Copyright © 1978, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers

by

Robert Steven Glanville

Computer Science Division
Department of Electrical Engineering and Computer Sciences

Technical Report No. UCB−CS−78−01
and
Electronics Research Laboratory
Memorandum No. ERL−M78/9

University of California, Berkeley
94720

A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers[†]

Robert Steven Glanville

Abstract

This dissertation presents a method for the construction of efficient code generators for high-level procedural programming languages from a symbolic description of the instruction set of the target computer. A table driven algorithm is given that translates a relatively low-level intermediate representation of a program into assembly or machine code for the target computer. A construction algorithm is presented that produces the required tables from a functional description of the target machine. By supplying an appropriate machine description, new tables can easily be created, thus retargeting a compiler for the new computer. Techniques are developed to prove the correctness of the resulting code generator based on the instruction set description.

The output of the front end of the compiler is assumed to be a linearized intermediate representation (IR) of the source program consisting of a sequence of parenthesis-free prefix expressions. Implementation decisions concerning representation and storage allocation, as well as all but the low-level, machine dependent optimizations are already incorporated into the IR. Each machine instruction is described by a prefix expression and an assembly or machine language template. The code generation algorithm performs a pattern-matching similar to parsing. However, unlike the situation in syntax analysis, target machine descriptions are normally highly ambiguous. By defining a property called **uniformity**, which is satisfied by most instruction sets, it is possible to give a concise characterization of the sequence of prefix expressions computed by an instruction set, to check that all possible inputs to the code generator fall within this class, and to produce a left-to-right deterministic linear-time code generator.

Ambiguities in the machine description are resolved in favor of choosing longer instruction patterns over shorter ones, thus effectively attempting to produce the object program that is shortest in terms of the number of instructions generated while containing the same sequence of operations. In practice this heuristic works very well. In comparison with existing compilers, the code generated by this algorithm is of equal or better quality (in terms of the size of the code produced). The instances in which existing compilers produce superior code

†Research sponsored by National Science Foundation Grant MCS74-07644-A03.

stem from optimizations, i.e. changes in the sequence of operations, that were not employed in this work. Most of these optimizations could be combined with our method of code generation. The code generation routines for most existing compilers are written by hand and use sequences of instructions identified by the implementer. By choosing code sequences in a systematic algorithmic fashion, our code generators are more consistent and more successful in using the full range of machine instructions, including many special purpose instructions.

Professor Susan L. Graham
Chairman of Committee

Acknowledgements

I wish to express my gratitude to the many people who helped make this dissertation a reality. The members of the faculties at Berkeley and Southern Methodist Universities provided the excellent educational environment so necessary in such an endeavor. My special thanks goes to my advisor, Professor Susan L. Graham, who provided the encouragement and guidance I needed, and to the members of my committee, Professors Lawrence Rowe and Ilan Adler.

It is impossible to properly acknowledge the numerous persons upon whose work, ideas, and stimulating discussion this dissertation is based. Of particular note are Jeff Barth and Bill Joy, who were graduate students at Berkeley concurrent with myself. I cannot begin to express my thanks to my wife, Linda, who found the time and energy to provide the moral support I needed to complete this dissertation, despite the fact that she was suffering through the same ordeal with her own dissertation. I would also like to thank my parents for encouraging me in my academic pursuits. Finally, I would like to thank Polynomial who contributed countless hours of her undivided attention to each and every detail of my dissertation, even as I wrote it.

The financial support of the National Science Foundation under grant MCS74-07644-A03 is gratefully appreciated.

Table of Contents

Chapter 1: Introduction 1
1.1 Goals of Automatic Code Research 2
1.2 Overview of Compilation 2
1.2.1 Analysis 2
1.2.2 Synthesis 4
1.3 Why Study Code Generation 5
1.4 Previous Work 6
1.4.1 Procedural Code Generation 6
1.4.1.1 UNCOL 6
1.4.1.2 Elson and Rake 7
1.4.1.3 Wilcox 9
1.4.1.4 Donegan 11
1.4.1.5 Analysis of Code Generation Languages 13
1.4.2 Code Generation by Semantic Machine Description 13
1.4.2.1 Miller 13
1.4.2.2 Weingart 14
1.4.2.3 Analysis of Semantic Code Generation 16
1.5 Summary of This Project 17
1.6 Comparison with Compiler-Compilers 17

Chapter 2: Design of a Machine Independent Compiler 19
2.1 Machine Independent Aspects 20
2.2 Parameterizable Machine Dependent Implementation Decisions 22
2.3 Code Generation 26
2.4 Optimization Phases 26
2.4.1 Constant Folding and Constant Propagation 28
2.4.2 Peephole Optimization 29
2.4.3 Common Subexpression Elimination 30
2.4.4 Other Optimizations 31

Chapter 3: The Target Machine Description 32
3.1 The Internal Representation (IR) 32
3.2 Modeling Computer Instruction Sets 36
3.3 The Structure of TMDL 41
3.4 A Sample Machine Description 43

Chapter 4: The Code Generation Algorithm 45
4.1 Shift-Reduce Parsing Algorithm 46
4.2 Adding Semantic Information to the Parser to Make a Code Generator 49
4.3 Register Allocation 51
4.4 A Simple Example 52
4.5 Automatic Construction of a Code Generating Parser 56
4.5.1 The Initial Table Construction 56
4.5.2 Correct Code is Always Generated 61
4.5.3 Looping 66
4.5.4 Complexity of Loop Detection by Preprocessor 68
4.5.5 Loop Elimination by Preprocessor 69
4.5.6 Blocking and Uniformity 71

4.5.7 Semantic Blocking 82

Chapter 5: Two Examples: The PDP-11 and The IBM 370 86
5.1 The Preprocessor and Coder Implementations 86
5.2 Generating PDP-11 Object Code 87
5.3 Generating IBM 370 Object Code 101

Chapter 6: Results and Conclusions 111
6.1 Postfix vs. Prefix 112
6.2 Areas for Future research 114

References 116

Appendix A: PDP-11 Machine Description 119

Appendix B: IBM 370 Machine Description 122

Figures

1.1 Block Diagram of a Compiler 3
1.2 GCL Floating Point Assignment Macro from [Elson70] 8
1.3 Code Generation Template from [Wilcox71] 10
1.4 CGPL Routines for ADD and SUB from [Donegan73] 12

2.1 Generation of a Machine Independent Compiler 20
2.2 Bootstrapping to a New Computer 22
2.3 PASCAL-P Compiler Targeting Information 25
2.4 Three Places for Modular Optimization 27
2.5 Some Peephole Optimizations 30

3.1 Sample Machine Description 44

4.1 Sample Instruction Set Description 53
4.2 Code Generation Table for Example 54
4.3 Initial State Computation for Example Instruction Set 62
4.4 Code Generation Table for Example 63
4.5 Graph Constructed to Eliminate Loop: **d**→...→**d** 70

5.1 A Machine Independent PASCAL Compiler 87
5.2 PDP-11 Preprocessor Execution Times 89
5.3 PDP-11 Code Generator Table Sizes 90
5.4 PDP-11 PASCAL Runtime Organization 92
5.5 PDP-11 PASCAL Procedure Entry and Exit Code 93
5.6 Matrix Multiplication Routines 94
5.7 Integer Read Routines 95
5.8 PDP-11 IR Translation of Test Pregrams 96
5.9 Assembly Listings for *matrixmult* Routines 97
5.10 Assembly Output for *readn* Routines 100
5.11 IBM 370 Preprocessor Execution Times 102
5.12 IBM 370 Code Generator Table Sizes 102
5.13 IBM 370 PASCAL Runtime Organization 103
5.14 IBM 370 Translation of Test Programs 104
5.15 ALGOL-W Comparison Routines 105
5.16 IBM 370 Assembly code for *matrixmult* 108
5.17 ALGOL-W Assembly code for *matrixmult* 109
5.18 IBM 370 Assembly code for *readn* Routines 110

Algorithms

4.1 The Code Generator 50
4.2 The Initial LR(0) Code Generator Constructor 58
4.3 The Initial SLR(1) Code Generator Constructor 60
4.4 Loop Detection 69
4.5 Loop Elimination 72
4.6 Uniformity Test for Algorithm 4.3 78
4.7 Blocking—Uniformity Test for Algorithm 4.2 80
4.8 Default List Construction 84

Chapter 1: Introduction

A compiler is a computer program which translates other programs, written in a particular programming language, into executable code for a specific target computer. This dissertation presents an approach to the problem of producing a retargetable (and transportable) compiler for a high level programming language. Such a compiler could be systematically reconfigured to generate object code for any one of a large class of commercially available computers. A major part of this research concerns the automatic creation of a suitable code generator for each target computer from a symbolic description of that computer's instruction set.

The overall design of a retargetable compiler that incorporates such a code generator is presented in Chapter 2. Particular emphasis is given to discussing the degree of machine dependence that is inherent in various portions of a compiler in a general implementation. Topics closely related to code generation, including register allocation and optimization, are also discussed. Finally, practical aspects are considered, such as the efficiency, modifiability, and maintainability of the resulting compiler.

The target machine description language, TMDL, used by the machine independent compiler, is described in Chapter 3. Since the instruction set description, a major part of TMDL, is closely linked to the internal representation (IR) output of the analyzer, the internal representation is also discussed. Motivation is given for the design decisions, as they are very much affected by the code generation algorithm used.

Chapter 4 presents a table driven pattern matching algorithm that generates object code by 'parsing' the output of the recognition phase of the compiler using the target computer's instruction set as the grammar 'rules'. A finite state pushdown automaton is constructed from the patterns defining the computer's instruction set. At compile time, the output of the parser is taken as the input to the automaton. The longest leftmost instruction pattern is found and the corresponding object instruction is emitted. An algorithm is presented that computes the tables required by the code generator from a semantic description of the target computer's instruction set. The properties of the code generation algorithm are then investigated to determine the type of code produced, what it efficiently codes, and how one can insure that a machine's instruction set will cover all possible outputs of the compiler. The correctness of the code generator and coder construction algorithms is addressed next. Sufficient conditions that the code generator will not loop, block on valid input, or generate incorrect code are given. The machine independent code generation algorithm is thus proven to always generate correct code for any valid IR input.

Example code generators for a PASCAL implementation are presented in Chapter 5. The target computers used are the PDP-11 and the IBM 370. The code generated by the coder for several sample programs is compared with the output of existing compilers on the target computers.

Chapter 6 concludes with the overall results and comments on the feasibility of using this method of code generation both in code generation research and in production compiling. Areas for future research are also discussed.

1.1. Goals of Automatic Coder Research

The major goal of this research is to design a compiler for a single, high-level language that can be systematically reconfigured to produce code for a variety of distinct target computers. The modularity necessary for this task is obtained primarily from the use of symbolic description of the target computer's instruction set. The compiler will be reconfigurable to 'favor' specific target computer architectures by allowing the implementer to choose between certain alternatives (such as how the run time stack is implemented) and to input specific facts about the computer (such as the number of registers). It is expected that machine independent local optimizations will be included in the compiler to allow it to produce good object code. Also, knowledge about the behavior of the particular code generation algorithm used should allow the compiler to arrange the intermediate text symbols, in a machine independent fashion, in a way that allows a simple-minded coder to do well. The work of [Loveman76] and [Carter75] suggests that such an approach is feasible. In Loveman's research source language transformations are used to improve the code generated; in Carter's work the intermediate code is transformed by various optimizations prior to code generation.

1.2. Overview of Compilation

Code generation is one aspect of the complex process of compilation. Because the terminology used to describe some critical points in the compilation process is not standardized, a brief review of compilation is presented next. The model used is similar to that of [Wilcox71]. Figure 1.1 gives an overview of the structure of the compiler model used.

1.2.1. Analysis

Analysis, the first portion of the compilation process, is the phase in which the compiler determines the structure and meaning of the source program being compiled. The original program, written for ease of human comprehension, is transformed into a form that is easy for the computer to manipulate. Analysis also provides the programmer with information about syntax

violations of the language present in the program. The analysis phase produces a set of tables describing the functional and representational aspects of the data manipulated by the program and an **abstract program tree (APT)** which represents the executable portion of the source program.

Analysis is divided into three subtasks: lexical analysis, syntactic analysis and semantic analysis. **Lexical analysis**, also referred to as **scanning**, is the process by which the compiler groups certain strings of characters into individual **tokens**. Identifiers, single and multiple character operators and numerical constants are examples of tokens. Comments and insignificant blanks are discarded. Each distinct token is given an integer number, allowing the compiler to work with fixed length integers instead of variable length character strings, thus simplifying the task. The output of the scanner is a list of tokens that represent the source program that is being compiled.

Syntactic analysis, or **parsing**, is the process by which the compiler determines the structure of the program. Normally, the tokens output by the lexical analyzer are treated as symbols in a context free language and parsed by one of the well known deterministic context free parsing algorithms. As the parse is performed, an APT is constructed that represents the program structure. An APT is similar to a parse tree. The interior nodes of a parse tree represent nonterminals in the context free grammar used to define the syntax of the program. All tokens appear as leaves of the tree. In an APT, tokens appear as interior nodes, and there are very few, if any, nodes that do not correspond to tokens in the source program.

Semantic analysis is the process of determining the meaning of a program. Attributes of variables, constants, functions and procedures are determined and put into the symbol table. In strongly typed languages, type checking is performed to determine the necessary coercions, and to insure that the use of a variable is consistent with its definition (an error message is issued if it is not). The meaning of ambiguous symbols such as +, which may be integer or real addition or perhaps set union, may also be determined.

1.2.2. Synthesis

Synthesis is the process in which the compiler builds an equivalent object code representation of the program in the machine language of the target computer. Internal representations for the data objects described in the symbol table are chosen, and object code sequences are generated that manipulate the data as described by the APT. The environment required by the language being compiled must also be simulated on the target computer. Any implementation dependent restrictions violated by the program are reported. Wilcox calls the synthesis phase the **code generation phase**, but in this dissertation the term code generation will refer to the

restricted task of choosing actual object code instructions to implement the program.

Synthesis is divided into the subtasks of storage allocation, translation, code generation and, optionally, optimization. **Storage allocation** is the assigning of memory locations to variables. **Translation** converts the APT into a low level **internal representation (IR)**. The IR can be thought of as the assembly language for some ideal, source language dependent computer. It is an efficient representation, and often contains operations that depend upon specifics of the particular language being compiled. **Optimization** is a translation of the IR into itself or another IR that will allow the code generator to produce better object code, according to some cost criteria. **Code generation** is the mapping of the final IR representation into instructions of the target computer.

The individual phases of real compilers are seldom as distinct as this model would indicate. In any particular implementation, distinct phases and tasks are often performed in parallel due to the strong interactions that must occur between phases or to allow a more efficient implementation. Translation is often performed by the parser as a consequence of each reduction. Storage allocation can be done as early as when the variable declarations are encountered, or as late as after the optimization phase to allow the additional information gained to better allocate available resources. And in a one pass compiler everything is done in parallel. In this dissertation, however, we will speak as though the phases occur sequentially, in the manner described, and note any difficulties that might be encountered in altering the model to correspond to real compilers.

1.3. Why Study Code Generation

Research in code generation has lagged behind the rest of compiler research for several reasons. In order to analyze a problem mathematically one has to build a precise model of exactly what is happening. For scanning and parsing, there are clean models that allow very complete mathematical analysis. Optimization research also has provided models of the program improvement process. However, while there have been significant results in treating code generation mathematically ([Newcomer75] or [Aho76] for example), research has been with idealized models of computers. Real computers tend to be rather messy to describe as they simply do not have mathematically elegant instruction sets. There are always special instructions that implement certain computations quite efficiently. Furthermore, there are a variety of architecturally distinct commercial computers. A single model would have to be complex indeed to include them all, and would lead to a less efficient implementation of a code generator than a model that was tailored around a single computer design.

The fact remains that real compilers must generate object code for real computers. The result is that most code generators are designed from the ground level by the implementors. Retargeting a compiler for a new computer may require a substantial amount of work, especially if reasonably good object code is to be generated. This dissertation does not attempt to build a formal model of computer instruction sets, but instead presents a method of generating code by pattern matching with a simplified representation of a computer's instruction set. It is hoped that this research will help in the creation of retargetable and transportable compilers.

1.4. Previous Work

We next turn to a summary of the efforts that have been made in the past to improve the code generation process. Previous research in code generation can be divided into two classes. The first class allows the user to provide information about the target machine in procedural form. The implementor describes the code generation process and makes all of the decisions as to what kind of code is to be generated. There are several special purpose code generation languages and interpreters that have been designed for this purpose. The second class uses information about the target machine supplied in a descriptive format, or data base. A code generator is automatically constructed from facts describing the resources of the target computer and the semantics of each instruction on that computer. The work carried out for this dissertation falls into the second class.

1.4.1. Procedural Code Generation

1.4.1.1. UNCOL

In the late 1950's, an attempt was made to define a UNiversal Computer Oriented Language, called UNCOL, to help make all languages easily available on all computers [Strong58] [Steel61]. UNCOL was an intermediate language which was to be used as a stepping stone in compiling any language. A program called a 'generator' would translate programs written in some source language into UNCOL, and a machine-dependent, language-independent program called a 'translator' would generate object code for a particular target computer from the UNCOL program. Thus, if a compiler was written for a new language it would become available immediately on all computers, either because the source language it was written in was already available on all computers, perhaps via UNCOL, or by translating the compiler into UNCOL and generating a version that would execute on the target computer through its already existing UNCOL to machine language translator. Likewise, for each new computer manufactured, once a single UNCOL to object code translator has been written, all existing languages would become available on that computer, as well as a wealth of existing application programs.

The UNCOL approach would substantially reduce the number of compilers required to have all languages available on all computers. For M languages to be available on each of N computers, there would only have to be $M + N$ translators written, instead of the usual $M \times N$. There would also be a certain inherent compatibility among implementations of a language due to the fact that there would be virtually a single compiler for that language.

Unfortunately this project failed. The variety of languages, computer architectures, and instruction sets available at that time proved to be too difficult to adequately represent in a single intermediate language. There was also some concern that such a two translator scheme would be relatively inefficient when compared to a standard compiler. In any event, UNCOL never enjoyed wide-spread use despite its authors' dedication to practicality.

1.4.1.2. Elson and Rake

Elson and Rake describe an implementation of a code generator which generates optimized code for an experimental PL/I compiler [Elson70]. The code generator interprets macro commands written in a special code generating language, GCL. Input to the coder is a syntax tree produced by the parser representing the program being compiled. General purpose tree referencing functions are provided to allow the coder to traverse the data structure easily.

The user provides a separate macro, coded in GCL, for each type of node that may appear in the tree. Each macro includes all required logic for optimization, code emission, and error detection and correction for that particular type of node. Flow of control is determined by conditional branches, parameterized subroutine invocations, and a multidimensional table look up facility that is used as a multidimensional computed GOTO. The language has an assembly language appearance and the examples given are difficult to comprehend, primarily because of the four dimensional computed GOTO's. For example, the floating point assignment macro is over 90 lines long and does not support double-double precision or complex data types. It is also optimized specifically for minimum execution time on the IBM 360 System/65. A portion of that macro appears in Fig. 1.2.

Elson and Rake's method has a number of advantages. The machine dependent code emitters provided by GCL ease the task of code emission. The quality of code produced by their implementation is quite good. The tree referencing primitives allow the implementor to test the context of a particular node to determine applicable optimizations. Powerful optimizations that would be difficult to detect in a linear representation may be readily implemented. For example, in the PL/I statement:

$$I = LENGTH(S1 \,||\, S2);$$

```
                  @FLOATASSIGN OMD
  START @FLOATASSIGN
  DCL (GOPT,LLEN,RLEN,WKCELL,RATR,LATR,LO,LB,LI,LL,LR,RO,RB,
    RI,RR,RL,WKCELL)CELL,GPR REG (FIXED)
*CHECK GLOBAL CELL WHICH HAS COMPILER OPTIONS
  IF (BIT(GOPT,OPTT) = 0|BIT(GOPT,MG5) = 0),OK
  MSG'@FLOATASSIGN OPTIMIZED ONLY FOR MOD 65,TIME OPTION'
*FIND BYTE LENGTHS OF SOURCE AND TARGET
*@FLOATLENGTH UTILITY EXPECTS CURSOR AT PARENT OF ARITH NODE
OK PUSH ARG(1)
  LINK @FLOATLENGTH(LLEN)
  POP
  PUSH ARG(2)
  LINK @FLOATLENGTH(RLEN)
  POP
  IF (LLEN = 16|RLEN = 16),NOT16
  MSG'DOUBLE DOUBLE LENGTH NOT SUPPORTED BY @FLOATASSIGN'
  RTN
NOT16 IF (ARG(1).ARG(1).COMPLEX|ARG(2).ARG(1).COMPLEX),NOTCPX
  MSG'COMPLEX NOT SUPPORTED BY @FLOATASSIGN'
  RTN
NOTCPX SET LALN = 2 - ARG(1).UNALIGNED
  SET RALG = 2 - ARG(2).UNALIGNED
*NOW DO TABLE LOOKUP AND GO TO RESULT LABEL TO
*SET UP REQUIREMENTS FOR SOURCE RESULT, DEPENDING
*ON LENGTHS AND ALLIGNMENTS
  LOOK ERR1,WKCELL,TBL1(LALN,LLEN/4,RALN,RLEN/4)
  GO TO WKCELL
ERR1 MSG'ERROR IN TBL1 LOOKUP IN @FLOATASSIGN'
  RTN
*FOLLOWING ARE THE RESULT LABELS OF LOOKUP
*TARGET 4 BYTES ALLIGNED, SOURCE ALLIGNED. ASK FOR
*RX REFERENCE OR FLOATING REGISTER
RXFR1 SET RATR = M'F0001000'
  GOTO LRX
*8 - BYTE RESULT NEEDED IN FLOATING REGISTER, SO SOURCE
*WILL DO SDR, LE OR LD OR MVC(4), SDR, LE
FRFW1 SET RATR = M'30000000'
*GET ADDRESSIBILITY OF TARGET AS RX OR RS REFERENCE
LRX SET LATR = M'C0000000'
  GOTO LINK
*REQUEST BOTH SOURCE AND TARGET AS RS REFERENCES
*SINCE MVC WILL BE DONE
RSI SET RATR = M'40000000'
 ...
TBL1 TBL(2,2,2,2) REF
  ARRY   RXFR1,RXFR1,FRFUL1,FRFUL1,FRFUL1,RS1,RS1,RS1,RS1,
       RS1,RS1,RS1,RS1,RS1,RS1,RS1
 ...
  END
```

Fig. 1.2. GCL floating point assignment macro from [Elson70].

the code generated does not actually concatenate strings *S1* and *S2*. Strength reduction is used to implement the concatenation operator as a much more efficient add instruction in the context of a length function. Code is generated that is equivalent to the statement:

$$I = LENGTH(S1) + LENGTH(S2);$$

This optimization results in substantially improved code. Finally, the code macros are easily altered to detect and exploit special cases that at first were thought unimportant.

Elson and Rake's method is not without its drawbacks. The fact that GCL is an interpreted language causes a certain amount of overhead. The macros tend to be large, complex and difficult to understand. They are, in fact, so large that they had to be paged in from disk in the authors' implementation. The size of the macros is due in part to the size and complexity of both the language being compiled and the instruction set of the IBM 360. The macros are machine dependent, relying heavily upon the target computer's architecture and instruction set not only with respect to optimization but also to type checking. The concatenation optimization mentioned previously, impressive as it is, is the result of a highly specific decision by the implementors to watch for this type of construct. The retargeting of this compiler for a new computer would require not only a complete rewriting of the code generation macros, but even a rewriting of part of the GCL itself, as there are specific procedures in GCL that depend upon the instruction formats of the target computer. This method relies heavily upon the implementor's ability to design and debug the code generation macros, quite a formidable task for a language/machine combination like PL/I on the IBM 360.

1.4.1.3. Wilcox

Wilcox describes a model for code generation for modern high level programming languages that is based on the code generator he designed for the Cornell PL/C compiler [Wilcox71]. Code generation is divided into four phases: storage allocation, translation (of the APT into tuples), global optimization, and coding. The analysis phase of the compiler constructs an abstract representation of the program. The translator produces a code sequence for an artificial source language machine, called an SLM, that is based on the operators and data types of the source language. Each node of the APT results in a sequence of such instructions. The coder produces object code for the target computer from the SLM program after an optional optimization phase. An implementation map is used to generate a sequence of object machine code instructions for each SLM instruction.

In order to generate object code for a particular computer, the user provides a subroutine for each SLM instruction. These routines are written in a coding language, ICL, especially

designed for the object machine. Statements in ICL are precompiled into templates which are interpreted at compile time by the code generator. The code template for binary addition on the IBM 360 presented by Wilcox is reproduced in Fig. 1.3. ICL is low level and assembly like.

```
ADDB    BR     A,ADDB1    ->ADDB1 If A is in a register       1
        BR     B,ADDB2    ->ADDB2 If B is in a register       2
        LGPR   A          Generate code to load A into        3
                          register
ADDB1   BR     B,ADDB3    ->ADDBB3 If B is in a register      4
        GRX    A,A,B      Generate A+B                        5
        B      ADDB4      ->Merge                             6

ADDB3   GRR    AR,A,B     Generate A+B                        7
ADDB4   FREE   B          Release resources assigned to B     8
ADDB5   POP    1          Remove B descriptor from stack      9
        EXIT

ADDB2   GRX    A,B,A      Generate A+B                       10
        FREE   A          Release resources assigned to A    11
        SET    A,B        A now designates result location   12
        B      ADDB5      ->Merge                            13

        Figure x.x Code Template for Binary Addition (one-pass)
```

Fig. 1.3. Code Generation Template from [Wilcox71].

To retarget a compiler for a new computer, a new ICL compiler has to be designed for that target computer and an interpreter has to be written. This is not as difficult as it may sound, as ICL is like an assembly language and therefore relatively easy to compile. Then a new set of ICL templates has to be written to govern code generation. This is obviously not as simple a task as one might hope when retargeting a compiler. But Wilcox was primarily interested in code generation techniques, not portability. It seems possible to produce better object code using this method than other methods more concerned with portability. ICL templates also allow the implementor a separate vehicle for specifying the code generation part of the compiler, quite possibly reducing the implementation and debugging effort required. However, the addition of a new optimization may require minor additions to the ICL used in order to collect additional necessary information. The implementor still has to make all of the local optimizations and instruction choices, and, like GCL used by Elson and Rake, this method relies heavily on the implementor's ability to design and debug code generation routines.

1.4.1.4. Donegan

In his dissertation, Donegan proposes an organization for a general code generation scheme, though it was not implemented [Donegan73]. The input to the code generator is a syntax tree and a symbol table. Code is generated directly from the tree without translation into a linearized form. To determine what code will be generated for each type of node in the syntax tree, a separate routine is written in a Code Generation Preprocessor Language, CGPL, designed by Donegan. Routines written in CGPL are translated by a preprocessor into PL/I source routines which can then be incorporated into a compiler.

The code generation algorithm accepts expressions of the language being compiled as a parameter and generates object code to compute that expression. Evaluation proceeds by a traversal of the syntax tree. As each operator is encountered, the coder is considered to be in a state that corresponds to the locations of its operands, such as in an accumulator or in memory. The operand state idea is based on earlier work done by [Miller71] which is summarized in a later section of this chapter. For each operator, the implementor specifies three sets of facts. The first is a list of special conditions that will apply to the operands upon entry to that CGPL procedure. An example would be that both operands cannot be in the accumulator of a single accumulator machine at the same time. The second is a set of state tuples from which code can be directly emitted for that operator, and the corresponding PL/I code to actually generate the correct code. The PL/I code is usually a single call to a particular code generation routine. The third set of inputs is a list of state transitions that transform operands from one state to another and PL/I code generation sequences to implement them. This information allows the code generator to automatically modify the state of the operands, bringing the coder into a terminal state from which code can be emitted. Example CGPL routines for integer addition and subtraction for a simple one accumulator computer with a stack appear in Fig. 1.4.

As stated earlier, the preprocessor generates PL/I routines as output from the CGPL input routines. It also generates a table for each operator that describes the state transition paths that will be used to generate code for all valid input state pairs. Thus the CGPL routine for a particular operator will not have to recompute transition paths each time it is invoked at code generation time. The preprocessor also checks to make sure that it is possible to generate code for all valid operand state pairs. Donegan discusses the problem of determining the shortest state transition path when several possible instruction sequences exist, based on supplied instruction costs.

CGPL is a higher level code generation language than those used in previous work. It allows a cleaner representation of the information than is possible in lower level languages, and consequently should require less implementation and debugging time. It has incorporated in it

```
$GENERATE ADD;
 DECLARE CONDITIONS (VAR.TREEPTR)MUTUALLY EXCLUSIVE,
        (INACC,ONSTACK) INTERNAL;
 TERMINAL
   (VAR,INACC)  ->  'GEN(ADD,OP1)',
   (INACC,VAR)  ->  'GEN(ADD,OP2)',
   (ONSTACK,INACC)  ->  'GEN(ADD,NIL)';
 TRANSITION
   (VAR,INACC)  ->  LOADUP,
   (TREEPTR,INACC)  ->  TRANS,
   (INACC,ONSTACK)  ->  'GEN(PUSH,NIL)';
END

$GENERATE SUB;
 DECLARE CONDITIONS
   (VAR,TREEPTR) MUTUALLY EXCLUSIVE,
   (INACC,ONSTACK) INTERNAL;
 TERMINAL
   (INACC,VAR)  ->  'GEN(SUB,OP2)',
   (INACC,ONSTACK)  ->  'GEN(SUB,NIL)';
 TRANSITION
   (VAR,INACC)  ->  LOADUP,
   (TREEPTR,INACC)  ->  TRANS,
   (INACC,ONSTACK)  ->  'GEN(PUSH,NIL)';
END;
```

Fig. 1.4. CGPL Routines for ADD and SUB from [Donegan73].

some ideas on automatically determining instruction sequences from a semantic description of what is needed to be done. In these two respects it is a distinct improvement over earlier work in code generation languages.

However, CGPL is not entirely portable due to its close tie to a single high level language and to the machine and implementation dependent state set and support routines. This author feels that the quality of the object code produced by the system is not much better than that produced by macro expansion. Operators are translated individually after all operands have been evaluated and without regard to the requirements of subsequent operators. There is no way to incorporate multiple operation instructions that might appear in a new computer's instruction set (such as memory to memory addition — equivalent to an add and a store, or a double indexed load — two adds and a load) without modifying the internal syntax tree to locate where they might be used.

An implementation of CGPL was done by Maltz as an extension to the PASCAL

programming language [Maltz77]. The language aspects were improved somewhat, especially the typing mechanism and syntax. Maltz's implementation also indicated that fair code could be generated, but that the drawbacks previously mentioned were present. CGPL is just another programming language that is specifically designed for writing code generators.

1.4.1.5. Analysis of Code Generation Languages

Code generation languages are not as successful as might be hoped. While they are an improvement over the method of writing a code generator in a standard computer language, since some bookkeeping is done automatically, the implementor still has to make all of the low-level decisions, and consequently can make the same errors, whether they result in incorrect or merely inefficient object code. This problem is magnified when more than one programmer is involved in writing the code generator. It is more desirable to allow the programmer to input a concise description of the target computer and let the code generator make all of the decisions, assuming such a task could be efficiently done. If that description is concise, a single programmer could write and maintain the code generator for a large compiler. Even if several programmers were involved, the task would be simpler to understand. Many logical errors would be eliminated, and the bulk of the remaining errors would be clerical in nature, representing some misconception as to how the actual computer hardware behaves.

1.4.2. Code Generation by Semantic Machine Description

1.4.2.1. Miller

Miller describes an early attempt to automatically create a code generator from a semantic description of a target machine [Miller71]. In his system, DMACS, a code generator is created in two steps. In the first step the semantics of the language are defined by a set of machine independent macros. In the second step information describing the structure and semantics of the target computer is defined. All of the operators in the language macros must have instruction sequences that implement them in the target machine description.

The code generation process is modeled as a finite state machine whose state is determined by the runtime location of the operands to a particular macro. States are called **permitted** if object code can be directly generated to implement that macro from that state. If the coder is not in a permitted state, a series of automatic transitions is made to bring it into such a state. In order to do this, the user specifies a set of state transition paths and object code that implements them, such as load, store, and register to register move instructions.

Data referencing is treated as a separate task. The language implementor defines data reference macros and the machine specifier describes the resources available on the target machine and how the distinct data classes are to be represented. It is then possible to have automatic allocation and accessing of data items.

The problem with this work is that many simplifying assumptions are made concerning the architecture of the target computer. Compilers for real machines are more complex than Miller's model allows. Also, he only addresses the problem of generating code for two parts of a compiler, expression evaluation and data referencing. Nevertheless, Miller opened a new area of automatic code generation, the semantic machine description approach. A great deal of research has been and continues to be carried out in this area.

1.4.2.2. Weingart

Weingart describes the code generator used in several implementations of the systems programming language IMP [Weingart73]. His method is based on matching the target computer's instruction set with the output of the parser to determine which instructions to generate. The instruction set is encoded into a tree structure to facilitate the pattern matching process. A tree traversing routine accepts patterns from the parser and searches the tree for equivalent patterns.

The user provides a description of the target computer's instructions in the form of a tree structured data base. Only those instructions that implement operators in the source language are used. Instructions that move data between registers or load data from memory into a register do not implement any such operation and must be handled separately. Weingart describes a program that helps to construct the data base for the user. The user also describes the format in which the instructions will be output. Special conversion patterns are required in addition to the instruction set description to allow the coder to start generating code for certain instances for which the machine has no single instruction. For example, consider the statement:

$$A := B + C;$$

The output of the recognizer is something like:

$$m\ m\ m\ +\ :=$$

where m stands for the address of a variable. On a machine without a multiple address add instruction the code generator would need some help in starting the code generation for this statement. Since instructions to load a single operand into a register are not automatically

utilized by the code generator, the user supplies a special pattern called a **conversion** pattern to match the '$m\ m\ +$' portion of the input and arbitrarily generate code to load one specific operand into a register. When generating code for an operator it is the user's responsibility to specify conversion patterns for any case in which code cannot be directly generated. The input string is then altered to reflect the load instruction:

$$m\ m\ r\ +\ :=$$

and code generation continues. The '$m\ r\ +$' portion of the input is then matched to an add instruction, code is issued, and the input is changed to:

$$m\ r\ :=$$

The major advantage of this method of code generation is that it relies on the instruction set of the target computer in deciding what code to emit. The clerical nature of describing an instruction set is less prone to error than the creative nature of writing code generation routines, macros, or templates. Every operational instruction is available to the coder, and with a careful ordering of the instruction tree, special cases for non-standard instructions are automatically discovered by the code generator. Weingart claims that good local code is generated. Finally, retargeting the compiler for a new computer is greatly simplified. One merely substitutes the new instruction set's description for the old as input to the tree builder and produces a data base for a code generator for the new machine.

The disadvantages of this method are the difficulty of creating and using an optimal tree structure for code generation and the lack of sufficient conditions to insure that a complete set of conversion patterns have been supplied. The efficiency of the code generation algorithm as well as the quality of the generated code are dependent upon an optimized version of the instruction tree. If the tree is not compressed as much as possible (by combining prefixes of similar instructions) then the coder will spend a lot of time searching the tree for patterns. If there is one instruction that is a more efficient but a special case of another instruction, then care must be taken to insure that the special case is matched first, or else it will never be used. For example, if there is an add immediate instruction that adds an integer constant to a register, and a more efficient increment instruction that adds 1 to a register, then it is desirable to use the increment instruction whenever possible. There is also a minor problem that would result from an incorrect description of an instruction. This would cause incorrect code to be generated, but would be relatively easy to detect as that instruction would always be issued incorrectly.

A more serious problem is the lack of a method to determine whether the necessary set of conversion patterns have been supplied. These patterns are not obtained directly from a description of the instruction set, though some progress in this area is surely possible. It is mandatory that the patterns be complete. Any omission would cause the coder to block at some point and it would not be able to generate code for some legal expression. Without a set of sufficient conditions that will insure that a complete set of conversion patterns have been supplied, there will be no guarantee that the coder will not block on valid input.

1.4.2.3. Analysis of Semantic Code Generation

Semantic code generation is a promising area in compiler research. It strives to free the compiler writer from having to painstakingly analyze numerous special cases when choosing code sequences, and allows the entire code generation process to be viewed from a higher, more abstract level. More of the implementor's time can be devoted to studying the problem instead of juggling the solution, analogous to coding in a high level language as opposed to assembly language. The fact that a code generator can be automatically created provides both the possibility of writing a practical, easily retargetable compiler and developing a convenient tool for investigating instruction set designs. Both are becoming increasingly important due to the number of microprogrammable computers appearing today. Such development tools are absolutely necessary if the increased flexibility provided by these new computers is to be fully utilized.

A major problem with code generation by semantic machine description is to determine the proper level at which to define the semantic model. A description at too high a level would possibly make implicit assumptions about the target computer that are not always true, leading to either inefficient or incorrect implementations. A description at too low a level would either be too difficult to use or would result in an inefficient code generator. Furthermore, certain computer architectures are difficult to model by a semantic description. Particularly notable are the 'super computers' of both past and present and the one chip microcomputers. The difficulty is perhaps inherent in their design rather than a shortcoming of semantic modeling. Such computers are a product of pushing available technology to its limits in order to achieve a cost or performance breakthrough. They were designed either for high speed or for low cost, not for programming ease. Finally, there is some question as to whether any completely general code generation scheme can produce reasonably good code in an acceptable amount of computer time.

1.5. Summary of This Project

The approach taken in this dissertation is to implement a machine independent compiler for a specific programming language that can systematically be tailored to generate code for any specific computer. The input to the compiler that gives it this flexibility consists of both a list of implementation decisions and a semantic description of the target computer. The implementation decisions govern the method by which certain language constructs are to be implemented, hopefully in a way that is most amenable to the target computer. The semantic machine description is at a relatively high level. To compensate for this fact the compiler is tailored to produce an internal representation that favors the target computer's architecture, according to the implementation decisions made. As many machine independent optimizations as are judged worth while may be included in the compiler (constant folding, etc.). Specific data concerning the target machine is used to guide other machine dependent optimizations. A preprocessing algorithm is presented that reads the target machine instruction set description, in the form of an ambiguous context free grammar, and outputs tables that a machine independent code generation algorithm can use to produce object code for the machine described. Prior work in the area of parsing table construction using ambiguous grammars proved useful [Aho75]. The IR input to the code generator is biased by the implementation decisions in a way that will allow a simple, machine independent code generation strategy to generate reasonably good code. Finally, algorithms are developed to eliminate the possibility of the coder looping or blocking on valid input, resulting in a proof of correctness for the resulting code generator. A more detailed description of this process is given in the next chapter.

1.6. Comparison with Compiler-Compilers

One may rightfully ask how this work differs from the compiler-compiler approach to compiler construction and language portability. It may appear to be just a subset of compiler-compiler research. This observation is in fact partially correct. However, the main difference is that compiler-compilers strive to be as general as possible, especially at the front end, allowing many radically different languages to be compiled, while this work is primarily concerned with the code generation phase of compiling. What has been done is, in effect, to fix the front half of a compiler-compiler system for a specific language and to concentrate on the relatively non-understood back half, primarily code generation. Practically speaking, in our design, the front end can be written as efficiently as standard one language / one machine compilers, something not currently possible with compiler-compilers. Consequently such a compiler can be used in a production environment. In a research environment, this work provides a valuable tool for experimentation with code generation schemes and related areas, including register allocation algorithms, determining the effectiveness of certain optimizations, and, perhaps most

importantly, allows the impact of different instruction sets on object code size and efficiency to be studied.

Chapter 2: Design of a Machine Independent Compiler

This chapter discusses the overall design of a machine independent compiler. A **compiler** is a program that translates programs written in a programming language into machine instructions that can be directly executed by the target computer. This is in contrast to an **interpreter** that translates source programs into pseudo code, which is in turn interpreted by another program when being run. A **machine independent compiler** can be reconfigured in a systematic and well-defined manner to generate object code that can be executed on any specific target computer.

In particular this chapter discusses a generalized retargetable compiler. A compiler is **retargetable** if there is a well-defined procedure by which it can be altered to generate object code for a variety of different target computers, even though the compiler itself might need to be executed on a particular computer. A **portable** compiler is a retargetable compiler that can be made to execute on the new target computer as well. The act of altering a compiler so that it will execute on and produce code for a new computer is called **transporting** the compiler to the new computer. The language compiled by a portable compiler is also said to be portable, as it can be made to run on any computer by the transporting of its compiler. There are several ways to actually transport a portable compiler [Welsh72] [Pasko73] [Nori74] [Poole74] [Glanville76], the most popular being variations of **bootstrapping** in which the compiler is written in the language which it compiles and then used to compile itself into code for the new target computer.

The organization of the compiler is described with respect to these areas:

1) language dependent — machine independent aspects,
2) machine dependent aspects that have a small number of reasonable implementations,
3) strongly machine dependent aspects (i.e. code generation), and
4) the use and effectiveness of specific optimizations.

Emphasis is placed on producing a compiler that can be retargeted for a new computer with minimum effort, yet allows fast compilation and the generation of reasonably good object code. It is argued that such a compiler can be written with almost the same effort required to implement a single compiler for a single computer, and that such a compiler is desirable from a language standard point of view as it would aid program portability. The final section discusses various machine independent and machine dependent but parameterizable optimizations that can optionally be included, with some observations as to their cost and effectiveness.

A compiler for a specific computer is generated by a preprocessor from a set of specifications that define the target computer and the implementation decisions to be used. The preprocessor can be thought of as a language dependent compiler-compiler. The input to the preprocessor is a list of implementation decisions, a list of machine instructions along with their semantics, and a set of parameters that describe the architecture of the target computer. The output consists of a program in some high level language which is the compiler. A general diagram of the compiler appears in Fig. 2.1. The implementor must also provide certain service routines that interface with the operating system under which the compiler will be run. These include routines for input and output, memory management, interprocess communication, and other local support.

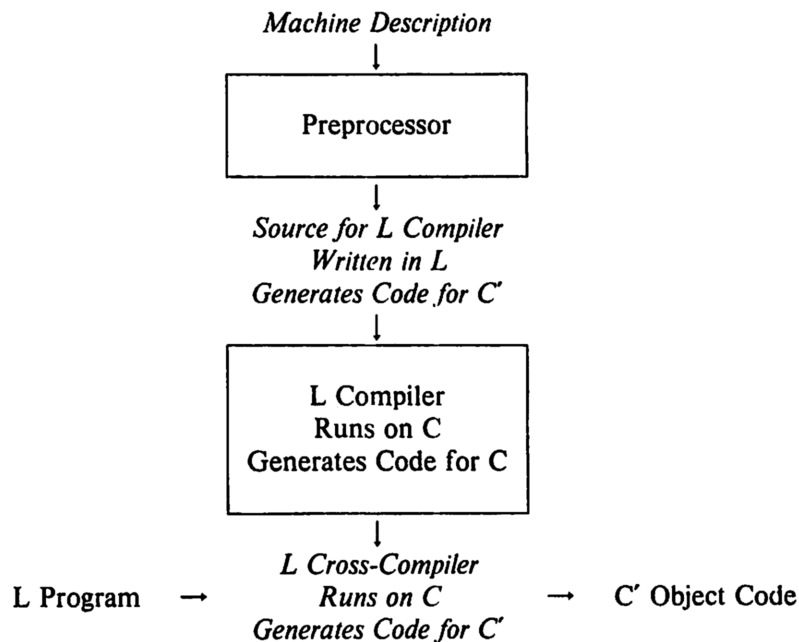


Fig. 2.1. Generation of a Machine Dependent Compiler.

2.1. Machine Independent Aspects

When compiling a program for a given language, there are certain tasks that must be done which are not dependent upon the target computer for which the code is to be generated. In fact, if a complete definition of the language is available, it is possible to write a parser and a semantic analyzer that would be adequate in a compiler for any target computer. The APT output of the recognition phase can also be in a machine independent language. Furthermore, a multitude of language dependent but computer independent optimizations can be performed,

either on the source language itself or on the intermediate text. These optimizations are discussed in Section 2.4.

In a machine independent compiler, machine independent aspects are programmed once and used in all implementations. When generating a new compiler for a new computer, the section of code for these aspects is simply copied. If the compiler is to be transported to new computers, instead of simply to be retargeted to produce code for them, the question arises of which language is to be used to write such a compiler. Assembly language is obviously not suitable, as it is both difficult to write in and, by nature, not readily transportable to new computers. High level programming languages, such as PL/I, FORTRAN or COBOL also have their problems. FORTRAN is available on most computers, but programs written in it are not necessarily portable. There are too many hardware and implementation dependent restrictions and 'enhancements', and it is not the ideal compiler implementation language. PL/I is better suited to compiler writing, but is available on only a few computers. Systems programming languages, such as PL/360, C, IMP, BLISS, and BCPL, are more suited to compiler writing but they are machine dependent and each is only available on a small number of computers.

There are two immediate solutions to this dilemma. The first one applies only to those languages in which a compiler may easily be written. This method requires that the preprocessor and the machine independent compiler be written in the language to be compiled. Then it is a relatively simple task to bootstrap these programs onto another computer using the following steps: The specifications for the new implementation are input to the preprocessor and the source code for a compiler for the new computer is generated. That source is then compiled on the existing compiler and the resulting object code is a cross compiler that executes on the old computer and generates code for the new computer. Next, the source for the new compiler is compiled on the cross compiler, and the result is a compiler for the new computer. This process is illustrated in Fig. 2.2.

The second method would be used to implement machine independent compilers for languages which are not totally suited to compiler writing. The machine independent compiler is written in a language that either is already available on the new computer or is implemented via a similar machine independent compiler. The language used to implement the compiler for the new language is then transported to the new computer in the manner described above, if that language is not already available on the new computer. The machine semantics and implementation details for the primary language being transported are then input to the preprocessor, and source for the compiler that generates code for the new computer is output. This can then be compiled and run on the new computer via the existing secondary language compiler.

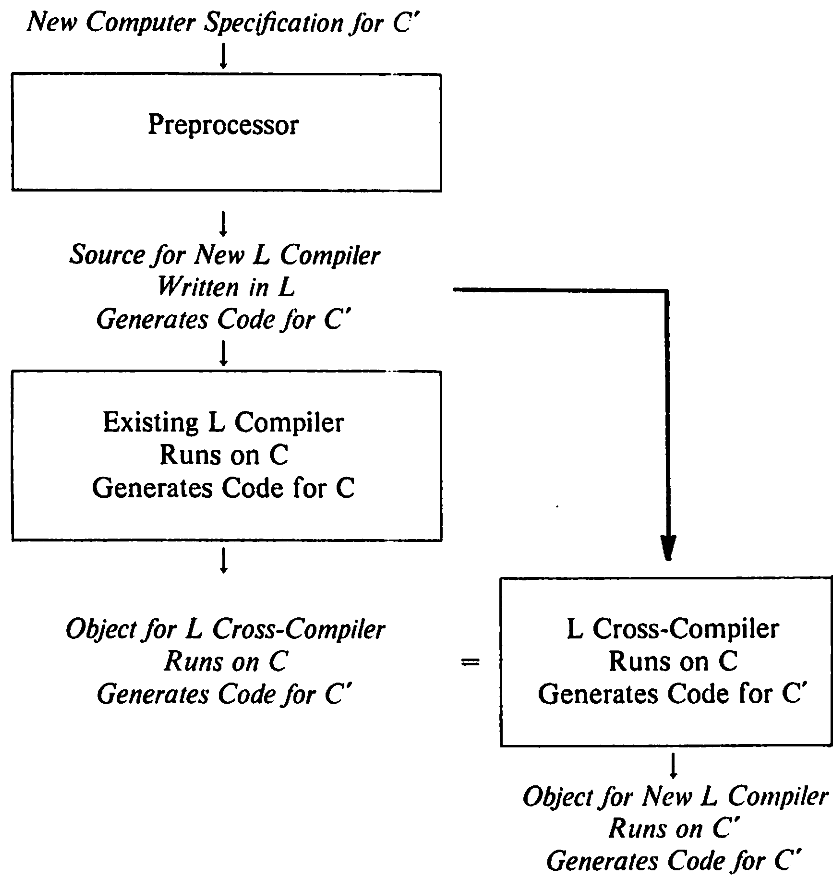


Fig. 2.2. Bootstrapping to a New Computer.

2.2. Parameterizable Machine Dependent Implementation Decisions

There are certain implementation decisions that, though a single choice could be used for all target computers, can drastically affect the efficiency of the implementation. Such decisions should be made with an actual target computer in mind. An example is the run time display in ALGOL-like languages. On a computer with sufficient index or base registers, it may be best to keep the entire display in registers. On a computer with a few base or general purpose registers, it may be best to keep only the global and local displays in registers and access the others via the complete display stored in memory. Single register computers might do better allocating a fixed array in memory to the currently active display. The single register solution would work correctly on any computer, but certainly not as efficiently on most. The number of reasonable ways to implement the display, however, seems to be limited to these three. In fact, for a given language, there are many implementation decisions that appear to have only a few good choices on existing computers.

The idea of automatically tailoring the first part of a compiler to a specific target computer via a list of attributes of that computer is not new. The PASCAL-P compiler uses just such a scheme to automate variable allocation [Nori74]. The P compiler is actually the first half of a complete compiler. Source programs are translated into assembly language for a hypothetical stack computer, SC, with variable allocation and addressing governed by the machine dependent parameters supplied. Either the code can be interpreted (an interpreter written in PASCAL is supplied with the P-compiler), or various methods may be used to translate the assembly code for SC into object code for the target computer. Since the P-compiler is written in PASCAL, it is possible to bootstrap a compiler onto the new computer once this has been done. The P-compiler has been successfully bootstrapped onto a number of computers using this method, including the Univac 1108 and the IBM-360 [Richmond74], the PDP-11 [Bron76], and the PDP-10 [Grosse76], to mention a few.

When specifying a particular implementation of the machine independent compiler discussed in this dissertation, the implementor supplys a checklist of such implementation decisions. The preprocessor then emits the corresponding sections of code, in whatever language was used to implement the compiler, that correspond to the decisions. Since many such decisions are represented by a slight change to the IR produced in the translation phase, almost as macro expansions, this should not be a difficult task. Again, consider the ALGOL runtime display. The display is used to access variables. As discussed at length in Chapter 3, the IR is a low level representation of the program being compiled. Included in it are the implementation details of that particular compiler. So, any change in the implementation of the display will result in a corresponding change in the IR code to access variables that is input to the code generator. But this is a very local alteration to the IR, as if it were a macro substitution at each variable access. No other portion of the compiler need know about this decision (aside from the register allocator, which must not allocate any base registers, used to hold the display, for other tasks). The code generator blindly translates IR symbols into object code instructions, not knowing what ends they might accomplish.

Another example of an implementation decision that could be made via a checklist concerns the evaluation of Boolean expressions. It may be that the target computer has a set of efficient compare instructions that computes a Boolean result from two operands. In that case, it would be desirable to generate straightforward prefix code for the IR using relational operations. Otherwise, it might be best for the compiler to compute the value using conditional jumps [Bauer68] [Gries71] [Aho77b]. For example, the following Boolean expression:

$$(I < J) \textbf{ and } (K \neq L)$$

could be represented in the two following pseudo target machine instruction sequences:

```
load    r1,I
lt      r1,J
load    r2,K
ne      r2,L
and     r1,r2
```

```
                load    r1,I
                comp    r1,J
                jge     l.false
                load    r1,K
                comp    r1,L
                jeq     r1,l.false
l.true          load    r1,=1
                jump    next.ins
l.false         load    r1,=0
next.ins
```

As the reader may have noticed, the two instruction sequences shown do not compute exactly the same expression. The first one evaluates the entire expression verbatim. It always compares K and L even though the result of the computation may have been determined by the comparison of I and J. (If one operand to a Boolean **and** operator is FALSE then the result will be FALSE regardless of the value of the other operand.) The second code sequence uses this fact to optimize the evaluation of the expression. If I is greater than or equal to J then a jump is made directly to the FALSE label.

There are several points to be considered concerning the evaluation of Boolean expressions. It is often a decision of the language designers to require full evaluation of Boolean expressions, to require the evaluation to stop as soon as the result can be determined (as in ALGOL-W [Bauer68] and BLISS-11 [Wulf75]), or to allow the implementor to make the decision (as in PASCAL [Jensen74]), either explicitly or accidentally. The last choice is undesirable since it may lead to problems when moving a program from one compiler to another. It is also important for the user to know which choice is being made. Consider the statement:

while ($i \leqslant MAXINDEX$) **and** ($A[i] \neq x$) **do** $i := i + 1$;

where A is an array with dimension 1..*MAXINDEX*. The execution of this statement will sometimes result in a subscript referencing error when full evaluation is done, but never when minimal evaluation is used. Such expressions are also affected by the order of evaluation when the minimal computation method is used. In any event, the implementor of a compiler using the machine independent compiler described in this dissertation will not be making a choice of whether to use a partial or complete evaluation of Boolean expressions, as that is predetermined by the language being compiled. Instead the implementor decides whether Boolean compares are best implemented by arithmetic operations or conditional jumps on a particular target computer. It should also be noted that the compiler can often optimize jumps in a machine

independent manner, depending upon whether the Boolean value being computed is used in a flow of control test as in:

if $(i < j)$ **and** $(k \neq l)$ **then** ...

or an assignment statement:

Boolvar := $(i < j)$ **and** $(k \neq l)$;

The difference stems from the fact that an **if** statement only needs to transfer control to one of two locations while the more general case of the use of a Boolean expression in the assignment statement requires some standard representation of TRUE or FALSE to be explicitly generated and stored into the variable *Boolvar*.

The PASCAL-P compiler requires 9 integer constants in order to specify a target computer. These constants are used to determine certain machine dependent limitations (such as maximum integer value accepted), how much storage to allocate for the basic types, and compiler limited constants (such as maximum string length). Figure 2.3 summarizes the required constants.

Constant	Usage
MAXINT	the largest integer which the compiler will process.
CHARSIZE, PTRSIZE, INTSIZE, BOOLSIZE, REALSIZE, SETSIZE	the number of basic, addressable storage units required to store the values of the indicated type. SETSIZE must be at least 59 as the P-compiler utilizes sets of that size. (Alignment boundaries are not handled).
DIGMAX	the maximum length string of characters which may be used to represent unsigned numeric constants. This allows the compiler to manipulate real values, which are not processed but copied to the SC code output.
STRLGTH	the maximum length string that can be handled by the computer. Must be greater than DIGMAX.

Fig. 2.3. PASCAL-P Compiler Targeting Information.

2.3. Code Generation

There are decisions that cannot be made without knowing certain facts about the target computer, most notably which object code instructions to issue and how much memory to allocate for variables and temporaries. The feasibility of a machine independent compiler depends upon the existence of a reasonably efficient generalized code generation algorithm. The efficiency of the code generator and of the code it generates will have a strong influence on the acceptance of the resulting compiler. The ease of specifying the instruction set semantics will largely determine the number of computers on which such compilers are implemented.

A major problem associated with a machine independent compiler is choosing which object code instructions to generate. This dissertation presents a model of computer instruction set semantics and a general code generation algorithm that facilitate the implementation of a code generator. The implementor supplies a list of the meanings of each instruction on the target computer to the preprocessor. The preprocessor outputs tables that are read by a machine independent code generator and are used to govern the object code generated at compile time. The task of determining which instruction or instructions to emit for a particular language construct is handled automatically by the algorithm.

2.4. Optimization Phases

This section discusses modular code optimizations and how they could be included in the overall design of the machine independent compiler. Modular optimization passes can be conditionally selected by the user when desired, thus eliminating extra overhead when efficient execution is less critical (e.g. during program development). They would also provide researchers with a convenient tool with which to study various optimization techniques and instruction set designs. Empirical studies could be made to measure both the compile time cost and the efficiency improvements of the optimizations on actual programs.

There are three places in the compiler that modular optimizations can conveniently be inserted. The first is as an APT to APT transformation that detects source language level optimizations. The second is an IR to IR translator that can utilize implementation dependent information, present in the IR, to perhaps produce better code than the first type of optimization. The last is peephole optimization that is used to improve the object code that has been generated by the code generator. Figure 2.4 shows the location of the optimizations in the compiler. Some of the optimizations presented are implementable in a totally language-dependent and machine-independent manner as a transformation of the APT. However, a much greater improvement in the code generated is possible if they are applied after the introduction of implementation dependent information. Some are only applicable in certain

implementations and not in others. In any event, it is usually desirable to introduce optimizations after a point at which the implementation dependent information is known.

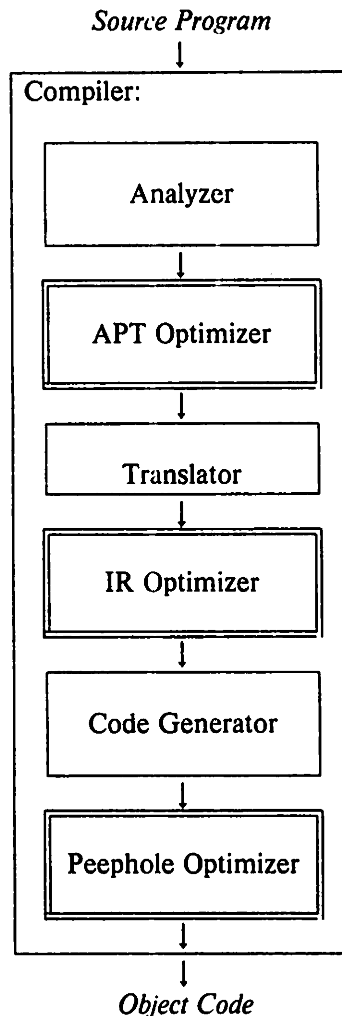


Fig. 2.4. Three Places for Modular Optimization.

The word 'optimization' is used rather freely here. Perhaps 'code improvement' or even 'transformation' is better suited. There is no clear-cut line between generating 'optimized code' and just plain 'good code'. In special cases, a valid 'optimization' may even result in pessimized code. Unlike constant folding, which can only result in shorter object code and a faster run time, common subexpression elimination may result in object code that actually 'costs' more in certain specific cases than if it were not done. This depends on the cost measure used, the architecture of the target computer, and its exact state when the object code is being produced. Generally, the resulting object code will be shorter and/or will execute more rapidly

after the transformation has been applied. In any event, the term 'optimization' will be used throughout this chapter in the preceding sense.

2.4.1. Constant Folding and Constant Propagation

One of the simplest optimizations to perform is constant arithmetic computations at compile time. In its basic form, this transformation consists of finding expression subtrees in the APT in which all leaves are constants and replacing each subtree by a single node representing the equivalent constant value. The replacement requires evaluating operators, such as '3*5', and certain functions with constant arguments, such as 'ARCTAN(1.0)*4.0'. User functions cannot be evaluated at compile time unless there is some mechanism to insure that they have no side effects and that they always return the same value when called with the same parameters. The PL/I function attribute *REDUCIBLE* is one method for the user to notify the compiler that these conditions hold [IBM65].

Constant propagation is also easy to perform. If a variable is assigned a constant value, its value is used later in the program, and there is no way for the variable not to have that value when control reaches the subsequent variable reference, then the constant may be substituted for the variable in that reference.

Constant folding can be expanded to include the more general situation in which the commutativity of an operator is used to discover otherwise hidden cases. However, one has to be careful not to violate the semantics of the source language. Ask any numerical analyst what certain 'legal' commutative reorderings of operands will do to the accuracy of certain floating point computations. Also more compile time is required to discover generalized constant folding.

A specific place that constant folding occurs frequently is in expressions used to access array elements or record fields. The address of the array element $A[i]$ may be computed as '$a + i$', where 'a' is the address of $A[0]$ (if it were to exist). Obviously, if 'i' is a constant, then computing the address $A[i]$ can be made much simpler. In FORTRAN the address would be a compile time constant, and in PASCAL the array element would be addressed in the same way that a simple variable is addressed. If there is a constant component to the index, then it could be added to 'a' at compile time to simplify the runtime expression. The address of a PASCAL-like record field, such as *REC.FLD*, is a constant offset from the beginning of the record, which often can be added to the constant offset of the variable of which it is a part at compile time.

Constant folding may not appear to improve the object code enough to be worth performing. However, constant expressions may be introduced by macro expansion or by inline

procedures. If there are constant parameters to a macro or inline procedure, then a considerable amount of optimization may be possible. Constant folding, along with dead code elimination (the removal of code that cannot be reached), can even reduce the amount of work required to implement a reasonably efficient compiler. A group at IBM is implementing an experimental optimizing PL/I compiler [Harrison77]. In PL/I, string concatenation is a complex operation to optimize. String variables may be variable or fixed length, on word or byte storage boundaries, have known or unknown compile time values, etc. Their solution to this problem is to write one general string concatenation procedure and include an inline copy at each occurrence of a concatenation operator, with the formal parameters replaced by the actuals. Repeated application of constant propagation and constant folding, live-dead analysis, and dead code elimination uses the information known at compile time to tailor the operations to that particular call, thus optimizing the operation.

Constant folding and constant propagation are language dependent but machine independent optimizations. These optimizations can be programmed once and used in all implementations. Either of the first two methods of adding them to the portable compiler is suitable. An IR to IR translator would be able to detect optimizations involving the target implementation, thus producing better code, but an APT to APT translator would be easier to implement.

2.4.2. Peephole Optimization

A substantial amount of optimization can be achieved by the use of a peephole optimizer [McKeeman65]. Such an optimizer transforms object code after it has been emitted by the code generator. An attempt is made to recognize certain instruction patterns and to replace them with more efficient code sequences. There are several distinct classes of patterns that may be used in a peephole optimizer. Some of them are listed in Fig. 2.5. For an example, consider the two statements:

$$A := B;\ \ C := A + 1;$$

Naive code generation might produce code similar to:

```
load   r1,b
store  r1,a
load   r1,a
add    r1,=1
store  r1,c
```

Clearly the second **load** instruction is unnecessary as the value of A is already in register **r1**. Likewise, the **add** instruction could be replaced by an increment instruction, if one existed.

- Eliminating redundant loads and stores
- Recognizing target machine idioms
- Removing unreachable code blocks
- Eliminating jumps to jump instructions
- Algebraic simplification
- Strength reduction

Fig. 2.5. Some peephole optimizations.

As will be seen in Chapter 4, the code generator used in the machine independent compiler is very good at utilizing idiom instructions, such as using increment to add 1. It is also capable of performing a limited amount of strength reduction (e.g. replacing a multiply instruction with a more efficient add or shift instruction). The other peephole optimizations, however, would seem to improve the generated code a great deal, and should be carefully considered when implementing a compiler [Wulf75].

2.4.3. Common Subexpression Elimination

Another language dependent, machine independent optimization is common subexpression elimination. When the same subexpression appears more than once in a program, and the compiler can determine that its value will be the same each time it is evaluated, then that expression is called a common subexpression (CSE) and only need be computed once. Its value can be stored in a temporary location and used upon encountering subsequent occurrences of the subexpression instead of actually recomputing it. The rules that govern when this optimization may be used are highly language dependent.

A CSE optimizer can be used in the machine independent compiler in several ways. The simplest would be to locate all instances of a CSE, compute the value once and store it in a temporary variable, and replace the uses of it by a reference to that variable [Schneck73] [Loveman76]. The code generator would not even be aware that such an optimization had taken place. Somewhat better code can be generated if the value is not stored into a temporary, but flagged in the IR as a CSE. Each use of it can be replaced by a "use CSE" operator, and the code generator can attempt to keep its value in a register throughout its lifetime. This matter is described in more detail in section 4.3.

2.4.4. Other Optimizations

More global optimizations, such as code motion, can also be performed as APT to APT transformations [Allen75] [Graham76] [Barth77]. The net result is almost a source level optimization, so the code generator is not affected. Thus, these optimizations could easily be utilized in a machine independent compiler.

Chapter 3: The Target Machine Description

This chapter describes the Target Machine Description Language, TMDL, that is read by the preprocessor and the Internal Representation, IR, for the machine independent compiler outlined in Chapter 2. TMDL is designed to be easy for the implementor to use. It is composed of sections that describe the resources of the target computer (such as the number and kind of actual machine registers), how those resources are used (including logical groupings of register classes and pairs, and which registers are free for the code generator to use), and the instruction set description. The major part of the discussion of TMDL concerns the instruction set description. Since it is closely tied to the IR, a description of the development and content of the IR is given in the first section. The instruction set description is presented next, followed by a discussion of the remainder of TMDL. The final section gives a partial description of a simple computer to give the reader a feeling for the nature of TMDL.

3.1. The Internal Representation (IR)

In order to generate object code in an automated system, it is necessary to define a semantic model of computer instruction sets. Such a model is restricted in two ways. It must interface with the computer (by describing the actual instructions) and with the semantic description of the language being compiled. It is not surprising, then, that the design of the IR used in a machine independent compiler is heavily influenced by the language used to describe machine instructions, and vice versa. Due to the parsing nature of the code generation algorithm, the IR must be constructed from the same set of symbols used to describe the instruction set semantics. In addition, the IR must have the property of **locality**, in that the meaning of an expression is not affected by the context in which it appears. The reason for this will become more apparent when the actual code generation algorithm is described. This section outlines the development of the IR; the next section describes the instruction set semantic language.

Consider the following PASCAL statement:

$$A := B + C \tag{3.1}$$

This statement by itself is ambiguous. It represents a few valid and a multitude of invalid PASCAL statements, depending on the types and level of declaration of the variables, whether they are fields of a record, function names, or even type names or undeclared identifiers. It should be noted that it is not the function of the code generator to perform type checking or to report

other semantic errors. It will be assumed that the IR string passed to the coder represents a semantically valid program, i.e. a legal program according to the rules of the language. To simplify the presentation, the type of all variables and operators in this section will be assumed to be integer.

A **source language prefix expression** is an unambiguous linear representation of the APT for a source language expression, having been linearized in Polish prefix order. In a prefix expression, the number of operands to each operator (label of an internal tree node) is fixed and each operator appears immediately prior to its operands. Parentheses, which are required in standard infix notation, are not required in prefix notation, and consequently are absent. Prefix notation is one of the simplest ways to unambiguously represent a source language statement in a known context. This representation is used as the input to the code generator by some compilers [Aho72a] [Gries71]. For an in depth mathematical study of parenthesis-free representations of trees, see [Meyers74]. The equivalent source language prefix expression for (3.1) is:

$$:= A + B\ C \tag{3.2}$$

This expression can be generated from the APT by a simple preorder walk [Knuth68]. In a compiler that is generating code from prefix expressions, A, B, and C would be pointers into a symbol table where their attributes are described. Such a prefix expression does not provide enough information for a code generator. Even knowing that A is a local variable, B is local to the next statically enclosing procedure, and C is global is not sufficient. The code generator must either have the details of the particular implementation built in (to be able to generate code, for example, that accesses the value of a variable) or have that information represented in the IR itself. The viewpoint taken in this dissertation is that the code generator's primary responsibility is to generate correct code, and that it should not be required to make implementation decisions. Thus, implementation details are incorporated into the IR.

Since (3.2) is at too high of a semantic level for the code generator to use, it must be translated into an equivalent sequence of basic machine operations that convey how the source language operations are to be implemented. (In particular, all coercions are explicit). This translation is seldom more than a macro expansion. When the value of a variable is needed, for example, it is replaced by an expression representing an access path to its value in that particular implementation. When its address is required, such as the case for the identifier A in (3.2), it is replaced by the corresponding expression. Unlike source language level prefix representations, IR is a very low-level and local language, allowing expressions to be evaluated without considering their contexts. The code generation algorithm presented in this dissertation

requires that the IR have this property. In contrast, source language prefix is usually not local. An identifier may represent either the value or the address of a variable, depending on which operand to an assignment operator it is. An exception would be BLISS [Wulf71] in which an identifier always represents the address of a variable and must be dereferenced to obtain its value.

The IR input to the code generator is a sequence of Polish prefix expressions composed from a finite set of symbols, V, called the **vocabulary**. These symbols are classified into the two major categories of **operators** and **operands**. In order to be able to define the set of well-formed IR expressions, the number of arguments to each operator must be both fixed and known†. Each operator is also classified as being either a **root-level** operator (those that may not appear in an operand to another operator) or an **internal** operator (those that must appear as an operand to another operator). Let **M** be a mapping of symbols in **V** into the integers, **M: V → integers**, with $\mathbf{M}(e)$ equal to $1-n$ if e is an n-ary operator, and equal to 1 if e is an operand. (Thus binary operators are mapped to -1, unary operators to 0, and operand symbols to 1.) If $E = e_1...e_k$ is a sequence of symbols in V, then $\mathbf{M}(E)$ is denoted $W = w_1...w_k$. E is a Polish **prefix expression** iff:

1) $w_1+w_2+...+w_k = 1$, and

2) $\forall$ j, $1 \leqslant j < k$, $w_1+...+w_j < 1$.

A **root-level** IR expression is a prefix expression, E, that satisfies the two additional conditions:

3) e_1 is a root-level operator, and

4) $e_2,...,e_k$ are not root-level operators.

An **internal** IR expression is a prefix expression containing no root-level operators. An **operand to an operator** is either a single symbol that is itself an operand, or a sequence of symbols in V that form a Polish prefix expression that contains no root-level operators. The operand immediately following a binary operator is termed the **left** operand; the other operand is the **right** operand. An **IR expression** is a Polish prefix expression that does not violate any of the above properties of its operators. An IR **program** is a sequence of root-level IR expressions that represent a translation of some valid source program. Prefix expressions are a linear representation of expression trees. Root-level operators are those that must appear as the root

†For simplicity, it is assumed in this dissertation that all operands are either unary or binary. All of our methods are readily generalized to n-ary operators.

of the tree in the equivalent tree representation of an IR expression, while internal operators are those that may not appear as the root.

Arithmetic expressions are represented in the IR by their equivalent Polish prefix representation. The value of a variable is represented by a prefix expression describing an access path to its value at execution time, and the address of a variable is represented by an expression describing a way to compute its actual address at execution time. Likewise, control structures and procedure and function invocations must be expressed as well-formed prefix expressions. This is accomplished for higher level control structures by representing them as a sequence of lower-level operations. A **for** loop, for example, would be broken up into the basic operations of assigning the initial value to the control variable, assigning the limit value to a temporary variable, incrementing the control variable, and conditionally jumping to repeat the loop. It is important to note that root-level IR expressions cannot be embedded in one another. If a conditional jump operator is root-level, then the code generator cannot load a variable into a register, perform a test on its value and jump accordingly, and then increment and store the value back into the variable†. The basic operations required to implement the PASCAL **for** loop:

for $D := E$ **to** F **do** ... ; (3.3)

could be represented as:

```
        D := E;
        TEMP := F;
        goto 2;
1:      ...
        D := succ(D);
2:      if D ≤ TEMP then goto 1;
```

Actual IR code for this **for** loop is given later in this section.

The generation of IR is target implementation dependent. We therefore will make a few assumptions about the target computer and the implementation for which code is to be generated in order to allow (3.2) to be translated into IR. Assume that global variables reside in memory locations fixed at load time, and that local variables reside in memory locations addressed by the sum of a run time base address and a fixed displacement. If the base for local variables is kept in a register, and each statically more global base is kept in a memory location

†This restriction is imposed to facilitate proof of correctness of code generators. It does not restrict the implementation, since one could have distinct conditional jump operators — one root-level and one internal — which correspond to the same machine instruction.

that is pointed to by its immediately inner base (i.e. as a linked-list of bases headed by the local base, as in [Wirth72]), then (3.2) may be rewritten as:

$$:= \; + \; a \; r \; + \uparrow \; + \; b \uparrow r \uparrow c \qquad (3.4)$$

The r's stand for the register that contains the local base, a b and c stand for address constants, $\uparrow$ is a unary operator that computes the value of the memory location addressed by its operand, $+$ is integer addition and $:=$ is a binary operator that stores the value of its second operand into the memory location addressed by the value of its first operand. The address of the global variable C is then 'c', and its value is '$\uparrow c$'. The address of the local variable A is the constant 'a' plus the run time base, i.e. '$+ \; a \; r$'. Since only the local base is present in a register, B's base is found by following the linked-list headed by the local base. In this case B is one static level out, so the value of B's base is '$\uparrow r$', the address of B is '$+ \; b \uparrow r$', and the value of B is '$\uparrow + \; b \uparrow r$'.

If D, E, and F are local variables in the above implementation, then the IR code for the **for** loop in (3.3) would be:

```
        := + d r ↑ + e r
        := + temp r ↑ + f r
        j L2
: L1    . . .
        := + d r + ↑ + d r 1
: L2    ≤ L1 ? ↑ + d r ↑ + temp r
```

The root-level unary operator j is an unconditional jump operator that transfers control to the symbolic label specified by its operand. The root-level unary operator : defines the location of the label specified by its operand. The internal binary operator ? compares the values of its operands and leaves the result in the condition code register. The root-level binary operator $\leqslant$ transfers control to the label specified by its first operand if the value in the condition code register indicates that the first of the last two values compared was less than or equal to the second. The symbols in the IR are sometimes qualified by semantic information. Such qualification is discussed in section 3.2.

3.2. Modeling Computer Instruction Sets

The semantic description of an instruction is a prefix expression on the same level as that of the IR. Base and index address calculations are expanded fully to reveal the precise value that is computed by the instruction. However, a bit level description of the operations performed and their operands is not used, as it is more detailed than required. Instead operators

are typed according to the length of their operands. On the IBM 370, for example, there are several integer addition and memory reference operators. Full word instructions perform a full 32 bit integer addition and reference 32 bit memory words. Half word instructions also perform a 32 bit addition but only reference 16 bits of memory for the second operand. Index and base calculations, however, perform only 24 bit arithmetic, since the value being computed will be used as a memory address. Of course a 32 bit add may be used in any place that a shorter result is required if overflow conditions are either handled separately or ignored, but a 32 bit memory reference operator cannot be used in place of a 16 bit one. The implementor may specify cases where operator substitution is possible and thus the code generation algorithm has a wider choice of instruction patterns available to use, e.g. to implement index additions. This improves the quality of the object code generated.

The description of an instruction consists of three parts: the instruction pattern, the location of the result, and assembly information. The **instruction pattern** is a prefix expression composed of IR symbols specifying what the instruction computes. The **location of the result** is either an operand symbol representing a register or condition code or is non-existent†. The Greek letter lambda, λ, is used to specify that the location of the result is non-existent. The **assembly information** specifies the target computer instruction that computes the expression equal to the instruction pattern. Assembly information may either be symbolic assembly language or binary machine code. Each instruction is thought of as computing an expression, either for the value of the expression or for any side effects it may produce. A load instruction would compute the obvious value of the contents of the memory location or locations loaded into a register, while a store instruction computes an expression that one is primarily interested in for its side effect. The value calculated and used as the memory location in which to store is normally discarded after its use. The value stored into memory is normally of no interest to the code generator once the store has taken place. For the purposes of this section we will concentrate on the problem of picking machine instructions for the generated code and we will avoid any attempt to use the values left in registers from previous computations. Information retention as such is a simplified form of common subexpression elimination, and is not considered the responsibility of the code generator‡. Thus the location of the result of a store instruction is non-existent in so far as the code generator is concerned, as the values computed are no longer needed in order to generate code for the remainder of the program. In a similar manner, the location of the result of a jump instruction or procedure call does not compute a

†Notice that a value may be termed non-existent in this sense even though it is contained in memory.

‡This is not to suggest that such retention cannot be exploited by the compiler. Detection of such common subexpressions would take place prior to code generation and the information would be explicit in the IR input. (See section 4.3)

value that will be used in evaluating the current expression and therefore is considered non-existent.

There is a relationship that must hold between the instruction pattern and the location of the result in a valid TMDL description. If the instruction pattern is a root-level expression, i.e. if the leftmost symbol is a root-level operator, then the instruction computes (or completes the computation of) a well formed IR expression. The location of the result for such instructions must be λ. All other instructions compute expressions without a root-level operator, and must have a non-λ operand symbol for the result location.

The description of a typical indexed load instruction would be:

Result	IR Pattern	Assembly Information	
r	$::= (\uparrow + k\ r)$	"**load** r,k,r"	(3.5)

The meaning is to load the contents of the memory location addressed by adding the address constant k to the value of a register, into another register. The instruction pattern used by the code generator for this instruction would be '$\uparrow + k\ r$'.

In (3.5) each of the registers that appear in the instruction pattern may correspond to any actual register in the target machine. There are no restrictions on which ones they may be. Certain instructions contain semantic restrictions on some of their registers in that they must be the same actual register as another register in the instruction pattern. These instructions cannot be fully described by context-free instruction patterns. An example of such an instruction is a two address memory-to-memory integer add instruction that adds the contents of two memory locations and places the sum in one of the locations. A description of such an instruction would be:

$\lambda ::= (:= k + \uparrow k \uparrow k)$ "**madd** k,k" (3.6)

Clearly some mechanism must be provided to inform the code generator that in order to be able to use this instruction the first k must be the same as one of the subsequent ks. The solution is to add semantic qualifiers to the symbols of the instruction set description language that specify equivalences between symbols of the same kind that appear in a single instruction. These qualifiers are specified by a dot, '.', followed by an integer index. A qualifier may appear after any input symbol that needs this kind of semantic information. The net result is that this instruction cannot 'match' unless all symbols of the same kind and index have equivalent value. For example, (3.6) would be described by:

$$\lambda ::= (:= k.1 + \uparrow k.1 \uparrow k.2) \qquad \text{"}\mathbf{madd}\ k,k\text{"} \qquad (3.7)$$
$$\lambda ::= (:= k.1 + \uparrow k.2 \uparrow k.1) \qquad \text{"}\mathbf{madd}\ k,k\text{"}$$

There are two patterns for this instruction since the operator + is commutative. Notice that $k.1$ and $k.2$ could be the same constant. The qualifications indicate which symbols must be the same, but not which are distinct. (If it were necessary to indicate distinctness, a suitable convention could, of course, be established.)

Additionally, some instructions, such as increment or clear instructions, require specific constants or registers to appear in their descriptions. A specific constant is indicated in an instruction description by following the symbol defining its class (such as k) by an equal sign, =, and the actual constant. Thus increment and clear instructions would be described by:

$$r ::= (+\ r\ k{=}1) \qquad \text{"}\mathbf{inc}\ \ r,r\text{"} \qquad (3.8)$$
$$r ::= (k{=}0) \qquad \text{"}\mathbf{clr}\ \ r\text{"}$$

The first instruction pattern indicates that the contents of a register is to be incremented by one and that the result may be placed into any register, since there is no semantic restriction on the result register. If the instruction were to require that the result be put back into the original register, then the description would be:

$$r.1 ::= (+\ r.1\ k{=}1) \qquad \text{"}\mathbf{inc}\ \ r\text{"} \qquad (3.9)$$

The assembly information field of an instruction description contains either a symbolic or a bit-level description of the object instruction being described. In this dissertation it will be assumed that symbolic assembly language is to be generated, though in an actual production compiler one may prefer to generate object modules directly. This assumption does, of course, imply an additional pass of invoking an assembler, but is conceptually the same and avoids a messy language for describing machine code instruction formats at the bit level. A final assembly pass does have its advantages and is used by several production compilers, such as the C compiler in the UNIX operating system [Ritchie76]. This organization allows the assembler to solve problems such as forward referencing, choosing long vs. short jump instructions [Szymanski78], generating loader and debugger compatible object modules, performing peephole optimization, and locating and utilizing machine dependent idioms.

In a symbolic instruction description, there must be a way to bind specific symbols in the instruction pattern to symbols in the symbolic description. This is indicated by appending a dot, '.', followed by the digit corresponding to the associated symbol in the instruction pattern to the symbol appearing in the assembly field. Since every field in the symbolic description

must be associated with a specific symbol in the instruction pattern, all symbols in the pattern that represent a class of values must be semantically qualified. Thus the complete description of the instructions in (3.7) through (3.9) would be:

λ ::= (:= *k*.1 + ↑ *k*.1 ↑ *k*.2)	"**madd**	*k*.1,*k*.2"
λ ::= (:= *k*.1 + ↑ *k*.2 ↑ *k*.1)	"**madd**	*k*.1,*k*.2"
r.1 ::= (+ *r*.2 *k*=1)	"**inc**	*r*.1,*r*.2"
r.1 ::= (+ *k*=1 *r*.2)	"**inc**	*r*.1,*r*.2"
r.1 ::= (*k*=0)	"**clr**	*r*.1"
r.1 ::= (+ *r*.1 *k*=1)	"**inc**	*r*.1"
r.1 ::= (+ *k*=1 *r*.1)	"**inc**	*r*.1"

Again, two descriptions are required for a complete listing of the commutative operations, so the **inc** and **madd** instructions each appear twice.

Similarly, semantic qualifications of certain operand symbols in the IR may be necessary. This normally occurs when a symbol may represent any single value in a given set. The exact value represented is indicated by appending a dot, '.', followed by that value. Thus, the IR input:

+ *r.5 k.100*

indicates that the actual register **r5** and the specific constant value of 100 are to be used in the final code generated.

Some instructions require that an operand be in an adjacent pair of registers. Register operands in a TMDL description that correspond to register pairs (see section 3.3 for a discussion of how this is done) must be treated differently in the assembly field of an instruction description. Normally, only one of the actual registers in a register pair appears in an assembly language program. Which one is specified in the assembly information field by following the register description by an additional dot, '.', and a 1 or 2 indicating whether the first or second register's name should be substituted in the code output. For example, letting **d** designate a double register, the double register right-shift instruction:

d.1 ::= (→ *d*.1 *k*.1) "**rsd** *d*.1.1,*k*.1"

indicates that the first register of a register pair is used in an assembly language description of a right-shift-double instruction.

The temptation to use the code generator to perform optimizations that are not directly related to choosing the output instructions has deliberately been resisted. It would be possible for the code generator to perform additional optimizations, such as constant folding, by a

suitable extension. For example, the instruction pattern:

$$k=(k.1+k.2) ::= (+\ k.1\ k.2) \quad \text{``''}$$

could be used to fold the addition of integer constants. The code generator is not the appropriate place to perform such optimization; the instruction pattern '+ *k*.1 *k*.2' is not related to the target machine but to the programming language, and therefore does not belong in a target machine description. It also needlessly complicates the code generator, requiring an interpreter to evaluate arithmetic expressions.

3.3. The Structure of TMDL

A TMDL program consists of four sections. The first is an option statement that is used for requesting printed output of the code generator's tables and for debugging. The second is a list of the registers available on the target computer. The third defines classes of registers to facilitate the instruction description, and contains a list of the terminal symbols expected as the input (the symbols in the IR) and the kind of semantic information they will contain. The last section is a description of the instructions on the target computer.

The option statement consists of the string '$options' followed by a comma separated list of the options that are to be set. It is terminated by a semicolon. Thus:

```
$options  trace,codetables;
```

would indicate that 'trace' and 'codetables' are to be set. This statement is optional, and may be omitted if no options are selected.

The second section lists the actual registers on the target computer. They are separated into two groups: allocatable and dedicated. The allocatable registers are available for the register allocator to use in any manner desired, while the dedicated registers are reserved for specific functions and can only be used when explicitly requested. A sample register description is:

```
$registers
  $allocatable {r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12};
  $dedicated   {r0,r1,r13,r14,r15};
```

When symbolic assembly information is used in the instruction set description, the register names will be output as they appear in the register description.

The next section describes the symbols that are to be used in the grammar for code generation and instruction description. It is divided into two parts for nonterminals and terminals.

The nonterminal section defines logical classes of either actual registers or ordered pairs of actual registers. Each logical class is bound to a nonterminal in the instruction set grammar. All members of a class are assumed by the register allocator and code generator to be indistinguishable. They are used interchangeably unless a particular register is specifically indicated in an instruction description or in the IR input. An actual register may appear in several logical register classes. A sample register class definition is:

```
$symbols
  $nonterminals
    r = r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
    d = <r2,r3>,<r4,r5>,<r6,r7>,<r8,r9>,<r10,r11>;
    e = r2,r4,r6,r8,r10;
    o = r3,r5,r7,r9,r11;
```

The nonterminal **r** defines the total set of registers on the target machine. The nonterminals **d**, **e**, and **o** designate the allocatable double, even, and odd registers, respectively.

The description of terminal symbols serves two purposes. It allows a minimal amount of error checking of the IR input and it classifies terminal symbols as either constants, unary operators, or binary operators. The latter information is used in determining 'escape routes' from potentially looping configurations in the code generator. The terminal symbols must include descriptions of all symbols that will appear in the IR. Otherwise there would be IR code sequences for which it would be impossible to generate code. A sample terminal symbol section for a simplified IR follows:

```
$terminals
  k: 0,32767;
  +,-,*,/,=: binary;
  :=,↑: unary;
```

The binding for *k* indicates that it is an operand, the values of which can range from 0 to 32767.

The final section of a TMDL program describes the instruction set of the target computer. It is headed by the string '$instructions' and concluded by '$end'. Since most computer character sets do not contain λ, a period, '.', may be used in the TMDL program instead.

3.4. A Sample Machine Description

An example target machine description will now be presented. The target computer used is a standard von Neumann computer with an array of identical general purpose registers. A somewhat idealized computer is used to avoid an unnecessarily complex code generation example in Chapter 4. The instruction set uses primarily a one address scheme with a single memory address field and two register fields. One register is used as an index register, when index mode is specified, and as an operand in register to register mode. The other register is the second operand and the location of the result of arithmetic operations. There are four primary addressing modes: immediate, direct, immediate indexed, and indexed direct. Additionally, there are two register addressing modes where either the second operand or its address in memory resides in that register. When a jump or store instruction is executed, the addressing scheme is changed to represent one additional level of indirection. The value of the operand is used as the address at which to store the result. The machine description with a subset of the actual instruction set appears in Fig. 3.1. For simplicity, only the operations of load, add, and store are used in this example.

The instruction '*r*.1 ::= (+ *k*.1 *r*.1) "**add** *r*.1,=*k*.1" ' is not included because the instruction '*r*.2 ::= (+ *k*.1 *r*.1) "**load** *r*.2,=*k*.1,*r*.1" ' does the same computation and is more general since the result of the addition may be placed in a different register. Also, the fact that the **add** instructions are commutative is reflected by the two patterns for each actual instruction, such as lines 17 and 18. It can be inferred from the instruction descriptions that := is a root-level operator, where as + and ↑ are internal operators.

```
TMDL Preprocessor  V0.0  01/Jun/77                    14 Jun 77   13:34:01

 1*  $options statesets,tables;
 2*
 3*  $registers
 4*       $allocatable {r0,r1,r2,r3,r4};
 5*       $dedicated   {r5,r6,r7};
 6*
 7*  $symbols
 8*       $nonterminals
 9*         r = r0,r1,r2,r3,r4,r5,r6,r7;
10*       $terminals
11*         k: 0,32767; + binary; ↑ unary; := binary;
12*
13*  $instructions
14*
15*       r.2 ::= (+ ↑ + k.1 r.1 r.2)              "add    r.2,k.1,r.1";
16*       r.1 ::= (+ r.1 ↑ + k.1 r.2)              "add    r.1,k.1,r.2";
17*       r.1 ::= (+ ↑ k.1 r.1)                    "add    r.1,k.1";
18*       r.1 ::= (+ r.1 ↑ k.1)                    "add    r.1,k.1";
19*       r.1 ::= (+ r.1 r.2)                      "add    r.1,r.2";
20*       r.2 ::= (+ r.1 r.2)                      "add    r.2,r.1";
21*       λ ::= (:= ↑ + k.1 r.1 r.2)               "store  r.2,*k.1,r.1";
22*       λ ::= (:= + k.1 r.1 r.2)                 "store  r.2,k.1,r.1";
23*       λ ::= (:= ↑ k.1 r.1)                     "store  r.1,*k.1";
24*       λ ::= (:= k.1 r.1)                       "store  r.1,k.1";
25*       λ ::= (:= r.1 r.2)                       "store  r.2,r.1";
26*       r.2 ::= (↑ + k.1 r.1)                    "load   r.2,k.1,r.1";
27*       r.2 ::= (+ k.1 r.1)                      "load   r.2,=k.1,r.1";
28*       r.2 ::= (+ r.1 k.1)                      "load   r.2,=k.1,r.1";
29*       r.2 ::= (↑ r.1)                          "load   r.2,*r.1";
30*       r.1 ::= (↑ k.1)                          "load   r.1,k.1";
31*       r.1 ::= (k.1)                            "load   r.1,=k.1";
32*
33*  $end
```

Fig. 3.1. Sample Machine Description.

Chapter 4: The Code Generation Algorithm

This chapter describes a syntax driven code generation algorithm used in the machine independent compiler outlined in Chapter 2. The code generator is based on a deterministic shift-reduce parser. The state set and moves of the parser are determined by the semantics of the instruction set of the target computer. Tables defining the automaton are automatically constructed by a preprocessor from a description of the instruction set for the target machine, specified in TMDL. This provides a high degree of flexibility in the code generator, a section of the compiler that is generally considered non-transportable. Sufficient conditions are developed that insure that the code produced by the code generator is correct. The input to the coder is a low level prefix internal representation of the program being compiled, tailored to a specific target implementation. The coder can be thought of as first performing a context free parse of the IR string using the instruction set as grammar rules. As each reduction is signalled, "semantic" routines are invoked. Finally, the corresponding object instruction is emitted.

In practice, the context free grammar defined by the instruction set of a computer is almost certain to be ambiguous. There are usually several different instruction sequences on any given computer that produce the same effect. The problem is further complicated by the desire to produce the best possible code. The situation is analogous to finding the shortest (according to some cost criteria) parse of an input string using an ambiguous grammar. Many optimizations used in compilers can be thought of as an attempt to determine the order of reductions that will produce the best code. Of course even finding the shortest parse defined by an ambiguous context free instruction set grammar in no way insures that optimal code has been generated. Common subexpression elimination, for example, which cannot be performed in a context free code generation scheme, can further reduce the cost of computing many expressions.

The code generation algorithm presented is not an optimizer. Instead, its goal is to translate the IR input into a sequence of object code instructions that **correctly** implements the desired operations on the target computer. It must at the same time avoid producing poor code. To do so, the algorithm uses a simple heuristic in an attempt to emit the instructions that correspond to the longest instruction patterns in the input string. Target machine dependent code sequences and implementation dependent constructs are used in generating the IR. This serves two purposes. It is the vehicle by which the code generator is informed of the details of how each language feature is to be implemented on a particular target machine. It also aids the code generator in matching longer instruction patterns that represent some special

instruction on that target machine, thus resulting in better code. Target computer parameterized optimizations can be used to improve the overall quality of code generated, as can language dependent, machine independent optimizations. One important property of the code generation algorithm is that it can utilize many kinds of optimization information, when present. Included are the abilities to take advantage of live-dead variable analysis and common subexpression information (see section 4.3).

4.1. Shift-Reduce Parsing Algorithm

The code generation algorithm used in the compiler described in Chapter 2 is a modified deterministic LR(1)-like shift-reduce parser [Aho72a]†. Because the context free grammar being used to generate code is usually ambiguous, standard context free parsing methods of linear complexity do not apply. This section describes the modified shift-reduce parsing algorithm. Subsequent sections explain how the parser is turned into a code generator, and how the tables for such a parser are constructed from an instruction set grammar. Several algorithms are developed that allow the parser constructor to determine the instances where the parser can loop, to eliminate the possibility of looping without altering the IR inputs for which code can be generated, and to test for and eliminate the possibility of the code generator blocking on valid input (or inform the user that the IR input must satisfy certain restrictions, to be checked by the implementor). It is therefore possible to determine whether the coder will generate code for all valid inputs or not.

The grammar used by the parsing algorithm is a set of TMDL instruction descriptions, as discussed in Chapter 3. The instruction grammar is slightly different from a conventional context-free grammar, but the differences are for notational convenience only. The vocabulary **V** is the set of symbols defined in the third section of the TMDL description, where **N** denotes the set of nonterminals and **T** denotes the terminal symbols. However, unlike the more usual situation, elements of **T** may also occur in **N**, meaning that some nonterminals can also occur in the IR. (Typically, this facility is used to specify base registers in the input.) The corresponding context-free grammar would have, for example, a non-terminal **r**, a terminal *r*, and a rule **r** ::= *r* with no associated code generation.

The other difference from conventional grammars is the special left hand side λ, which designates "no result". To obtain a corresponding conventional grammar, one would replace λ by a nonterminal, say L, and add the rules $S ::= L \mid S\,L$. In other terms, the instruction grammar generates or describes a sequence of derivations from λ. (Note that λ is not an element of

†The reader is assumed to be familiar with LR(1) parsing and parser construction.

N or V.)

In the discussion that follows, we refer to our notation as a **grammar G**. The language generated by G in the fashion just described is denoted **L(G)**.

A shift-reduce parser consists of a pushdown stack and a finite control with a finite set of states **Q**. The pushdown stack contains the start state $\mathbf{q}_0$ at the bottom, followed by 0 or more repetitions of pairs of entries, consisting of a vocabulary symbol followed by a state symbol†. The state symbol at the top of the stack always designates the current state. The finite state control determines the action of the coder as a function of its current state and the current input symbol. Input is read and pushed onto the stack until the right hand part of a grammar rule is on the top of the stack (disregarding the intervening state symbols). The parser may then perform a reduction using one of the variants of that rule. (The choice is made by a semantic routine.) The top symbols corresponding to the right hand part of the grammar rule (and intervening state symbols) are popped and the nonterminal on the left of that rule is pushed onto the stack, followed by the new current state. The parser then continues to shift and reduce until it has reduced the entire input string to a single start symbol, at which point it reports success and halts. The parser also detects erroneous input and informs the user. (Such errors normally indicate bugs in earlier stages of the compiler.)

At any given time the parser is in a specific **state, q,** in **Q**. After each action of shifting or reducing, the parser moves to a possibly different state. The action of the parser is determined by two functions, called **ACTION** and **NEXT**. These functions are defined on **Q** × **V**. Values of NEXT are single states in **Q**, and values of ACTION come from the set {**shift**, **reduce** *R*, **accept**, **error**}. When the parser is in state **q** and the next input symbol is *x*, the action of the parser is uniquely determined by the value of ACTION(**q**,*x*). If the action is **shift**, then *x* is pushed onto the stack, followed by the unique state symbol **q′** = NEXT(**q**,*x*), the input is advanced, and the parser moves to state **q′**. If the action is **reduce** *R*, then the parser performs a reduction using a grammar rule **r** ::= α in the set *R*. All rules in *R* have the same syntactic right hand part but may differ in their semantic restrictions or their left hand parts. The particular rule used is determined from the semantics of the rules and the symbols on the stack (as described in section 4.2). The instruction pattern α will always be on the top of the stack at the start of the reduction. The parser then pops the stack $2|\alpha|$ times, uncovering the state **q′** that it was in just prior to reading the first symbol in the prefix expression just computed. The left hand part of the rule, **r**, is pushed onto the stack, followed by **q″** = NEXT(**q′**,**r**), moving the parser into state **q″**, and the process continues. If the action is

†As usual, the vocabulary symbols could, in theory, be omitted. However, we find them convenient.

accept then the parser halts and reports that it has successfully parsed the input string†. If the action is **error** then it reports that there is no parse for the input string and halts.

A **configuration** of the parser is a snapshot of the parser at an instance while it is parsing. It is shown as:

(**stack** → *input$*)

where **stack** is the sequence from bottom to top of the symbols on the pushdown stack, *input* is the unread part of the input, *$* is an implicit end of input marker (not normally shown when input remains), and the top stack symbol **q** is the current state. Unread input symbols are shown in *italics*, and symbols on the stack are shown in **bold** type. The **initial configuration** is the configuration in which the parser always begins, with the start state symbol $\mathbf{q_0}$ on the stack, none of the input having been read, and in the start state $\mathbf{q_0}$. A **move** of the parser is the performing of a single action, taking it from one configuration to another. A move is represented as:

$$(\mathbf{s_1} \rightarrow i_1) \vdash (\mathbf{s_2} \rightarrow i_2)$$

A single move operation is represented by the operator $\vdash$, *n* moves in succession by $\vdash^n$, one or more moves by $\vdash^+$, and zero or more by $\vdash^*$. Thus, a **shift** operation is represented as:

$$(\ldots\mathbf{aq_k bq_i} \rightarrow xyz\ldots) \vdash (\ldots\mathbf{aq_k bq_i xq_j} \rightarrow yz\ldots)$$

where ACTION($\mathbf{q_i}$,x) = **shift** and NEXT($\mathbf{q_i}$,x) = $\mathbf{q_j}$. A **reduce** would be:

$$(\ldots\mathbf{aq_k b_1 q_l \ldots b_m q_i} \rightarrow xyz\ldots) \vdash (\ldots\mathbf{aq_k wq_j} \rightarrow xyz\ldots)$$

using the rule '**w** ::= $\mathbf{b_1 \ldots b_m}$' for the reduction, where NEXT($\mathbf{q_k}$,w) = $\mathbf{q_j}$. A total parse to acceptance would be:

$$(\mathbf{q_0} \rightarrow input) \vdash^* (\mathbf{q_i} \rightarrow \$)$$

with ($\mathbf{q_0}$ → *input*) being the initial configuration and ACTION($\mathbf{q_i}$,$\$$) being **accept**. In the code generator under discussion the only **accept** action occurs when the parser is in the initial state,

†Note that by definition the action is **accept** only when no input remains.

q_0. The code generation algorithm appears on the following page as Algorithm 4.1. The construction of the ACTION and NEXT functions is discussed in section 4.5.

4.2. Adding Semantic Information to the Parser to Make a Code Generator

The shift-reduce parsing algorithm presented so far is not sufficient to select instructions for the target machine. The primary function addressed so far has been the matching of instruction patterns in a machine description with input strings in an intermediate language. Missing is a set of provisions for handling the semantic information needed to generate final object code, such as which register **r** stands for or which constant *k* is. This shortcoming can be handled by carrying semantic information along with the state information on the coder's stack. For example, when a **shift** is performed with the input symbol **r** standing for a register, the specific register represented by that **r** is also pushed onto the stack. Then when a **reduce** is done, the semantic information necessary to generate a final instruction can be read off the stack.

The addition of semantic restrictions to the instruction description allows a greater number of special instructions to be described. It also complicates the meaning of the **reduce** operation in the shift-reduce code generation algorithm. A single instruction pattern may correspond to more than one instruction as the introduction of semantic restrictions may require the duplication of some instruction patterns by forcing some commutative operations to be represented by two identical patterns (e.g. **add** *r*.1,*r*.2 and **add** *r*.2,*r*.1) and by allowing one instruction to be specified as a special case of another (e.g. **inc** *r*.1 and **add** *r*.1,=*k*.1). It also may be that two distinct instructions compute the same expression but leave the result in registers of different classes (e.g. **loadx** *x*.1,α and **loadr** *r*.1,α). Thus, when a **reduce** is performed, there may be a list of rules from which to choose instead of a single rule.

The rule used when a **reduce** operation contains more than one rule is picked by a simple heuristic. Rules are ordered by the preprocessor into a 'best instruction first' sequence. At code generation time, the set of rules for a specific **reduce** action is tested in that order until an instruction is found that is semantically compatible with the information on the top of the stack. Since all instructions in a reduce set have the same instruction pattern, the 'cheapest', according to some cost criteria, instructions are tested first. If the length of an object instruction in bits is used as the cost factor, a 16 bit long increment instruction will be tested before a 32 bit long add immediate instruction, since the basic patterns are identical but the 'cost' of the increment instruction is less.

It is also possible, if all the instructions in the set have semantic restrictions, that more than one instruction is generated. This situation is discussed in section 4.5.7.

Algorithm 4.1 — The Code Generator

Input: ACTION and NEXT functions (represented in matrices) derived from machine **M**'s description, and the IR of a program, **P**, being compiled.

Output: An assembly language program for **P** on machine **M**.

Method: Perform a Shift-Reduce parse of the IR input, emitting target instructions whenever reductions are performed.

Initialization: Set the parser's state, S, and the stack to $\mathbf{q}_0$.

Step 1: Set the look-ahead symbol, *u*, to the next input symbol. If all the input has been read, set *u* to the end of input symbol, *$*.

Step 2: Perform the action in ACTION[S,*u*]:

shift: Push *u* onto the stack. Then set S to the value of NEXT[S,*u*], push S onto the stack, and advance the input one symbol. Go to step *1*.

reduce *R*: Output an instruction from set *R* or an equivalent sequence of instructions (see section 4.5.7). If the rule used is '$\mathbf{r} ::= \alpha$' then pop the stack $2|\alpha|$ times and set S to the state on the top of the resulting stack. If $\mathbf{r} \neq \lambda$ then push **r** onto the stack, set S to the value of NEXT[S,**r**] and push S onto the stack. Go to step *2*†.

accept: The code generator halts. Code has been generated for the entire IR input string. This can only happen when all of the input has been read (i.e. *u* is *$*) and the stack is empty (i.e. S is $\mathbf{q}_0$).

error: Issue an error message and halt. The IR input has no parse using the underlying grammar for M's instruction set.

---

†If $\mathbf{r} \in \mathbf{N}$, then ACTION[S,**r**] must be **shift**.

4.3. Register Allocation

Register allocation is handled automatically by the code generator. The machine description specifies how many and what kind of registers exist on the target computer, and which ones are available to the register allocator. Each nonterminal in the grammar, in general, represents a logical register or a class of logical registers. Each logical register is associated with an actual machine register or pair of registers. In this way information such as the fact that **d0** is the register pair **<r0,r1>** is included in the machine description.

Each actual register has a descriptor associated with it that describes its status. This description includes a Boolean flag indicating whether the register is allocatable or not and a use count field that indicates the number of times that the value in that register will be used before it is discarded. Normally, i.e. without common subexpression elimination, this value is either zero or one. A register is **busy** if it has a non-zero use count, and is **free** if it has a zero use count. Any free register may be allocated. When a register is allocated its use count is incremented by one, and after its use it is decremented by one. Initially the use count for all allocatable registers is set to zero.

Register allocation occurs as a subtask to a **reduce** operation. After an instruction pattern has been semantically verified, the use count for each actual register in the instruction pattern is decremented. Then, if the result of the instruction is non-λ, and if the result register is not semantically linked to any other register in the instruction pattern, the register allocator is asked for a register of the appropriate class. The register allocator then returns a free register in that class, after setting its use count to 1. If the result register, **r**, is semantically linked to a register in the instruction pattern, then **r** must be used as the target register and its use count set to one (since it contains a newly computed value). Note that in the presence of common subexpressions the semantic test for instruction compatibility must include a check to see if there will be an error in this case. If **r** is linked to another register, then **r** must have no other references outside the scope of that instruction pattern. If it does, i.e. if the use count is greater than the number of references to **r** in that instruction pattern, then since the execution of that instruction would alter the contents of **r** the other uses would then be incorrect. When this occurs, the code generator allocates a new register, **r'**, in the same class as **r**, issues an instruction to move the contents of **r** to **r′** and then issues the first instruction using **r′** as the target register.

A slight addition to the register allocation scheme is required to utilize common subexpression (CSE) information. A binary operator, ○, is added to the IR that takes an integer constant as its first operand and an arbitrary expression as its second. The meaning of this operator is that the second operand is a common subexpression that is to be used the number of times specified by its first operand. Each occurrence of the operator ○, i. e. each designation of an

expression as a CSE, is implicitly numbered sequentially from 1 as it is read. The first place that a CSE is used is where it is computed, as an ordinary operand. Subsequent uses are indicated by a 'use CSE' operator, ⊗, which is a unary operator taking a constant as its operand, indicating which CSE it represents. For example, the two statements:

$$A := B + C; \quad D := B + C;$$

would have an IR representation of:

$$:= k.a \bigcirc k.2 + \uparrow k.b \uparrow k.c \quad := k.d \otimes k.1$$

Upon encountering a define CSE operator, the code generator sets the use count of the register containing the CSE's value to the number of times that it will be used. Thus, that value will be preserved until its final use because that register will remain busy. This is equivalent to adding a special grammar rule:

$$r.1 ::= (\bigcirc\ k.1\ r.1)$$

for each logical register class to the machine description with the semantics that the use count for register $r.1$ is set to $k.1$. The mechanism does not have to actually be implemented in this manner — it is simpler to 'hard-wire' it into the code generator. Likewise, when a use CSE operator is encountered, the actual register that contains that value is substituted into the IR stream, equivalent to a grammar rule of the form:

$$r.1 ::= (\otimes\ k.1)$$

with the associated semantics of using the actual register that contains CSE number $k.1$ for $r.1$.

4.4. A Simple Example

A simple code generation example is presented next. The target machine is the one described at the end of Chapter 3. The instruction set description is repeated in Fig. 4.1 for easier reference. A table summarizing the parser produced appears in Fig. 4.2. In the table, the ACTION and NEXT functions are represented in the following format: Columns of the table are headed by symbols in **V** and rows correspond to states in **Q**. An entry in row **q**, column ν of the table is of the form '**a**:**b**', where **a** always represents ACTION($\mathbf{q}, \nu$) and **b** is NEXT($\mathbf{q}, \nu$) when **a** is **shift**, and the rule set by which to reduce when **a** is **reduce**. The

ACTION values **shift, reduce, accept, error,** are abbreviated respectively S, R, A, E.

1*	*r*.2 ::= (+ ↑ + *k*.1 *r*.1 *r*.2)	"**add**	*r*.2,*k*.1,*r*.1";
2*	*r*.1 ::= (+ *r*.1 ↑ + *k*.1 *r*.2)	"**add**	*r*.1,*k*.1,*r*.2";
3*	*r*.1 ::= (+ ↑ *k*.1 *r*.1)	"**add**	*r*.1,*k*.1";
4*	*r*.1 ::= (+ *r*.1 ↑ *k*.1)	"**add**	*r*.1,*k*.1";
5*	*r*.1 ::= (+ *r*.1 *r*.2)	"**add**	*r*.1,*r*.2";
6*	*r*.2 ::= (+ *r*.1 *r*.2)	"**add**	*r*.2,*r*.1";
7*	λ ::= (:= ↑ + *k*.1 *r*.1 *r*.2)	"**store**	*r*.2,**k*.1,*r*.1";
8*	λ ::= (:= + *k*.1 *r*.1 *r*.2)	"**store**	*r*.2,*k*.1,*r*.1";
9*	λ ::= (:= ↑ *k*.1 *r*.1)	"**store**	*r*.1,**k*.1";
10*	λ ::= (:= *k*.1 *r*.1)	"**store**	*r*.1,*k*.1";
11*	λ ::= (:= *r*.1 *r*.2)	"**store**	*r*.2,*r*.1";
12*	*r*.2 ::= (↑ + *k*.1 *r*.1)	"**load**	*r*.2,*k*.1,*r*.1";
13*	*r*.2 ::= (+ *k*.1 *r*.1)	"**load**	*r*.2,=*k*.1,*r*.1";
14*	*r*.2 ::= (+ *r*.1 *k*.1)	"**load**	*r*.2,=*k*.1,*r*.1";
15*	*r*.2 ::= (↑ *r*.1)	"**load**	*r*.2,**r*.1";
16*	*r*.1 ::= (↑ *k*.1)	"**load**	*r*.1,*k*.1";
17*	*r*.1 ::= (*k*.1)	"**load**	*r*.1,=*k*.1";

Fig. 4.1. Sample Instruction Set Description.

Code will now be generated for the expression:

$$A := B + C$$

where the variables are addressed as those in (3.4). The equivalent IR expression is:

$$:= + \ a \ r.7 \ + \uparrow \ + \ b \uparrow r.7 \uparrow c$$

(By convention, **r7** is being used as the base register.) The initial configuration of the coder is that it is in the start state, with an empty stack, and reading the input symbol :=, where the lower case alphabetic characters *a*, *b*, and *c* stand for address constants '*k*'. The semantic information associated with input and stack symbols is also shown in the configurations, but the stack symbols themselves are omitted. The actual value of a symbol is appended to it with a dot. The initial configuration appears as:

$$(\mathbf{1} \ \rightarrow \ := + \ k.a \ r.7 \ + \uparrow \ + \ k.b \uparrow r.7 \uparrow k.c)$$

The action specified by ACTION(**1**,:=) is **shift** and NEXT(**1**,:=) is **2**, so the next configuration of the coder is:

Code Generator Move Table						
	$	r	k	+	↑	:=
1*	ACCEPT					S: 2
2*		S: 3	S: 4	S: 5	S: 6	
3*		S: 7	S: 8	S: 9	S: 10	
4*		S: 11	S: 8	S: 9	S: 10	
5*		S: 12	S: 13	S: 9	S: 14	
6*		S: 15	S: 16	S: 17	S: 10	
7*	R: 11	R: 11	R: 11	R: 11	R: 11	R: 11
8*	R: 17	R: 17	R: 17	R: 17	R: 17	R: 17
9*		S: 12	S: 18	S: 9	S: 14	
10*		S: 15	S: 19	S: 20	S: 10	
11*	R: 10	R: 10	R: 10	R: 10	R: 10	R: 10
12*		S: 21	S: 22	S: 9	S: 23	
13*		S: 24	S: 8	S: 9	S: 10	
14*		S: 15	S: 25	S: 26	S: 10	
15*	R: 15	R: 15	R: 15	R: 15	R: 15	R: 15
16*		S: 27	S: 8	S: 9	S: 10	
17*		S: 12	S: 28	S: 9	S: 14	
18*		S: 29	S: 8	S: 9	S: 10	
19*	R: 16	R: 16	R: 16	R: 16	R: 16	R: 16
20*		S: 12	S: 30	S: 9	S: 14	
21*	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}	R: {5,6}
22*	R: 14	R: 14	R: 14	R: 14	R: 14	R: 14
23*		S: 15	S: 31	S: 32	S: 10	
24*		S: 33	S: 8	S: 9	S: 10	
25*		S: 34	S: 8	S: 9	S: 10	
26*		S: 12	S: 35	S: 9	S: 14	
27*	R: 9	R: 9	R: 9	R: 9	R: 9	R: 9
28*		S: 36	S: 8	S: 9	S: 10	
29*	R: 13	R: 13	R: 13	R: 13	R: 13	R: 13
30*		S: 37	S: 8	S: 9	S: 10	
31*	R: 4	R: 4	R: 4	R: 4	R: 4	R: 4
32*		S: 12	S: 38	S: 9	S: 14	
33*	R: 8	R: 8	R: 8	R: 8	R: 8	R: 8
34*	R: 3	R: 3	R: 3	R: 3	R: 3	R: 3
35*		S: 39	S: 8	S: 9	S: 10	
36*		S: 40	S: 8	S: 9	S: 10	
37*	R: 12	R: 12	R: 12	R: 12	R: 12	R: 12
38*		S: 41	S: 8	S: 9	S: 10	
39*		S: 42	S: 8	S: 9	S: 10	
40*	R: 7	R: 7	R: 7	R: 7	R: 7	R: 7
41*	R: 2	R: 2	R: 2	R: 2	R: 2	R: 2
42*	R: 1	R: 1	R: 1	R: 1	R: 1	R: 1

Fig. 4.2. Code Generation Table for Example.

(1 2 → + *k.a r.7* + ↑ + *k.b* ↑ *r.7* ↑ *k.c*)

In a similar manner, the next 9 configurations are:

(1 2 5 → *k.a r.7* + ↑ + *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a → *r.7* + ↑ + *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 → + ↑ + *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 → ↑ + *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 → + *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 → *k.b* ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 35.b → ↑ *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 35.b 10 → *r.7* ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 35.b 10 15.7 → ↑ *k.c*) ⊢

"load r1,*r7"

In this configuration the coder issues instruction 15: "**load** *r*.2,**r*.1" which has the pattern '*r*.2 ::= (↑ *r*.1)'. In the instruction pattern the logical register '*r*.1' is semantically linked to the actual register **r7**. The logical register '*r*.2' is not bound to any register in the instruction pattern, so the register allocator is used to determine that it will be bound to the actual register **r1**. Then the top 2 states are popped off the stack, revealing state 35 on the top of the stack. The actual register **r1** is bound to the result symbol '*r*.2', the current state is set to NEXT(35,r) = 39, and 39.1 is pushed onto the stack. The remainder of the coder's configurations follow:

(1 2 5 13.a 24.7 9 14 26 35.b 39.1 → ↑ *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 35.b 39.1 10 → *k.c*) ⊢
(1 2 5 13.a 24.7 9 14 26 35.b 39.1 10 19.c → $) ⊢

"load r2,*c*"

(1 2 5 13.a 24.7 9 14 26 35.b 39.1 42.2 → $) ⊢

"add r2,*b*,r1"

(1 2 5 13.a 24.7 33.2 → $) ⊢

"store r2,*a*,r7"

(1 → $) accept

4.5. Automatic Construction of a Code Generating Parser

The algorithm for building the table used by the code generator is presented next. For conceptual clarity, we present the algorithm in several stages — i.e., an initial table is computed which is then modified. These modifications could be incorporated in the initial algorithm. The discussion of each modification motivates its necessity and indicates how this incorporation is done.

4.5.1. The Initial Table Construction

This section presents an algorithm for computing, from an instruction set description, the initial ACTION and NEXT functions for a parser that will generate object code. The method is similar to the algorithm for building SLR(1) parse tables.

It should be emphasized that the table construction algorithm presented will accept any context free grammar as input, not necessarily unambiguous, and will produce a deterministic, shift-reduce parser as its output. However, in the general case, the language accepted by the resulting parser is not guaranteed to be the same language defined by the grammar. If the grammar satisfies certain sufficient conditions presented later in this chapter, then it is possible to build the tables in such a way that the language accepted by the parser is the same as that defined by the grammar. Other linear parsing algorithms accepting ambiguous context free grammars allow a much wider set of grammars to be used, but fail to provide any assurance that the resulting parser will not loop or block on valid input [Aho75]. While it is relatively safe to use such methods in carefully controlled situations where the consequences are well understood, (such as the dangling **else** problem, or defining the precedence or associativity of specific infix operators), it is not at all acceptable to use them to construct code generators in a manner similar to that of this dissertation. The class of ambiguities defined by an arbitrary instruction set description is not guaranteed to be accepted by such a parser. It is mandatory that sufficient conditions be known to test the correctness of the resulting code generator before they are used in compilers.

The other algorithms presented here also have limited usefulness on non instruction set grammars. The loop detection algorithm will correctly locate all existing loops in the tables for grammars with no lambda rules, but the loop removal and blocking removal algorithms depend upon the fact that the language being parsed is a sequence of Polish prefix expressions, and that the symbols have fixed arity.

First, a few definitions will be given. Letting **G** be the instruction description grammar, $\mathbf{V}^*$ represents strings of 0 or more symbols in **V** and lower case Greek letters from the first of the alphabet, α β & γ, represent members of $\mathbf{V}^*$. If $\mathbf{v} ::= \alpha\ \beta$ is a grammar rule in **G**, where

either **v** ∈ **N** or **v** = λ, then [v → α · β] is an LR(0) **item**. **I** is the set of LR(0) items for a grammar **G**. Each state, **q**, in **Q** is associated with a non-empty subset of **I**, referred to as the **value of q**. The **core** of **q** is the set of items in **q** of the form [v → α · β] with α ≠ λ.

The first stage for the algorithm for creating a code generator appears as Algorithm 4.2. It is very similar to an LR(0) table constructor. The state set generated is identical to the LR(0) state set produced by an LR(0) or SLR(1) table constructor. The only difference is the way the algorithm computes the ACTION function when there is a **shift/reduce** or **reduce/reduce** conflict. Conflicts will always exist when the input grammar is ambiguous because the set of grammars that are SLR(1) does not include any ambiguous grammars. **Shift** entries for which there are no conflicts are the same as in an LR(0) or SLR(1) table. **Shift/reduce** conflicts are settled in favor of the **shift**. This choice is an attempt to allow the coder to match longer instruction patterns. If **reduce** were picked, the coder would issue only the simplest instructions. **Reduce/reduce** conflicts are resolved in favor of the longest instruction pattern. Some instruction set descriptions have multiple rules with the same pattern. That is, the rules have the same instruction pattern, but different semantic restrictions or result symbols. All such rules are entered into the reduce set *R* of a single entry in the table. The particular instruction issued is determined at code generation time. The preprocessor arranges multiple rule entries in a 'best instruction first' ordering, however, in an attempt to generate better code. Semantic processing and reduction choices are discussed in more detail in section 4.5.7.

In Algorithm 4.2 (and in the implementation), the resolution of shift-reduce conflicts is done in LR(0) fashion. In other words, if a state has a **shift/reduce** conflict, no **reduce** moves are signalled, even for lookahead symbols for which no **shifts** are possible. Similarly, **reduce/reduce** conflicts are resolved without consideration of lookahead symbols and **reduce** actions are indicated for all possible input symbols. Thus the algorithm does not compute follow sets. (For **x** ∈ **V**, the follow set FOLLOW(x) is the set of input symbols that can follow a given grammar symbol in a sentential form.)

In a state with no possible shifts, carrying out the reduction without inspecting the input is always safe. If the lookahead were inspected, the possible actions would be either **reduce** or **error**. If the action should be **error**, i.e. the next input cannot validly follow, then the error will eventually be indicated, before the input is advanced (see [Aho72a] for discussion of this point). Since errors occur only from compiler bugs, such deferral of error detection does not degrade the algorithm.

It turns out that for many instruction grammars, the LR(0)-like construction suffices also for states having LR(0) **shift/reduce** conflicts because an SLR(1) conflict-resolved state having both **shift** moves and **reduce** moves cannot arise. However, for some grammars, **reduce** moves

Algorithm 4.2 — The Initial LR(0) Code Generator Constructor

Input: A TMDL instruction set description for machine **M**.

Output: The initial ACTION and NEXT functions (represented in matrices) for a code generator for machine **M**.

Method: Construct a set of LR(0) states for the context free grammar underlying the given instructions. As it is being done, fill in the NEXT and ACTION values to produce the code generator, resolving conflicts.

```
procedure generatestates;
begin
    q_0 ← close({i ∈ I | i = [λ → • α]});
    k ← 0;  n ← 0; /* highest state number */
    while k ⩽ n do begin
        if ∃ i ∈ q_k with i = [x → α • β], β ≠ λ, then
            ∀ v ∈ V do
                if ∃ i ∈ q_k with i = [x → α • v β] then
                    ACTION[q_k,v] ← shift;
                    q' ← close({[x → α v • β] | [x → α • v β] ∈ q_k});
                    if ∃ q_j = q' then NEXT[q_k,v] = q_j
                    else n ← n + 1; q_n ← q'; NEXT[q_k,v] ← q_n
                else ACTION[q_k,v] ← error
        else R ← {[x → α •] ∈ q_k | ∄ i ∈ q_k, i = [x' → α' •], length(α') > length(α)};
            ∀ v ∈ V do ACTION[q_k,v] ← reduce R;
        k ← k + 1;
    end;
    ACTION[q_0,$] ← accept;
end generatestates;

function close(q);
begin
    repeat
        q ← q + {[x → • α] | [y → β • x γ] ∈ q, x ≠ λ}
    until q does not change;
    return(q);
end close;
```

for non-**shift** lookaheads are filled in at a later stage by the blocking-uniformity algorithm (Algorithm 4.7). Algorithm 4.3 is an SLR(1)-like constructor in which the **reduce** moves are filled in by the initial constructor. This point is discussed further in section 4.5.6 (which also contains a more precise definition of FOLLOW).

Using Algorithm 4.2, we will now build a code generator for the following simplified instruction set:

1* *r*.1 ::= (*k*.1)	"**load**	*r*.1,=*k*.1"
2* *r*.1 ::= (↑ *r*.2)	"**eval**	*r*.1,*r*.2"
3* *r*.1 ::= (+ *r*.1 *r*.2)	"**add**	*r*.1,*r*.2"
4* *r*.1 ::= (+ *r*.2 *r*.1)	"**add**	*r*.1,*r*.2"
5* λ ::= (:= *r*.1 *r*.2)	"**store**	*r*.2,*r*.1"
6* *r*.1 ::= (↑ + *k*.2 *r*.2)	"**load**	*r*.1,*k*.2,*r*.2"

State **0** initially contains the item [λ → • := r r] and does not change during closure. The only valid look ahead symbols are := and *$*. State **0** is thus:

0* [λ → • := r r]

$: **accept**
:=: **shift 1**

with any ACTIONs not listed being **error**. As with all code generators created by Algorithm 4.2 or Algorithm 4.3, the only accepting configuration is when the parser is in the initial state, **0**, reading the end of file symbol, *$*. Equivalently, all input must have been read and the stack must be empty. This insures that the entire IR string has been fully processed.

State **1** initially contains the single item [λ → := • r r]. It was introduced when the dot was advanced over the assignment operator in the item in State **0**. Since the dot precedes the variable **r**, the closure of State **1** adds all items of the form [r → • α]. No further change occurs in the closure and the resulting state is:

Algorithm 4.3 — The Initial SLR(1) Code Generator Constructor

Input: A TMDL instruction set description for machine **M**.

Output: The initial ACTION and NEXT functions (represented in matrices) for a code generator for machine **M**.

Method: Construct a set of SLR(1) states for the context free grammar underlying the given instructions. As it is being done, fill in the NEXT and ACTION values to produce the code generator, resolving conflicts.

```
procedure generatestates;
begin
   q_0 ← close({i ∈ I | i = [λ → · α]});
   k ← 0;  n ← 0; /* highest state number */
   while k ≤ n do begin
      ∀ v ∈ V do
         if ∃ i ∈ q_k with i = [x → α · v β] then
            ACTION[q_k,v] ← shift;
            q' ← close({[x → α v · β] | [x → α · v β] ∈ q_k});
            if ∃ q_j = q' then NEXT[q_k,v] = q_j
            else n ← n + 1; q_n ← q'; NEXT[q_k,v] ← q_n;
         else if ∃ i ∈ q_k with i = [x → α ·] then
            R ← {[x → α ·] ∈ q_k | v ∈ FOLLOW(x), ∄ i ∈ q_k, i = [x' → α' ·] such that
                 both length(α') > length(α) and v ∈ FOLLOW(x')};
            ACTION[q_k,v] ← reduce R;
         else ACTION[q_k,v] ← error;
      k ← k + 1;
   end;
   ACTION[q_0,$] ← accept;
end generatestates;

function close(q);
begin
   repeat
      q ← q + {[x → · α] | [y → β · x γ] ∈ q, x ≠ λ}
   until q does not change;
   return(q);
end close;
```

1* [λ → := • r r]
[r → • k]
[r → • ↑ r]
[r → • + r r]
[r → • ↑ + k r]

r: **shift 2**
k: **shift 3**
+*:* **shift 4**
↑*:* **shift 5**

A full description of the initial state computation is given in Fig. 4.3. We omit the set notation for **reduce** actions and use a sequence of rule numbers instead. Notice that state **10** has two **reduce** actions with the same syntax but different semantics. The table describing the initial ACTION and NEXT functions, which are also the final functions for this simple example, is given in Fig. 4.4.

4.5.2. Correct Code is Always Generated

This section will prove that, under a few basic assumptions, the code generated by the code generation algorithm is correct. The major problem is showing that instructions are issued in the correct order. An instruction to add the contents of two registers only implements the addition specified by a source language program if the current contents of those two registers corresponds to the operands of that addition. It will be assumed that the IR input to the code generator is correct, i.e. that it represents some valid source program, and therefore must be a list of prefix expressions. The implementor either can insure that the IR generator is correct or can insert a 'well-formedness' test in the IR generator, based on the mapping given in Chapter 3. It will also be assumed that the TMDL machine description accurately describes the target computer. The preprocessor has no way of detecting errors in a machine description, save certain errors of omission. Finally, it will be assumed that for 'correct' IR input, the coder always reaches an **accept** state. This fact is established in section 4.5.6. We begin by proving a few basic lemmas.

Lemma 4.1: For any given configuration, the concatenation of the vocabulary symbols on the stack, the bottom symbol being on the left, with the unread input forms a sequence of root-level IR expressions.

Proof: By induction on the number of moves made by the parser. We will show that the

0*

[λ → . := r r]

$: accept
:=: shift 1

1*

[λ → := . r r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 2
k: shift 3
+: shift 4
↑: shift 5

2*

[λ → := r . r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 6
k: shift 3
+: shift 4
↑: shift 5

3*

[r → k .]

all: reduce 1

4*

[r → + . r r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 7
k: shift 3
+: shift 4
↑: shift 5

5*

[r → ↑ . r]
[r → ↑ . + k r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 8
k: shift 3
+: shift 9
↑: shift 5

6*

[λ → := r r .]

all: reduce 5

7*

[r → + r . r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 10
k: shift 3
+: shift 4
↑: shift 5

8*

[r → ↑ r .]

all: reduce 2

9*

[r → + . r r]
[r → ↑ + . k r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 7
k: shift 11
+: shift 4
↑: shift 5

10*

[r → + r r .]

all: reduce 3 4 ******

11*

[r → k .]
[r → ↑ + k . r]
[r → . k]
[r → . ↑ r]
[r → . + r r]
[r → . ↑ + k r]

r: shift 12
k: shift 3
+: shift 4
↑: shift 5

12*

[r → ↑ + k r .]

all: reduce 6

Fig. 4.3. Initial State Computation for Example Instruction Set.

	$	r	k	+	↑	:=
0	A:					S: 1
1		S: 2	S: 3	S: 4	S: 5	
2		S: 6	S: 3	S: 4	S: 5	
3	R: 1	R: 1	R: 1	R: 1	R: 1	R: 1
4		S: 7	S: 3	S: 4	S: 5	
5		S: 8	S: 3	S: 9	S: 5	
6	R: 5	R: 5	R: 5	R: 5	R: 5	R: 5
7		S:10	S: 3	S: 4	S: 5	
8	R: 2	R: 2	R: 2	R: 2	R: 2	R: 2
9		S: 7	S:11	S:4	S: 5	
10†	R: *R*	R: *R*	R: *R*	R: *R*	R: *R*	R: *R*
11		S:12	S: 3	S: 4	S: 5	
12	R: 6	R: 6	R: 6	R: 6	R: 6	R: 6

†$R = \{3,4\}$.

Fig. 4.4. Code Generator Table for Example.

weights assigned to IR expressions in Chapter 3 always obey the definition of prefix expressions.

Inductive Hypothesis: The concatenation of the stack and remaining input symbols forms a sequence of root-level expressions.

Basis: In the initial configuration the stack contains the single entry $\mathbf{q}_0$ which is not a vocabulary symbol. The input is a sequence of root-level expressions, and the *I.H.* is trivially upheld.

Inductive Step: Assume that the *I.H.* holds for a given configuration. We will show that it then holds after any next move. There are 4 cases to consider, corresponding to the 4 distinct actions:

shift: The leftmost symbol from the remaining input is read and pushed onto the stack. The string resulting from the concatenation of the stack and input symbols does not change.

reduce: Let $e_1 ... e_{j-1} e_j ... e_k e_{k+1} ... e_m$ be the symbols in the configuration of the parser immediately prior to this **reduce** action, with $e_j ... e_k$ being the instruction pattern and e_r the location of the result for this reduction. Since the instruction pattern is a prefix expression, $w_j + ... + w_k = 1$. According to the table construction algorithm, the action **reduce** is

only entered into the tables when the top most symbols on the stack correspond to the instruction pattern for rule i. There are two cases to be considered.

Case 1: The location of the result of the instruction issued is λ. The instruction pattern for such instructions must be a root-level expression. This action removes the entire expression from the stack, and leaves nothing in its place. The resulting concatenation, $e_1 \ldots e_{j-1} e_{k+1} \ldots e_m$, is therefore still a sequence of root-level IR expressions.

Case 2: The location of the result of the instruction issued is non-λ. The instruction pattern is an internal prefix expression. It therefore must be an argument to another operator in a root-level expression, $e_i \ldots e_{j-1} e_j \ldots e_k e_{k+1} \ldots e_l$. By 1) and 2) of the definition of prefix expressions in Chapter 3, we have $w_i + \ldots + w_l = 1$ and $w_i + \ldots + w_{j-1} < 1$. The **reduce** action will remove $e_j \ldots e_k$ from the stack, and replace it with e_r, (which must be an operand symbol), so $w_r = 1$. The resulting expressions is $e_i \ldots e_{j-1} e_r e_{k+1} \ldots e_l$. Since $w_r = 1 = w_j + \ldots + w_k$, we have: $w_i + \ldots + w_{j-1} + w_r + w_{k+1} + \ldots + w_l = 1$, and 1) is satisfied. Since $e_i \ldots e_k$ is a prefix expression, $\forall$ x, i $\leqslant$ x $<$ k, $w_i + \ldots + w_x < 1$, so $w_i + \ldots + w_{j-1} + w_r = w_i + \ldots + w_{j-1} + w_j + \ldots + w_x < 1$ and $w_i + \ldots + w_{j-1} + w_r + w_{k+1} + \ldots + w_n$ $= w_i + \ldots + w_{j-1} + w_j + \ldots + w_k + w_{k+1} + \ldots + w_n < 1$, n $<$ l, so 2) is upheld. 3) is upheld, since e_i is a root-level operator, and 4) holds since no symbol in $e_{i+1} \ldots e_k$ is a root-level operator, and $e_{i+1} \ldots e_{j-1} e_r e_{k+1} \ldots e_l$ is a prefix expression and therefore $e_1 \ldots e_{j-1} e_r e_{k+1} \ldots e_m$ is still a sequence of root-level IR expressions.

accept: The parser halts without altering the input or stack. The *I.H.* holds trivially.

error: Even though this cannot happen for valid input, this action would leave the input and stack unaltered. □

Lemma 4.2: The evaluation of an operation is done after or concurrent with the evaluation of all of its operands.

Proof: By induction on the number of operators in the subtree computed by the expression headed by that operator.

Inductive Hypothesis: For expressions with k or fewer operators, the evaluation of an operator is not done prior to the evaluation of any of its operands.

Basis: There are no operators in the expression, i.e. it is a single operand symbol. Until

it is used in an instruction that computes a larger expression, the value of the operand is not altered but simply moved from register to register. Then the larger expression is computed by a single instruction, and the *I.H.* trivially holds.

Inductive Step: Assume true for all expressions with k or fewer operators. For an expression with $k + 1$ operators, the *I.H.* must hold for each operand to the primary operator, as the operands will contain at most k operators. Once the instruction involving the primary operator has been issued, the entire sub-expression headed by it will be reduced to either a single symbol or to λ. (This follows from the fact that instruction patterns are deterministic prefix expressions with a rigid format.) That operator is therefore evaluated either simultaneously or after its operands, and the *I.H.* holds. (A more rigorous proof of this Lemma would use another inductive proof here on the number of operators in the last instruction issued.) □

Lemma 4.3: The stack is always void of any IR symbols when the coder is in the initial state, $\mathbf{q}_0$.

Proof: State $\mathbf{q}_0$ only contains items of the form $[\lambda \rightarrow \cdot\ \alpha]$. Therefore no entry in ACTION is **shift** $\mathbf{q}_0$, for each **shift** action advances the dot over a symbol in at least one item, $[r \rightarrow \alpha \cdot a\ \beta]$ to $[r \rightarrow \alpha\ a \cdot \beta]$, which will be included in the next state. Therefore the only way to enter state $\mathbf{q}_0$ is to pop all of the remaining symbols off the stack, revealing the initial entry, $\mathbf{q}_0$, on the bottom. □

Lemma 4.4: Each root-level IR expression is completely evaluated before any symbol in the next sequential expression is shifted onto the stack.

Proof: The only state in which items of the form $[\lambda \rightarrow \cdot\ \alpha]$ are included is the initial state. The only occurrence of root-level operators in instruction patterns are as the left-most symbol in a root-level expression, which are exactly those instructions where the result is λ. Therefore, the only state containing an item with the dot preceding a root-level operator is $\mathbf{q}_0$. By Lemma 4.3, the stack is always empty when the parser is in state $\mathbf{q}_0$, therefore all input must be fully processed prior to shifting over a root-level operator. But the IR input is a sequence of root-level prefix expressions, each containing exactly one root-level operator as its first symbol, so, equivalently, each sequential root-level IR expression is completely evaluated before any symbol in the next expression is read. □

Lemma 4.5: When an instruction, *i*, is issued by a **reduce** move, the top symbols on the stack are exactly the symbols in the instruction pattern for *i*.

Proof: The lemma follows from the LR(k) properties of the algorithm (see section 4.5.6). □

Theorem 4.1: The code produced by the code generator upon making an **accept** move correctly implements the original IR input.

Proof: By Lemma 4.5, an instruction is issued only when the top of the stack corresponds to the symbols in that instruction's pattern. Those symbols are then popped and the result symbol is used as the next input symbol. The instruction issued will compute that piece of the prefix expression when executed, assuming that the machine description is correct. The only **accept** move is in state $\mathbf{q}_0$ reading *$*, and by Lemma 4.3 the code generator's stack is always empty when it is in state $\mathbf{q}_0$. Therefore whenever the code generator **accepts** the input, it has parsed it entirely, and issued all of the necessary instructions to implement it. The only remaining problem is that the instructions might be issued in the wrong order. Lemma 4.2 states that code is issued to implement an operation only after, or at the same time as, code is issued to compute the value of its operands. Assuming that the semantic information properly keeps track of the contents of registers, the value of a root-level expression is therefore computed correctly. Since each root-level IR expression is implemented sequentially, the entire input is implemented in the correct order, and the code generator does indeed compute the IR input correctly. □

4.5.3. Looping

A shift-reduce parser is said to **loop** if it makes an unbounded number of moves without reading any further input. In general, there are two ways for looping to occur. If a shift-reduce parser ever repeats a configuration, it will be in a loop, endlessly repeating the sequence of moves that it just made, beginning and ending with that configuration. Several facts can be observed about such looping sequences in general. There is no way to 'back-up' the input. Once an input symbol has been read, by performing a **shift**, the input string will be shortened for all future configurations. No previous configuration can ever be entered again. Therefore no input symbol can be read during a loop, i.e. no **shifts** can be performed on new input symbols. One way for a shift-reduce parser to loop is by reducing on chain rules. A **chain rule** is a rule of the form '**x** ::= **y**' where **y** is a single symbol in **V**. Such a reduction leaves the size of

the stack and input unchanged, so an unbounded sequence of chain reductions is possible. Another way a shift-reduce parser can loop is to add symbols to the stack by reducing **lambda rules** (rules with an empty right-hand part), and then reduce those symbols back out, returning to a previous configuration. It is possible for chain rules to be interspersed among the moves in such a sequence. Code generation grammars, however, have no lambda rules and therefore cannot loop in this manner.

It is also possible for the parser to loop without returning to a previous configuration. This could only happen if the parser continually increases the size of its stack, without reading any input; it is not possible without reducing lambda rules.

Detecting the possibility of such looping by examining the tables of a shift-reduce parser is, in general, quite complex [Aho72a]. But code generation grammars contain no lambda rules, allowing the possibility of looping to be determined in a much simpler manner, by examining the ACTION and NEXT functions.

Briefly, the only way that a code generating parser can loop is by entering into a series of chain-rule reductions that bring it into a previous configuration. If a reduction of a rule with a longer right-hand part is performed, then the stack will have grown smaller. Since there are no lambda rules, the only way to increase the stack to its previous size is by performing a series of **shifts**. But that will advance the input, and there will be no way to return to a previous configuration. Thus, a reduction using a rule with a right hand side longer than one symbol will prevent the coder from returning to a previous configuration and therefore prevent it from looping. Similarly, the stack cannot grow without bounds. This can happen only in the presence of lambda rules. In fact, the size of the stack cannot exceed the length of the original input string, as the only way to increase the size of the stack is to perform a **shift**, which reads an input symbol.

It can be determined whether a code generator can loop by examining its ACTION and NEXT functions and finding all places that a chain rule reduction occurs. To see how this is done, consider the following generalized sequence of moves that form a loop:

$(\ldots \mathbf{q d q_1} \rightarrow w \ldots) \vdash$	**reduce** $(\mathbf{d_1} ::= \mathbf{d})$
$(\ldots \mathbf{q d_1 q_2} \rightarrow w \ldots) \vdash^{k-2}$	
$(\ldots \mathbf{q d_{k-1} q_k} \rightarrow w \ldots) \vdash$	**reduce** $(\mathbf{d} ::= \mathbf{d_{k-1}})$
$(\ldots \mathbf{q d q_1} \rightarrow w \ldots)$	{ Same as first configuration above }

The values of ACTION and NEXT causing these moves are summarized in the following table:

	d_{k-1}	...	d_1	d	w
q	$S{:}q_k$	...	$S{:}q_2$	$S{:}q_1$	
q_1					$R{:}\{d_1 ::= d\}$
q_2					$R{:}\{d_2 ::= d_1\}$
...					
q_k					$R{:}\{d ::= d_{k-1}\}$

We will now discuss how to detect potential looping patterns in the parser's tables. The method takes each state, **q**, and checks for potential chain rule loops from that state by defining a relation characterizing the parser move:

$$(\ldots \mathbf{aqd_iq_k} \rightarrow w\ldots) \vdash (\ldots \mathbf{aqd_jq_l} \rightarrow w\ldots)$$

and then taking the transitive closure of that relation to find the loops. We define the relation $\mathbf{CHAINLOOP}_q$ on $N \times N$ to be:

$\mathbf{d_i}$ $\mathrm{CHAINLOOP}_q$ $\mathbf{d_j}$ $\Leftrightarrow$
$\exists\, \mathbf{q_k} \in \mathbf{Q}$ such that $\mathrm{ACTION}(\mathbf{q},\mathbf{d_i}) = \mathbf{shift}$, $\mathrm{NEXT}(\mathbf{q},\mathbf{d_i}) = \mathbf{q_k}$,
$\mathrm{ACTION}(\mathbf{q_k},w) = \mathbf{reduce}\{\mathbf{d_j} ::= \mathbf{d_i}\}$,
$\mathrm{ACTION}(\mathbf{q},\mathbf{d_j}) = \mathbf{shift}$, $\mathrm{NEXT}(\mathbf{q},\mathbf{d_j}) = \mathbf{q_l}$.

where w is an input symbol. The parser will be able to loop if and only if there is a variable $\mathbf{d_i}$ with $\mathbf{d_i}$ $\mathrm{CHAINLOOP}_q^+ \mathbf{d_i}$.

It is possible to check the code generator tables for the possibility of looping. In short, one computes for all states **q**, $\mathrm{CHAINLOOP}_q$. If there is a variable, **d**, with **d** $\mathrm{CHAINLOOP}_q^+$**d**, then the coder may loop when in a configuration of the form $(\ldots \mathbf{qdq_i} \rightarrow w\ldots)$. The algorithm for computing CHAINLOOP appears as Algorithm 4.4.

4.5.4. Complexity of Loop Detection by Preprocessor

The complexity of loop detection by the preprocessor is examined next. If N represents the number of variables in the grammar, then each transitive closure takes time $O(N^3)$. If Q is the number of states and V the number of symbols in the total vocabulary, then the complexity of this test by computing CHAINLOOP is in the worst case QN^3, which is greater than N^4. (Typically, Q is on the order of several hundred and N is 5 to 10.) If the grammar contains no

Algorithm 4.4 — Loop Detection

Input: NEXT and ACTION functions for a shift-reduce parser.

Output: A list of looping configurations.

Method: Compute CHAINLOOP_q for each $\mathbf{q} \in \mathbf{Q}$ and check for the existence of a $\mathbf{d} \in \mathbf{V}$ with $\mathbf{d}\ \text{CHAINLOOP}_q^+ \mathbf{d}$.

```
procedure loopdetection;
begin
    ∀ q ∈ Q do begin
        CHAINLOOP ← ∅;
        ∀ d ∈ N  do begin
            if ACTION[q,d] = shift and NEXT[q,d] = q' then
                if ACTION[q',*] = reduce R then
                    ∀ chainrules ν ::= d ∈ R do begin
                        CHAINLOOP ← CHAINLOOP + {(d,ν)};
                    end;
        end;
        ∀ d ∈ N do begin
            if (d,d) ∈ CHAINLOOP+ then
                q' ← NEXT[q,d];
                output("Looping configuration: (...qdq' → w...)");
        end;
    end;
end loopdetection;
```

chain rule loops, then the parser cannot loop. Even if the grammar does contain chain loops, the parser may not loop, since conflict resolution may have eliminated the potentially looping chain reductions. In general, CHAINLOOP will be extremely sparse so the closure is easily computed by graph traversal. Since N is normally quite small, a straightforward bit-wise implementation is also feasible.

4.5.5. Loop Elimination by Preprocessor

It is possible for the preprocessor to automatically eliminate potential looping in a code generating parser by introducing new states into the parser's tables which suppress unnecessary chain reductions. In fact, by a slight addition to Algorithm 4.4 all loops may be eliminated in a single pass as they are discovered.

To show how loop elimination is done, assume that we have discovered a loop including configuration (**...qdq**$'$ $\rightarrow$ *w*...) with **d** CHAINLOOP$^{+}_{\mathbf{q}}$**d**. By definition, root-level expressions (hence, rules with the left hand part λ) must begin with an operator. Consequently there is no **d** $\in$ N such that **d** CHAINLOOP$^{+}_{q_0}$ **d**. Thus $\mathbf{q} \neq \mathbf{q}_0$. By definition of CHAINLOOP, ACTION(**q**,**d**) is **shift**, NEXT(**q**,**d**) = **q**$'$ and ACTION(**q**$'$,*w*) is **reduce** *R*, where $\mathbf{u}_1$ ::= **d** is in *R* (for some $\mathbf{u}_1$ $\in$ N). Therefore [$u_1 \rightarrow \cdot$ d] must be in **q**. Likewise, all of the items of the form [u $\rightarrow \cdot$ v], **v** $\in$ N, involved in the looping must also be in **q**.

To break the loop, we consider why the chain item [$u_1 \rightarrow \cdot$ d] is included in state **q** by the constructor. Since $\mathbf{q} \neq \mathbf{q}_0$, the chain item is added by the closure operation. Hence there is some item [$x \rightarrow \alpha \cdot v\, \beta$], where $\alpha \neq \lambda$ and **v** $\in$ N, in the core of **q** such that for some γ, $v \Rightarrow^* \mathbf{u}_1\gamma$. But by a simple inductive argument, $\mathbf{u}_1\gamma$ must be a prefix expression, since the right hand part of each rule is a prefix expression. Hence $\gamma = \lambda$ and $\mathbf{v} \Rightarrow^* \mathbf{u}_1$ by a sequence of applications of chain rules v ::= $\mathbf{u}_k$, $\mathbf{u}_k$ ::= $\mathbf{u}_{k-1}$, ..., $\mathbf{u}_2$ ::= $\mathbf{u}_1$, each having a corresponding chain item in **q**. Although $\mathbf{u}_1$ ::= **d** is a rule included in the loop, the other chain rules in the derivation $v \Rightarrow^* \mathbf{u}_1$ may or may not be. The loop is completed by a set of items of the form: {[$d \rightarrow \cdot\, w_1$] [$w_1 \rightarrow \cdot\, w_2$] ... [$w_i \rightarrow \cdot\, u_j$]} for some $i \leq j \leq k$ or $\mathbf{u}_j = v$ (see Fig. 4.5). In any case the looping is prevented by modifying the states in the loop so that **d** is necessarily reduced to **v** (i.e. the derivation $v \Rightarrow^* \mathbf{u}_1 \Rightarrow^* \mathbf{d}$ is reconstructed) and then v is shifted. This is accomplished by removing the item [$w_i \rightarrow \cdot\, u_j$] from state **q**. The effect of shifting **v** is to enter a state containing item [$x \rightarrow \alpha\, v \cdot \beta$]. The next move of the parser is either a reduction by a non-chain rule or a shift of an input symbol — either kind of move breaks the loop.

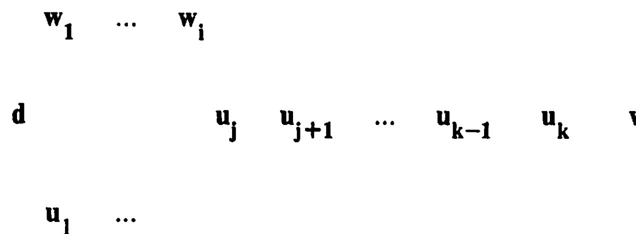


Fig. 4.5. Graph Constructed to Eliminate Loop: **d**$\rightarrow$...$\rightarrow$**d**.

Modifying the coder's tables to prevent looping is performed in the following way: A directed graph is constructed from the chain rules involving only nonterminals. As shown in Figure 4.5, the nodes of the graph correspond to nonterminals in the grammar, and arcs exist in the graph from **v** to **u** for each chain rule of the form **u** ::= v. Each arc is given a positive,

non-zero length that is used as the 'cost' of the code that will be generated for such a reduction. If there are several semantically restricted forms of the rule, one could use the average or expected cost of the variants. Alternately, a constant cost of 1 works quite well. For each state **q″** such that ACTION($\mathbf{q},\mathbf{u}_j$) = **shift**, NEXT($\mathbf{q},\mathbf{u}_j$) = **q″** and ACTION(**q″**,w) = **reduce** {$\mathbf{w}_i ::= \mathbf{u}_j$, $\mathbf{u}_{j+1} ::= \mathbf{u}_j$} a new state **q‴** is constructed that is the same as **q″** except that it contains only those chain rules which transform the nonterminal $\mathbf{u}_j$ into a nonterminal that is closer to a nonterminal **v**, with an item of the form $[x \rightarrow \alpha \cdot v \beta] \in \mathbf{q}$, i.e. $\mathbf{w}_i ::= \mathbf{u}_j$ is removed from the **reduce** set for **q″** to make the **reduce** set for **q‴**. NEXT($\mathbf{q},\mathbf{u}_j$) is then set to the newly created state. Thus, any chain rule reduction done immediately after shifting from state **q** will reduce the distance between the symbol on the top of the stack and the closest symbol that will allow an item in the core of **q** to be extended. Since the minimum distance is zero, this process must terminate in at worst $|N|-1$ chain rule reductions, and the loop has been eliminated. Since no moves other than multiple-possibility reductions have been eliminated, the language accepted by the automaton is unchanged. This algorithm is given as Algorithm 4.5.

4.5.6. Blocking and Uniformity

Once the possibility of looping has been eliminated, the only way that the code generating algorithm presented in this dissertation can fail to generate code for the input (i.e. can fail to reach the **accept** action) is by blocking. A coder is said to **block** when it performs an **error** action. In this section a sufficient condition for instruction grammars is presented, together with an algorithm for testing whether an instruction set satisfies the condition, and a corresponding sufficient condition for the IR that is straightforward to check. It is argued that if both the IR and the instruction set satisfy the sufficient conditions, then the coder cannot block. We then indicate how the conditions might be weakened if necessary for a particular instruction set. In order to simplify the presentation, we assume in this section that no **reduce** set R contains two or more rules with different left hand parts. This restriction is removed in section 4.5.7.

An instruction set grammar is said to be **uniform** if it satisfies the following condition:

> Any left (similarly right) operand of a binary operator **b** is a valid left (respectively right) operand of **b** in *any* prefix expression of $\mathbf{V}^*$ containing **b**. Any operand of a unary operator **u** is a valid operand of **u** in *any* prefix expression of $\mathbf{V}^*$ containing **u**.

An instruction set is uniform if its grammar is uniform. The essential idea of uniformity is that operands to an operator are valid independent of context.

Algorithm 4.5 — Loop Elimination

Input: NEXT and ACTION matrices, the set of LR(0) items, and a list of all chain rules and their costs for a shift-reduce code generator.

Output: A new set of NEXT and ACTION matrices for a shift-reduce code generator that cannot loop.

Method: Locate loops by computing CHAINLOOP and remove some reductions so that looping is no longer possible.

```
/* Compute DISTANCE[u,v] = cost of shortest code sequence from u to v */

∀ u,v ∈ N do DISTANCE[u,v] ← ∞;
∀ chain rule u ::= v ∈ G do
    DISTANCE[u,v] ← cost[v ::= u];
∀ w ∈ N do
    ∀ u ∈ N do
        ∀ v ∈ N do
            if DISTANCE[u,v] > DISTANCE[u,w] + DISTANCE[w,v] then
                DISTANCE[u,v] ← DISTANCE[u,w] + DISTANCE[w,v];
if ∀ v ∈ N DISTANCE[v,v] = ∞ then return; /* there are no loops */

∀ q ∈ Q do begin
    Compute CHAINLOOP_q using Algorithm 4.4;
    if ∃ v ∈ N with (v,v) ∈ CHAINLOOP_q^+ then begin

        /* Compute MINDISTANCE_q[v] ∀ v */

        ∀ v ∈ N do MINDISTANCE[v] ← ∞;
        ∀ x ∈ N with [y → α · x β] ∈ q, α ≠ λ, do begin
            MINDISTANCE[x] ← 0;
            ∀ v ∈ N if DISTANCE[v,x] < MINDISTANCE[v] then
                MINDISTANCE[v] ← DISTANCE[v,x];
        end;

        ∀ v ∈ N do begin
            q' ← NEXT[q,v]; R ← ∅;
            n ← (numstates ← numstates + 1); q_n ← q';
            ∀ rule 'v' ::= v' ∈ ACTION[q',v], do begin
                if MINDISTANCE[v'] < MINDISTANCE[v] then
                    Add all rules 'v' ::= v' to R;
            end;
            ACTION[q_n,v] ← reduce R; NEXT[q,v] ← q_n;
        end;
    end;
end;
```

We give some examples to clarify this notion. Consider the six rule example of section 4.5.1. The possible operands of $\uparrow$, $+$, and $:=$ are all registers (or prefix expressions that result in registers) except for the left operand of $+$ and the operand of $\uparrow$ in rule 6. However the left operand of $+$, a constant, can be transformed into a register by rule 1. Also, the operand of $\uparrow$ in rule 6 is a special case of the operand of $\uparrow$ in rule 2 (i.e. this is an ambiguity which will be resolved by using rule 6 whenever possible). Thus, whenever the operators $\uparrow$, $+$, $:=$ occur, their operands are any expressions corresponding to registers (i.e. generated by **r** in the grammar).

An example of an instruction set which is not uniform is:

1* $r ::= (+ \uparrow k\ k)$
2* $r ::= (+\ k \uparrow k)$
3* $\lambda ::= (:= r\ r)$

In this example k is a valid first operand of $+$ only if the second operand is '$\uparrow k$' and vice-versa.

We next give an algorithm to determine whether an instruction grammar is uniform. Rather than testing the grammar, the algorithm is a blocking test of the code generator. We present the algorithm and also argue that the uniformity test fails (i.e. blocking can occur) if and only if the grammar is not uniform.

To describe the test easily, we need some terminology to refer to parts of a prefix expression. Let $e_1 ... e_i ... e_k$ be a prefix expression, where $k \geq 2$ and $1 \leq i \leq k$. As is well known, the prefix expression corresponds to a (unique) binary tree with nodes labelled by the e_j's, where e_1 labels the root and non-leaf nodes are labelled by operators. The subtrees rooted by children of an operator node represent its operands (see, for example, [Meyers74]).

We define the following function and predicate for each such prefix expression $e_1 ... e_k$: For $1 < i \leq k$, **parent**$(e_i, e_1 ... e_i ... e_k) = e_j$ where e_j is the parent of e_i in the tree corresponding to the prefix expression. (Clearly $j < i$). For $1 < i \leq k$, the value of **leftchild**$(e_i, e_1 ... e_i ... e_k)$ is TRUE if e_i is the left child of its parent in the tree corresponding to the prefix expression and FALSE otherwise (namely, if e_i is the right child). The single child of a unary node is assumed to be a left child. It is easily shown that for any prefix expression $e_1 ... e_k$ and for $1 < i \leq k$,

1) *leftchild*$(e_i, e_1 ... e_i ... e_k)$ = TRUE iff e_{i-1} is an operator, and
2) *leftchild*$(e_i, e_1 ... e_i ... e_k)$ = FALSE iff e_{i-1} is an operand symbol.

Using *parent* and *leftchild*, we define the relations used by the uniformity test. The

definitions are presented with respect to a particular instruction grammar **G**. Let $\mathbf{B} \subseteq \mathbf{V}$ be the binary operators; let $\mathbf{U} \subseteq \mathbf{V}$ be the unary operators. Define the relation **LEFT** on $(\mathbf{B} \cup \mathbf{U}) \times \mathbf{V}$ as:

b LEFT **v** iff

$\exists$ a rule $\mathbf{r} ::= \alpha\mathbf{bv}\beta$ for some $\mathbf{r} \in \mathbf{N} \cup \{\lambda\}$.

It is a simple consequence of the properties of prefix expressions, that if **b** is an operator, *parent*$(\mathbf{v},\alpha\mathbf{bv}\beta) = \mathbf{b}$ and *leftchild*$(\mathbf{v},\alpha\mathbf{bv}\beta) =$ TRUE. Define the relation **RIGHT** on $\mathbf{B} \times \mathbf{V}$ as:

b RIGHT **v** iff

$\exists$ a rule $\mathbf{r} ::= \alpha\mathbf{b}\beta\mathbf{v}\gamma$

for some $\mathbf{r} \in \mathbf{N} \cup \{\lambda\}$ and some prefix expression $\beta \neq \lambda$.

It is easily shown that if **b** is a binary operator and β is a prefix expression, *parent*$(\mathbf{v},\alpha\mathbf{b}\beta\mathbf{v}\gamma) = \mathbf{b}$ and *leftchild*$(\mathbf{v},\alpha\mathbf{b}\beta\mathbf{v}\gamma) =$ FALSE.

In addition, we need the usual grammatical leftmost and rightmost descendent and follow relations. Define the relation **FIRST** on $\mathbf{V} \times \mathbf{V}$ as:

u FIRST **v** iff

$\exists$ a derivation $\mathbf{u} \Longrightarrow^* \mathbf{v}\alpha$ for some $\alpha \in \mathbf{V}^*$, $\mathbf{u} \in \mathbf{V}$.

Thus **u** FIRST **v** if there is some derivation in **G** from **u** that yields **v** as the leftmost symbol. Similarly, define **LAST** on $\mathbf{V} \times \mathbf{V}$ as:

u LAST **v** iff

$\exists$ a derivation $\mathbf{u} \Longrightarrow^* \alpha\mathbf{v}$ for some $\alpha \in \mathbf{V}^*$, $\mathbf{u} \in \mathbf{V}$.

By the usual product of relations, **b** LEFT FIRST **v** iff **v** can appear as the first symbol in the left operand to **b** in a derivation in **G**, and **b** RIGHT FIRST **v** iff **v** can appear as the first symbol in the right operand to **b** in a derivation in **G**. For $\mathbf{u} \in \mathbf{V}$, let

$$\text{FOLLOW1}(\mathbf{u}) = \{\mathbf{v} \mid \text{for some } \mathbf{r} ::= \alpha\mathbf{xy}\beta,\ \mathbf{r} \in \mathbf{N} \cup \lambda,\ \mathbf{x} \text{ LAST } \mathbf{u} \text{ and } \mathbf{y} \text{ FIRST } \mathbf{v}\}.$$

Let

$$\lambda\text{-LAST} = \{\mathbf{u} \mid \exists \text{ rule } \lambda ::= \alpha\mathbf{x} \text{ for some } \alpha \in \mathbf{V}^* \text{ and } \mathbf{x} \text{ LAST } \mathbf{u}\}.$$

(It is easily shown that λ-LAST consists only of operand symbols.) Then for $\mathbf{u} \in \mathbf{V}$,

$$\text{FOLLOW}(\mathbf{u}) = \begin{cases} \text{FOLLOW1}(\mathbf{u}) \text{ if } \mathbf{u} \notin \lambda\text{-LAST} \\ \text{FOLLOW1}(\mathbf{u}) \cup \{\text{root-level operators}\} \text{ if } \mathbf{u} \in \lambda\text{-LAST}. \end{cases}$$

The definitions have the following consequences:

Lemma 4.6: Let **G** be an instruction set grammar. Let $\mathbf{u},\mathbf{v} \in \mathbf{V}$. Then $\mathbf{v} \in$ FOLLOW1(**u**) iff **v** can follow **u** in a prefix expression generated by the grammar. Also $\mathbf{v} \in$ FOLLOW(**u**) iff **v** can follow **u** in a sequence of prefix expressions generated by the grammar.

Proof: Left to the reader. □

Lemma 4.7: Let **G** be an instruction set grammar.

1) Let **u** be an operator. Then $\forall\, \mathbf{v} \in \mathbf{V}$, $\mathbf{v} \in$ FOLLOW(**u**) iff **u** LEFT FIRST **v**.

2) Let **u** be an operand symbol. Then $\forall\, \mathbf{v} \in \mathbf{V}$, $\mathbf{v} \in$ FOLLOW1(**u**) iff $\exists\, \mathbf{r} ::= \alpha\mathbf{b}\beta\mathbf{x}\mathbf{y}\gamma$, where $\mathbf{r} \in \mathbf{N} \cup \{\lambda\}$, such that **b** is a binary operator, $\beta\mathbf{x}$ is a prefix expression, **x** LAST **u**, and **y** FIRST **v**, in which case **b** RIGHT FIRST **v**.

Proof: Left to the reader. □

Using the definitions, we can characterize uniformity by the following lemma.

Lemma 4.8: The following conditions are equivalent:

1) **G** is a uniform grammar.

2) Let **Q** be the set of LR(0) items generated for **G**.†

2a) $\forall\, \mathbf{u},\mathbf{v} \in \mathbf{V}$, if **u** LEFT FIRST **v** then $\forall\, \mathbf{q} \in \mathbf{Q}$ containing $[\mathrm{r} \rightarrow \alpha \cdot \mathrm{x}\, \beta]$ where *parent*$(\mathbf{x},\alpha\mathbf{x}\beta) = \mathbf{u}$ and *leftchild*$(\mathbf{x},\alpha\mathbf{x}\beta)$ = TRUE there is some $[\mathrm{r}' \rightarrow \alpha' \cdot \mathrm{x}'\, \beta']$ in **q** such that *parent*$(\mathbf{x}',\alpha'\mathbf{x}'\beta') = \mathbf{u}$, *leftchild*$(\mathbf{x}',\alpha'\mathbf{x}'\beta')$ = TRUE, and **x′** FIRST **v**, and

2b) $\forall\, \mathbf{u},\mathbf{v}$ in **G**, if **u** RIGHT FIRST **v** then $\forall\, \mathbf{q} \in \mathbf{Q}$ containing $[\mathrm{r} \rightarrow \alpha \cdot \mathrm{x}\, \beta]$ where

†The reader should recall from the theory of LR-parsing that such a set of items can be generated for <u>any</u> context free grammar, even though a consequent parser generation method might yield a parser with conflicts (i.e. a nondeterministic parser).

$\alpha \neq \lambda$, *parent*$(\mathbf{x},\alpha\mathbf{x}\beta)$ = **u**, **u** $\in$ **B**, and *leftchild*$(\mathbf{x},\alpha\mathbf{x}\beta)$ = FALSE, either there is some $[\mathrm{r}' \rightarrow \alpha' \cdot \mathrm{x}' \beta']$ in **q**, $\alpha \neq \lambda$ such that *parent*$(\mathbf{x}',\alpha'\mathbf{x}'\beta')$ = **u**, *leftchild*$(\mathbf{x}',\alpha'\mathbf{x}'\beta')$ = FALSE, and **x**′ FIRST **v** or there is some $[\mathrm{r}'' \rightarrow \gamma \cdot]$ in **q**, $\gamma \neq \lambda$ such that **v** $\in$ FOLLOW(**r**″).

Proof: (Informal) 1 → 2. Suppose **G** is a uniform grammar. Let **Q** be the set of LR(0) items generated from **G** and let **w** $\in$ **V**. It follows from the correct prefix property of the LR(0) construction that every state **q** $\in$ **Q** containing an item of the form $[\mathrm{r} \rightarrow \alpha\ \mathrm{w} \cdot \beta]$ (i.e. an item with an occurrence of **w** before the dot) represents a state of the automaton after recognizing one of a regular set of initial segments of prefix expressions containing **w**, where the initial segment ends in **w**. Furthermore, every item $[\mathrm{r}' \rightarrow \delta \cdot \epsilon]$ in such a state **q** has the property that either δ is a suffix of α**w** or α**w** is a suffix of δ. Let **u** be an operator occurring in at least one instruction.

1 → 2a. Let **v** be any symbol such that **u** LEFT FIRST **v**. Then **v** is the first symbol of a left operand of **u** in some prefix expression. Since **G** is uniform, **v** must be a possible first symbol of a left operand in any prefix expression containing **u**. Let **q** $\in$ **Q** be any state containing an item of the form $[\mathrm{r} \rightarrow \alpha\ \mathrm{u} \cdot \mathrm{x}\ \beta]$. Since the operator **u** must be followed in any prefix expression by a left operand and since it must be possible for the left operand to start with **v**, under the standard 'canonical' parser construction, state **q** must signal a **shift** action or a **reduce** action when the next input symbol is **v**. Since the right hand part of every rule is a prefix expression, there is no item in **q** of the form $[\mathrm{r} \rightarrow \alpha\ \mathrm{u} \cdot]$ (i.e. no rule ends with an operator). Consequently **q** must contain an item $[\mathrm{r}' \rightarrow \alpha'\ \mathrm{u} \cdot \mathrm{x}'\ \beta']$ the closure of which signals a **shift** on **v**. Thus, **x**′ FIRST **v**. Clearly *parent*$(\mathbf{x}',\alpha'\mathbf{u}\mathbf{x}'\beta')$ = **u** and *leftchild*$(\mathbf{x}',\alpha'\mathbf{u}\mathbf{x}'\beta')$ = TRUE.

1 → 2b. Let **u** be a binary operator and let **v** be any symbol such that **u** RIGHT FIRST **v**. Then **v** is the first symbol of a right operand of **u** in some prefix expression. Since **G** is uniform, **v** must be a possible first symbol of a second operand in any prefix expression containing **u**. Let **q** $\in$ **Q** be any state containing an item of the form $[\mathrm{r} \rightarrow \alpha \cdot \mathrm{x}\ \beta]$ where $\alpha \neq \lambda$, *parent*$(\mathbf{x},\alpha\mathbf{x}\beta)$ = **u**, **u** $\in$ **B**, and *leftchild*$(\mathbf{x},\alpha\mathbf{x}\beta)$ = FALSE. By reasoning analogous to the previous section of the proof, when in state **q**, the automaton has just recognized a sequence of input symbols ending in **u** followed by its first argument (a prefix expression). Since the automaton must next accept a prefix expression starting with **v**, there must be the possibility of a **shift** action or a **reduce** action from state **q** when the next symbol is **v**. It follows from the construction

of **Q** that **q** must contain an item $[r' \rightarrow \alpha' \cdot \gamma]$, $\alpha' \neq \lambda$ where either $\gamma = \lambda$ and $v \in$ FOLLOW(r′) or $\gamma = x'\beta'$ and x′ FIRST v. It remains to show that if the item has the form $[r' \rightarrow \alpha' \cdot x' \beta']$, then *parent*$(x', \alpha' x' \beta')$ = **u** and *leftchild*$(x', \alpha' x' \beta')$ = FALSE. Since *leftchild*$(x, \alpha x \beta)$ = FALSE, α must end with an operand symbol. Since α and α' end with the same symbol, *leftchild*$(x', \alpha' x' \beta')$ = FALSE. Since $\alpha x \beta$ and $\alpha' x' \beta'$ are prefix expressions and either α is a suffix of α' or α' is a suffix of α, a straightforward argument establishes that *parent*$(x', \alpha' x' \beta')$ = *parent*$(x, \alpha x \beta)$ = **u**.

2 → 1. Suppose **G** is a grammar which satisfies conditions 2a and 2b. Then for every **u**,**v** such that **u** is an operator and **v** is a valid first symbol of a left operand (similarly right operand) of **u**, whenever the automaton has recognized **u** (respectively, and its first operand), i.e. whenever the automaton is in a state with an item having a dot following **u** (resp, with an item that is valid for the prefix of a sentential form ending in $u\beta$ where β is a prefix expression), **v** is a possible next symbol (see [Aho72a]). Since **v** is either a complete operand or another operator one can argue by induction on the size of the operand tree that the grammar must be uniform. □

Suppose Algorithm 4.3 (the SLR(1)-like algorithm) is used to compute the table for the code generator. We know from the theory of LR parsing (see, for example, [Aho72a]) and the fact that the right parts of rules are prefix expressions that for any $q \in Q$, $u \in V$,

1) ACTION(**q**,**u**) = **shift** iff **q** contains some item $[r \rightarrow \alpha \cdot x \beta]$ where $x \Rightarrow^* u\gamma$, $\gamma \in V^*$ (i.e. **x** FIRST **u**).

2) If **q** contains some item $[r \rightarrow \alpha \cdot]$ then ACTION(**q**,**u**) = **reduce** *R* for every $u \in$ FOLLOW(r) such that **q** contains no item $[r' \rightarrow \alpha' \cdot x' \beta']$ where x′ FIRST **u**.

If Algorithm 4.2 (the LR(0)-like algorithm) is used, then 1) above is still true but the second condition becomes:

2′) If **q** contains no items $[r \rightarrow \alpha \cdot \beta]$, $\beta \neq \lambda$ and **q** contains some item $[r \rightarrow \alpha \cdot]$ then $\forall\, u \in V$, ACTION(**q**,**u**) = **reduce** *R*. If **q** contains some item $[r \rightarrow \alpha \cdot \beta]$, $\beta \neq \lambda$ then $\forall\, u \in V$ such that ACTION(**q**,**u**) ≠ **shift**, we have ACTION(**q**,**u**) = **error**.

Using these properties and lemma 4.8 we obtain:

Lemma 4.9: Let the code generator tables be computed by Algorithm 4.3. The following conditions are equivalent:

1) **G** is a uniform grammar.

2a) $\forall$ **u**,**v** $\in$ **V**, if **u** LEFT FIRST **v** then $\forall\, q \in Q$ containing $[r \rightarrow \alpha \cdot x \beta]$, where $r \in N \cup \{\lambda\}$, *parent*$(x, \alpha x \beta)$ = **u** and *leftchild*$(x, \alpha x \beta)$ = TRUE, it follows that

ACTION(q,v) = **shift**, and

2b) $\forall$ **u**,**v** in **G**, if **u** RIGHT FIRST **v** then $\forall$ **q** $\in$ **Q** containing [r $\rightarrow \alpha \cdot$ x β] where **r** $\in$ **N** $\cup$ {λ}, *parent*(**x**,α**x**β) = **u**, **u** $\in$ **B** and *leftchild*(**x**,α**x**β) = FALSE, ACTION(q,v) $\in$ {**shift**,**reduce** *R*}. □

The idea of the uniformity test is the following: For every operator, the LEFT FIRST and RIGHT FIRST relations are computed. The algorithm then inspects the states for items [r $\rightarrow \alpha \cdot$ x β] as specified in lemma 4.9, finds the appropriate parent, and checks that for all first symbols of left and right operands, either a **shift** or a **reduce** is signalled. Note that it suffices to check for **error** because only state $\mathbf{q_0}$ can have an **accept** action. Also note that the possibility of a **reduce** in 2a) is ruled out by the form of the grammar (i.e. **u** must be an operator and must be the last symbol in α, and no right hand side of a rule can end in an operator).

Algorithm 4.6 — Uniformity Test for Algorithm 4.3

Input: ACTION and NEXT matrices computed by Algorithm 4.3, the relations LEFT FIRST and RIGHT FIRST, and the set of LR(0) items for a shift-reduce code generator.

Output: A list of table error entries, if any nonuniformity exists.

Method: Check each state for **error** actions on follow relationships.

```
procedure uniformitycheck;
begin
    ∀ q ∈ Q do begin
        ∀ items of the form [r → α • v β] ∈ q, α ≠ λ do begin
            x ← parent(v,αvβ);
            if leftchild(v,αvβ) then
                ∀ u with x LEFT FIRST u do begin
                    if ACTION[q,u] = error then
                        output("Not Uniform");
                end;
            else (* x ∈ B and v begins the second operand *)
                ∀ u with x RIGHT FIRST u do begin
                    if ACTION[q,u] = error then
                        output("Not Uniform");
                end;
        end;
    end;
end uniformitycheck;
```

It follows easily from lemma 4.9 that:

Theorem 4.2: Algorithm 4.6 generates output if and only if the instruction grammar used by Algorithm 4.3 to generate the ACTION table is non-uniform. □

Suppose Algorithm 4.2 is used to compute the table tested by the uniformity test. Let us compare the tables generated by Algorithms 4.2 and 4.3. Every **shift** action for Algorithm 4.2 will be a **shift** action for Algorithm 4.3 and conversely. However some **reduce** actions for Algorithm 4.2 could be **error** actions for Algorithm 4.3. These moves would be in states having no **shift** actions and in fact represent error situations with delayed detection. They cannot invalidate the uniformity test. In addition, some **reduce** actions for Algorithm 4.3 could be **error** actions for Algorithm 4.2. This situation arises in states having a nonterminal before the dot in each core item and at least one **shift** action (note that this is a state in which a right operand is expected). In this case, the **reduce** action must be inserted in the table generated by Algorithm 4.2. This addition has been incorporated into Algorithm 4.7.

It follows from Lemma 4.9, Theorem 4.2, and the LR properties of the construction that:

Theorem 4.3: Algorithm 4.7 generates error output if and only if the instruction grammar used by Algorithm 4.2 to generate the ACTION table is non-uniform. □

It appears from the commercially available instruction sets we looked at, that the need for the table modifications made by Algorithm 4.7 rarely arises. We give an example in which additional **reduce** actions are needed in section 5.2, although the necessity for such additions resulted from the fact that only a partial instruction set description from an existing machine was supplied.

Suppose the grammar is found to be uniform. Furthermore, suppose that a particular IR input $\alpha \mathbf{u} \mathbf{v} \beta$ is a sequence of root-level expressions. It follows from lemmas 4.7 and 4.9 that for any operator **u**, for any $\mathbf{v} \in \mathbf{V}$, and for any state **q** such that for some $\mathbf{q}' \in \mathbf{Q}$, ACTION(**q′**,**u**) = **shift** and NEXT(**q′**,**u**) = **q**, ACTION(**q**,**v**) = **error** iff **u not** LEFT FIRST **v**. Recall that in either Algorithm 4.2 or Algorithm 4.3, for any state signalling no **shifts**, **reduce** actions are indicated for any symbol which can follow the left-hand symbol of a rule being reduced. It follows from this fact and lemmas 4.7 and 4.9 that for any operand symbol **u**, for any $\mathbf{v} \in \mathbf{V}$, and for any state **q** such that for some $\mathbf{q}' \in \mathbf{Q}$, ACTION(**q′**,**u**) = **shift** and NEXT(**q′**,**u**) = **q**, ACTION(**q**,**v**) = **error** iff for each item $[\mathbf{r} \rightarrow \alpha \cdot \gamma]$ in **q**, either $\gamma = \lambda$, $\mathbf{r} \neq \lambda$ and $\mathbf{v} \notin$ FOLLOW(**r**) or $\gamma = \mathbf{x}\beta$, *parent*(x,αxβ) = **b**, *leftchild*(x,αxβ) = FALSE and **b not** RIGHT FIRST **v**. In the former instance ($\gamma = \lambda$), there can be no rule $\mathbf{r}' ::= \alpha' \mathbf{b} \beta \mathbf{x} \mathbf{y} \gamma$ such that **y** FIRST **v**, *parent*(y,αbβxyγ) = **b** and **x** FIRST **r**. It follows that when the code generator enters state **q** and the parsing stack (omitting the state symbols) has the form $\alpha \mathbf{b} \sigma$ where σ is a (non-empty) prefix expression and **b** is a binary operator, **b not** RIGHT FIRST **v**.

Algorithm 4.7 — Blocking—Uniformity Test for Algorithm 4.2

Input: ACTION and NEXT matrices computed by Algorithm 4.3, the relations LEFT FIRST and RIGHT FIRST, and the set of LR(0) items for a shift-reduce code generator.

Output: A modified ACTION table and a list of table error entries, if any nonuniformity exists.

Method: Check each state for **error** actions on follow relationships.

```
procedure uniformitycheck;
begin
  ∀ q ∈ Q do begin
     ∀ items of the form [r → α · v β] ∈ q, α ≠ λ do begin
        x ← parent(v,α vβ);
        if leftchild(v,α vβ) then
           ∀ u with x LEFT FIRST u do begin
              if ACTION[q,u] = error then
                 output("Not Uniform");
           end;
        else (* x ∈ B and v begins the second operand *)
           ∀ u with x RIGHT FIRST u do begin
              if ACTION[q,u] = error then
                 if ∃ i ∈ q with i = [x' → α'] such that u ∈ FOLLOW(x') then
                    R ← {[x → α · ] ∈ q_k | v ∈ FOLLOW(x)
                       ∄ i ∈ q_k, i = [x' → α' ·] such that
                          both length(α') > length(α) and v ∈ FOLLOW(x')};
                    ACTION[q_k,v] ← reduce R;
                 else output("Not Uniform");
           end;
     end;
  end;
end uniformitycheck;
```

Since the code generator cannot loop, it can fail to generate code for an input IR sequence only if it blocks. By the remarks above, blocking is characterized by the LEFT FIRST and RIGHT FIRST relations on the instruction grammar. Consequently, any input IR sequence $E = e_1 e_2 ... e_k$ $k \geq 2$ is said to be **valid** for the code generator if:

1) E is a sequence of root-level expressions,
2) $\forall e_i \in \mathbf{B} \cup \mathbf{U}$, e_i LEFT FIRST e_{i+1}, and
3) $\forall e_i \in \mathbf{B}$, e_i RIGHT FIRST e_j where $e_{i+1} ... e_{j-1}$ is an (internal) prefix expression.

In terms of retargetability, one might prefer to state the situation another way. Let $LEFT_{IR}$, $RIGHT_{IR}$, and $FIRST_{IR}$ be the relations satisfied by the IR. In other words, substituting $LEFT_{IR}$ for LEFT, etc., the IR satisfies 1), 2), and 3) above. Let $LEFT_{CG}$, $RIGHT_{CG}$, and $FIRST_{CG}$ be the relations of a uniform instruction grammar described previously. Then an intermediate representation (IR) is valid for a code generator (CG) provided that:

$LEFT_{IR} \subseteq LEFT_{CG}$,
$RIGHT_{IR} \subseteq RIGHT_{CG}$, and
$FIRST_{IR} \subseteq FIRST_{CG}$.

Since the compiler probably cannot generate all IR expressions generated (i.e. computed) by a uniform instruction set, the inclusion might be proper in some instances.

It is the responsibility of the implementor (or of some other part of a compiler-writing system) to specify $LEFT_{IR}$, $RIGHT_{IR}$, and $FIRST_{IR}$. However, specifying these relations is considerably simpler that characterizing the set of strings that are possible IR expressions.

For most actual instruction sets we have looked at, one can use simpler computations of the relations LEFT FIRST and RIGHT FIRST than the ones suggested by the definitions. Define relation **LT** on $(\mathbf{B} \cup \mathbf{U}) \times \mathbf{V}$ by:

b LT **u** iff
$\exists$ a rule $\mathbf{r} ::= \mathbf{bu}\beta$ for some $\mathbf{r} \in \mathbf{N} \cup \{\lambda\}$.

Define **RT** on $\mathbf{B} \times \mathbf{V}$ by:

b RT **u** iff
$\exists$ a rule $\mathbf{r} ::= \mathbf{b}\beta\mathbf{u}\gamma$ for some $\mathbf{r} \in \mathbf{N} \cup \{\lambda\}$
and some prefix expression $\beta \neq \lambda$.

The difference between LT and LEFT is that to compute LT, one inspects only the leftmost operator of each rule. (Similarly for RT and RIGHT). Clearly, LT $\subseteq$ LEFT and RT $\subseteq$ RIGHT. Consequently LT FIRST $\subseteq$ LEFT FIRST and RT FIRST $\subseteq$ RIGHT FIRST.

Suppose LT is used instead of LEFT and RT is used instead of RIGHT in Algorithm 4.6. If the containment of LT in LEFT or of RT in RIGHT is proper, then certain actions will not be inspected by the modified algorithm. However, if the computation performed by lemma 4.7 is replaced by the computation obtained by substituting LT, RT for LEFT, RIGHT respectively in lemma 4.7, then the resulting but somewhat less precise computation can be used in the IR

validity check. In other words, if $LEFT_{IR} \subseteq LT_{CG}$, $RIGHT_{IR} \subseteq RT_{CG}$, and $FIRST_{IR} \subseteq FIRST_{CG}$, then the code generator produces code for all inputs (but it might never be possible to use certain sequences of instructions).

We now state a theorem that says the code generator will always produce correct output for any valid IR input.

Theorem 4.4: If **G** is a uniform grammar, α is a valid IR string according to **G**, and the code generator constructed for **G** by the algorithms in Chapter 4 is shown to contain no loops, then that code generator will always **accept** α and generate correct code for α.

Proof: Theorem 4.1 states that correct code is always generated (i.e. incorrect code is never generated). By a simple counting argument it can be shown that a non-looping code generator must eventually **accept** an input string or else block (i.e. perform an **error** action). Since α is valid, the code generator will not block. Therefore it must **accept**, and consequently will have generated correct code for the entire input. □

Notice that we have indirectly justified the conflict-resolution rules used by Algorithms 4.2 and 4.3, namely that **shift/reduce** conflicts are resolved in favor of **shifts** and **reduce/reduce** conflicts are resolved in favor of the longest reduction.

Suppose the grammar is not uniform. Then certain operands to certain operators can occcur in some contexts but not in others. If the implementer can identify these operators, operands, and contexts, and can insure that the same restrictions apply to the IR, then our method can still be used. Since non-uniform instruction sets do not seem to occur in real machines, we have not attempted to study weaker conditions that uniformity.

4.5.7. Semantic Blocking

If the code generator is performing a **reduce** R, but each rule in R contains a semantic restriction that is not satisfied, then the coder would **semantically block**. This can usually be avoided by the use of a **default list** of shorter instructions that contain no semantic restrictions and together compute the desired expression. This list is automatically generated by the preprocessor. Again consider the memory-to-memory add instruction†. The basic instruction pattern is '$:= k + \uparrow k \uparrow k$'. If this pattern occurs in the IR, but the constants associated with each of the three k's are distinct, then the instruction cannot be used. If there are instructions with the patterns '$r ::= \uparrow k$', '$r ::= + \uparrow k\ r$', and '$\lambda ::= := k\ r$', then they could be issued, simulating a non-restricted memory-to-memory add instruction, and the code generation could

†$\lambda ::= (:= k.1 + \uparrow k.1 \uparrow k.2)$ "**madd** $k.2,k.1$"
$\lambda ::= (:= k.1 + \uparrow k.2 \uparrow k.1)$ "**madd** $k.2,k.1$"

proceed from there as though the longer instruction had been issued. In fact, a default list of instruction patterns that contains shorter, possibly semantically restricted instructions is used. This allows the code generator to utilize any shorter special instructions instead of always reverting to 'worst case' code when the longest instruction is not suitable. If no instruction on the reduce list for a specific sub-pattern is applicable, then that pattern's default list is recursively utilized.

Default instruction lists for **reduce** states are automatically constructed by the preprocessor. These lists are obtained by simulating the action of the coder using as input each instruction pattern that contains only semantically restricted instructions in its reduce list, using only those rules with right-hand sides that are shorter than the pattern under consideration. The construction, in effect, builds a code generator for the subset of shorter instructions patterns, and simulates the resulting code generator's actions, noting where each instruction is issued. This process appears as Algorithm 4.8.

In the presence of semantically restricted instruction patterns, the code generation Algorithm 4.1 will choose from a list of rules in step *2* when a **reduce** action is performed. Assuming that this list has already been sorted by the preprocessor, the rules must be tested sequentially until one is found that is semantically compatible. In the event that no instruction is acceptable, the default list of reductions is used to implement that reduction.

There are several classes of semantic restrictions that may have to be satisfied. Constants in the IR input may have to be equal to specific values, such as 1 in an increment instruction, and logical registers may have to be equivalent to specific actual registers. Multiple occurrences of a symbol in any class may have to refer to the same actual value or register. If the result is to appear in a register, then there must not be any references to the value in that register outside of that instruction pattern. Such references could exist only if some sort of common subexpression elimination has been done. (Register allocation is discussed in section 4.3.) Finally, any additional semantic restrictions required to properly describe a particular target computer may be added to the code generator. The proof of correctness only requires that a default instruction or list of instructions be available for each restricted instruction, so that it will always be possible to generate code.

If two instructions have the same instruction pattern but different result locations, the actual instruction used is arbitrary. Adherence to the uniformity condition insures that the coder will still accept the input regardless of the choice made. The item $[x \rightarrow \beta \cdot]$ will not be included in a state of a uniform instruction set unless a reduction using it on a valid input $\alpha\beta\gamma \Rightarrow \alpha x\gamma$ is still valid. For practical reasons, one may preorder the instructions by cost, by the instruction that leaves the result in a register class with more actual registers (in an attempt

Algorithm 4.8 — Default List Construction

Input: The table for a shift-reduce code generator. The parameter **q** is the semantically restricted state.

Output: A list of default instructions for the semantically restricted **reduce** move.

Method: Build a code generator without the semantically restricted instruction and simulate the action of the code generator reading the restricted instruction pattern as input, noting where instructions are issued. The kth symbol of pattern ρ is designated $\rho[k]$. The state at the top of the stack is designated **tos**.

```
procedure defaultlistconstructor(q);
begin
    /* ACTION[q,*] = reduce R, and all instructions in R semantically restricted */

    /* Find the Starting State for this pattern */

    ρ ← R.H.S. of instruction pattern for rules in R followed by '$';
    q' ← first state with [v → · ρ] in it;
    m ← {[v → α · β] ∈ I | v ∈ V, |αβ| ≤ |ρ|};
    f ← {[v → · α] ∈ I | v ∈ V};  k ← 1;

    /* Simulate the action of a code generator on the input ρ
        using only patterns that are shorter than ρ */

    while k ≤ |ρ| do begin
        repeat
            case ACTION[q',ρ[k]] of
                accept: k ← ∞; Readflag ← TRUE;
                shift: q'' ← close(m ∩ (q'−f))
                       if ∃ an item [x → α · v β] ∈ q'' with v = ρ[k] then
                           q' ← NEXT[q', ρ[k]]; Readflag ← TRUE; push(q');
                       else i ← "x ::= α" with [x → α ·] ∈ q'' for maximum |α|;
                           output("reduce i at k"); pop |α| times;
                           q' ← NEXT[tos,x]; push(q');
                reduce: i ← "x ::= α" ∈ R; output("reduce i at k");
                       pop |α| times; q' ← NEXT[tos,x]; push(q');
                error: output("Cannot produce code for ρ");
            end;
        until Readflag;
        k ← k + 1;
    end;
end defaultlistconstructor;
```

to avoid having to save registers in temporaries), or by any other ordering desired, and the coder would still function correctly. It is also possible to examine the operator corresponding to *parent*$(\mathbf{x},\alpha\mathbf{x}\gamma)$ to determine the instruction that leaves the result in a location that is closer to a valid operand to that operator, thus possibly avoiding a subsequent register move instruction.

Chapter 5: Two Examples: The PDP-11 and The IBM 370

This chapter describes partial implementations of code generators for PASCAL on two target computers using the machine independent compiler techniques described in this dissertation. The target computers are the PDP-11 and the IBM 370. These computers were picked for the trial implementations because of their general availability and their contrasting architectures and instruction sets. PASCAL was used for the source language because of its relatively clean design and increasing popularity, the author's familiarity with it, and the existence of PASCAL compilers and interpreters locally.

The preprocessor system and code generator described in this dissertation were fully implemented. The implementations are described in section 5.1. A full PASCAL compiler was not implemented in order to make the project manageable by a single person. Instead, a very straightforward translation of source programs into IR was done by hand in order to generate code. The remaining two sections describe the sample implementations, the problems encountered in producing them, and the kind of code they generate.

5.1. The Preprocessor and Coder Implementations

The TMDL preprocessor and code generator are written in Unix PASCAL and execute on a PDP-11/70 via an interpreter [Joy77]. The preprocessor consists of four passes. This organization was necessitated by the limited address space of 28K words (program + data) of the PDP-11, not by the preprocessor algorithm. The first pass performs the scanning, parsing, and semantic analysis for TMDL and produces diagnostic messages and source listings for the user. A small amount of processing is performed on the instruction patterns to allow more efficient semantic checking by the code generator. Using Algorithm 4.2, the second pass computes the LR(0) item set and SLR(1)-like state set necessary to construct the code generator, and the initial NEXT and ACTION matrices and **reduce** lists used by the code generator. The third pass uses Algorithm 4.7 to determine whether the coder will block, and if so, whether the blocking can be prevented. The fourth pass performs loop detection and elimination (using Algorithms 4.4 and 4.5), and constructs the necessary default instruction lists using Algorithm 4.8. Listing and debugging options are also available. The code generator reads the files produced by the preprocessor and the IR input and generates symbolic assembly language output. At present, the IR is implemented as a source level language consisting of ASCII characters, much like the examples in this dissertation. While this is not the most efficient IR representation possible, it does allow test strings to be more easily constructed by hand. A production compiler would of course utilize a binary coding for the IR, and would also be compiled instead of interpreted.

Some graduate students at Berkeley are presently constructing a front end for a machine independent PASCAL compiler using the front end of the Unix PASCAL interpreter. The interpreter is multiple pass and builds an APT in the process of compiling a program. It is a relatively easy task to flatten the APT into a suitable IR. Thus, with a reasonable amount of effort a make-shift front end is being constructed, resulting in a complete, if somewhat patchwork, machine independent compiler, allowing real programs to be compiled. Figure 5.1 outlines such a completed implementation.

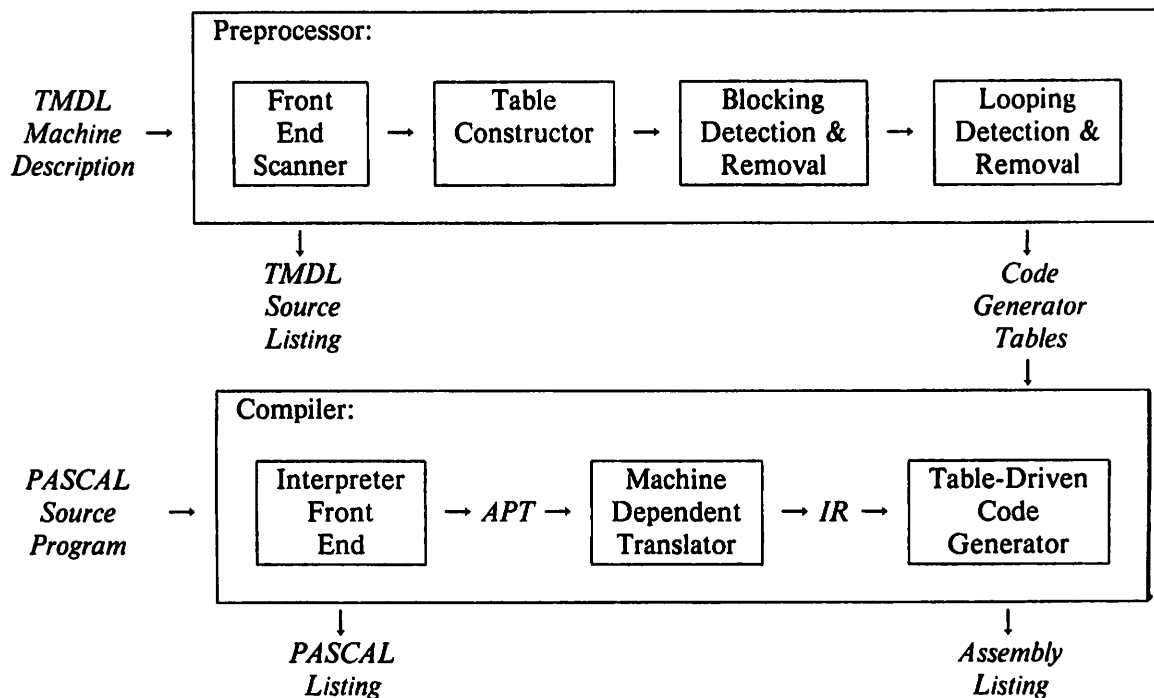


Fig. 5.1. A Machine Independent PASCAL Compiler.

5.2. Generating PDP-11 Object Code

A TMDL description of the majority of the instruction set of the PDP-11/70 computer was used to create a code generator for the PDP-11. The PDP-11 architecture is very enlightening concerning TMDL and the Shift-Reduce code generation algorithm. It illustrates the considerable power of semantic machine description systems, particularly TMDL. Notable is the ease with which complex instructions can be described, (e.g. the memory-to-memory, indirect, indexed integer subtract instruction on the PDP-11). (No comment is made at this time as to the utility of such an instruction.) Describing the PDP-11 also uncovers deficiencies in TMDL and the code generation algorithm. These are primarily associated with the

awkwardness in handling instructions that have computational side effects (such as the auto increment and decrement modes on general purpose registers when used as index registers) and instructions that differentiate between odd and even numbered registers (such as the integer multiply and divide instructions).

The TMDL machine description for the PDP-11 appears in Appendix A. There are 106 distinct instructions and variants described in this partial list. There are 10 pseudo-instructions that are used exclusively in determining whether a register is odd or even numbered in preparation for a multiply or divide operation; these instructions generate no code. A few addressing modes were left out of the description for some instructions in order to fit the preprocessor into core, as were the I/O, floating-point, procedure invocation, and byte-length memory referencing instructions. There are 26 distinct symbols used in the vocabulary (in addition to end-of-file, $). Five represent register classes: **r** any general purpose register, **d** an odd-even pair of allocatable registers, **o** and **e** odd and even numbered allocatable registers, and **c** the condition code register. Of the eight general purpose registers on the PDP-11, the first five are allocatable, **r5** is the local base register, **r6** is the stack pointer (also referred to as **sp**), and **r7** is actually the program counter, **pc**, and not a general purpose register at all. Two symbols *k* and *l* are used to represent constants, one for 16-bit, word-length constants and the other for compiler generated label numbers. There is no reason that these symbols could not be merged into a single symbol; they are only distinguished in order to clarify the instruction set description for the reader. There are seven jump operators: <, >, ⩽, ⩾, =, ≠, and *j*. The first six represent conditional jump binary operators. A transfer of control to the label defined by the first argument is taken if the indicated relationship holds between the last two values compared, according to the current state of the condition code register. The unconditional jump operator, *j*, is a unary operator taking only a label number as an operand. The symbol : is a unary operator that defines the location of the label specified by the value of its operand. The comparison operator, *?*, compares the values of its two operands and sets the condition code register accordingly. The dereferencing symbol, ↑, is a unary operator that computes the value of the 16-bit memory location addressed by its operand. The store operator, :=, stores the value of its second operand into the 16-bit memory location addressed by its first operand. The remaining 8 symbols, +, −, *, /, *m*, &, |, and !, are operators representing integer addition, subtraction, multiplication, division, and negation, and Boolean **and**, **or**, and **not**, respectively.

The resulting code generator has 217 active states. Five looping configurations were detected and eliminated from the initial coder's tables of 217 states, causing 5 new states to be created, for a maximum state count of 222. This left 2 original states unreferenced. Four of the newly created states were identical, including reduce lists, and could be combined into a single state, for a net of 217 states. The final coder was shown to be loopless by Algorithm 4.4.

One potentially blocking state was discovered and was altered to prevent the possibility of its blocking. The specifics of this blocking are discussed in detail later in this section.

The total execution time for creating the PDP-11 code generator was 315 seconds. A summary of the execution times for each pass and function is given in Fig. 5.2. These timings are for interpretive execution with all runtime checks enabled. Since subranges are used extensively in the programs, the runtime range-checking overhead was significant. A run with runtime checks disabled required only 233 seconds. The fact that the programs were being executed on an interpreter is responsible for an estimated factor of 10 to 20 in the execution time. The estimated execution time for a compiled version of the preprocessor would be 16 to 32 seconds with runtime checks enabled, a remarkably low time.

Module	Function	Execution Time (Seconds)	
		With Tests	Without Tests
PASS1	Processing TMDL Source	10.45	9.78
	Writing File	1.88	1.83
PASS2	Creating States	283.63	205.07
	Writing Files	3.35	3.30
PASS3	Eliminating Blocks	6.95	6.22
PASS4	Eliminating Loops	3.32	2.50
	Making Default Lists	5.90	4.52
Total:		315.48	233.22

Fig. 5.2. PDP-11 Preprocessor Execution Times.

The size of the resulting code tables is also pleasingly small, requiring 6300 bytes of storage. A large percentage of the basic storage requirement was eliminated by employing a simple table compaction technique. Instead of keeping the ACTION and NEXT tables as full arrays, identical rows were merged, and each state was given an index into the resulting merged array. Since there are only 7 unique rows in the ACTION table for the PDP-11 description, the requirement for the necessary 189 entries (27 entries/row × 7 rows) using only 2 bits per entry is just 48 bytes. Each state is required to have an index into this array of 3 bits in length, so an additional 82 bytes is required for the 217 values. There are 134 distinct rows in NEXT, for a total of 3618 bytes plus 217 for the index array. The default list requires 90 bytes plus a 217 byte index, and the reduce list requires 360 bytes plus the 217 byte index. The assembly information fields in the instructions require an additional 1440 bytes, for a grand total of just under 6300 bytes, a quite small code generator indeed. The sizes are summarized in Fig. 5.3.

Item	Entries	Bits	Total Bytes
ACTION array	189	2	48
Index to ACTION	217	3	82
NEXT array	3618	8	3618
Index to NEXT	217	8	217
Reduce List	360	8	360
Index to Reduce	217	8	217
Default List	90	8	90
Index to Default	217	8	217
Assembly Information	-	-	1440
Total:			6289

Fig. 5.3. PDP-11 Code Generator Table Sizes.

The potentially blocking state was intentionally introduced into the machine description to illustrate the blocking detection and removal mechanism. It represents an incomplete rather than erroneous instruction set description, so the preprocessor was able to accommodate it. The instruction which causes the 'trouble' is an indirect, indexed store constant instruction described by:

λ ::= (:= ↑ + *k*.1 *r*.1 *k*.2) "**mov** *k*.2,**k*.1(*r*.1)"

Since there is no other instruction description that stores into an indirect, indexed memory location, the only item that can be extended once the pattern ':= ↑ + *k r*' has been shifted into the stack, is [λ → := ↑ + k r • k], indicating that initially the only valid next symbol would be a constant. But any arithmetic expression potentially could be stored into an indirect, indexed memory location, so any initial expression symbol should be accepted as a valid next symbol. State **137**, where the block occurred, was initially:

137* [e → r •]
[o → r •]
[d → r •]
[r → + k r •]
[r → ↑ + k r •]
[λ → := ↑ + k r • k]

k: **shift 166**

The dot in the last item precedes the second operand to :=, so all symbols **x** with := RIGHT FIRST **x** should have an action other than **error**. By removing the non-completed item introduced by the troublesome instruction, and recomputing the ACTION for the

additional symbols, the action **reduce** '$r ::= \uparrow + k\ r$' was inserted into the table for all such entries. The block was thus eliminated, as the code generator could continue from that state for any valid next symbol. Note that this is the action that would have occurred on the input '$:= \uparrow + k\ r\ k$' had the indirect, indexed store constant instruction not been included in the machine description in the first place. The coder will, however, issue that instruction when the necessary pattern appears, so the quality of code produced by the code generator is not degraded by making this change.

The runtime environment required for PASCAL is described next. The PDP-11 has addressing modes that increment or decrement a general purpose register when it is used as a memory address. These instructions allow main memory to be treated like a stack in a very simple and efficient manner. The UNIX operating system uses an inverted stack (where the top of the stack is at the lowest memory address) to hold saved registers, operands, and return addresses when calling systems routines, so this convention was adopted for PASCAL. By convention, general purpose register **sp** (**r6**) points to the next free location on the top of the stack. When a value is pushed onto the stack, **sp** is first decremented by 2 (the PDP-11 is byte addressable) and the the resulting value is used as the memory location into which to store the value. The base address for local variables is kept in **r5**, and is normally greater than **sp**. The base address for the statically enclosing procedure is stored in the memory location pointed to by **r5**, and the base address for each successively enclosing procedure is pointed to by the last active, enclosed procedure's base. The resulting linked-list facilitates the accessing of non-local bases, allowing a simple indirect memory reference to chain to the base address of the next statically enclosing procedure. The first segment of memory below **sp** is the free space, followed by the heap, the global variables (at constant memory locations), and the executable code. This organization is depicted in Fig. 5.4. The calling and return code sequences necessary to maintain PASCAL's runtime environment are shown in Fig. 5.5.

Two sample programs are examined next to investigate the quality of code generated by the code generator. Unfortunately, there was no PDP-11 PASCAL compiler available with which to compare the resulting code. Therefore the closest equivalent programs were written in C, the systems programming language for UNIX, and the resulting code used for the comparison. An idiom optimizer is used by the C compiler to optimize the assembly language output prior to assembly. It was trivial to utilize this post code generation optimizer to further improve the output of our code generator, an unexpected bonus. Two PASCAL programs were hand translated into IR and input to the code generator. They appear along with their IR and C counterparts in Fig. 5.6 through Fig. 5.8. The first subroutine performs matrix multiplication and the second is an integer read function. It should be noted that more efficient C programs could be written. C allows the programmer to explicitly optimize programs extensively at the

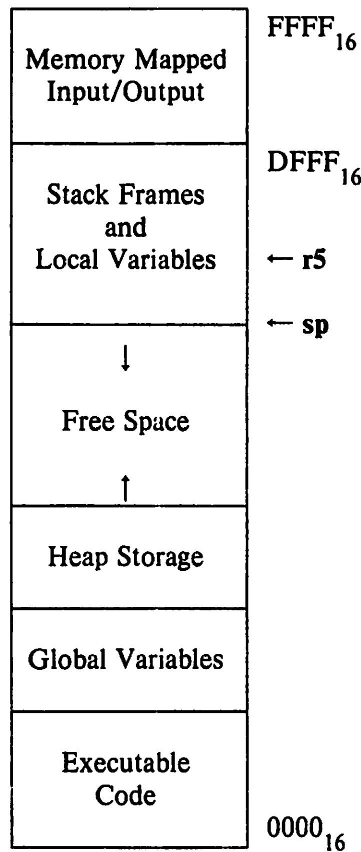
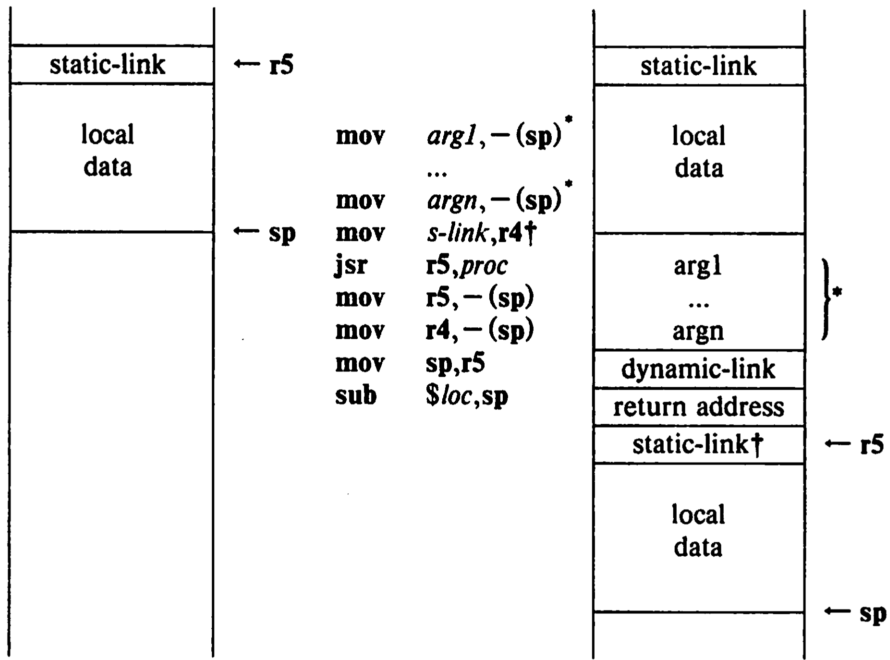


Fig. 5.4. PDP-11 PASCAL Runtime Organization.

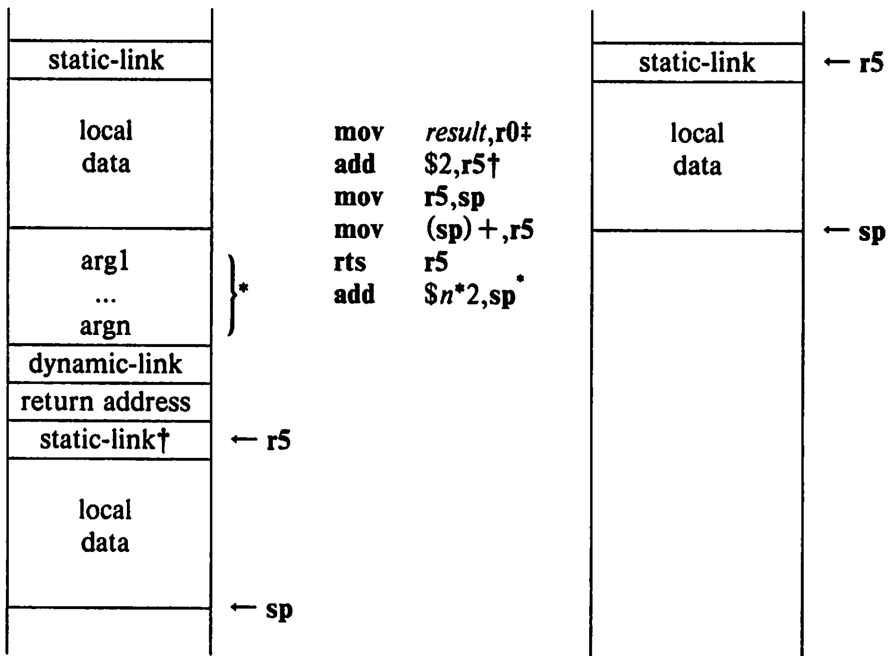
source level. But it would be an unfair test to compare the output of the PASCAL compiler with the code generated by the C compiler for such a program, so the closest C equivalent to the PASCAL programs are used.

Only code for the body of the procedures is used in the comparisons. Since C routines call the subroutines *csav* and *cret* to handle entry and exit code, the actual C assembly language output for procedure entry and exit is smaller but slower than inline code. Additionally, the runtime environment required for PASCAL is more complex and requires additional overhead to maintain. Therefore the only meaningful comparison of the routine sizes would exclude entry and exit code.

The assembly code produced for *matrixmult* by the machine independent code generator and the C compiler are shown in Fig. 5.9. The machine independent compiler generated code that is substantially better, requiring 65 words of memory compared to the 73 words required by the C compiler. The only code generated by the C compiler that is better is the use of a shift instruction to implement the doubling operations required in indexing integer arrays. This



a) Procedure invocation code sequence



b) Procedure exit code sequence

*Present only with parameters.

†Not needed if called procedure is at top level.

‡Used only for functions.

Fig. 5.5. PDP-11 PASCAL Procedure Entry and Exit Code.

```
{ PASCAL program }

const MAXINDEX = 9;
type matrix = array[0..MAXINDEX] of
                   array[0..MAXINDEX] of integer;

procedure matrixmult(var a,b,c: matrix);
     var i,j,k,sum: integer;
begin
     for i := 0 to MAXINDEX do
          for j := 0 to MAXINDEX do begin
               sum := 0;
               for k := 0 to MAXINDEX do
                    sum := sum + a[i][k] * b[k][j];
               c[i][j] := sum;
          end;
end { matrixmult };


/* C program */

matrixmult(a,b,c)
int a[10][10],b[10][10],c[10][10];
{    int i,j,k,sum;
     for (i=0;i<=9;i=i+1) {
          for (j=0;j<=9;j=j+1) {
               sum = 0;
               for (k=0;k<=9;k=k+1) {
                    sum = sum + a[i][k] * b[k][j];
               }
               c[i][j] = sum;
          }
     }
}
```

Fig. 5.6. Matrix Multiplication Routines.

instruction could be generated by the machine independent compiler if additional instruction patterns had been included to use a shift instruction in the assembly field, when the constant in the multiply is equal to the semantically restricted value of 2, as in:

‘*o*.1 ::= (* *o*.1 *k*=2) “**als** *o*.1” ’;

No new states would be added to the tables since the new instructions' patterns would be semantically restricted versions of existing patterns. The net size of the resulting code would be 62 words, only 85 per-cent the size of the C compiler's code.

```
{ PASCAL program }

var ch: char;

function readn: integer;
      var lval,base: integer;
begin
      while ch = ' ' do read(ch);
      if (ch ≤ '9') and (ch ≥ '0') then begin
            if ch = '0' then base := 8
            else base := 10;
            lval := 0;
            repeat
                  lval := lval * base + ord(ch) - ord('0');
                  read(ch);
            until (ch < '0') or (ord(ch) - ord('0') ≥ base);
            readn := lval;
      end else readn := -1;
end { readn };
```

```
/* C program */

int ch;

readn()
{     int answer,lval,base;
      while (ch == ' ') ch = getchar();
      if ((ch <= '9') && (ch >= '0')) {
            if (ch == '0') base = 8;
            else base = 10;
            lval = 0;
            do {
                  lval = lval*base + ch - '0';
                  ch = getchar();
            } while ((ch >= '0') && ((ch - '0') < base));
            answer = lval;
      } else answer = -1;
      return(answer);
}
```

Fig. 5.7. Integer Read Routines.

```
            := + k.I r.5 k.0
            j l.2
: l.1       := + k.J r.5 k.0
            j l.4
: l.3       := + k.SUM r.5 k.0
            := + k.K r.5 k.0
            j l.6
: l.5       := + k.sum r.5 + ↑ + k.SUM r.5
                * ↑ + ↑ + k.A r.5 * k.2 + * ↑ + k.I r.5 k.M ↑ + k.K r.5
                    ↑ + ↑ + k.B r.5 * k.2 + * ↑ + k.K r.5 k.M ↑ + k.J r.5
            := + k.K r.5 + ↑ + k.K r.5 k.1
: l.6       ≤ l.5 ? ↑ + k.K r.5 k.M
            := + ↑ + k.C r.5 * k.2 + * ↑ + k.I r.5 k.M ↑ + k.J r.5 ↑ + k.SUM r.5
            := + k.J r.5 + ↑ + k.J r.5 k.1
: l.4       ≤ l.3 ? ↑ + k.J r.5 k.M
            := + k.I r.5 + ↑ + k.I r.5 k.1
: l.2       ≤ l.1 ? ↑ + k.I r.5 k.M
```

a) IR translation of *matrixmult*

```
: l.1       ≠ l.2 ? ↑ k.CH k.' '
            j l.1
: l.2       > l.3 ? ↑ k.CH k.'9'
            < l.3 ? ↑ k.CH k.'0'
            ≠ l.5 ? ↑ k.CH k.'0'
            := + k.BASE r.5 k.8
            j l.6
: l.5       := + k.BASE r.5 k.10
: l.6       := + k.LVAL r.5 k.0
: l.7       := + k.LVAL r.5 − + * ↑ + k.LVAL r.5 ↑ + k.BASE r.5 ↑ k.CH k.'0'
            < l.8 ? ↑ k.CH k.'0'
            < l.7 ? − ↑ k.CH k.'0' ↑ + k.BASE r.5
: l.8       := + k.READN r.5 ↑ + k.LVAL r.5
            j l.4
: l.3       := + k.READN r.5 k.−1
: l.4
```

b) IR translation of *readn*

Fig. 5.8. PDP-11 IR Translation of Test Programs.

```
        clr     -2(r5)
        jbr     L2
L1:     clr     -4(r5)
        jbr     L4
L3:     clr     -10(r5)
        clr     -6(r5)
        jbr     L6
L5:     mov     $12,r0
        mul     -2(r5),r0
        add     -6(r5),r0
        mul     $2,r0
        add     12(r5),r0
        mov     $12,r1
        mul     -6(r5),r1
        add     -4(r5),r1
        mul     $2,r1
        add     10(r5),r1
        mov     (r1),r1
        mul     (r0),r1
        add     r1,-10(r5)
        inc     -6(r5)
L6:     cmp     -6(r5),$11
        jle     L5
        mov     $12,r0
        mul     -2(r5),r0
        add     -4(r5),r0
        mul     $2,r0
        add     6(r5),r0
        mov     -10(r5),(r0)
        inc     -4(r5)
L4:     cmp     -4(r5),$11
        jle     L3
        inc     -2(r5)
L2:     cmp     -2(r5),$11
        jle     L1
```

a) Code Generator's Code

```
        clr     -10(r5)
L2:     cmp     $11,-10(r5)
        jlt     L3
        clr     -12(r5)
L5:     cmp     $11,-12(r5)
        jlt     L6
        clr     -16(r5)
        clr     -14(r5)
L8:     cmp     $11,-14(r5)
        jlt     L9
        mov     -10(r5),r1
        mul     $12,r1
        add     -14(r5),r1
        asl     r1
        add     4(r5),r1
        mov     (r1),r1
        mov     -14(r5),r3
        mul     $12,r3
        add     -12(r5),r3
        asl     r3
        add     6(r5),r3
        mul     (r3),r1
        add     -16(r5),r1
        mov     r1,-16(r5)
L10:    mov     -14(r5),r0
        inc     r0
        mov     r0,-14(r5)
        jbr     L8
L9:     mov     -10(r5),r1
        mul     $12,r1
        add     -12(r5),r1
        asl     r1
        add     10(r5),r1
        mov     -16(r5),(r1)
L7:     mov     -12(r5),r0
        inc     r0
        mov     r0,-12(r5)
        jbr     L5
L6:L4:  mov     -10(r5),r0
        inc     r0
        mov     r0,-10(r5)
        jbr     L2
L3:L1:
```

b) C Compiler's Code

Fig. 5.9. Assembly Listings for *matrixmult* Routines.

One might expect that the post-code generation optimization pass available for C compilations would make up the difference. This is not entirely the case. The optimized code modules were 62 (59 if a shift were used) and 66 words in length. A closer look reveals the reasons why. The C optimizer is primarily concerned with global optimizations involving the rearrangement of code blocks to minimize conditional tests and jumps in loops, and with eliminating jumps to jumps. (The PDP-11 has single and double word jump instructions, the former having a limited range of ± 128 words, so this optimization is important.) The compiler is responsible for recognizing idioms such as incrementing a variable or adding to memory. However, the compiler does not discover the full complement of such patterns *unless they are explicitly indicated in the source program*. The machine independent code generation algorithm, on the other hand, recognizes all special single instruction idioms regardless of their context. Two examples from the matrix multiply routines exemplify this point. The first one occurs when loop control variables are incremented and the second involves adding the value of an expression to a variable. In the C compiler, adding 1 to a local variable is only converted into an increment instruction when it is explicitly written as an increment operator. The naive statement:

$$i = i + 1;$$

yields the following 5 words of code:

```
mov   i(r5),r1
inc   r1
mov   r1,i(r5)
```

The C compiler does detect the fact that adding 1 can be best implemented by an increment instruction, but only when the value being incremented is in a register. Therefore, the compiler will never discover that the increment memory instruction could be used. If the programmer has explicitly indicated that the variable i is to be incremented, as in:

$$i++;$$

then the compiler does issue the locally optimal code:

```
inc i(r5)
```

The affect is equivalent to adding a unary operator to the IR that increments the memory

location addressed by its operand. The machine independent code generation algorithm, however, produces an increment instruction in all possible cases without the aid of such an operator. A similar source language optimization in C allows the programmer to indicate that the value of an expression is to be added to a variable. The naive C statement:

sum = *sum* + <*expr*>;

generates suboptimal code, while locally optimal code results from the equivalent statement:

sum =+ <*expr*>;

In the first case, the C compiler computes the value of <*expr*>, leaving the result in a register, adds the value of *sum* to that register, and stores the result into *sum*. In the latter case an add to memory instruction replaces the add and the store, saving two words of code. The code generated by the machine independent code generation algorithm produces the optimized version, again without the aid of a new IR operator.

Figure 5.10 shows the assembly code produced by both code generators for the second example routine, *readn*. The code generated is almost identical in every respect. The machine independent code generator produced 64 words of code while the C compiler produced 65. The difference is due to an unnecessary jump instruction generated by the C compiler that jumps to the next sequential memory location. Again, more efficient C programs can be written by utilizing features and operators that are not present in PASCAL. Since PASCAL does not have a **return** statement, the value of a function is stored in a local variable until the function is exited. The C program was also written in this manner, though slightly better code is produced by the use of **return** statements (62 words). The post code generation optimizer does remove the unnecessary jump instruction from the C compilers output, and the resulting code for each compiler is 61 words, identical except for register and label numbers and the reassigning of the relative addresses of local variables.

The machine independent code generation algorithm utilizes many special purpose, machine dependent instructions, such as increment memory, without the aid of a special source language operator or other means of locating such occurrences. This observation would indicate that a PASCAL compiler using such a code generator would produce code almost as good as a systems programming language like C. Furthermore, the programmer is not forced to write cryptic and unreadable programs using those special operators in order to produce the most efficient programs. The 'natural' representation of the statement is just as good, even without the extra cost of an expensive optimization pass. Thus, a code generation algorithm

```
L1:     cmp   _ch,$40
        jne   L2
        jsr   pc,_getchar
        mov   r0,_ch
        jbr   L1
L2:     cmp   _ch,$71
        jgt   L3
        cmp   _ch,$60
        jlt   L3
        cmp   _ch,$60
        jne   L5
        mov   $10,-6(r5)
        jbr   L6
L5:     mov   $12,-6(r5)
L6:     clr   -4(r5)
L7:     mov   -6(r5),r0
        mul   -4(r5),r0
        add   _ch,r0
        sub   $60,r0
        mov   r0,-4(r5)
        jsr   pc,_getchar
        mov   r0,_ch
        cmp   _ch,$60
        jlt   L8
        mov   _ch,r0
        sub   $60,r0
        cmp   r0,-6(r5)
        jlt   L7
L8:     mov   -4(r5),-2(r5)
        jbr   L4
L3:     mov   $-1,-2(r5)
L4:     mov   -2(r5),r0
```

a) Code Generator's Output

```
L2:     cmp   $40,_ch
        jne   L3
        jsr   pc,_getchar
        mov   r0,_ch
        jbr   L2
L3:     cmp   $71,_ch
        jlt   L4
        cmp   $60,_ch
        jgt   L4
        cmp   $60,_ch
        jne   L5
        mov   $10,-14(r5)
        jbr   L6
L5:     mov   $12,-14(r5)
L6:     clr   -12(r5)
L9:     mov   -12(r5),r1
        mul   -14(r5),r1
        add   _ch,r1
        add   $-60,r1
        mov   r1,-12(r5)
        jsr   pc,_getchar
        mov   r0,_ch
L7:     cmp   $60,_ch
        jgt   L10000
        mov   _ch,r0
        add   $-60,r0
        cmp   -14(r5),r0
        jgt   L9
L10000:L8:    mov -12(r5),-10(r5)
        jbr   L10
L4:     mov   $-1,-10(r5)
L10:    mov   -10(r5),r0
        jbr   L1
L1:
```

b) C Compiler's Output

Fig. 5.10. Assembly output for *readn* routines.

that was primarily designed to provide ease of use in describing the code to be generated, and to allow retargeting and transporting of a language to be done with minimal effort, has in addition succeeded in producing very respectable object code.

5.3. Generating IBM 370 Object Code

A TMDL description of a portion of the instruction set for the IBM system/370 computer was used to generate a coder for that machine. Enough instructions were described to allow code to be generated for the sample programs of section 5.2. Equivalent ALGOL-W programs were written and compiled on the IBM 370 in order to compare the code generated. The results appear in this section.

The IBM 370 is a more conventional computer than the PDP-11. It does, however, have an enormous instruction set, including a wealth of extremely specialized instructions such as edit, search, decimal arithmetic, convert-to-binary/packed-decimal, etc. These special instructions were not used in the machine description — there is no equivalent construct in PASCAL. Some of those instructions would require a small PASCAL procedure equivalent. This is not a deficiency of the code generation scheme; such instructions are only utilized in special places even by assembly language programmers. The remaining instructions constitute standard memory-register and register-register operations.

The TMDL machine description for the IBM 370 appears in Appendix B. It contains 58 distinct instructions and variants, 3 of which are pseudo-instructions and generate no code. Of the 27 symbols used (in addition to $\$$), there are only 4 that differ in meaning from those used in the PDP-11 description. The operand k represents a 12 bit constant instead of 16, and the memory reference operators $\uparrow$ and := reference 32 bit quantities instead of 16. The operator a indicates 24 bit integer addition, and is used primarily in indexing operations, while the operator + indicates integer addition of 32 bit operands. The fact that the more general operator can be used to implement an index addition is indicated by listing each instruction that implements + as also implementing a, as in:

```
r.1 ::= (+ r.1 r.2)          "ar  r.1,r.2"
r.1 ::= (a r.1 r.2)          "ar  r.1,r.2"
```

This duplication of instruction descriptions accounts for 4 of the instruction patterns.

The resulting code generator has 164 states. Two looping configurations were detected and eliminated from the basic tables of 164 states, creating 2 new states. This left 1 original state unreferenced. The two newly created states were identical, and could be combined, for a net result of 164 states. The final coder was shown to be loopless. No blocking configurations were discovered, and no default lists had to be constructed. The total execution time for the construction was 165 seconds (or 111 seconds with run-time checks disabled), an estimated equivalent of 8 to 16 seconds for a compiled version. The resulting tables occupy 4700 bytes. Execution timings and table sizes are summarized in Fig. 5.11 and Fig. 5.12.

Module	Function	Execution Time (Seconds)	
		With Tests	Without Tests
PASS1	Processing TMDL Source	5.82	4.40
	Writing File	1.10	.87
PASS2	Creating States	147.43	98.45
	Writing Files	2.98	2.00
PASS3	Eliminating Blocks	5.87	4.60
PASS4	Eliminating Loops	2.37	1.40
	Making Default Lists	.27	.15
Total:		165.84	111.87

Fig. 5.11. IBM 370 Preprocessor Execution Times.

Item	Entries	Bits	Total Bytes
ACTION array	140	2	35
Index to ACTION	166	3	63
NEXT array	3164	8	3164
Index to NEXT	166	8	166
Reduce List	222	8	222
Index to Reduce	166	8	166
Default List	0	8	0
Index to Default	-	-	0
Assembly Information	-	-	820
Total:			4636

Fig. 5.12. IBM 370 Code Generator Table Sizes.

The runtime environment supporting PASCAL is similar to that used by other PASCAL implementations [Wirth71,72]. The IBM 370 has an array of 16 general purpose registers. This would allow the display to be kept in registers if so desired. However, in interest of efficient procedure entry and exit code sequences, only the local and global displays are kept in registers. The intervening displays are seldom accessed by programs according to a study by [Wirth72], so this decision does not slow the execution of procedure bodies unacceptably. The dedicated registers are allocated in the following manner: The display for local variables resides in **r14** while the display for global variables resides in **r13**. There are no absolute addressing instruction modes in the IBM 370 instruction set. Memory is addressed as base + displacement + optional index, so the global variables also must have a display. Program jumps are also base-relative, so an additional register **r15** is used to hold the display for the currently executing procedure. This value is loaded upon procedure entry and exit but remains constant within a

procedure, limiting the maximum size of the body of a procedure to 4096 bytes. The top of the run-time stack and the bottom of the heap are pointed to by registers **r12** and **r11**, respectively. Figure 5.13 shows the overall organization of memory.

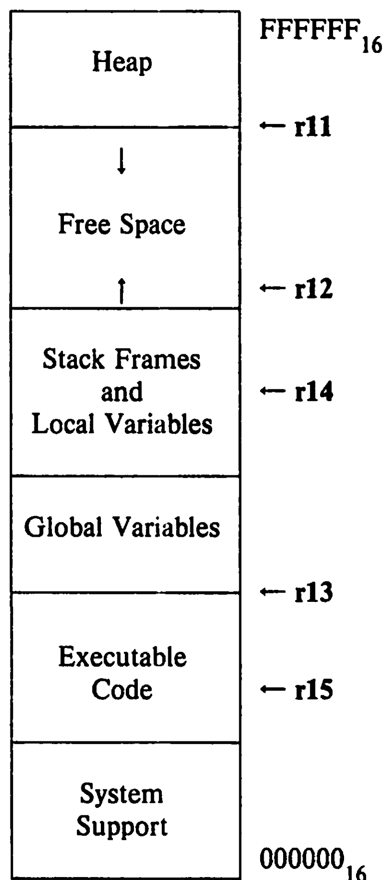


Fig. 5.13. IBM 370 PASCAL Runtime Organization.

The code generated for the two sample programs *matrixmult* and *readn* is presented next. The closest equivalent compiler available was the ALGOL-W compiler [Bauer68], so equivalent programs were written in ALGOL-W, compiled, and the resulting object code used for a comparison. The IR code for the two routines is given in Fig. 5.14. It is almost exactly the same as that for the PDP-11. The only major difference is that global variables are addressed by the sum of a constant offset and a base register. The equivalent ALGOL-W routines are given in Fig. 5.15.

Not a lot of effort was put into designing the IBM 370 code generator. There was no such computer immediately available and few people knowledgable in IBM 370's were around to help make decisions about the best organization for PASCAL. The TMDL description was

```
            := a k.I r.15 k.0
            j l.2
: l.1       := a k.J r.15 k.0
            j l.4
: l.3       := a k.SUM r.15 k.0
            := a k.K r.15 k.0
            j l.6
: l.5       := a k.SUM r.15  + ↑ a k.SUM r.15
                  * ↑ a ↑ a k.A r.15 * k.2 + * ↑ a k.I r.15 k.M ↑ a k.K r.15
                      ↑ a ↑ a k.B r.15 * k.2 + * ↑ a k.K r.15 k.M ↑ a k.J r.15
            := a k.K r.15 + ↑ a k.K r.15 k.1
: l.6       ≤ 1.5 ? ↑ a k.K r.15 k.M
            := a ↑ a k.C r.15 * k.2 + * ↑ a k.I r.15 k.M ↑ a k.J r.15 ↑ a k.SUM r.15
            := a k.J r.15 + ↑ a k.J r.15 k.1
: l.4       ≤ l.3 ? ↑ a k.J r.15 k.M
            := a k.I r.15 + ↑ a k.I r.15 k.1
: l.2       ≤ l.1 ? ↑ a k.I r.15 k.M
```

a) IR translation of *matrixmult*

```
: l.1       ≠ l.2 ? ↑ a k.CH r.13 k.' '
            j l.1
: l.2       > l.3 ? ↑ a k.CH r.13 k.'9'
            < l.3 ? ↑ a k.CH r.13 k.'0'
            ≠ l.5 ? ↑ a k.CH r.13 k.'0'
            := a k.BASE r.14 k.8
            j l.6
: l.5       := a k.BASE r.14 k.10
: l.6       := a k.LVAL r.14 k.0
: l.7       := a k.LVAL r.14 - + * ↑ a k.LVAL r.14 ↑ a k.BASE r.14 ↑ a k.CH r.13 k.'0'
            < l.8 ? ↑ a k.CH r.13 k.'0'
            < l.7 ? - ↑ a k.CH r.13 k.'0' ↑ a k.BASE r.14
: l.8       := a k.READN r.14 ↑ a k.LVAL r.14
            j l.4
: l.3       := a k.READN r.14 m k.1
: l.4
```

b) IR translation of *readn*

Fig. 5.14. IBM 370 Translation of Test Programs.

```
procedure matmult(integer array a,b,c(*,*));
begin
    integer sum,i,j,k;
    for i := 0 until 9 do
        for j := 0 until 9 do begin
            sum := 0;
            for k := 0 until 9 do
                sum := sum + a(i,k)*b(k,j);
            c(i,j) := sum
        end
end.
```

a) ALGOL-W *matrixmult* procedure

```
begin
    string(1) ch;

    integer procedure readn;
    begin
        integer answer,lval,base;
        while ch = " " do read(ch);
        if (ch <= "9") and (ch >= "0") then begin
            if ch = "0" then base := 8
            else base := 10;
            lval := 0;
            while (ch >= "0") and ((ch - "0") < base) do begin
                lval := lval * base + ch - "0";
                read(ch)
            end;
            answer := lval
        end else answer := -1;
        answer
    end;
end.
```

b) ALGOL-W *readn* procedure

Fig. 5.15. ALGOL-W Comparison Routines.

generated by the author from a hardware manual on the IBM 360 [IBM66]. Consequently a few simplifying assumptions were made to ease the implementation. It turns out that many of the same simplifications have been used in implementing other compilers on the IBM 360, including the ALGOL-W compiler and at least one PASCAL compiler. These simplifications include limiting the size of the code for a procedure and the directly addressable local variable space to 4096 bytes each. In any event, a TMDL description of the IBM 370 was written and the code generator was altered to produce IBM 370 assembly language code with a minimum of effort.

The assembly code output by the machine independent code generator is examined next. The output for *matrixmult* appears in Fig. 5.16. It consists of 60 instructions, and occupies 212 bytes. The equivalent ALGOL-W code appears in Fig. 5.17, also consists of 60 instructions, and occupies 224 bytes. As with the comparisons with C in section 5.2, only the code generated for the bodies of the procedures is shown, as the parameter passing and environment housekeeping performed by the two languages differs somewhat. The output of the ALGOL-W compiler is somewhat larger, primarily due to the fact that the **step** and limit for the **for** loops are kept in memory despite the fact that they are compile-time constants. The use of a **BXLE** instruction at the end of each **for** loop, however, almost makes up for the extra 16 bytes of initialization code per loop. Including the instruction description:

$$\lambda ::= (\leqslant l.1 \ ? \ r.1 \ r.2) \quad \text{"}\mathbf{BXLE}\ r.1,r.2,\mathrm{L}l.1(\mathbf{r15})\text{"};$$

in the IBM 370 TMDL description would result in its replacing a **CR** and **BC** instruction pair three times, shortening the code by a total of 6 bytes.

The assembly language output for both *readn* routines appears in Fig. 5.18. The machine independent compiler generated 45 instructions requiring a total of 164 bytes of memory, exclusive of the two calls to the *read* routine. The ALGOL-W compiler generated 44 instructions and three literals requiring 170+3 bytes. The difference stems from the fact that the global character variable *ch* is treated as a string of length one by the ALGOL-W compiler but is treated as a scalar variable by PASCAL. ALGOL-W issues the more efficient **CLC** instruction when comparing *ch* to a literal constant, but issues less efficient code when generating character constants for use in a computation. Literal constants are built by the pair of instructions:

```
SR   r3,r3
IC   r3,0026(r14)
```

where 0026(**r14**) addresses the character constant. The **SR** instruction first clears **r3** and then

the **IC** instruction inserts a single character into the lowest byte. The machine independent code generator issues the single instruction:

LA r3,"0"

to load any constant in the range 0 to 4095.

```
        SR   r2,r2
        ST   r2,001C(r14)
        BC   14,L2(r15)
L1:     SR   r2,r2
        ST   r2,0020(r14)
        BC   14,L4(r15)
L3:     SR   r2,r2
        ST   r2,0028(r14)
        SR   r2,r2
        ST   r2,0024(r14)
        BC   14,L6(r15)
L5:     LA   r2,2
        LA   r3,12
        LR   r4,r3
        M    r4,001C(r14)
        A    r4,0024(r14)
        MR   r4,r2
        A    r4,0010(r14)
        L    r2,0000(r4)
        LA   r3,2
        LA   r4,12
        M    r4,0024(r14)
        A    r4,0020(r14)
        MR   r4,r3
        A    r4,0014(r14)
        L    r3,0000(r4)
        LR   r4,r3
        MR   r4,r2
        A    r4,0028(r14)
        ST   r4,0028(r14)
        LA   r2,1
        A    r2,0024(r14)
        ST   r2,0024(r14)
L6:     L    r2,0024(r14)
        LA   r3,11
        CR   r2,r3
        BC   12,L5(r15)
        LA   r2,2
        LA   r3,12
        LR   r4,r3
        M    r4,001C(r14)
        A    r4,0020(r14)
        MR   r4,r2
        A    r4,0018(r14)
        L    r2,0028(r14)
        ST   r2,0000(r4)
        LA   r2,1
        A    r2,0020(r14)
        ST   r2,0020(r14)
L4:     L    r2,0020(r14)
        LA   r3,11
        CR   r2,r3
        BC   12,L3(r15)
        LA   r2,1
        A    r2,001C(r14)
        ST   r2,001C(r14)
L2:     L    r2,001C(r14)
        LA   r3,11
        CR   r2,r3
        BC   12,L1(r15)
```

Fig. 5.16. IBM 370 Assembly code for *matrixmult*.

```
      LA    r2,1               LM    0,1,00C8(r11)
      ST    r2,00B8(r11)       BXLE  2,0,L5(r14)
      LA    r2,9         L6:   L     r2,00AC(r11)
      ST    r2,00BC(r11)       MH    r2,004E(r11)
      SR    r2,r2              L     r3,00B0(r11)
      C     r2,00BC(r11)       MH    r3,005A(r11)
      BC    2,L2(r14)          AR    r2,r3
L1:   ST    r2,00AC(r11)       AL    r2,0048(r11)
      LA    r2,1               L     r3,009C(r11)
      ST    r2,00C0(r11)       ST    r3,0000(r2)
      LA    r2,9               L     r2,00B0(r11)
      ST    r2,00C4(r11)       LM    0,1,00C0(r11)
      SR    r2,r2              BXLE  2,0,L3(r14)
      C     r2,00C4(r11) L4:   L     r2,00AC(r11)
      BC    2,L4(r14)          LM    0,1,00B8(r11)
L3:   ST    r2,00B0(r11)       BXLE  2,0,L1(r14)
      SR    r2,r2        L2:
      ST    r2,009C(r11)
      LA    r2,1
      ST    r2,00C8(r11)
      LA    r2,9
      ST    r2,00CC(r11)
      SR    r2,r2
      C     r2,00CC(r11)
      BC    2,L6(r14)
L5:   ST    r2,00B4(r11)
      L     r2,00AC(r11)
      MH    r2,0086(r11)
      L     r3,00B4(r11)
      MH    r3,0092(r11)
      AR    r2,r3
      AL    r2,0080(r11)
      L     r3,0000(r2)
      L     r4,00B4(r11)
      MH    r4,006A(r11)
      L     r5,00B0(r11)
      MH    r5,0076(r11)
      AR    r4,r5
      AL    r4,0064(r11)
      M     r2,0000(r4)
      SLDA  r2,0020
      A     r2,009C(r11)
      ST    r2,009C(r11)
      L     r2,00B4(r11)
```

Fig. 5.17. ALGOL-W Assembly code for *matrixmult.*

```
L1:   L     r2,ch(r13)
      LA    r3,' '
      CR    r2,r3
      BC    6,L2(r15)
         ... read(ch) ...
      BC    14,L1(r15)
L2:   L     r2,ch(r13)
      LA    r3,'9'
      CR    r2,r3
      BC    2,L3(r15)
      L     r2,ch(r13)
      LA    r3,'0'
      CR    r2,r3
      BC    4,L3(r15)
      L     r2,ch(r13)
      LA    r3,'0'
      CR    r2,r3
      BC    6,L5(r15)
      LA    r2,8
      ST    r2,0010(r14)
      BC    14,L6(r15)
L5:   LA    r2,10
      ST    r2,0010(r14)
L6:   SR    r2,r2
      ST    r2,0014(r14)
L7:   L     r2,0010(r14)
      M     r2,0014(r14)
      A     r2,ch(r13)
      LA    r3,'0'
      SR    r2,r3
      ST    r2,0014(r14)
         ... read(ch) ...
      L     r2,ch(r13)
      LA    r3,'0'
      CR    r2,r3
      BC    4,L8(r15)
      L     r2,ch(r13)
      LA    r3,'0'
      SR    r2,r3
      C     r2,0010(r14)
      BC    4,L7(r15)
L8:   L     r2,0014(r14)
      ST    r2,000C(r14)
      BC    14,L4(r15)
L3:   LA    r2,1
      LCR   r2,r2
      ST    r2,000C(r14)
L4:
```

a) Code Generator's Output

```
      DS    ' ','0','9'
L1:   CLC   0024(0,r14),ch(r12)
      BC    7,L2(r14)
         ... read(ch) ...
      BC    15,L1(r14)
L2:   CLC   0025(0,r14),ch(r12)
      BC    4,L7(r14)
      CLC   0026(r14),ch(r12)
      BC    2,L7(r14)
      CLC   0026(0,r14),ch(r12)
      BC    7,L3(r14)
      LA    r2,8
      ST    r2,0038(r11)
      BC    15,L4(r14)
L3:   LA    r2,10
      ST    r2,0038(r11)
L4:   SR    r2,r2
      ST    r2,34(r11)
L5:   CLC   0026(0,r14),ch(r12)
      BC    2,L6(r14)
      MVC   003C(0,r11),ch(r12)
      SR    r2,r2
      IC    r2,003C(r11)
      SR    r3,r3
      IC    r3,0026(r14)
      SR    r2,r3
      C     r2,0038(r11)
      BC    11,L6(r14)
      MVC   003D(0,r11),ch(r12)
      SR    r2,r2
      IC    r2,003D(r11)
      L     r5,0034(r11)
      M     r4,0038(r11)
      SLDA  r4,0020
      AR    r2,r4
      SR    r3,r3
      IC    r3,0026(r14)
      SR    r2,r3
      ST    r2,0034(r11)
         ... read(ch) ...
      BC    15,L5(r14)
L6:   L     r2,0034(r11)
      ST    r2,0030(r11)
      BC    15,L8(r14)
L7:   LA    r2,1
      LCR   r2,r2
      ST    r2,0030(r11)
L8:
```

b) ALGOL-W Compiler's Output

Fig. 5.18. IBM 370 Assembly code for *readn* Routines.

Chapter 6: Results and Conclusions

The machine independent code generation algorithm and code generator construction algorithms presented represent a step towards the automatic generation of correct compilers. The advantages of such a tool are many. The size of a TMDL machine description is only a fraction of the size of the equivalent source level routines. Furthermore, such a description represents a major improvement in clarity and modifiability. A level of assurance that the code generator always generates correct code is available that is not possible in other code generators, short of an impractical, exhaustive proof for each routine used. The ease with which a new machine description can be written and substituted for the existing one, thus creating a cross compiler, is not available in standard code generation schemes. Similarly, it is straightforward to alter the machine description to include additional instructions available on a upward compatible computer in a computer line, or to utilize instruction sets which are locally extended (e.g. via microprogramming). Detecting where special instructions can be used is automatically handled, without a tedious case by case analysis by the implementer. The total size of the resulting code generator (program + tables) is considerably smaller than procedural code generators, and can be further decreased in size by the utilization of additional table compaction techniques. Finally, as an added bonus, extremely good object code is generated, with the future addition of all popular optimizations desired cleanly fitting into the compiler.

As with most code generation schemes, this method does not work well with some of the more awkward computers in existence. The messier the architecture, the more headaches for the code generator implementer, would seem to be the central theme. Of particular note is the CDC-6000 series of computers (and all upward compatible artifacts). The high degree of complexity of the special functions provided by particular registers (often as side effects to some other action) has caused other compiler writers considerable grief [Wirth72] [Ammann77]. In addition, one must be concerned with the optimizations that must be used to fully utilize the parallelism of the arithmetic unit, or the instruction cache, on some models. The newly emerging micro-computers contain many of the same design mistakes made in earlier, large scale computers. (With software costs as high as they are, it is unfortunate that industry is reluctant to try to improve the machines on which software must run. Newer technologies are only used to implement a faster or cheaper version of older, archaic architectures.)

The primary reason that such computers present problems to compiler writers is that there are too many exceptions. Special functions and side effects are associated with specific registers. Different instructions have different ways of handling addressing modes, often with supposedly complementary instructions lacking true symmetry [Wirth68]. Almost every single

register has some unique property. This is true even on cleaner designs (such as the PDP-11) to a certain degree [Bron76]. Whether a register is odd or even is important to multiply and divide instructions. The archaic use of a condition code register, as opposed to the computation of a Boolean value that is left in a register, is totally unsatisfactory, and causes a great deal of trouble, since there are almost always shorter code sequences for many special cases. In many computers supposedly "general purpose" register **r0** cannot be used as an index register. The cost of making each register in an array of general purpose registers functionally the same surely cannot be prohibitive, especially in the anti-inflationary presence of diminishing hardware costs. The machine independent code generation algorithm is yet another argument in favor of cleaner computer architectures, though it can handle many of the existing anomolies. As more elegant parsing tools were developed, more was understood about parsing, and consequently the syntax of programming languages came to be more cleanly designed. There is no reason not to alter the architecture of future computers to reflect advances made in code generation techniques. The impact on compilation speed and complexity, as well as the speed and size of the resulting code, should be just as great and as positive.

6.1. Postfix vs. Prefix

The decision to describe the instruction descriptions as prefix expressions and consequently to require the IR to be a prefix language is not without foundation. In fact, postfix was initially preferred. If the item sets defining the states of the shift-reduce parser used to generate code are examined, the states roughly correspond to the set of possible instruction pattern prefixes. On a computer with several addressing modes, a prefix description would have separate patterns for each operator and addressing mode since the operator is the first symbol in the description. A postfix description would allow the states corresponding to the addressing parts of all operations to be combined, since the operator is the last symbol in the description. Equivalently, there are fewer strings corresponding to valid prefixes of a postfix instruction description than a prefix description, and consequently fewer states in the code generator. Additionally, a postfix IR is easier to generate than a prefix IR, and can be done in a one pass scheme. However, when an attempt was made to generate code for real computers, the code generation algorithm did not work as well with a postfix description.

The major problem with a postfix description is that not enough information is known at the time an operand is encountered to allow the code generator to properly handle it. It is not known which operand the current expression is to what operator. Most computers have some instructions that require one or more operands to obey exceptional conditions, such as being in an odd-numbered register, while not similarly restricting the operands of other instructions. In a postfix representation, the coder will stack both operands prior to encountering the operator.

It cannot be determined that the special conditions must apply until both operands have been shifted onto the stack. If the first operand does not meet those special conditions there is no way to transform it so that it will. The coder cannot modify any expression unless it is on the top of the stack. The algorithm can be allowed to 'back-up' and fix the problem. But this requires many more states to be added to the coder so that it can remember which way it is supposed to be going, and needlessly complicates both the preprocessor and the code generator. When a prefix representation is used, this problem does not occur. An operator is encountered prior to its operands and any special conditions that they must meet are implicitly remembered by the state of the code generator.

Consider the following integer add and divide instructions:

r.1 ::= (/ *d*.1 *r*.2)	"**div** *d*.1.1,*r*.2"
r.1 ::= (+ *r*.1 *r*.2)	"**add** *r*.1,*r*.2"

Assume that all integer divide instructions require that the dividend be in a double register, **d**. Then in a prefix description the state containing the item [r → / • d r] will remain on the stack until after the next symbol has been converted into a **d**. Once the first operand has been computed, that operand will be changed into a **d** and the coder will be able to continue. In a postfix description, however, the states containing the item [r → • d r /] will also contain all items of the form [r → • α], including [r → • r r +]. Again assume that the first operand to a divide operator has been computed and is in a register, **r**. The coder will then **shift** to a state containing [r → r • r +] but not [r → d • r /] since it has not seen the operator and does not know to convert the **r** into a **d**. After computing the second operand in another register, the coder will move to a state containing [r → r r • +] but not [r → d r • /] and, in the case of a divide, would block on the next input symbol, /.

Allowing the code generator to back up and fix single symbols will solve the problem for the previous example, but consider the following instruction set:

r.1 ::= (+ *r*.1 *r*.2)	"**add** *r*.1,*r*.2"
r.1 ::= (+ ↑ *k*.1 *r*.1)	"**add** *r*.1,*k*.1"
r.1 ::= (+ *r*.1 ↑ *k*.1)	"**add** *r*.1,*k*.1"
r.1 ::= (− *r*.1 *r*.2)	"**sub** *r*.1,*r*.2"
r.1 ::= (− *r*.1 ↑ *k*.1)	"**sub** *r*.1,*k*.1"
r.1 ::= (↑ *k*.1)	"**load** *r*.1,*k*.1"

If the prefix code generator is generating code for the expression '− ↑ *k r*', once it has read the − it will be in a state containing the items:

q: [r → − . r r]
[r → − . r ↑ k]
[r → . ↑ k]
. . .
[r → . α]

After performing a **shift** on ↑, it will be in a state without any item with the dot after a −. Therefore, it must reduce the expression '↑ *k*' to '*r*' before it can extend an item in the core of q. Thus the code generator 'knows' that it must always load the first operand to a subtract instruction into a register before it can issue that instruction. In the postfix case, the IR expression would be '*k* ↑ *r* −' and the coder would shift over the string '*k* ↑ *r*' and block in the state:

q′: [r → k ↑ r . +]
[r → r . r −]
[r → r . k ↑ −]
[r → r . α]
. . .
[r → . β]

Thus, in a postfix representation the code generator would have to have the ability to back up and fix expressions once the operator was encountered. Basically, a postfix parser is optimistically shifting on any input for which there is an operator that can follow, and must correct any misjudgments that it makes; a prefix parser is only shifting over input for which it knows that any valid input can be accepted, and cannot get into such trouble. Therefore, prefix expressions are used to describe instruction sets. It is unfortunate that computers do not treat all operands to all operators uniformly, for then this problem would disappear.

6.2. Areas for Future Research

The most obvious extension of this research is to complete the implementation of a total compiler using the existing preprocessor and code generator. In fact, plans are currently under way to implement an APT to IR translator and use the existing PASCAL interpreter's front end, resulting in a machine independent PASCAL compiler. The code generator and preprocessor can then be bootstrapped into executable rather than interpretable code, as they are written in PASCAL. The front end of the PASCAL interpreter is written in C, and would have to be rewritten in PASCAL in order to successfully bootstrap the entire compiler onto another computer.

There are numerous related areas that should prove fruitful for further research. Several topics concern adaptions or extensions of the machine independent code generation algorithm to other requirements, and others are in the area of instruction set design and computer

architecture. One obvious extension of the algorithm to cover existing computers would be to consider arithmetic stack computers [Burroughs64] [HP73]. Stack code can probably be generated by extending the semantics of a **shift** move to include ordering information, i.e. to keep track of which operand to each operator is first placed on the stack. This modification would still allow the code generator to handle multiple operator instructions, but would retain the information necessary to substitute a reverse subtract instruction for a standard subtract instruction when needed. Another extension to the algorithm would be to allow the machine description to force strict ordering on the evaluation of operands to certain operators. Such ordering can probably be achieved by simply ignoring all instruction patterns that allow the left (or right) operand to be deferred in evaluation, by not including items with the left operand to such an operator in the set used in the closure operation during parse table construction. The constructor would be allowed to defer operations only if all operands were run-time constant expressions (such as base + offset computations) so that no future computation could alter that value prior to its use.

With the increasing number of user microprogrammable computers, the possibility of conducting experiments in instruction set design are within the reach of many users. The machine independent code generator is just the tool that will allow the experimenter to adapt a compiler to new instruction sets with ease, thus avoiding the rewriting of all software. The code generator also allows special instructions to be added to specific computers to handle special circumstances. The code generator will be able to incorporate new instructions with minimal effort.

References

[Aho72a] Aho, A.V., and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling*, Volume 1,2. Prentice-Hall, Englewood Cliffs, N.J. (1972).

[Aho72b] Aho, A.V., and Ullman, J.D., 'Optimization of Straight Line Code,' *Siam J. Computing*, **1**:1 (1972), 1-19.

[Aho75] Aho, A.V., Johnson, S.C., and Ullman, J.D., 'Deterministic Parsing of Ambiguous Grammars,' *Comm. ACM*, **18**:**8**, (August 1975), 441-452.

[Aho76] Aho, A.V. and Johnson, S.C., 'Optimal Code Generation for Expression Trees,' *Journal ACM*, **23**:3 (July 1976), 488-501.

[Aho77a] Aho, A.V., Johnson, S.C., and Ullman, J.D., 'Code Generation for Expressions with Common Subexpressions,' *Journal ACM*, **24**:**1** (January 1977), 146-161.

[Aho77b] Aho, A.V., and Ullman, J.D., *Principles of Compiler Design*, Addison Wesley, Reading, Massachusetts, (1977).

[Allen75] Allen, F.E., 'Interprocedural Analysis and the Information Derived by It,' *Programming Methodology, Lecture Notes in Computer Science*, **23**, Springer-Verlag, New York, (1975), 291-321.

[Ammann77] Ammann, U., 'On Code Generation in a PASCAL Compiler,' *Software — Practice and Experience*, **7**:3, (June-July 1977), 391-423.

[Barth77] Barth, J.M., *A Practical Interprocedural Data Flow Analysis and its Applications*, Ph. D. dissertation, Department of Computer Science, University of California, Berkeley, (May 1977).

[Bauer68] Bauer, H., Becker, S., and Graham, S., 'ALGOL-W Implementation,' *Technical Report CS-98, Computer Science Department, Stanford University*, Palo Alto, California, (May 1968).

[Bratman61] Bratman, H., 'An Alternate Form of the Uncol Diagram,' *Comm. ACM*, **4**:3, (March 1961), 142.

[Bron76] Bron, C., and DeVries, W., 'A PASCAL Compiler for PDP−11 Minicomputers,' *Software — Practice and Exprience*, **6**:**1**, (January-March 1976), 109-116.

[Burroughs64] *Burroughs B5500 Extended Algol Reference Manual*, Burroughs Corporation, Detroit, Michigan, (1964).

[Carter75] Carter, J.L., 'A Case Study of a New Compiling Code Generation Technique,' *IBM Research Report RC5666*, T. J. Watson Research Center, Yorktown Heights, N.Y., (1975).

[Donegan73] Donegan, M.K., *An Approach to the Automatic Generation of Code Generators*, Ph. D. dissertation, Rice University, Houston, Texas, (May 1973).

[Elson70] Elson, M., and Rake, S.T., 'Code Generation Techniques for Large-Language Compilers,' *IBM Systems Journal*, **9**:3, (1970), 166-188.

[Glanville76] Glanville, R.S., *Design and Implementation of Portable Compilers*, M.S. report, Computer Science Report UCB-CS-76-46, University of California, Berkeley, (June 1976).

[Graham76] Graham, S.L. and Wegman, M., 'A Fast and Usually Linear Algorithm for Global Flow Analysis,' *Journal Acm*, **23**:**1**, (January 1976), 172-202.

[Gries71] Gries, D., *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, (1971).

[Grosse76] Groose-Lindemann, C.O., and Nagel, H.H., 'Postlude to a PASCAL-Compiler Bootstrap on a DEC System-10,' *Software — Practice and Experience*, **6:1**, (January-March 1976), 29-42.

[Harrison77] Harrison, W., 'A New Strategy for Code Generation — the General Purpose Optimizing Compiler,' *Fourth ACM Symposium on Principles of Programming Languages*, Los Angales, California, (January 1977), 29-37.

[HP73] *HP3000 Reference Manual*, Hewlett-Packard Company, Palo Alto, California, (1973).

[IBM65] *IBM System/360 Operating System: PL/I Languae Specifications*, IBM Corporation Manual (GC28-6571), Data Processing Division, White Plains, New York, (1965).

[IBM66] *IBM System/360 Principles of Operation*, IBM Corporation Manual (A22-6821-3), Poughkeepsie, N.Y., (1966).

[Jensen74] Jensen, K., and Wirth, N., 'PASCAL User Manual and Report,' in *Lecture Notes in Computer Science*, **18**, Springer-Verlag, New York, (1974).

[Joy77] Joy, Wm.N., Graham, S.L., and Haley, C.B., *Unix PASCAL User's Manual*, Computer Science Department, University of California, Berkeley, California, (September 1977).

[Knuth68] Knuth, D.E., *The Art of Computer Programming: Vol. 1: Fundamental Algorithms*, Addison Wesley, Reading Massachusetts, (1968).

[Lancaster76] Lancaster, R.L., and Schneider, V.B., 'Quick Compiler Construction using Uniform Code Generators,' *Software — Practice and Experience*, **6:1**, (January-March 1976), 83-91.

[Loveman76] Loveman, D.B., 'Program Improvement by Source to Source Transformation,' *Third ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia, (January 1976), 140-152.

[Lowry69] Lowry, E.S., and Medlock, C.W., 'Object Code Optimization,' *Comm. ACM*, **12:1**, (January 1969), 13-22.

[Maltz77] Maltz, I.J., *Implementation of a Code Generator Preprocessor*, M.S. report, Computer Science Division, EECS, University of California, Berkeley, (August 1977).

[McKeeman65] McKeeman, W.M., 'Peephole Optimization,' *Comm. ACM*, **8:7**, (July 1965), 443-444.

[McKeeman70] McKeeman, W.M., Horning, J.J., and Wortman, D.B., *A Compiler Generator*, Prentice Hall, (1970).

[Meyers74] Meyers, W.J., 'Linear Representations of Tree Structure: A Mathematical Theory of Parenthesis-Free Notations,' *Technical Report STAN-CS-74-222, Computer Science Department, Stanford University*, Palo Alto, California, (July 1974).

[Miller71] Miller, P.L., 'Automatic Creation of a Code Generator from a Machine Description,' *Technical Report MAC TR-85, Project MAC, MIT*, Cambridge, Mass., (May 1971).

[Newcomer75] Newcomer, J.M., *Machine-Independent Generation of Optimal Local Code*, Ph. D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, (May 1975).

[Nori74] Nori, K.V., Ammann, U., Jensen, K., and Nageli, H.H., 'The PASCAL (P) Compiler: Implementation Notes,' *Berichte der Fachgruppe Computer-Wissenschaften, Eidgenössische Technische Hochschule*, Zurich, **10**, (December 1974).

[Pasko73] Pasko, H.J., 'A Pseudo-Machine for Code Generation,' *Technical Report CSRG-30, University of Toronto*, Ontario, Canada, (December 1973).

[Poole74] Poole, P.C., 'Portable and Adaptable Compilers, Compiler Construction, An Advanced Course,' Ed. G. Goos and J. Hartmanis, in *Lecture Notes in Computer Science*, **21**, Springer-Verlag, New York, (1974), 427-497.

[Richmond74] Richmond, G.H, Editor, *PASCAL Newsletter*, **2**, (May 1974).

[Ritchie76] Ritchie, D.M., 'A Tour Through the UNIX C Compiler,' *unpublished internal memorendum, Bell Laboratories*, Murray Hill, New Jersey, (1976).

[Schneck73] Schneck, P.B., 'A FORTRAN to FORTRAN Optimizing Compiler,' *Computer J.*, **16**:4, (1973), 322-330.

[Steel61] Steel, T.B., Jr., 'A First Version of UNCOL,' *Proceedings WJCC*, **19**, (1961), 371-378.

[Strong58] Strong, J., *et al.*, 'The Problem of Programming Communication with Changing Machines: A Proposed Solution,' *Comm. ACM*, **1**:**8**, (August 1958), 12-18.

[Szymanski78] Szymanski, T.G., 'Assembling Code for Machines with Span-Dependent Instructions,' *Comm. ACM*, to appear.

[Weingart73] Weingart, S.W., *An Efficient and Systematic Method of Compiler Code Generation*, Ph. D. dissertation, Yale University, New Haven, Conn., (June 1973).

[Welsh72] Welsh, J., and Quinn, C., 'A PASCAL Compiler for ICL 1900 Series Computer,' *Software — Practice and Experience*, **2**:**1**, (January-March 1972), 73-77.

[Wilcox71] Wilcox, T.R., *Generating Machine Code for High-Level Programming Languages*, Ph. D. dissertation, Technical Report 71-103, Department of Computer Science, Cornell University, Ithaca, N.Y., (September 1971).

[Wirth68] Wirth, N., 'PL360, A Programming Language for the 360 Computers,' *Journal ACM*, **15**:**1**, (January 1968), 37-74.

[Wirth71] Wirth, N., 'The Design of a PASCAL Compiler,' *Software — Practice and Experience*, **1**:3, (1971), 309-333.

[Wirth72] Wirth, N., 'On "PASCAL", Code Generation, and the CDC 6000 Computer,' *Technical Report STAN-CS-72-257, Computer Science Department, Stanford University*, Palo Alto, California, (February 1972).

[Wulf71] Wulf, W.A., Russel, D.B., and Habermann, A.N., 'BLISS: A Language for Systems Programming,' *Comm. ACM*, **14**:**12**, (December 1971), 780-790.

[Wulf75] Wulf, W., *et al.*, *The Design of an Optimizing Compiler*, American Elsevier Publishing Co., Inc., New York, (1975).

Appendix A: PDP-11 TMDL Description

```
$options statesets,tables,loops,items;

$registers
    $allocatable {r0,r1,r2,r3,r4,cc};
    $dedicated   {r5,sp,pc};

$symbols

    $variables
        r = r0,r1,r2,r3,r4,r5,sp,pc;
        d = <r0,r1>,<r2,r3>;
        o = r1,r3;
        e = r0,r2;
        c = cc;

    $terminals
        k: 0,32767; l: 0,1023;
        + binary; − binary; * binary; / binary; ↑ unary; := binary;
        j unary; : unary; ? binary;
        < binary; > binary; ⩽ binary; ⩾ binary; = binary; ≠ binary;
        & binary; | binary; ! unary; m unary;

$instructions

r.1 ::= (k.1)                          "mov   $k.1,r.1";
r.1 ::= (↑ k.1)                        "mov   *k.1,r.1";
λ ::= (:= k.1 r.1)                     "mov   r.1,*k.1";
λ ::= (:= k.1 ↑ k.2)                   "mov   *k.2,*k.1";
r.2 ::= (↑ + k.1 r.1)                  "mov   k.1(r.1),r.2";
λ ::= (:= + k.1 r.1 r.2)               "mov   r.2,k.1(r.1)";
r.2 ::= (↑ r.1)                        "mov   (r.1),r.2";
λ ::= (:= r.1 r.2)                     "mov   r.2,(r.1)";
λ ::= (:= r.1 ↑ r.2)                   "mov   (r.2),(r.1)";
r.2 ::= (↑ ↑ + k.1 r.1)                "mov   *k.1(r.1),r.2";
λ ::= (:= k.1 k.2)                     "mov   $k.2,*k.1";
λ ::= (:= + k.1 r.1 k.2)               "mov   $k.2,k.1(r.1)";
λ ::= (:= ↑ + k.1 r.1 k.2)             "mov   $k.2,*k.1(r.1)";
λ ::= (:= r.1 k.1)                     "mov   $k.1,(r.1)";
λ ::= (:= r.1 ↑ + k.2 r.2)             "mov   k.2(r.2),(r.1)";
λ ::= (:= r.1 ↑ k.1)                   "mov   *k.1,(r.1)";
λ ::= (:= + k.1 r.1 λ+ k.2 r.2)        "mov   k.2(r.2),k.1(r.1)";

r.1 ::= (+ r.1 r.2)                    "add   r.2,r.1";
r.2 ::= (+ r.1 r.2)                    "add   r.1,r.2";
r.1 ::= (+ k.1 r.1)                    "add   $k.1,r.1";
r.1 ::= (+ r.1 k.1)                    "add   $k.1,r.1";
r.1 ::= (+ ↑ k.1 r.1)                  "add   *k.1,r.1";
r.1 ::= (+ r.1 ↑ k.1)                  "add   *k.1,r.1";
λ ::= (:= k.1 + ↑ k.1 r.1)             "add   r.1,*k.1";
λ ::= (:= k.1 + r.1 ↑ k.1)             "add   r.1,*k.1";
```

```
r.2 ::= (+ ↑ + k.1 r.1 r.2)                        "add    k.1(r.1),r.2";
r.1 ::= (+ r.1 ↑ + k.2 r.2)                        "add    k.2(r.2),r.1";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 r.2)             "add    r.2,k.1(r.1)";
λ ::= (:= + k.1 r.1 + r.2 ↑ + k.1 r.1)             "add    r.2,k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 ↑ + k.2 r.2)     "add    k.2(r.2),k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.2 r.2 ↑ + k.1 r.1)     "add    k.2(r.2),k.1(r.1)";
λ ::= (:= + k.1 r.1 + k.2 ↑ + k.1 r.1)             "add    $k.2,k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 k.2)             "add    $k.2,k.1(r.1)";

r.1 ::= (− r.1 r.2)                                "sub    r.2,r.1";
r.1 ::= (− r.1 k.1)                                "sub    $k.1,r.1";
r.1 ::= (− r.1 ↑ k.1 )                             "sub    *k.1,r.1";
λ ::= (:= k.1 − ↑ k.1 r.1)                         "sub    r.1,*k.1";
r.1 ::= (− r.1 ↑ + k.2 r.2)                        "sub    k.2(r.2),r.1";
λ ::= (:= + k.1 r.1 − ↑ + k.1 r.1 r.2)             "sub    r.2,k.1(r.1)";
λ ::= (:= + k.1 r.1 − ↑ + k.1 r.1 ↑ + k.2 r.2)     "sub    k.2(r.2),k.1(r.1)";
λ ::= (:= + k.1 r.1 − ↑ + k.1 r.1 k.2)             "sub    $k.2,k.1(r.1)";

r.1 ::= (k=0)                                      "clr    r.1";
λ ::= (:= r.1 k=0)                                 "clr    (r.1)";
λ ::= (:= k.1 k=0)                                 "clr    *k.1";
λ ::= (:= + k.1 r.1 k=0)                           "clr    k.1(r.1)";

r.1 ::= (+ r.1 k=1)                                "inc    r.1";
r.1 ::= (+ k=1 r.1)                                "inc    r.1";
λ ::= (:= k.1 + ↑ k.1 k=1)                         "inc    *k.1";
λ ::= (:= k.1 + k=1 ↑ k.1)                         "inc    *k.1";
λ ::= (:= + k.1 r.1 + k=1 ↑ + k.1 r.1)             "inc    k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 k=1)             "inc    k.1(r.1)";

r.1 ::= (− r.1 k=1)                                "dec    r.1";
λ ::= (:= k.1 − ↑ k.1 k=1)                         "dec    *k.1";
λ ::= (:= + k.1 r.1 − ↑ + k.1 r.1 k=1)             "dec    k.1(r.1)";

e=0 ::= (r=0)                                      "";
o=0 ::= (r=1)                                      "";
e=1 ::= (r=2)                                      "";
o=1 ::= (r=3)                                      "";
r=0 ::= (e=0)                                      "";
r=1 ::= (o=0)                                      "";
r=2 ::= (e=1)                                      "";
r=3 ::= (o=1)                                      "";
r=0 ::= (d=0)                                      "";
r=2 ::= (d=1)                                      "";

e.1 ::= (r.1)                                      "mov    r.1,e.1";
o.1 ::= (r.1)                                      "mov    r.1,o.1";
d.1 ::= (r.1)                                      "mov    r.1,d.1.2; sxt d.1.1";

d.1 ::= (* e.1 r.1)                                "mul    r.1,e.1";
d.1 ::= (* r.1 e.1)                                "mul    r.1,e.1";
o.1 ::= (* o.1 r.1)                                "mul    r.1,o.1";
o.1 ::= (* r.1 o.1)                                "mul    r.1,o.1";
d.1 ::= (* e.1 k.1)                                "mul    $k.1,e.1";
```

```
d.1 ::= (* k.1 e.1)                 "mul   $k.1,e.1";
o.1 ::= (* o.1 k.1)                 "mul   $k.1,o.1";
o.1 ::= (* k.1 o.1)                 "mul   $k.1,o.1";
o.1 ::= (* o.1 ↑ r.1)               "mul   (r.1),o.1";
o.1 ::= (* ↑ r.1 o.1)               "mul   (r.1),o.1";
e.1 ::= (* e.1 ↑ r.1)               "mul   (r.1),e.1";
e.1 ::= (* ↑ r.1 e.1)               "mul   (r.1),e.1";
d.1 ::= (* e.1 ↑ + k.1 r.1)         "mul   k.1(r.1),e.1";
d.1 ::= (* ↑ + k.1 r.1 e.1)         "mul   k.1(r.1),e.1";
o.1 ::= (* o.1 ↑ + k.1 r.1)         "mul   k.1(r.1),o.1";
o.1 ::= (* ↑ + k.1 r.1 o.1)         "mul   k.1(r.1),o.1";

r.2 ::= (/ d.1 r.1)                 "div   r.1,d.1.1";
r.2 ::= (/ d.1 k.1)                 "div   $k.1,d.1.1";
r.2 ::= (/ d.1 ↑ + k.1 r.1)         "div   k.1(r.1),d.1.1";

λ ::= (: l.1)                       "Ll.1:";
c.1 ::= (? r.1 r.2)                 "cmp   r.1,r.2";
c.1 ::= (? k.1 ↑ k.2)               "cmp   $k.1,*k.2";
c.1 ::= (? ↑ k.1 k.2)               "cmp   *k.1,$k.2";
c.1 ::= (? k.1 ↑ + k.2 r.2)         "cmp   $k.1,k.2(r.2)";
c.1 ::= (? ↑ + k.1 r.1 k.2)         "cmp   k.1(r.1),$k.2";
c.1 ::= (? k.1 r.1)                 "cmp   $k.1,r.1";
c.1 ::= (? r.1 k.1)                 "cmp   r.1,$k.1";
c.1 ::= (? r.1 ↑ k.1)               "cmp   r.1,*k.1";
c.1 ::= (? ↑ k.1 r.1)               "cmp   *k.1,r.1";
c.1 ::= (? ↑ + k.1 r.1 r.2)         "cmp   k.1(r.1),r.2";
c.1 ::= (? r.1 ↑ + k.2 r.2)         "cmp   r.1,k.2(r.2)";

λ ::= (j l.1)                       "jbr   Ll.1";
λ ::= (< l.1 c.1)                   "jlt   Ll.1";
λ ::= (> l.1 c.1)                   "jgt   Ll.1";
λ ::= (= l.1 c.1)                   "jeq   Ll.1";
λ ::= (≤ l.1 c.1)                   "jle   Ll.1";
λ ::= (≥ l.1 c.1)                   "jge   Ll.1";
λ ::= (≠ l.1 c.1)                   "jne   Ll.1";

r.1 ::= (m r.1)                     "neg   r.1";
r.1 ::= (& r.1 r.2)                 "bit   r.2,r.1";
r.2 ::= (& r.1 r.2)                 "bit   r.1,r.2";
r.1 ::= (| r.1 r.2)                 "bis   r.2,r.1";
r.2 ::= (| r.1 r.2)                 "bis   r.1,r.2";
r.1 ::= (! r.1)                     "xor   $1,r.1";

$end
```

Appendix B: IBM 370 TMDL Description

```
$options statesets,tables,items,loops;

$registers
    $allocatable {cc,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12};
    $dedicated   {r0,r1,r13,r14,r15};

$symbols

    $variables
        r = r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
        d = <r2,r3>,<r4,r5>,<r6,r7>,<r8,r9>,<r10,r11>;
        e = r2,r4,r6,r8,r10;
        o = r3,r5,r7,r9,r11;
        c = cc;

    $terminals
        k: 0,4095;
        a binary; + binary; − binary; * binary; / binary; ↑ unary; := binary;
        j unary; ? binary;
        < binary; > binary; ≤ binary; ≥ binary; ≠ binary; = binary;
        & binary; | binary; ! unary; m unary;

$instructions

    r.2 ::= (↑ a k.1 r.1)                 "L    r.2,k.1(r.1)";
    r.3 ::= (↑ a a k.1 r.1 r.2)           "L    r.3,k.1(r.1,r.2)";
    r.3 ::= (↑ a k.1 a r.1 r.2)           "L    r.3,k.1(r.1,r.2)";
    r.3 ::= (↑ a r.1 a k.1 r.2)           "L    r.3,k.1(r.1,r.2)";
    r.2 ::= (↑ r.1)                       "L    r.2,0(r.1)";

    r.1 ::= (k.1)                         "LA   r.1,k.1";
    r.2 ::= (a k.1 r.1)                   "LA   r.2,k.1(r.1)";
    r.3 ::= (a a k.1 r.1 r.2)             "LA   r.3,k.1(r.1,r.2)";
    r.3 ::= (a k.1 a r.1 r.2)             "LA   r.3,k.1(r.1,r.2)";
    r.3 ::= (a r.1 a k.1 r.2)             "LA   r.3,k.1(r.1,r.2)";

    λ ::= (:= a k.1 r.1 r.2)              "ST   r.2,k.1(r.1)";
    λ ::= (:= a a k.1 r.1 r.2 r.3)        "ST   r.3,k.1(r.1,r.2)";
    λ ::= (:= a k.1 a r.1 r.2 r.3)        "ST   r.3,k.1(r.1,r.2)";
    λ ::= (:= a r.1 a k.1 r.2 r.3)        "ST   r.3,k.1(r.1,r.2)";
    λ ::= (:= r.1 r.2)                    "ST   r.2,0(r.1)";

    r.1 ::= (+ r.1 r.2)                   "AR   r.1,r.2";
    r.2 ::= (+ r.1 r.2)                   "AR   r.2,r.1";
    r.2 ::= (+ ↑ a k.1 r.1 r.2)           "A    r.2,k.1(r.1)";
    r.1 ::= (+ r.1 ↑ a k.2 r.2)           "A    r.1,k.2(r.2)";
    r.1 ::= (a r.1 r.2)                   "AR   r.1,r.2";
    r.2 ::= (a r.1 r.2)                   "AR   r.2,r.1";
    r.2 ::= (a ↑ a k.1 r.1 r.2)           "A    r.2,k.1(r.1)";
    r.1 ::= (a r.1 ↑ a k.2 r.2)           "A    r.1,k.2(r.2)";
```

```
r.1 ::= (− r.1 r.2)              "SR    r.1,r.2";
r.1 ::= (− r.1 ↑ a k.2 r.2)      "S     r.1,k.2(r.2)";

e.1 ::= (r.1)                    "LR    e.1,r.1";
r.1 ::= (e.1)                    "";
r.1 ::= (o.1)                    "";
d.1 ::= (e.1)                    "SLDL e.1,32";
d.1 ::= (o.1)                    "SRDA o.1,32";
r.1 ::= (d.1)                    "";

d.1 ::= (* e.1 r.1)              "MR    e.1,r.1";
d.1 ::= (* r.1 e.1)              "MR    e.1,r.1";
d.1 ::= (* e.1 ↑ a k.1 r.1)      "M     e.1,k.1(r.1)";
d.1 ::= (* ↑ a k.1 r.1 e.1)      "M     e.1,k.1(r.1)";

o.1 ::= (/ d.1 r.1)              "DR    d.1.1,r.1";
o.1 ::= (/ d.1 ↑ a k.1 r.1)      "D     d.1.1,k.1(r.1)";

r.1 ::= (& r.1 r.2)              "NR    r.1,r.2";
r.2 ::= (& r.1 r.2)              "NR    r.2,r.1";
r.1 ::= (& r.1 ↑ a k.2 r.2)      "N     r.1,k.2(r.2)";
r.2 ::= (& ↑ a k.1 r.1 r.2)      "N     r.2,k.1(r.1)";

r.1 ::= (| r.1 r.2)              "OR    r.1,r.2";
r.2 ::= (| r.1 r.2)              "OR    r.2,r.1";
r.1 ::= (| r.1 ↑ a k.2 r.2)      "O     r.1,k.2(r.2)";
r.2 ::= (| ↑ a k.1 r.1 r.2)      "O     r.2,k.1(r.1)";

r.2 ::= (! r.1)                  "NOT   r.2,r.1";

r.2 ::= (m r.1)                  "LCR   r.2,r.1";

λ ::= (j r.1)                    "BCR   14,r.1";
λ ::= (j a k.1 r.1)              "BC    14,k.1(r.1)";

c.1 ::= (? r.1 r.2)              "CR    r.1,r.2";
c.1 ::= (? r.1 ↑ a k.2 r.2)      "C     r.1,k.2(r.2)";

λ ::= (< r.1 c.1)                "BCR   4,r.1";
λ ::= (< a k.1 r.1 c.1)          "BC    4,k.1(r.1)";

λ ::= (> r.1 c.1)                "BCR   2,r.1";
λ ::= (> a k.1 r.1 c.1)          "BC    2,k.1(r.1)";

λ ::= (≤ r.1 c.1)                "BCR   12,r.1";
λ ::= (≤ a k.1 r.1 c.1)          "BC    12,k.1(r.1)";

λ ::= (≥ r.1 c.1)                "BCR   10,r.1";
λ ::= (≥ a k.1 r.1 c.1)          "BC    10,k.1(r.1)";

λ ::= (≠ r.1 c.1)                "BCR   6,r.1";
λ ::= (≠ a k.1 r.1 c.1)          "BC    6,k.1(r.1)";
```

```
λ ::= (= r.1 c.1)            "BCR  8,r.1";
λ ::= (= a k.1 r.1 c.1)      "BC   8,k.1(r.1)";

r.1 ::= (k=0)                "SR   r.1,r.1";

$end
```