite

```
1
2
3
4     no      5
5     id      z
6  assign    5  4
7     id      b
8     id      c
9     minus   8  7
10    id      y
11  assign    10 9
12
13
14
15
16    no      1
17    id      x
18    minus   17 16
19  assign    17 18
```

After Rewrite

```
1
2
3
4     no      5
5     id      z
6  assign    5  4
7     id      b
8     id      c
9     minus   8  7
10    id      y
11  assign    10 9
12
13
14
15
16
17    id      x
18
19    decr    17
```

3. A Usage Count Rewrite

The following example eliminates the common subexpression

$a + b$

from the source code expressions:

```
x := a + b;      incr x;      y := a + b;
```

The output generated is in the same format as for the two previous examples with the following exception: when a duplicate expression is found, the triple table, i.e. the input, is printed in two stages. The first stage shows where the duplicate expression is. The second stage after the transformation is complete. This is **not** two transformations, it is simply extra output halfway through the single transformation.

Enter the source filename: UCcsetrip.src

```
1   : id a
2   : id b
3   : plus 1, 2
4   : id x
5   : assign 4, 3
6   : id w
7   : incr 6
8   : id a
9   : id b
10  : plus 8, 9
11  : id y
12  : assign 11, 10
```

12 triples read.

After Rewrite

1st Stage:

```
1      id      a
2      id      b
3      plus    1 2
4      id      x
5      assign  4 3
6      id      w
7      incr    6
8      dupl    1
9      id      b
10     plus    8 9
11     id      y
12     assign  11 10
```

2nd Stage:

1	id		a
2	id		b
3	plus	1	2
4	id		x
5	assign	4	3
6	id		w
7	incr	6	
8			
9	id		b
10	plus	1	9
11	id		y
12	assign	11	10

After Rewrite

1st Stage:

1	id		a
2	id		b
3	plus	1	2
4	id		x
5	assign	4	3
6	id		w
7	incr	6	
8			
9	dupl		2
10	plus	1	9
11	id		y
12	assign	11	10

2nd Stage:

1	id		a
2	id		b
3	plus	1	2
4	id		x
5	assign	4	3
6	id		w
7	incr	6	
8			
9			
10	plus	1	2
11	id		y
12	assign	11	10

After Rewrite

1st Stage:

1	id		a
2	id		b
3	plus	1 2	
4	id		x
5	assign	4 3	
6	id		w
7	incr	6	
8			
9			
10	dupl		3
11	id		y
12	assign	11 10	

2nd Stage:

1	id		a
2	id		b
3	plus	1 2	
4	id		x
5	assign	4 3	
6	id		w
7	incr	6	
8			
9			
10			
11	id		y
12	assign	11 3	

4. Another Usage Count Rewrite

The following example eliminates the common subexpression

b
from the original input

```
x := a + b;      incr a;      y := a + b;
```

Note that in this example the intervening instruction `incr a` affects the value of `a + b` so it would no longer have the same value as when it was assigned to `x`. In this case `b` is the only common subexpression.

Enter the source filename: UCncsetrip.src

```
1 : id a
2 : id b
3 : plus 1, 2
4 : id x
5 : assign 4, 3
6 : id a
7 : incr 6
8 : id a
9 : id b
10 : plus 8, 9
11 : id y
12 : assign 11, 10
```

12 triples read.

After Rewrite

1st Stage:

```
1      id      a
2      id      b
3      plus    1 2
4      id      x
5      assign  4 3
6      dupl    1
7      incr    6
8      id      a
9      id      b
10     plus    8 9
11     id      y
12     assign  11 10
```

2nd Stage:

```
1      id      a
2      id      b
3      plus    1 2
4      id      x
5      assign  4 3
6
7      incr    1
8      id      a
9      id      b
10     plus    8 9
11     id      y
12     assign  11 10
```

After Rewrite

1st Stage:

1	id	a
2	id	b
3	plus	1 2
4	id	x
5	assign	4 3
6		
7	incr	1
8	id	a
9	dupl	2
10	plus	8 9
11	id	y
12	assign	11 10

2nd Stage:

1	id	a
2	id	b
3	plus	1 2
4	id	x
5	assign	4 3
6		
7	incr	1
8	id	a
9		
10	plus	8 2
11	id	y
12	assign	11 10

5. An Instruction Selection Rewrite

The initial input in the instruction selection phase is slightly modified from earlier phases. A usage count is "passed" forward from the previous phase. Since three separate grammars were used this usage count was added by hand. Therefore the usage count appears as the last number in the column. It does not appear in subsequent rewrites by function of the routine that displays the input.

The following example transforms the source code instructions:

```
z := b + a;    w := c;    x := (common subexpression a + b)
```

into their equivalent machine code representation. The register used to hold the $b + a$ is reused in the assignment to x .

Enter the source filename: cg2atrip.src

```
1  : decl int 1 a
2  : decl int 1 b
3  : decl int 1 c
4  : decl int 1 w
5  : decl int 1 x
6  : decl int 1 z
7  : id a           1
8  : id b           1
9  : plus 8, 7      2
10 : id z           1
11 : assign 10, 9   0
12 : id c           1
13 : id w           1
14 : assign 13, 12  0
15 : id x           1
16 : assign 15, 9   0
```

16 triples read.

After Rewrite

```
1  decl      a
2  decl      b
3  decl      c
4  decl      w
5  decl      x
6  decl      z
7  id        a
8  id        b
9  plus      8 7
10 ind       17 18
11 assign    10 9
12 id        c
13 id        w
14 assign    13 12
15 id        x
16 assign    15 9
17 id        z
18 reg       0
```

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	id		a
8	ind	19 20	
9	plus	8 7	
10	ind	17 18	
11	assign	10 9	
12	id		c
13	id		w
14	assign	13 12	
15	id		x
16	assign	15 9	
17	id		z
18	reg		0
19	id		b
20	reg		0

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	id		a
8	loadDA	22 21	
9	plus	8 7	
10	ind	17 18	
11	assign	10 9	
12	id		c
13	id		w
14	assign	13 12	
15	id		x
16	assign	15 9	
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19 20	
22	reg		14

After Rewrite

1	decl	a
2	decl	b
3	decl	c
4	decl	w
5	decl	x
6	decl	z
7	ind	23 24
8	loadDA	22 21
9	plus	8 7
10	ind	17 18
11	assign	10 9
12	id	c
13	id	w
14	assign	13 12
15	id	x
16	assign	15 9
17	id	z
18	reg	0
19	id	b
20	reg	0
21	ind	19 20
22	reg	14
23	id	a
24	reg	0

After Rewrite

1	decl	a
2	decl	b
3	decl	c
4	decl	w
5	decl	x
6	decl	z
7	ind	23 24
8	loadDA	22 21
9	addDA	8 7
10	ind	17 18
11	assign	10 9
12	id	c
13	id	w
14	assign	13 12
15	id	x
16	assign	15 9
17	id	z
18	reg	0
19	id	b
20	reg	0
21	ind	19 20
22	reg	14
23	id	a
24	reg	0

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	ind	23	24
8	loadDA	22	21
9	addDA	8	7
10	ind	17	18
11	store	10	9
12	id		c
13	id		w
14	assign	13	12
15	id		x
16	assign	15	9
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19	20
22	reg		14
23	id		a
24	reg		0

After Rewrite

1	decl	a
2	decl	b
3	decl	c
4	decl	w
5	decl	x
6	decl	z
7	ind	23 24
8	loadDA	22 21
9	addDA	8 7
10	ind	17 18
11	store	10 9
12	id	c
13	ind	25 26
14	assign	13 12
15	id	x
16	assign	15 9
17	id	z
18	reg	0
19	id	b
20	reg	0
21	ind	19 20
22	reg	14
23	id	a
24	reg	0
25	id	w
26	reg	0

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	ind	23 24	
8	loadDA	22 21	
9	addDA	8 7	
10	ind	17 18	
11	store	10 9	
12	ind	27 28	
13	ind	25 26	
14	assign	13 12	
15	id		x
16	assign	15 9	
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19 20	
22	reg		14
23	id		a
24	reg		0
25	id		w
26	reg		0
27	id		c
28	reg		0

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	ind	23	24
8	loadDA	22	21
9	addDA	8	7
10	ind	17	18
11	store	10	9
12	loadDA	30	29
13	ind	25	26
14	assign	13	12
15	id		x
16	assign	15	9
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19	20
22	reg		14
23	id		a
24	reg		0
25	id		w
26	reg		0
27	id		c
28	reg		0
29	ind	27	28
30	reg		13

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	ind	23	24
8	loadDA	22	21
9	addDA	8	7
10	ind	17	18
11	store	10	9
12	loadDA	30	29
13	ind	25	26
14	store	13	12
15	id		x
16	assign	15	9
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19	20
22	reg		14
23	id		a
24	reg		0
25	id		w
26	reg		0
27	id		c
28	reg		0
29	ind	27	28
30	reg		13

After Rewrite

1	decl		a
2	decl		b
3	decl		c
4	decl		w
5	decl		x
6	decl		z
7	ind	23	24
8	loadDA	22	21
9	addDA	8	7
10	ind	17	18
11	store	10	9
12	loadDA	30	29
13	ind	25	26
14	store	13	12
15	ind	31	32
16	assign	15	9
17	id		z
18	reg		0
19	id		b
20	reg		0
21	ind	19	20
22	reg		14
23	id		a
24	reg		0
25	id		w
26	reg		0
27	id		c
28	reg		0
29	ind	27	28
30	reg		13
31	id		x
32	reg		0

After Rewrite

1	decl	a
2	decl	b
3	decl	c
4	decl	w
5	decl	x
6	decl	z
7	ind	23 24
8	loadDA	22 21
9	addDA	8 7
10	ind	17 18
11	store	10 9
12	loadDA	30 29
13	ind	25 26
14	store	13 12
15	ind	31 32
16	store	15 9
17	id	z
18	reg	0
19	id	b
20	reg	0
21	ind	19 20
22	reg	14
23	id	a
24	reg	0
25	id	w
26	reg	0
27	id	c
28	reg	0
29	ind	27 28
30	reg	13
31	id	x
32	reg	0

6. Another Instruction Selection Rewrite

The following example transforms the source code instructions
z := b + a; x := (common subexpression b)
into their equivalent machine code representation. The register used to hold the id b must be copied to be reused in the assignment to x, otherwise it would be corrupted by the addition operation. The last number in the column is the usage count passed from an earlier transformation.

Enter the source filename: cg2btrip.src

```
1 : decl int 1 a
2 : decl int 1 b
3 : decl int 1 z
4 : decl int 1 x
5 : id a 1
6 : id b 2
7 : plus 6, 5 1
8 : id z 1
9 : assign 8, 7 0
10 : id x 1
11 : assign 10, 6 0
```

11 triples read.

After Rewrite

```
1 decl a
2 decl b
3 decl z
4 decl x
5 id a
6 id b
7 plus 6 5
8 ind 25 26
9 assign 8 7
10 id x
11 assign 10 6
12 id z
13 reg 0
```

After Rewrite

```
1 decl a
2 decl b
3 decl z
4 decl x
5 id a
6 ind 14 15
7 plus 6 5
8 ind 25 26
9 assign 8 7
10 id x
11 assign 10 6
12 id z
13 reg 0
14 id b
15 reg 0
```

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	id		a
6	loadDA	17	16
7	plus	6	5
8	ind	25	26
9	assign	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	plus	6	5
8	ind	25	26
9	assign	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	plus	20	5
8	ind	25	26
9	assign	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	copy	6	

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	plus	20	5
8	ind	25	26
9	assign	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	loadR	21	6
21	reg		13

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	addDA	20	5
8	ind	25	26
9	assign	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	loadR	21	6
21	reg		13

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	addDA	20	5
8	ind	25	26
9	store	8	7
10	id		x
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	loadR	21	6
21	reg		13

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	addDA	20	5
8	ind	25	26
9	store	8	7
10	ind	22	23
11	assign	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	loadR	21	6
21	reg		13
22	id		x
23	reg		0

After Rewrite

1	decl		a
2	decl		b
3	decl		z
4	decl		x
5	ind	18	19
6	loadDA	17	16
7	addDA	20	5
8	ind	25	26
9	store	8	7
10	ind	22	23
11	store	10	6
12	id		z
13	reg		0
14	id		b
15	reg		0
16	ind	14	15
17	reg		14
18	id		a
19	reg		0
20	loadR	21	6
21	reg		13
22	id		x
23	reg		0

Appendix E

Reparse Backup Example Revisited

In this appendix, we show by example why reparse backup continues one nonterminal past the root of the source tree. This appendix continues the reparse backup example of section 3, Chapter 7 and is not a self-contained appendix. This appendix considers what happens in the section 3 example if the "nonterminal backup" rule is not followed. Our discussion begins in the middle of the section 3 example, with the parse state shown in Figure 7.4 of Chapter 7. The reader is advised to read section 3, Chapter 7 up to that point, and then continue with this appendix.

Consider the tree transformation for the decrement instruction,

$$\begin{aligned} \langle \text{ASSIGN} \rangle &: \text{assign } \langle \text{var}_1 \rangle \text{ minus } \langle \text{var}_2 \rangle \langle \text{no} \rangle \uparrow \text{val} \\ &\quad ?\text{same}(\langle \text{var}_1 \rangle, \langle \text{var}_2 \rangle) ?\text{equal}(\downarrow \text{val}, 1) \\ &\quad \Rightarrow \text{decr } \langle \text{var}_1 \rangle \end{aligned}$$

shown previously in part (b) of Figure 7.6, the original input, before constant folding

```
assign x minus x plus 1 0
```

and the parse state before constant folding shown in Figure 7.4.

Notice that in Figure 7.4 the second `x` in the original input has been reduced to `<expr>`. This reduction took place on the basis of the `plus` symbol in the input and the reduction rules out the application of the decrement instruction transformation, which requires a `<var>` where we now have an `<expr>`. However, the constant folding transformation removes the `plus` from the input as if it never existed, in effect treating it as a syntactic error. But if we do not backup one nonterminal past the root of the source tree, the parse state in Figure E.1 results.

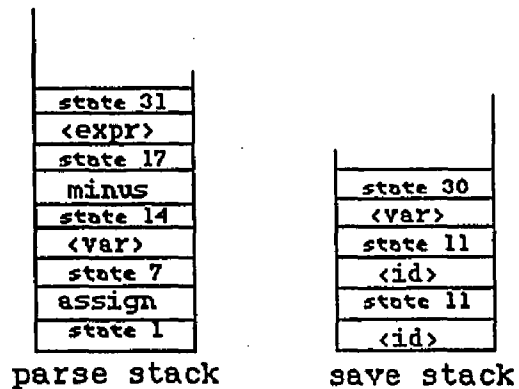


Figure E.1: Parse State after Incorrect Reparse Backup over Constant Folding Transformation

This parse state is bad in three respects. First, it does not represent the state that would result on the input `assign x minus x 1` because the reduction of `x` to `<expr>` remains in force. Secondly, this parse state results in suboptimal code as the tree transformation for the decrement instruction will not apply. Finally, the end result of such an incongruous parse state can be disastrous, as the code generator could fail with parsing errors.

In the case shown in Figure E.1, chance has thrown the parser into a state from which it can recover, using the transformations shown in Figure E.2, but suboptimal code results.

- 1) <EXPR> : <no> ⇒
- 2) <EXPR> : minus <expr₁> <expr₂>
 ?same(<expr₁>, <expr₂>) ⇒ <no>↓0
- 3) <EXPR> : minus <expr₁> <expr₂> ⇒
- 4) <ASSIGN> : assign <var> <expr> ⇒

Figure E.2: Expression Transformations

The input produced by the constant fold transformation is 1. The input is recognized as a `<no>` and subsequently reduced to an `<expr>` by rule 1 of Figure E.2, which results in the parse state shown below in Figure E.3

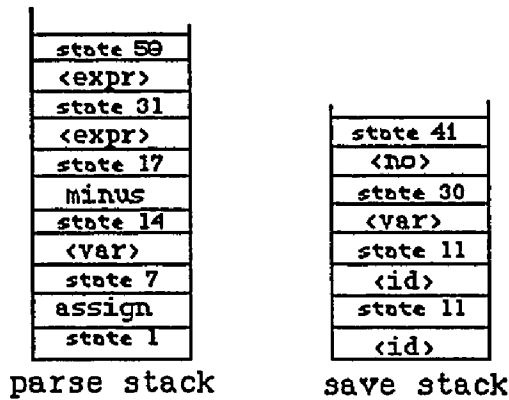


Figure E.3: Parse State after Reduce by Rule 1

In the parse state of Figure E.3, there is a possibility of reducing by rule 2 or rule 3, but since the expressions are not the same the semantic conditions of rule 2 are not met, and we reduce by rule 3 which results in the parse state shown in Figure E.4.

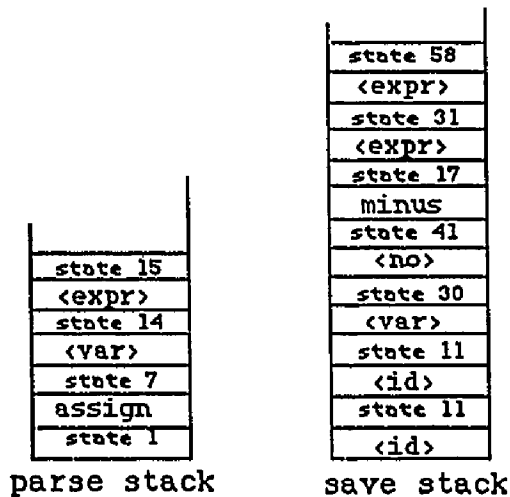


Figure E.4: Parse State after Reduce by Rule 2

The parse state of Figure E.4 allows application of rule 4 which will reduce the parse stack to <assign>. This is not the desired result of `decr x`. Subsequent instruction selection transformations on the IC will not produce a decrement instruction, but will produce machine code to compute the subtraction and store the result in `x`.

Appendix F

TTL and a Backtracking LR Parser Produce a Recognizer for any Recursively Enumerable Language

In this appendix, we show that a TTL specification for a backtracking LR parser produces a recognizer for any recursively enumerable language.

1. Approach and Notation

Languages can be defined two ways: by specifying a *generator* for the language, or by defining a *recognizer* for the language. For example, a grammar is a language generator whereas a push-down automaton is a recognizer. The following table defines the Chomsky hierarchy of languages by generator and by recognizer [AhU172, p. 96]:

<u>GENERATOR</u>	<u>RECOGNIZER</u>
Right-Linear Grammar automaton	one-way deterministic finite
Context-Free Grammar	one-way nondeterministic pushdown automaton.
Context-sensitive Grammar bounded	two-way nondeterministic linear automaton.
Unrestricted Grammar	a Turing machine.

Figure F.1: Language Generators and Recognizers

A language L is recursively enumerable if L is generated by an unrestricted grammar or if L is recognized by a Turing Machine.

The approach taken in this appendix is based on the observation that the question, "can a TTL specification and a backtracking LR parser define a recognizer for any recursively enumerable (unrestricted) language?" is equivalent to the question "can a TTL specification and

a backtracking LR parser simulate a Turing machine?" The point being that if a TTL specification and a backtracking LR parser can simulate any Turing machine, then it can recognize any recursively enumerable language by simulating the Turing machine that recognizes the language.

To answer the question "can a TTL specification and a backtracking LR parser simulate a Turing machine", we give a general algorithm which translates a Turing machine into a TTL specification, as shown in Figure F.2.

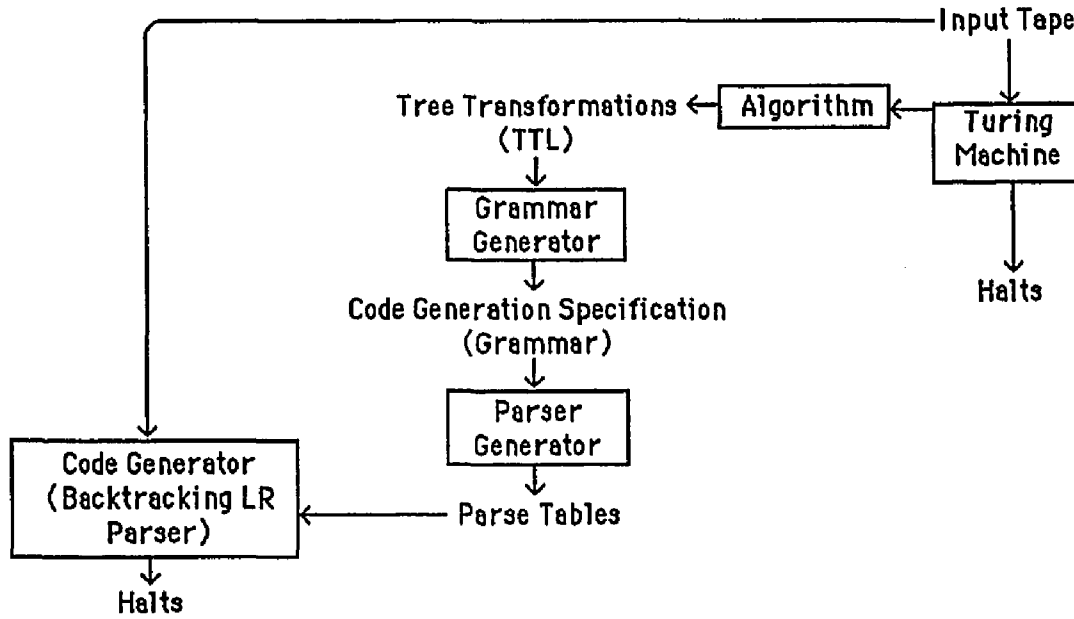


Figure F.2: Turing Machine Simulation

A Turing machine is formally denoted by [HoU179]:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where

Q is the finite set of states,

Γ is the finite set of allowable tape symbols,

B , a symbol of Γ , is the blank,

Σ , a subset of Γ not including B , is the set of input symbols,

δ is the next move function, a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L,R\}$

(δ may be undefined for some arguments),

q_0 in Q is the start state,

F is a subset of Q is the set of final states.

The Turing machine operates with a tape head that scans one symbol on the input tape at a time. On each scan, the Turing machine changes state, prints a new symbol on the tape to replace the scanned symbol, and moves its tape head left or right. The next move function, δ , defines the new state, new symbol, and tape head movement based on the scanned symbol and the current state.

An *instantaneous description* (ID) of a Turing machine is a representation of the current state of the machine as $\alpha_1 q_i \alpha_2$ where q_i is the current state of M , α_1 is the contents of the tape up to the symbol to the left of the tape head, and α_2 is the contents of the tape from the tape head to the right. The leftmost symbol in α_2 is the symbol currently scanned by the tape head.

We simulate a Turing machine by parsing the instantaneous description of the Turing machine and simulating the effect of the Turing machine's next move function on the instantaneous description. Consequently, Figure F.2 is slightly misleading in that the input tape to the Turing machine is not passed directly to the backtracking LR parser, rather the input tape is converted into an instantaneous description of the initial state of the Turing machine which is passed to the backtracking LR parser as shown in Figure F.3.

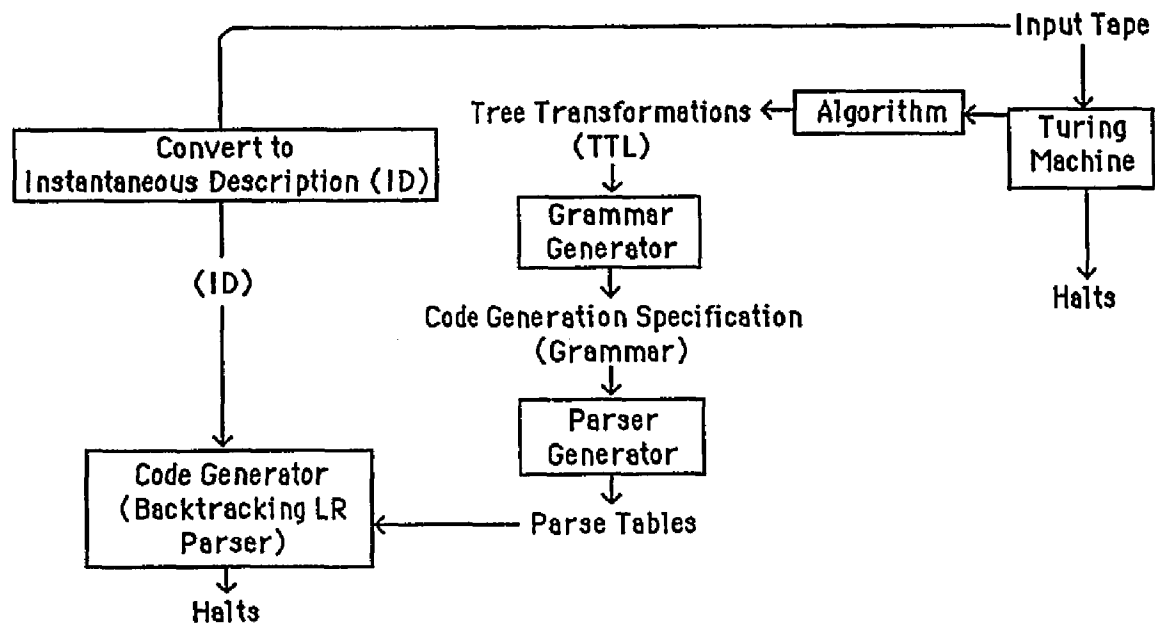


Figure F.3: Turing Machine Simulation Using ID

The remainder of this appendix describes first an example simulation of a Turing machine and then a general algorithm which converts a Turing machine into a TTL specification for a backtracking LR parser.

2. Turing Machine to TTL Specification Algorithm

The TTL specification of the example in Section 3 has a very regular structure, and in what follows we give a general algorithm for translating a Turing machine into a TTL specification for a backtracking LR parser that will simulate the Turing machine.

The most important aspect of simulating a Turing machine is representing the next move function δ . For each possible Turing machine move in δ , the following TTL rules are generated:

$$\delta(q, x) = (q', x', R)$$

$$\langle \text{move} \rangle: qx \rightarrow x'q'$$

$$\delta(q, y) = (q', y', L)$$

$$\langle \text{move} \rangle: qy \rightarrow \sigma y'$$

and

$\forall a: a \in \Gamma$, generate the rule:

$$\langle \text{move} \rangle: a\sigma \rightarrow q'a$$

where σ is a unique symbol

Thus moving the tape head to the right generates a single TTL rule, whereas moving the tape head to the left generates $|\Gamma| + 1$ TTL rules.

Several additional TTL rules are also required to complete the TTL specification such that the parser generator system used will create a backtracking LR parser that halts when the Turing machine simulation ends in a halt state. These are as follows:

$\langle \text{Halt} \rangle \quad : \langle \text{final} \rangle \langle \text{eof} \rangle \rightarrow$

and $\forall a: a \in \Gamma$, generate the two rules:

$\langle \text{final} \rangle: a \langle \text{final} \rangle \rightarrow \langle \text{final} \rangle$

$\langle \text{final} \rangle: \langle \text{final} \rangle a \rightarrow \langle \text{final} \rangle$

The latter pairs of rules enable the parser to clear the parse stack which is a requirement for the parser to halt. Rules are also required which specify the final states as follows:

$\forall q: q \in F$, generate the rule:

$\langle \text{final} \rangle: q \rightarrow$

Finally, Turing machines require an input tape which is infinite to the right and initially filled with blanks. Although we might require that the initial instantaneous description input to the parser be padded on the right with a sufficient number of blanks to enable the Turing machine to operate (for the given input string), an infinite tape can be simulated by including the TTL rules:

$\forall q: q \in Q$, generate the rule:

$\langle \text{move} \rangle: q \langle \text{eof} \rangle \rightarrow qB \langle \text{eof} \rangle$

The preceding algorithm is general as it makes no assumptions about the Turing machine converted to TTL or the language the Turing machine recognizes. Consequently, the algorithm can be used to convert any Turing machine into a TTL specification for a backtracking LR parser that simulates the Turing machine.

3. Turing Machine Simulation Example

This example illustrates how a Turing machine recognizing the language L , $L = a^n b^n c^n$, can be simulated using a TTL specification and a backtracking LR parser. A Turing machine recognizing $L = a^n b^n c^n$ is given below:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}\}$$

$$\Gamma = \{B, a, b, c, d\}$$

$$\Sigma = \{a, b, c, d\}$$

$$F = \{q_{12}\}$$

$\delta(q_0, a)$	$= (q_1, B, R)$	(* Erase a *)
$\delta(q_1, a)$	$= (q_1, a, R)$	(* Find matching b *)
$\delta(q_1, b)$	$= (q_2, b, R)$	
$\delta(q_2, b)$	$= (q_2, b, R)$	
$\delta(q_2, c)$	$= (q_3, c, L)$	
$\delta(q_2, d)$	$= (q_3, d, L)$	
$\delta(q_3, b)$	$= (q_4, d, L)$	(* Erase b and write d *)
$\delta(q_4, b)$	$= (q_4, b, L)$	(* Back to start *)
$\delta(q_4, a)$	$= (q_5, a, L)$	
$\delta(q_5, a)$	$= (q_5, a, L)$	
$\delta(q_5, B)$	$= (q_0, B, R)$	(* Begin again *)
$\delta(q_4, B)$	$= (q_6, B, R)$	(* Must match b's(now d's) and c's*)
$\delta(q_6, d)$	$= (q_7, B, R)$	(* Erase d *)
$\delta(q_7, d)$	$= (q_7, d, R)$	(* Find matching c *)
$\delta(q_7, c)$	$= (q_8, c, R)$	
$\delta(q_8, c)$	$= (q_8, c, R)$	

$\delta(q_8, B)$	$= (q_9, B, L)$	(* Erase c *)
$\delta(q_9, c)$	$= (q_{10}, B, L)$	
$\delta(q_{10}, c)$	$= (q_{10}, c, L)$	(* Back to start of "bc"
$\delta(q_{10}, d)$	$= (q_{11}, d, L)$	match*)
$\delta(q_{11}, d)$	$= (q_{11}, d, L)$	
$\delta(q_{11}, B)$	$= (q_6, B, R)$	(* Begin again *)
$\delta(q_{10}, B)$	$= (q_{12}, B, R)$	(* Halt *)

This Turing machine matches a's and b's, replacing a's with blanks and the matching b's with d's, then matches the d's and c's replacing the d's with blanks and the matching c's with blanks as well. The Turing machine's operation on a sample input is shown below using the instantaneous description notation.

Example Operation:

Bq ₀ aabbccBBBBBBB.....			
BBq ₁ abbccB	BBaq ₁ bbccB	BBabq ₂ bccB	BBabbq ₂ cc
BBabq ₃ bccB	BBaq ₄ bdccB	BBq ₄ abdccB	Bq ₅ BabdccB
BBq ₀ abdccB	BBBq ₁ bdccB	BBBbq ₂ dccB	BBBq ₃ bdccB
BBq ₄ BddccB	BBBq ₆ ddccB	BBBBq ₇ dccB	BBBBdq ₇ ccB
BBBBdcq ₈ cB	BBBBdcccq ₈ B	BBBBdcq ₉ cB	BBBBdq ₁₀ cBB
BBBBq ₁₀ dcBB	BBBq ₁₁ BdcBB	BBBBq ₆ dcBB	BBBBBq ₇ cBB
BBBBBcq ₈ BB	BBBBBq ₉ cBB	BBBBq ₁₀ BBBB	BBBBBq ₁₂ BBB.....

To simulate the Turing machine, we write a TTL description which parses the instantaneous description, using rewrite trees to model how the Turing machine's next move function affects the instantaneous description. For example, we model

$$\delta(q_0, a) = (q_1, B, R)$$

with the TTL rule,

$$\langle \text{move} \rangle: q_0a \rightarrow Bq_1$$

so that when the parser pattern matches input of the form q₀a, the parser substitutes input of the form Bq₁ using the TTL rule. The complete TTL rules for the Turing machine recognizing $L = a^n b^n c^n$ follow:

Equivalent TTL:

$\langle \text{Halt} \rangle$: $\langle \text{final} \rangle \langle \text{eof} \rangle \rightarrow$

$\langle \text{move} \rangle$: $q_0a \rightarrow Bq_1$

$\langle \text{move} \rangle$: $q_1a \rightarrow aq_1$

$\langle \text{move} \rangle$: $q_1b \rightarrow bq_2$

$\langle \text{move} \rangle$: $q_2b \rightarrow bq_2$

$\langle \text{move} \rangle$: $q_2c \rightarrow \sigma_1c$

$\langle \text{move} \rangle$: $a\sigma_1 \rightarrow q_3a$

$\langle \text{move} \rangle$: $b\sigma_1 \rightarrow q_3b$

$\langle \text{move} \rangle$: $c\sigma_1 \rightarrow q_3c$

$\langle \text{move} \rangle$: $d\sigma_1 \rightarrow q_3d$

$\langle \text{move} \rangle$: $B\sigma_1 \rightarrow q_3B$

$\langle \text{move} \rangle$: $q_2d \rightarrow \sigma_2d$

$\langle \text{move} \rangle$: $a\sigma_2 \rightarrow q_3a$

$\langle \text{move} \rangle$: $b\sigma_2 \rightarrow q_3b$

$\langle \text{move} \rangle$: $c\sigma_2 \rightarrow q_3c$

$\langle \text{move} \rangle$: $d\sigma_2 \rightarrow q_3d$

$\langle \text{move} \rangle$: $B\sigma_2 \rightarrow q_3B$

$\langle \text{move} \rangle$: $q_3b \rightarrow \sigma_3d$

$\langle \text{move} \rangle$: $a\sigma_3 \rightarrow q_4a$

$\langle \text{move} \rangle$: $b\sigma_3 \rightarrow q_4b$

$\langle \text{move} \rangle$: $c\sigma_3 \rightarrow q_3c$

$\langle \text{move} \rangle$: $d\sigma_3 \rightarrow q_3d$

$\langle \text{move} \rangle$: $B\sigma_3 \rightarrow q_4B$

$\langle \text{move} \rangle$: $q_4b \rightarrow \sigma_4b$

$\langle \text{move} \rangle$: $a\sigma_4 \rightarrow q_4a$

$\langle \text{move} \rangle$: $b\sigma_4 \rightarrow q_4b$

$\langle \text{move} \rangle$: $c\sigma_4 \rightarrow q_4c$

$\langle \text{move} \rangle$: $d\sigma_4 \rightarrow q_4d$

$\langle \text{move} \rangle$: $B\sigma_4 \rightarrow q_4B$

$\langle \text{move} \rangle$: $q_4a \rightarrow \sigma_5a$

$\langle \text{move} \rangle$: $a\sigma_5 \rightarrow q_5a$

<move>: $b\sigma_5 \rightarrow q_5b$
<move>: $c\sigma_5 \rightarrow q_5c$
<move>: $d\sigma_5 \rightarrow q_5d$
<move>: $B\sigma_5 \rightarrow q_5B$

<move>: $q_5a \rightarrow \sigma_6a$
<move>: $a\sigma_6 \rightarrow q_5a$
<move>: $b\sigma_6 \rightarrow q_5b$
<move>: $c\sigma_6 \rightarrow q_5c$
<move>: $d\sigma_6 \rightarrow q_5d$
<move>: $B\sigma_6 \rightarrow q_5B$

<move>: $q_5B \rightarrow Bq_0$

<move>: $q_4B \rightarrow Bq_6$

<move>: $q_6d \rightarrow Bq_7$
<move>: $q_7d \rightarrow dq_7$
<move>: $q_7c \rightarrow cq_8$
<move>: $q_8c \rightarrow cq_8$

<move>: $q_8B \rightarrow \sigma_7B$
<move>: $a\sigma_7 \rightarrow q_9a$
<move>: $b\sigma_7 \rightarrow q_9b$
<move>: $c\sigma_7 \rightarrow q_9c$
<move>: $d\sigma_7 \rightarrow q_9d$
<move>: $B\sigma_7 \rightarrow q_9B$

<move>: $q_9c \rightarrow \sigma_8B$
<move>: $a\sigma_8 \rightarrow q_{10a}$
<move>: $b\sigma_8 \rightarrow q_{10b}$
<move>: $c\sigma_8 \rightarrow q_{10c}$
<move>: $d\sigma_8 \rightarrow q_{10d}$
<move>: $B\sigma_8 \rightarrow q_{10B}$

<move>: $q_{10c} \rightarrow \sigma_9c$
<move>: $a\sigma_9 \rightarrow q_{10a}$
<move>: $b\sigma_9 \rightarrow q_{10b}$

<move>: $c\sigma_9 \rightarrow q_{10}c$
<move>: $d\sigma_9 \rightarrow q_{10}d$
<move>: $B\sigma_9 \rightarrow q_{10}B$

<move>: $q_{10}d \rightarrow \sigma_{10}d$
<move>: $a\sigma_{10} \rightarrow q_{11}a$
<move>: $b\sigma_{10} \rightarrow q_{11}b$
<move>: $c\sigma_{10} \rightarrow q_{11}c$
<move>: $d\sigma_{10} \rightarrow q_{11}d$
<move>: $B\sigma_{10} \rightarrow q_{11}B$

<move>: $q_{11}d \rightarrow \sigma_{11}d$
<move>: $a\sigma_{11} \rightarrow q_{11}a$
<move>: $b\sigma_{11} \rightarrow q_{11}b$
<move>: $c\sigma_{11} \rightarrow q_{11}c$
<move>: $d\sigma_{11} \rightarrow q_{11}d$
<move>: $B\sigma_{11} \rightarrow q_{11}B$

<move>: $q_{11}B \rightarrow Bq_6$

<move>: $q_{10}B \rightarrow Bq_{12}$

<final> : $q_{12} \rightarrow$

<final> : $a \langle \text{final} \rangle \rightarrow$
<final> : $\langle \text{final} \rangle a \rightarrow$
<final> : $b \langle \text{final} \rangle \rightarrow$
<final> : $\langle \text{final} \rangle b \rightarrow$
<final> : $c \langle \text{final} \rangle \rightarrow$
<final> : $\langle \text{final} \rangle c \rightarrow$
<final> : $d \langle \text{final} \rangle \rightarrow$
<final> : $\langle \text{final} \rangle d \rightarrow$
<final> : $B \langle \text{final} \rangle \rightarrow$
<final> : $\langle \text{final} \rangle B \rightarrow$

The parse trace that follows shows action of the backtracking LR parser that results from the preceding TTL specification. The input to the parser is the instantaneous description for the initial state of the Turing machine for the string aabbcc.

Parse Trace:

Stack (Bottom to Top)

Bq₀a
B
BBq₁a
BB
BBaq₁b
BBa
BBabq₂b
BBab
BBabbq₂c
BBabb
BBabbσ₁
BBab
BBabq₃b
BBab
BBabσ₃
BBa
BBaq₄b
BBa
BBaσ₄
BB
BBq₄a
BB
BBσ₅
B
Bq₅B
B
BBq₀a
BB
BBBq₁b
BBB
BBBbq₂d
BBBb
BBBbσ₂
BBB
BBBq₃b
BBB

Input

Bq₀aabbccB<eof>
abbccB<eof>
Bq₁abbccB<eof>
bbccB<eof>
aq₁bbccB<eof>
bccB<eof>
bq₂bccB<eof>
ccB<eof>
bq₂ccB<eof>
cB<eof>
σ₁ccB<eof>
ccB<eof>
q₃bccB<eof>
ccB<eof>
σ₃dccB<eof>
dccB<eof>
q₄bdccB<eof>
dccB<eof>
σ₄bdccB<eof>
bdccB<eof>
q₄abdccB<eof>
bdccB<eof>
σ₅abdccB<eof>
abdccB<eof>
q₅BabdccB<eof>
abdccB<eof>
Bq₀abdccB<eof>
bdccB<eof>
Bq₁bdccB<eof>
dccB<eof>
bq₂dccB<eof>
ccB<eof>
σ₂dccB<eof>
dccB<eof>
q₃bdccB<eof>
dccB<eof>
σ₃ddccB<eof>

BBB σ_3	ddccB<eof>
BB	q ₄ BddccB<eof>
BBq ₄ B	ddccB<eof>
BB	Bq ₆ ddccB<eof>
BBBq ₆ d	dccB<eof>
BBB	Bq ₇ dccB<eof>
BBBBq ₇ d	ccB<eof>
BBBB	dq ₇ ccB<eof>
BBBBdq ₇ c	cB<eof>
BBBBd	cq ₈ cB<eof>
BBBBdcq ₈ c	B<eof>
BBBBdc	cq ₈ B<eof>
BBBBdccq ₈ B	<eof>
BBBBdcc	σ_7 B<eof>
BBBBdcc σ_7	B<eof>
BBBBdc	q ₉ cB<eof>
BBBBdcq ₉ c	B<eof>
BBBBdc	σ_8 BB<eof>
BBBBdc σ_8	BB<eof>
BBBBd	q ₁₀ cBB<eof>
BBBBdq ₁₀ c	BB<eof>
BBBBd	σ_9 cBB<eof>
BBBBd σ_9	cBB<eof>
BBBB	q ₁₀ dcBB<eof>
BBBBq ₁₀ d	cBB<eof>
BBBB	σ_{10} dcBB<eof>
BBBB σ_{10}	dcBB<eof>
BBB	q ₁₁ BdcBB<eof>
BBBq ₁₁ B	dcBB<eof>
BBB	Bq ₆ dcBB<eof>
BBBBq ₆ d	cBB<eof>
BBBB	Bq ₇ cBB<eof>
BBBBBq ₇ c	BB<eof>
BBBBB	cq ₈ BB<eof>
BBBBBcq ₈ B	B<eof>
BBBBBc	σ_7 BB<eof>
BBBBBc σ_7	BB<eof>
BBBBB	q ₉ cBB<eof>
BBBBBq ₉ c	BB<eof>
BBBBB	σ_8 BBB<eof>
BBBBB σ_8	BBB<eof>

BBBB	q ₁₀ BBBB<eof>
BBBBq ₁₀ B	BBB<eof>
BBBB	Bq ₁₂ BBB<eof>
BBBBBq ₁₂	BBB<eof>
BBBBB<final>	BBB<eof>
⋮	
⋮	
<final><eof>	
<Halt>	

The parse can be viewed as operating in two phases. The first phase simulates the Turing machine until a final state is reached, and the second phase clears the parse stack and the input so that the parser will halt.

Bibliography

- Aho, A.V. and M. Ganapathi, "Efficient Tree Pattern Matching: An Aid to Code Generation," Proceedings 12th POPL Conference, ACM, 1985, Pages 334-340.
- Aho, A.V. , M. Ganapathi and Steven Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", ACM TOPLAS, Vol. 11, No. 4, October 1989, Pages 491-516.
- Aho, A.V. and S.C. Johnson, "Optimal Code Generation for Expression Trees," Journal of the ACM, vol. 23, No. 3, 1976, Pages 488-501. Also in Proceedings of ACM symposium on Theory of Computing, 1975, Pages. 207-217.
- Aho, A.V., S.C. Johnson and J.D. Ullman, "Code Generation for Machines with Multiregister Operations," Proceedings of the 4th ACM Symposium on Principles of Programming Languages, January 1977, Pages 21-28.
- Aho, A.V., R. Sethi and J.D. Ullman, 1986b. Compilers, Principles, Techniques and Tools. Addison-Wesley, Reading, Mass. (especially, Chapter 10 Data Flow Analysis)
- Aho, A.V., and J.D. Ullman, 1972. The Theory of Parsing, Translation, and Compiling Volume I: Parsing. Prentice-Hall, Inc, Englewood Cliffs, N.J.
- Aigrain, P., S.L. Graham, R.R. Henry, M.K. McKusick and E. Pelegri-Llopert, "Experience with a Graham-Glanville Style Code Generator", Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, June 1984, Pages 13-24.
- Burke, M.G., and Gerald Fisher, "A Practical Method for LR and LL Syntactic Error Diagnosis," ACM TOPLAS, Vol. 9, No. 2, April 1987, Pages 164-197.

- Carter, J.L., "A Case Study of a New Code Generation Technique for Compilers," *Communications of the ACM*, Vol. 20, No. 12, December 1977, Pages 914-920.
- Cattell, R.G.G., "Automatic Derivation of Code Generators from Machine Descriptions," *ACM TOPLAS*, Vol. 2, No. 2, April 1980, Pages 173-190.
- Cattell, R.G.G., "Formalization and Automatic Derivation of Code Generators", PhD Dissertation, Carnegie Mellon University, 1978.
- Chaitin, G.J., et al., "Register Allocation Via Coloring," *Computer Languages*, Vol. 6, 1981, Pages 47-57.
- Chaitin, G.J., "Register Allocation & Spilling Via Graph Coloring," *ACM (Proc. Spec. Interest Group Program. Lang. Symp. Compiler Constr., Boston, Mass., as part of Auslander and Hopkins "An Overview of the PL.8 Compiler")* , 1982, Pages 98-105.
- Chow, F. and J. Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, *SIGPLAN Notices*, Vol. 19, No. 6, June 1984,.
- Collins, W.R. and R.E. Noonan, "The Mystro System: a comprehensive translator toolkit," Department of Computer Science, College of William and Mary, Final Report, 1985, Langley Research Center Grant NASG-1435.
- Davidson, J.W. and C.W. Fraser, "Code Selection through Object Code Optimization," *ACM TOPLAS*, Vol. 6, No. 4, October 1984, Pages 505-526.
- Davidson, J.W. and C.W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer," *ACM TOPLAS*, Vol. 2, No. 2, April 1980, Pages 191-202.
- Davidson, J.W. and C.W. Fraser, "Register Allocation and Exhaustive Peephole Optimization," *Software Practice and Experience*, Vol. 14, No. 9, September 1984, Pages 857-865.

- Davis, R. and J. King, 1976, "An Overview of Production Systems," Machine Intelligence, Vol. 8 (Elcock, E.W. and Michie, D. eds.), Pages 300-332. Wiley & Sons, New York.
- Dhamdhere, D.M. and J.S. Keith, "Characterization of Program Loops in Code Optimization," Computer Languages, Vol. 8, No. 2, 1983, Pages 69-76.
- Fraser, C.W. and A. Wendt, "Integrating Code Generation and Optimization," Proceedings Symposium on Compiler Construction, ACM, Palo Alto, California, June 1986.
- Fraser, C.W., "A Compact, Machine-Independent Peephole Optimizer," Conference Record of the 6th Annual Symposium on Principles of Programming Languages, January 1979, Pages 1-6.
- Fraser, C.W., "Automatic Generation of Code Generators", PhD Dissertation, Yale University, 1977.
- Freiburghouse, R.A., "Register Allocation Via Usage Counts," Communication of the ACM, Vol. 17, No. 11, November 1974, Pages 638-642.
- Ganapathi, M., "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD Dissertation, University of Wisconsin-Madison, 1980.
- Ganapathi, M. and C.N. Fischer, "Integrating Code Generation and Peephole Optimization," Acta Informatica, Vol. 25, No. 1, January 1988, Pages 85-109.
- Ganapathi, M. and C.N. Fischer, "Affix Grammar Driven Code Generation," ACM TOPLAS, Vol. 7, No. 4, October 1985, Pages 560-599.
- Giegerich, Robert, "A Formal Framework for the Derivation of Machine-Specific Optimizers," ACM TOPLAS, Vol. 5, No. 3, July 1983, Pages 478-498.
- Glanville, R.S. and S.L. Graham, "A New Method for Compiler Code Generation," Conference Record of the 5th Annual Symposium on Principles of Programming Languages, January 1978, Pages 231-240.

- Grasmeder, C.A., "Jonathon: Automatic Code Generation Based on a Machine Description," Computer Science Honors Thesis, The College of William and Mary, Department of Math and Computer Science, 1982.
- Graham, S.L., R.R. Henry and R.A. Schulman, "An Experiment in Table Driven Code Generation", ACM Sigplan Notices, Vol. 17, No. 6, June 1982, Pages 32-43.
- Harrison, W., "A New Strategy for Code Generation-the General Purpose Optimizing Compiler," Technical Report RC 6283 (#26968) 11/10/76, Computer Science, 10 pages.
- Hennessy, J. and M. Ganapathi, "Advances in Compiler Technology," Annual Review of Computer Science., Vol. 1, 1986, Pages 83-106.
- Hennessy, J. and T. Gross, "Postpass Code Optimization of Pipeline Constraints", ACM TOPLAS, Vol. 5, No. 3, July 1983, Pages 422-448.
- Henry, R.R., "Graham-Glanville Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, CA, May 1984.
- Hopcroft, J.E., and J.D. Ullman, 1979. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, Mass.
- Horspool, R.N., "An Alternative to the Graham-Glanville Code-Generation Method", IEEE Software, Vol. 4, No. 3, May 1987, Pages 33-40.
- Horspool, R.N. and M. Whitney, "Even Faster LR Parsing" Software-Practice and Experience, Vol. 20, No. 6, June 1990, Pages 515-535.
- Horwitz, S., A. Demers, and T. Teitelbaum, "An Efficient General Iterative Algorithm for Dataflow Analysis," Acta Informatica, 24, 1987, Pages 679-694.

- Keller, Wolfgang, "Automated Generation of Code Using Backtracking Parsers for Attributed Grammars", SIGPLAN Notices, Vol. 26, No. 2, February 1991.
- Knuth, Donald E., "Semantics of Context-Free Languages", Mathematical Systems Theory, Vol. 2, No. 2, Pages 127-145, Springer-Verlag New York, 1968.
- Rohrich, Johannes, "Methods for the Automatic Construction of Error Correcting Parsers", Acta Informatica, Vol. 15, 1980, Pages 115-139.
- Sethi, R. and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions", Journal of the ACM, vol. 17, No. 4, October 1970, Pages 715-728. Reprinted as pp. 229-247 in Compiler Techniques, ed. B.W. Pollack, Auerbach, Princeton, NJ (1972).
- Slothouber, Louis P., "Adaptation of LR Parsing to Production System Interpretation", PhD Dissertation, College of William and Mary, 1989.
- Spector, D. and P.K. Turner, "Limitations of Graham-Glanville Style Code Generation", SIGPLAN Notices, Vol. 22, No. 2, February 1987.
- Tanenbaum, A., van Staveren, H., and Stevenson, J., "Using Peephole Optimization on Intermediate Code", ACM TOPLAS, Vol. 4, No. 1, January 1982, Pages 21-36.
- Waterman, D.A. and F. Hayes-Roth, 1978, "An Overview of Pattern-Directed Inference Systems", Pattern-Directed Inference Systems, Academic Press, London, Pages 3-22.
- Watt, D.A., "The Parsing Problem for Affix Grammars", Acta Informatica, Vol. 8, 1977, Pages 1-20.

VITA

Laurie Anne Smith King

Born in Atlantic, Iowa, 1 November 1958. Graduated from Blacksburg High School in Blacksburg, Virginia, June 1976, B.A. Virginia Polytechnic Institute and State University, June 1980, M.S., The College of William and Mary in Virginia, May 1983. Ph.D. candidate in Computer Science, The College of William and Mary in Virginia. The acceptance of this dissertation, Code Generation Using a Backtracking LR Parser, will complete the requirements for this degree.

The author was a fulltime instructor for The College of William and Mary in Virginia from Fall 1983 until Summer 1988 and will be an assistant professor of computer science at Ithaca College in Ithaca, N.Y. beginning Fall 1992.