Simplifying Code Generation Through Peephole Optimizations

Jack W. Davidson

Computer Science Technical Report TR-87-07
May 1, 1987

Simplifying Code Generation Through Peephole Optimization*

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Producing compilers that generate good object code is difficult. The early phase of the compiler, syntactical and lexical analysis, have been automated. The latter phases, code generation and optimization, are more difficult because of the wide range of machine architectures. This dissertation describes a technique for the rapid implementation of production-quality compilers through the use of a machine-independent retargetable peephole optimizer, PO. PO is retargeted by providing a description of the new machine.

PO simplifies many of the tasks associated with developing compilers. It simplifies code generation by eliminating most of the case-analysis typically necessary to produce good code. It simplifies the optimization phase by collecting several disparate optimizations and generalizing them as peephole optimizations. PO also demonstrates that traditional optimizations, such as register allocation, common subexpression elimination, and removal of unreachable code, may be done more thoroughly and completely when information about the target machine is available.

May 1, 1987

Department of Computer Science

The University of Virginia

Charlottesville, VA 22903

*This work was supported by the National Science Foundation under Grant MCS-7802545.

Table of Contents

ACKNOWLEDGEMENTS

The work described in this dissertation was supported by the National Science Foundation under Grant MCS-7802545.

I would like to thank the members of my committee, Chris Fraser, Ralph Griswold, and Dave Hanson for their support of this research. Special thanks go to Chris and Dave for their constant support and encouragement. My advisor, Chris, deserves special mention for all the time and effort he put into this work. No matter how busy he was, he always found time to discuss problems and solutions.

Finally, I would like to thank all my friends for their support, especially Audrey.

Chapter 1
Introduction

In the last few years there has been a great deal of research directed at automating compiler construction, particularly the code generation phase. There are several reasons for these efforts:

1. Because of advances in hardware design and construction, new machine architectures are being produced at a surprising rate. Unfortunately, compiler and software development has not progressed at the same rate.
2. Despite lower hardware costs and increased machine speeds, high-quality code is still desirable [WULF81].
3. Advances in programming language design have led to large, complex languages that need the support of good compilers.
4. The early phases of compiler construction, lexical analysis and syntax analysis, have been successfully automated.

Automating code generation is difficult because there are many machines each with different characteristics. There are one-, two- and three-address machines, general-register machines, stack machines, array processors, and composites of these. Even among machines of the same architectural class there are major differences. Instruction sets differ, and each architecture has its own idiosyncrasies: certain registers have restricted uses, division must be done in even-odd register pairs, some instructions set condition codes while others do not, etc. This makes it difficult to decide which instructions to use to produce the best code. In the past, code generators used extensive case analysis to generate optimal code sequences. Such case analysis is ad hoc and difficult to validate.

The goal of this research has been development of a retargetable compiler for a high-level language that produces production-quality code. This has been achieved through a retargetable peephole optimizer [McKE65] that automates most of the case analysis typically performed by code generators. As a vehicle for testing and validating the ideas of this dissertation, a compiler for a high-level programming language has been developed that can be retargeted for a new machine with two to three days of effort.

1.1 Previous Work

There are several criteria for evaluating automatic code generation techniques and portable compilers. These are:

1. Speed
2. Quality of code
3. Machine applicability
4. Implementation cost

For the code generator to be practical, it must produce code at a reasonable rate. Few users will tolerate a slow compiler. Similarly, few users tolerate inefficient code. A technique that accommodates a wide variety of architectures is better than one that handles a restricted set. Similarly, the less effort to retarget the compiler the better.

The only way code generation techniques can be accurately evaluated is through extensive use. Of the work reviewed below, only the BCPL, Pascal, and the Portable C Compiler have been extensively tested.

It is surprising that although many code generation techniques are presented as being applicable to a wide range of architectures, usually only one or two machines are presented as evidence of this.

Previous work on producing retargetable compilers and code generation falls into two categories: intermediate language approaches and table-driven approaches.

1.1.1 Intermediate Language Approach

Some of the first successful work in developing portable software used intermediate languages. These works are based on an earlier effort called UNCOL [STRO58, STEE61], an acronym for UNiversal Computer Oriented Language. In the UNCOL approach, compilers emit machine-independent UNCOL, which is translated to machine-specific object code. To retarget the compiler for a new machine, the program that translates UNCOL to object code is rewritten. This project failed because it was too difficult to represent a wide variety of architectures and languages in a single intermediate language.

Closely related to the intermediate language approach is abstract machine modeling [NEWE72]. An abstract machine along with a language that describes the basic operations and data types of the machine are developed. The language is often called an intermediate language. SIL, the SNOBOL4 Implementation Language [GRIS72], is used to implement a portable version of SNOBOL4 [GRIS71]. AIM1 [NEWE72] and STAGE2 [WAIT73] were used to implement a BASIC [DART70] interpreter. Janus [HADD78] has been used to implement a set of machine-independent mathematical software.

Intermediate languages have also been used to develop retargetable compilers. The code generator of the compiler produces code for an abstract machine. This code is then translated to assembly language for the target machine, typically through macro expansion or a similar process called code expansion. BCPL and Pascal are two typical languages for which this approach proved successful.

1.1.1.1 BCPL

BCPL [RICH71, RICH77] is a relatively simple systems programming language that has one type, the binary bit pattern. The BCPL compiler translates BCPL to OCODE, the assembly language for the BCPL abstract machine. Because there is only one type, OCODE is relatively simple and has only 56 operation codes ('opcodes').

BCPL is moved to a new machine by rewriting the code expander that translates each OCODE statement into a sequence of assembly language instructions for the target machine. This produces an inefficient version of the compiler. More sophisticated code expanders that translate OCODE into assembly language can then be written in BCPL. These new expanders can perform optimizations to produce more efficient versions of the compiler.

It requires three to five months of effort to move BCPL to a new machine if the implementor has no previous knowledge of BCPL [RICH71]. Part of the difficulty is that several versions of the OCODE to assembly language translator had to be written before an efficient version of the compiler was produced. While OCODE is well suited to executing BCPL on an abstract machine, it is not suited to real machines. Nonetheless, BCPL is a successful portable compiler and has been transferred to over ten machines.

1.1.1.2 Pascal

The Pascal-P compiler [NORI81] is a retargetable compiler for a 'standard subset' of Pascal [WIRT75]. The compiler is written in the subset of Pascal it compiles. It emits object code for a hypothetical stack machine. The assembly language for the stack machine is called PCODE. To retarget the compiler for a new machine, the code expander must be rewritten. Pascal-P takes longer to retarget than BCPL. Six months [BERR78] seems to be a realistic figure. This is probably due to Pascal having more types than BCPL which is reflected by the size of the abstract machine's instruction set. Pascal-P's PCODE has 125 opcodes (when types are included) as opposed to OCODE's 56.

There are several other retargetable Pascal compilers. These implementations are more complicated but capable of generating more efficient object code. One part of the compiler that is more complicated is the intermediate language. So that it can be efficiently mapped to the target machines, special purpose opcodes have been added to the intermediate language [NELS79]. The UCSD Pascal compiler [SHIL78], PASCALJ [HADD78], and the VU Pascal compiler [TANE80] are examples. To further increase the efficiency of the generated code, optimizers for the hypothetical stack machine's assembly language are often included as part of the distribution package [PERK79, TANE80].

1.1.1.3 MACRO SPITBOL

While BCPL and Pascal achieve portability by compiling themselves and generating an intermediate language, MACRO SPITBOL [DEWA77] takes a different approach. The compiler, the interpreter, and system routines are written in a machine-independent macro assembly language, called MINIMAL. The MACRO SPITBOL system is retargeted for a new machine by writing a translator that transforms MINIMAL statements to the target machine's assembly language.

Implementations of MACRO SPITBOL exist on a wide range of machines. The PDP-11, DEC-10, CDC Cyber series, and Burroughs B1700 are a few of the machines that have been accommodated. MACRO SPITBOL is unique in that the retargeted code can be faster than code that is tailored for the specific machine. Benchmarks of a MACRO SPITBOL implementation for the DEC-10 and SITBOL [GIMP73], a DEC-10 specific implementation of SNOBOL4, reveal that MACRO SPITBOL runs faster than SITBOL [GRIS77].

1.1.2 Table-driven Approach

Table-driven approaches to code generation modularize code generation by organizing code generation information in tabular form. The main differences in table-driven methods are how the tables are built and how the tables are used to generate code. This approach to code generation offers the same advantages as table-driven lexical and syntax analysis: ease of use, reliability, enhanced portability, and modularity.

1.1.2.1 Portable C Compiler

C [KERN78] is the systems programming language for the UNIX operating system [RITC74]. The Portable C Compiler [JOHN78] uses many of the ideas of a previous work by Snyder [SNYD74]. The goals of this work coincide with the goals of this research. These are to produce a portable compiler that is easily retargeted, produces good code, and is fast enough to be used as a production compiler.

The early phases of the compiler are mostly machine-independent and are implemented using many of the tools available with the UNIX operating system [JOHN80]. To simplify code generation and show that it is correct, Johnson abstracts the relevant features of code generation into two models. The expression tree produced by the parser models the computation to be performed, while which registers are busy might be the model of the machine's state. Code generation is simplified to choosing the proper 'transformations' to perform on the expression tree to reduce it to a single node.

A transformation is a rewriting rule for the expression tree. Transformations are organized into templates that represent the change to the expression tree, the associated change to the machine state, and instructions that implement those changes. Templates are organized in a table so that transformations can be rapidly matched to expression trees. If the machine state and the expression tree are correct, and if the semantics of the transformation to the expression tree match the semantics of the instructions emitted, the generated code must be correct.

This code generation method is both practical and flexible. The compiler has been implemented on over a dozen machines including the Honeywell 6000, IBM 370, and Interdata 8/32 [JOHN78]. It produces good code and is used as a production compiler.

The compiler is retargeted by writing templates for the new machine. Johnson reports no statistics on how long it takes to retarget the compiler. Building new templates is an ad hoc process, and the quality of the code and the speed of the code generator depend on the design of the templates.

1.1.2.2 Glanville's Work

One of the newer and more promising approaches to automatic code generation is due to Robert S. Glanville and Susan L. Graham [GLAN77, GLAN78, GRAH80]. Their technique involves a table-driven algorithm that translates a low-level intermediate representation of a program into assembly language for the target machine. The tables are constructed from a description of the target machine's instruction set.

Building the code generation tables is similar to building parse tables from a grammar. In fact, the theory of context-free parsing was used in developing the algorithm [AHO72]. As a result, the algorithm is fast and easily understood.

The code generator produces locally good code. In his dissertation, Glanville discusses code generators for two machines, the PDP-11 and the IBM 370. Although these machines are quite different, it is difficult to evaluate the ability of the algorithm to accommodate a wide variety of machines. Glanville expects to have difficulties with 'awkward' architectures such as the CDC Cyber series machines.

A more serious problem appears in the machine descriptions. Because context-free parsing techniques are used to construct the tables, it is necessary to describe each instruction with every addressing mode. For example, the PDP-11 has eight word-addressing modes. In some double-operand instructions every mode can be used in the source and destination. For a double-operand instruction such as mov there are 64 possible addressing mode combinations. This means that a full description of the PDP-11 must have 64 patterns to describe the mov instruction. Complete descriptions are large and tedious to write.

1.1.2.3 Ganapathi's Work

Ganapathi's work [GANA80] is similar to Glanville's. While Glanville uses LR grammars, Ganapathi uses attribute grammars [KNUT68]. The attributes are used to pass code generation information up the tree. This added information can be used to generate better code and perform machine-dependent optimizations.

Ganapathi produced code generators for the PDP-11 and the VAX-11/780. These machines are so similar that it is impossible to evaluate the ability of the technique to accommodate various architectures. The machine descriptions are very long and hard to understand. Ganapathi's PDP-11 description is seven pages long, whereas Glanville's is only three pages. Similarly, Ganapathi's VAX-11/780 description is 11 pages long.

Ganapathi uses the attributes to perform many machine-dependent optimizations. These optimizations are hand-coded into the semantic actions of the machine description. Consequently, the quality of the code is somewhat better than that of the methods reviewed thus far.

1.1.2.4 Production-Quality Compiler-Compiler Project

The Production-Quality Compiler-Compiler (PQCC) project [LEVE80] is somewhat more ambitious than the previously mentioned works. PQCC's goal is to automate all phases of compiler construction. The practical result would be a truly automatic compiler generation system. The project has focused on two areas of compiler writing: code optimization and code generation. Of main interest here is the work on code generation.

Code selection in the PQCC project is performed by the code generator generator developed by Cattell [CATT78, CATT79, CATT80]. Code generation is similar to the technique used in Johnson's Portable C Compiler. Templates are matched against a tree representing the program. The work differs from Johnson's in that templates are generated automatically from a machine description.

Templates are generated using heuristic search methods borrowed from artificial intelligence [ERNS69]. The heuristics cannot guarantee that the best instruction sequences will be found, or that any sequence will be found at all, but as a practical matter, the heuristics seem to be quite effective. Code generation is extremely fast; Cattell reports producing 2000 instructions per second on a DEC-10.

Many code generation issues are not dealt with by Cattell's code generator. Such issues as register allocation, register assignment, temporary assignment, and storage assignment are dealt with by other phases of the compiler. This spreads machine-dependent information throughout the compiler. Work is in progress by the PQCC group to build generators that derive this machine-dependent information from machine descriptions [LEVE80].

1.1.2.5 Fraser's Work

Fraser [FRAS77] describes a code generator generator whose organization is similar to Cattell's. Both methods use machine descriptions to generate templates. Fraser's approach differs from Cattell's in the manner of template generation.

Cattell's approach is to formalize code selection, while Fraser uses human programming knowledge to aid code selection. Each approach has strengths and weaknesses. Cattell [CATT78] notes that possibly the best approach would be a combination of formal methods along with human-knowledge information to aid code selection when formal methods fail. Fraser implemented a experimental version of his code generator generator and code generator. The method produced locally good code but was very slow.

1.1.2.6 Donegan's Work

Donegan [DONE79] describes a language for writing code generators called CGGL (pronounced sea-gull). It extends his earlier work in this area [DONE73]. This approach has elements of both table-driven and intermediate language approaches. The compiler produces an intermediate language that is translated to machine code by a code expander. The code expander is constructed from a CGGL program that describes the semantics of each intermediate language opcode for the target machine.

A finite-state automation that models code generation is constructed from a CGGL program. For instance, data movement is expressed as finite-state machine transitions. Optimal code generation involves finding the minimal path from the start state to the goal state for each intermediate language opcode.

The code expander is realized by converting the finite-state automaton into a Pascal program for generating code. Currently CGGL only handles simple architectures. Attempts to model more complex machines, such as the CDC Cyber series, result in an explosion of the number of states in the finite state automaton. Donegan makes no statements about the speed of the code generators produced from CGGL programs or how long it takes to produce a code generator.

1.1.2.7 Lamb's Work

Lamb [LAMB81] describes a set of tools that aid the construction of peephole optimizers. A peephole optimizer replaces poor code sequences in the object code with better ones. For example, one common peephole optimization is the replacement of pairs of instructions with a singleton that has the same effect. Lamb uses a pattern language to describe poor code sequences and their replacements. The language resembles assembly language with the capability to specify arbitrary patterns and conditions necessary for the replacement to occur. These patterns are converted by a translator to subroutines in the implementation language, BLISS-10 [WULF71]. The subroutines are called by a set of specialized pattern-matching routines. The optimizer is retargeted by writing patterns for the new machine.

Writing patterns is an ad hoc process. Unoptimized generated code is scanned for poor code fragments. Patterns that should be included may be omitted if they do not appear in the scanned code samples. A more critical problem is that the person who writes the patterns must ensure that patterns match only when appropriate. Failure to do so can produce incorrect code. Currently an optimizer for one machine, the VAX-11/780, has been implemented. Statistics for this machine show the optimizer to be reasonably fast. As more machines are accommodated, the technique's flexibility will become easier to evaluate.

1.2 Overview of this Research

This work contains elements of both the intermediate language and table-driven approaches. It differs, however, radically in its approach to achieving its goal of production-quality code. Most of the approaches described above use the traditional model of compilation consisting of five phases [Aho80]:

- lexical analysis
- syntax analysis
- semantic analysis
- code optimization
- code generation

Typically, the code optimization phase attempts to transform an intermediate representation of the source program into one that is faster, or smaller, or both. Code generation translates the optimized intermediate representation into target-machine code.

This research modifies the model of compilation by reversing the last two phases: code optimization and code generation. This has several effects. Code generation is simplified to a naive translation of the source-language program to machine code. The code optimization phase transforms the naive machine code produced by the code generator to production-quality machine code.

Code optimization is also simplified. All optimizations performed consist of peephole optimizations on a machine-dependent representation of the program. This research shows that many optimizations typically performed at a higher level are subsumed by a few machine-independent peephole optimizations. In addition, machine-level optimizations are shown to be more thorough than their high-level counterparts.

A block diagram of the organization of the compiler appears in Figure 1. The first phase of the compiler is machine-independent and produces intermediate code for an abstract machine. The second phase, called the code expander (back end), translates the intermediate language to target machine code. It is machine specific and must be rewritten for each new machine. The last phase optimizes the machine code and outputs assembly language. The optimizer, called PO, extends earlier work by Fraser [Fras79]. PO combines instructions by symbolically simulating pairs and using a machine description to verify that the result is valid. PO is retargeted by writing a new machine description. The compiler has been retargeted by the author for several machines in as few as three days. The compiler produces code that is as good as or better than all of the previously mentioned methods except PQCC.

1.3 Guide to the Dissertation

This dissertation is organized in much the same way as the compiler. Chapter 2 describes the intermediate language produced by the compiler. Chapter 3 introduces the register transfer notation used to describe instructions and explains how machines are described. Chapter 4 describes the phase of the compiler that translates the intermediate language to code for the target machine. Chapter 5, the bulk of the dissertation, describes the retargetable peephole optimizer, PO. Chapter 6 reviews the results of this work and outlines areas for future research.

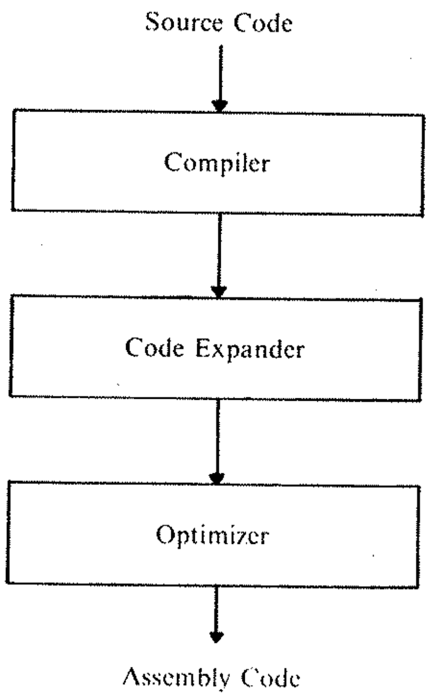

Figure 1. Compiler Organization

Chapter 2
The Intermediate Language

Machine-independence in the compiler is achieved through abstract machine modeling [Korn80, Newe72]. The abstract machine is designed to support the Y programming language [Hans81], but it is sufficiently general so it could be used to implement compilers for other languages such as C, Pascal, or Ada [Ichb79] with minor modifications.

The abstract machine interprets a simple postfix code. There are no general registers, and all operations are performed on the operands on the stack. It supports three types: integer, real, and character, and arrays of these types. There are three addressing modes: local, global, and parameter. Scalar parameters are passed by value. Arrays are passed by reference. Addresses are treated as a special type, and there is a set of operations for address calculations. Other mechanisms, such as passing arrays by value, can be built upon this foundation.

2.1 The Union Approach

Conventional abstract machines might be called 'union' machines because their instruction sets are collections of features from the target machines needed to support the target language. For example, if any target machine has three-address operations, and if it is desired that the generated code use them when possible, these three-address operations would be included in the instruction set of the abstract machine. This is necessary because it is difficult to exploit three-address target machine instructions when the abstract machine has only two-address instructions. As with most software design processes, trade-offs are made between convenience and efficiency. The result is an abstract machine that closely parallels the real machines in terms of data types, addressing modes, and types of instructions. The intermediate language is large, but it can effectively use the instruction sets of most target machines.

There are several problems with this technique. Intermediate languages that are developed using this approach tend to be unwieldy. Many machines have special-case instructions that are more efficient than their general counterparts. To make effective use of the target machine's instruction set, these instructions are often included in the intermediate language. This makes the intermediate language large, and consequently the code expander is also large.

Another problem with the union approach, which is shared with many compilers, is that the code generation phase is complicated. In order to generate code to use the abstract machine's special case instructions, the code generator or code expander must do extensive case analysis. In order to emit a clear instruction instead of the more general store instruction, for example, the code generator or expander must check if the value being stored is zero.

A third problem with the union approach is that inefficient code is produced. For example, many machines have a condition-code register that is set to reflect the result of the last operation. Few abstract machines have explicit operations on the condition-code register, making it difficult for the compiler or expander to take advantage of the condition code being set by a previous operation. For example the code fragment,

```
i = j
if (i == 0)
    ...
```

might be translated to the following intermediate code

```
store    j,i       store j in i
cmpne    i,0,l1    jump to label l1 if i is nonzero
...
llab     l1        define label l1
```

This code might be translated by an expander into the following PDP-11 assembly language.

```
        mov    j,i     move value of j to i
        cmp    i,#0    compare i to 0
        bne    l1      jump to label l1 if not equal
        ...
l1:
```

Comparing the value i to zero is unnecessary, since the condition code register is set when j is stored in i. Such situations are typically remedied by peephole optimizers [WULF75].

Finally, a fourth problem with the union approach is the difficulty of adding new machines with new features to the set of target machines. To do so may require redesigning the abstract machine so that code generators can take advantage of the new features. For example, the VAX-11/780 has instructions for manipulating queues. Several years ago such operations were rarely included in machine instruction sets. To exploit new operations, the abstract machine must be modified to include them. Because it is difficult to predict new machine features, union abstract machines are unstable.

2.2 The Intersection Approach

An alternate approach to abstract machine design is to include only the most general source-language operations required to support the language. This method might be called the 'intersection' approach. While the union approach is motivated by the source language and target machines, the intersection approach is motivated mainly by the source language.

For example, many machines have 'clear memory' instructions, but they are not included in an intersection machine's instruction repertoire. This is because there is a general way to clear a memory location—load a zero and then store it. Similarly, the abstract machine has add and subtract instructions, but not increment and decrement instructions. Consequently, intersection abstract machines have smaller instruction sets than union machines. In general, intersection intermediate languages provide only one way to generate code for any source language operation.

The generated code for the two kinds of abstract machines for the code fragment

```
i = i - 1
```

is shown below.

```
          Union Machine                         Intersection Machine
pushi i   push value of i           pushla i    push local address of i
dec       decrement i               pushla i    push local address of i
popi  i   store value back          pushi       load an integer using
          into i                                address on top of stack
                                    pushic 1    push a constant 1
      or                            subi        subtract two integers
                                    popi        store integer on top of stack in
dec   i                                         location pointed to by stack - 1
```

Constructing an abstract machine using the intersection approach has several beneficial effects. Because the intermediate language is small, the expander is small, simplifying its implementation.

The absence of special-case instructions simplifies the code generation phase of the compiler. In the example above, the union machine requires the code generator to check the operands of the subtract operation so it can emit a decrement instruction. Intersection machines do not require such case analysis.

The only detrimental effect of the intersection method is the poor quality of the generated code. If each line of abstract-machine code in the above example is translated to one line of target-machine assembly code, the union machine produces one or three instructions, the intersection machine, six. Both code fragments need peephole optimization, but the intersection example needs extensive optimization just to reduce its size to that of the union example. This optimization, however, is shown in Chapter 5 to be manageable.

2.3 Discussion

The abstract machine used in this research is an intersection machine. It has 99 opcodes. In contrast, EM-1, the union intermediate language for the VU Pascal compiler, has 219 opcodes [Tane80]. A complete list of the abstract machine instructions along with a description of their operations appears in Appendix A.

The abstract machine designed for Y supports a broad range of machine architectures. Cross-compilers that use the abstract machine exist for the PDP-11, DEC-10, the CDC CYBER series computers, and Intel's 8080 microprocessor.

Certain architectures seem better suited to the Y abstract machine than others. Byte-addressable machines with stacks are particularly easy to accommodate. The abstract machine is more difficult to map onto machines with no registers or with irregular register sets such as the Intel's 8080 or the MOS Technology's 6502, but such machines can be accommodated.

To illustrate the ease of generating code using the Y intermediate language, two examples of source code along with the intermediate code produced by the compiler are shown below.

code generated for a = b * c + 1

```
pushla  a   push address of local variable a
pushla  b   push address of local variable b
pushi       load an integer using address on top of stack
pushla  c   push address of local variable c
pushi       load an integer using address on top of stack
muli        multiply two integers
pushic  1   push a constant 1
addi        add two integers
popi        store integer on top of stack in location pointed to by stack − 1
```

code generated for if (j) i = in[j]

```
pushla  j    push address of local variable j
pushi        get value of j
pushic  0    push constant 0
cmpie   l1   compare and jump to label l1 if equal
pushla  i    push address of local variable i
pushla  j    push address of local variable j
pushi        get value of j
pushla  in   push address of local variable in
index   2    convert in and index j to an address where in's element size is 2
adda    −2   add offset of −2 to address
pushi        get array value
popi         store value from top of stack into i
llab    l1
```

Chapter 3
Machine Descriptions

PO is retargeted for a new machine by providing a new machine description. A machine description describes the syntax and effects of each target machine instruction. From the machine description, a recognizer and transducer are automatically constructed. PO uses the recognizer to check that a register transfer represents a valid target machine instruction. It uses the transducer to translate the internal representation of instructions into the target machine's assembly language.

3.1 Instruction-Set Processor

The effects of instructions are described using ISP-like notation [Bell71]. For instance, the PDP-10 instruction

```
add 3,loc
```

is expressed in ISP as

```
r[3] ← r[3] + m[loc];
```

which indicates that the contents of memory location loc is added to register 3. The register transfer

```
PC ← if r[3] > m[loc] then PC+1 else PC;†
```

represents the DEC-10 instruction

```
camg 3,loc
```

The instruction skips the next location if the contents of register 3 exceeds the contents of memory location loc. PC denotes the program counter, and is the only name that has special meaning to PO. The program counter is assumed to be incremented automatically, so this effect need not be made explicit.

Many machine instructions have multiple effects. Such instructions may contain several register transfers. For example, the DEC-10 instruction

```
aoje 5,L1
```

is expressed in ISP as

```
r[5] ← r[5] + 1; PC ← if r[5] + 1 = 0 then L1 else PC;
```

The first register transfer specifies that register 5 is incremented; the second specifies that if the result is zero a jump is made to label L1, otherwise execution continues with the next instruction. The '+ 1' is needed in the second register transfer because all register transfers are done in parallel.

Details irrelevant to the machine description may be omitted. For example, the PDP-11 instruction

```
tst r1
```

is expressed as

† The actual implementation syntax differs slightly.

```
        NZ ← r[1] ? 0;
```

where NZ represents the condition-code register. The contents of register 1 are compared to zero and the condition code is set accordingly. Code generators do not need to know how the condition code represents the result of comparisons, so the semantics of the '?' operator need not be specified. The DEC-10 ibp instruction, which increments a byte pointer, performs several complicated operations. Various fields in the byte pointer are manipulated, based on the size of the byte. Such detail is irrelevant to PO, which need know only that a memory location is being modified in the same way other byte pointer instructions manipulate them. Thus,

```
        ibp loc
```

is expressed as

```
        m[loc] ← ibp(m[loc]);
```

which specifies that the byte pointer at memory location loc is incremented. Similarly, the 'load byte' instruction

```
        ldb 3,loc
```

is expressed as

```
        r[3] ← ldb(m[loc])
```

From the ISP for these two instructions, it is easy to see how the ildb instruction (increment and load byte) should be expressed. Composing the register transfers yields

```
        r[3] ← ldb(ibp(m[loc]))
```

which describes the instruction

```
        ildb 3,loc
```

3.2 Machine Descriptions

A machine description is a grammar for syntax-directed translation between ISP register transfers and the machine's assembly language. Because PO is retargeted for a new machine by rewriting the machine description, machine descriptions have been kept simple to write. Typically, a machine description takes about a day to write by someone familiar with the target machine.

Awkward machine features can be disguised by the machine description. Many microprocessors have addressing modes that can only reach nearby words. These addressing modes are valuable because they are faster and take less space than their general counterparts. Assemblers often offer pseudo-instructions so the user does not have to decide which addressing variant is appropriate. Machine descriptions can do the same thing by allowing all addressing modes to any address. The machine description or the assembler can translate them to the appropriate addressing mode depending on the distance of the instruction to the source or destination. Similarly, many microprocessors have conditional branches that can only reach nearby locations. The machine description can disguise this feature by allowing conditional branches to arbitrary targets and let the assembler translate them to two-instruction sequences as necessary.

The machine description need not include every instruction in the machine's instruction set. Instructions that PO has little chance of replacing (e.g. subroutine jump instructions, block moves, and I/O instructions) may be omitted. Instructions that are never generated may also be omitted. Experience, however, has shown that it is difficult to predict the transformations PO will make on the generated code. PO can do its best job when it knows all instructions except those it has little chance of replacing.

A machine description is divided logically into two parts. The first part describes the machine's addressing modes, and the second part describes the instructions. This division provides a natural way to describe machines. The addressing modes are described without regard for the instructions that use them. The machine operations and the addressing modes are combined to describe the machine's instruction set.

Structuring machine descriptions in this way produces concise, understandable machine descriptions. Machine descriptions usually comprise two to four pages.

In this work, the recognizer and transducer are realized by transforming the machine description into a grammar for Lex [LESK79], a lexical-analyzer generator. From its input, Lex generates subroutines that implement the recognizer and transducer. Because Lex recognizes only regular expressions, the machine must be described using regular expressions. Most machines, however, have context sensitive components in their instructions. For instance the DEC-10, add instruction can be described as

```
    RG ← RG + M;
```

where RG denotes a register and M denotes a memory location. When a nonterminal appears twice in a production, the strings matched by the patterns must be identical. In the above example, both instances of RG must match the same string.

At times it is necessary to override the context sensitivity checking performed by the recognizer. This can be done by defining several instances of the same pattern each with a different name. For example, the description of the CDC Cyber instruction for moving one X register to another is

```
    x[RN1] ← x[RN2];
```

Although RN1 and RN2 (register numbers) represent the same pattern, they need not match identical strings.

3.3 A Sample Machine Description

Portions of the machine description for the PDP-11 are given below. Following each part of the description is a explanation of the description. To minimize detail, only a portion of the machine description is given.

```
RN        [0−7]+
XDENT     ((("_"|"L")[A−Za−z0−9_]+)|(−?[0−9]+)
IDENT     XDENT(" "[−+]" "XDENT)*
LABEL     "L"[L0−9]+
```

This part of the description defines regular expressions used throughout the description of the addressing modes. The first definition describes a register number: a sequence of one or more digits between 0 and 7, i.e. an octal number. The second definition describes a component of an identifier: an underscore or a capital L followed by one or more letters, digits, or underscores, or a number preceded by an optional negative sign. IDENT defines an identifier: an XDENT followed by zero or more occurrences of the operators + or − followed by a XDENT. The last definition is for a label: a capital L followed by a sequence of one or more L's or digits. A description of the addressing modes follows.

```
REG     :=  r[RN]                    :=  rRN

LB      :=  LABEL                    :=  LABEL

        :=  0                        :=  $0

        :=  1                        :=  $1

ID      :=  IDENT                    :=  $IDENT

WORD    :=  m[IDENT]                 :=  IDENT
        :=  m[r[RN] + IDENT]         :=  IDENT(rRN)
        :=  m[r[RN]++]               :=  (rRN)+
        :=  m[--r[RN]]               :=  -(rRN)
        :=  m[r[RN]]                 :=  (rRN)
        :=  m[m[r[RN]++]]            :=  *(rRN)+
        :=  m[m[--r[RN]]]            :=  *-(rRN)
        :=  m[m[r[RN] + IDENT]]      :=  *IDENT(rRN)
        :=  m[m[IDENT]]              :=  *IDENT
        :=  m[m[r[RN]]]              :=  *(rRN)

REL     :=  ==                       :=  eq
        :=  !=                       :=  ne
        :=  >=                       :=  ge
        :=  <=                       :=  le
        :=  <                        :=  lt
        :=  >                        :=  gt

SO      :=  <<
```

Each line of the definition has three fields: the token returned by the recognizer if the register transfer pattern is matched, the ISP syntax for the pattern, and the assembler syntax for the pattern. For instance, the first line defines a register: r followed by the register number in brackets. The corresponding assembler syntax is r followed by the register number. When a register is recognized, the token RG is returned.

If the first field is empty, the token returned is the string matched, as shown in the third line of the address definitions. In this case, a 0 is returned when a 0 is recognized. The assembler syntax field is also optional, as illustrated by the definition of SO, the shift operator. There is no corresponding assembler syntax so the field is omitted.

In the addressing modes definition, it is necessary to distinquish between a 0, a 1, and an ID. The pattern for ID can be written so that it does not match a 0 or a 1, but that complicates the pattern. To make things easier, the Lex rules for resolving ambiguous expressions are used.

1. The longest match is preferred.
2. Among rules that match the same number of characters, the rule given first is preferred.

Because the rules for 0 and 1 appear first, their tokens are returned instead of the token ID. Similarly, rule 1 ensures that the token SO is returned when the characters << are encountered instead of two occurrences of REL.

The final part of the address definitions groups the previously described tokens and single characters into classes:

```
RG1   :=  RG
RG2   :=  RG
DSTW  :=  REG | WORD
SRCW  :=  REG | ID | WORD | 0 | 1
```

RG1 and RG2 are two instances of registers. DSTW is a register or a word, and SRCW is a register, an identifier, a word, a 0, or a 1. Only tokens and single characters can appear on the right-hand side of these definitions. These groupings allow concise definition of instructions by combining similar addressing modes and operators into single groups.

The syntax for instruction definition is

instruction expression := *instruction definition*

An *instruction definition* is:

```
action | {
        [ test condition: ] action
        [ test condition: ] action
        ...
        }
```

where brackets indicate an optional field and | separate alternatives. Ellipses (...) indicate indefinite repetition of items. Italicized words are nonterminals defined in the following paragraphs.

An *instruction expression* is the ISP representation of the instruction. *Action*'s are executed if their associated *test condition* is true. *Test conditions* are evaluated in the order they appear. A series of test conditions can be thought of as an else-if chain or a case expression. If a test condition is omitted, the associated action is always executed.

The simplest type of instruction definition is one that has no test conditions and only one action. The action is the assembly language to output if the instruction expression is matched. An instruction expression matches the input string if it performs all register transfers requested, and the rest of its register transfers perform harmless functions. An example is the description of the PDP-11 compare instruction, cmp.

```
NZ ← DSTW ? SRCW;   :=   cmp  DSTW,SRCW
```

Within test conditions it is possible to reference substrings of the input string that matched components of the instruction expression. For instance, the following machine description fragment describes the PDP-11 instructions asr and ash.

```
RG ← RG SO SRCW;NZ ← RG SO SRCW ? 0;   :=   {
  !strcmp(SRCW, "−1"): asr RG
  ash SRCW,RG
  }
```

In this example, the test condition allows one *instruction expression* to identify two instructions. If the shift count is −1, then the asr instruction is matched, otherwise the more general ash instruction is matched.

Test conditions may call system-supplied and user-supplied procedures. An example is strcmp, which is used above to compare two strings; it returns true if they are equal, and false otherwise. The definition of the multiply instruction, mul, uses several different user-supplied routines:

```
RG ← RG * SRCW;NZ ← RG * SRCW ? 0;   :=   {
  !odd(RG) || !ispwr2(SRCW): ABORT
  !strcmp(SRCW, "2"): asl RG
  pwr2(SRCW): ash SRCW,RG
  mul SRCW,RG
  }
```

Odd returns true if its argument is an odd-numbered register – 16-bit multiplication on the PDP-11 must be done in an odd-numbered register. Ispwr2 returns true if its argument is a power of two. Pwr2 returns true if its argument is a power of two, and as a side effect converts its argument to the $\log_2$ of itself. It is used to determine whether the faster shift operation can be used instead of a multiply.

The special action ABORT indicates that although the input matched the instruction expression, it cannot possibly be a valid instruction, and that the recognition process should be terminated immediately. An ABORT action must be the first action in an instruction definition, as shown in the description of the multiply instruction. For example, 16-bit multiplication must be performed in an odd-numbered register, but if the multiplier is a power of two, it can be performed by a shift operation in any register. If neither of these two conditions is met, the input is not a valid instruction and is rejected.

Machine descriptions are converted to input suitable for Lex by a machine description processor written in SNOBOL4 [Gris71]. Using regular expressions to describe machine instructions and Lex to construct a recognizer and transducer is one successful approach to the problem of recognizing legal instructions. A previous implementation [Davi80] used SNOBOL4 patterns and pattern matching to recognize legal instructions. It was much too slow for production use. The speed of the optimizer hinges on the recognition and translation process. Developing new notations for describing machine instruction sets and concomitant methods for recognizing machine instructions is an area of further research.

3.4 Comparison with Other Work

Several of the works discussed in Chapter 1 use machine descriptions to achieve machine independence. Since each method depends heavily on the descriptions, it is worthwhile to compare and contrast the various description methods. The PDP-11 add instruction is used as a basis of comparison.

3.4.1 Glanville's Work

Glanville's description of the add instruction is:

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

r.1 and r.2 represent registers; k.1 and k.2 represent constants. ↑ is a unary operator that retrieves the value of the memory location addressed by its operand. The assembly language instruction to output if the production matches the input appears to the right of the production.

3.4.2 Ganapathi's Work

Ganapathi's description of the add instruction is:

```
Word↑ → + Word↑a Word↑r Istemp(↓r) EMIT (↓'add' ↓a ↓r)
      → + Word↑a Word↑b GETTEMP(↓'word' ↑r)
                 EMIT (↓'mov' ↓b ↓r)
                 EMIT (↓'add' ↓a ↓r)
```

↑ denotes a synthetic attribute that passes information up the parse tree. ↓ denotes an inherited attribute, and it passes information down the parse tree. The variables a, b, and r are attribute variables. Word represents the word-addressing modes available on the PDP-11. *Istemp* is a disambiguating predicate that determines whether the current production can apply. *Istemp* is passed the attributes of the destination of the addition and verifies that it can be destroyed by the addition. GETTMP allocates a temporary, and EMIT outputs the assembly language instruction if the production matches the input.

3.4.3 Cattell's Work

Cattell's description of the add instruction is:

```
(; (← $1:DST (+ $1:DST $2:SRC)) (← %N (LSS (+ $1:DST $2:SRC) 0))
  (← %Z (EQL (+ $1:DST $2:SRC) 0))
```

DST and SRC represent the word-addressing modes available on the PDP-11. They are described in a separate part of the description.

3.4.4 Comparison

The description used in this research is:

```
DSTW ← DSTW + SRCW;NZ ← DSTW + SRCW ? 0;   :=   add    SRCW,DSTW
```

DSTW and SRCW represent the word-addressing modes available on the PDP-11. NZ represents the condition code register. The assembly language instruction to output if the instruction expression matches the input follows the ':='.

This description is somewhat easier to read than the others because it uses infix notation instead of prefix. Ganapathi's description requires that the user implement the disambiguating predicates and actions for each machine. While this gives this method flexibility in performing machine-dependent optimizations, it complicates writing descriptions.

Glanville's description is substantially longer than the others because the addressing modes have not been factored out of the instruction definition.

Both Glanville's and Ganapathi's descriptions are incomplete in that the setting of the condition code is not described. In Glanville's case, this means that code cannot be generated that performs an addition solely for the side effect of setting the condition code. Ganapathi has a separate mechanism that scans for instructions that are only used to set the condition code. If it sets the condition code exactly as the preceding instruction, the instruction is suppressed. Since his description of the add instruction does not specify that the instruction sets the condition code, this information must come from some other source.

The description method used in this research provides a simple, yet robust method of describing machine instructions. Uninteresting details of the machine architecture can be suppressed, while complicated but necessary details can easily be described. Machine descriptions for the PDP-11, PDP-10, CDC Cyber series machines, and the 8080 appear in Appendices B, C, D, and E, respectively.

Chapter 4
The Code Expander

The code expander translates the intermediate language produced by the code generator into register transfer lists ('RTLs') for the target machine. Because the code expander must be rewritten to retarget the compiler for a new machine, it has been kept very simple. Nevertheless, it still translates the intermediate language into RTLs in a way that supports the final goal—production-quality code.

The remainder of this chapter describes rules and guidelines that have been adopted to simplify the code expander and provide input to PO that results in high-quality code. A rule must be rigorously observed. Failure to observe a rule can result in generation of erroneous code. Guidelines are suggestions and hints from an experienced user. They can be ignored, but code quality may suffer.

4.1 Form Rules

The expander produces a series of records as output, some of which can be ignored by PO. Each record has four fields: a sequence number, a label, a RTL, and a dead-variable list. All but the first of these fields may be empty.

The sequence number is a unique integer that PO uses to construct pointers into the list of RTLs. Labels should have a consistent format so that they can be easily identified. PO uses the RTL and the dead-variable list to dynamically discover the effects of instructions. The dead-variable list contains variables and registers whose contents are no longer valid, or of no further interest.

The expander can emit code that is ignored by PO by marking the record with a special tag. Debugging code is an example: code that logs statement numbers, posts routine names, and outputs diagnostic information. If PO deleted, rearranged, or modified such code, the user could be confused by the unexpected absence, rearrangement, or modification of the expected output.

It is sometimes difficult to describe a machine operation using the ISP notation. These can be expressed in assembly language and tagged so that PO ignores them. Other information, such as assembler pseudo-operations used to declare variables, allocate space, and define constants are of no use to PO and can be tagged.

4.2 Valid Instruction Rule

The code expander translates each abstract machine opcode into one or more valid RTLs for the target machine. A valid RTL represents a legal instruction from the machine description. If the expander emits an invalid RTL, the phase of PO that translates RTLs into assembly language reports an error.

4.3 Label Rule

The code expander identifies labels introduced during code expansion to ensure that incorrect optimizations are not made. Code expanders for machines with 'skip' instructions also produce code expansion labels for the implicit target of the skip. The section on labels in Chapter 5 explains the effects labels have on the various phases of optimization.

4.4 Side Effect Rule

The code expander identifies all side effects caused by the expanded code. If all side effects of the generated code are not identified, missing information causes PO to make erroneous changes to the program.

If an instruction causes the contents of a register to be destroyed, the code expander must pass this information to PO. For instance, on the DEC-10 the blt instruction (block transfer) destroys the contents of the register used to hold the source and destination addresses. On some CDC machines, the integer divide instruction is actually a macro that uses certain registers as scratch locations. Subroutine calls often have side effects that must be indicated. Depending on the calling sequence, the contents of certain registers may be destroyed by subroutine calls. The expander places such registers on the dead-variable list so that PO knows their contents are invalid.

4.5 Register Rules and Guidelines

4.5.1 Register Allocation

The code expander maps the abstract machine onto the target machine. For stack architectures, the mapping is straightforward. For general-register architectures, the mapping is more difficult because the abstract machine is stack oriented. Conventional code expanders are complicated by allocation of the target machine's registers. They must decide which values should be in registers and what to do if the available machine registers are exhausted. They also must maintain the status of the real machine registers and temporary locations—which ones are in use and what they contain.

The policy adopted in the expanders used with PO solves all of the problems above. All values are initially loaded into registers. This, however, would quickly exhaust the supply of registers. Consequently, expanders assume that there are as many registers as needed. Each time a new register is needed, one is generated. Once PO has reduced the number of registers needed and determined which values should remain in registers, a later pass, described in Chapter 5, maps the pseudo-registers onto the set of real machine registers.

All bookkeeping involving registers and temporary locations traditionally done by expanders is eliminated. Mapping the abstract stack machine to a register machine requires simply simulating the stack in the infinite supply of registers.

The following is a fragment of Y code, the intermediate code emitted by the Y compiler, and the RTLs emitted by the DEC-10 expander.

```
j = i   →   pushla  j   →   r[16] ← r[15] + j;
            pushla  i   →   r[17] ← r[15] + i;
            pushi       →   r[18] ← m[r[17]];
            popi        →   m[r[16]] ← r[18];
```

Pseudo-registers are numbered starting after the highest numbered register of the target machine. This allows the code expander to use the target machine's registers if necessary. In the above example, register 15 is the target machine register that serves as the pointer to local variables. The stack is simulated in pseudo-registers 16, 17, 18, etc.

4.5.2 Machine Register Rule

A machine's hardware registers may be used in generating code (e.g. register 15 in the example above). Before a hardware register is loaded, it must be placed on the previous dead-variable list. This ensures that PO does not rearrange the generated code so that the use of a hardware register is moved past a subsequent load. This is unnecessary for pseudo-registers if the intermediate result guideline below is followed.

4.5.3 Annotation Guidelines

PO eliminates redundant loads by tracking values that are loaded into registers, and when possible, reusing a value in a register instead of reloading it. Depending on the target machine, there may be values for which it is better to keep the load instruction. For example, on the CDC Cyber series machines, zero need not be retained in a register, since register B0 is always zero. These values, of course, depend on the particular target machine. Determining this set of values is left to the expander. For instance, on the DEC-10 it is better to not allocate a register to a constant that is eighteen bits or smaller since these are accessed inexpensively via 'immediate' mode operations, such as addi and movei. On the PDP-11, sixteen-bit constants can be accessed by indexing off the program counter via immediate mode addressing. In this case, it is less expensive to keep a constant that is used several times in a register and save the memory references, but this consumes scarce registers. For the PDP-11 expander used in this research, RTLs that load integer constants are annotated. Registers are saved for values that are expensive to load or compute.

Loads that should *not* be deleted are annotated. Annotation is a simple marking of the RTL. Using annotation only affects the quality of generated code; omitting annotation does not affect the correctness of the code. Experience indicates that annotation is more important on some machines than on others. For example, the code on the PDP-11 suffers considerably if annotation is not used, while code on the DEC-10 is still quite good, because the DEC-10 has more registers.

4.5.4 Intermediate Result Guidelines

Often a value that is loaded into a register is destroyed by an operation that uses that value. This is true on many two-address machines. For instance, on the DEC-10 the add instruction destroys one of the instruction's operands. If that value is used later, it must be reloaded. Expanders can determine which values can be destroyed only by looking ahead in the instruction stream. To avoid this complication and still produce good code, all values and intermediate results are preserved. Later phases of optimization determine which values can be destroyed. A small example clarifies this guideline. Again there is a Y code fragment, the intermediate code emitted by the Y compiler, and the RTLs generated by the PDP-11 expander.

```
j = i + 1   →   pushla  j   →   r[8] ← r[5];
                                r[8] ← r[8] + j;
                pushla  i   →   r[9] ← r[5];
                                r[9] ← r[9] + i;
                pushi       →   r[10] ← m[r[9]];
                pushic  1   →   r[11] ← 1;
                addi        →   r[12] ← r[10];
                                r[12] ← r[12] + r[11];
                popi        →   m[r[8]] ← r[12];
```

Register 5 is the frame pointer. The RTLs of interest are those generated for addi. On the PDP-11, an add instruction destroys one of the operands. If that value is subsequently used it has to be reloaded. To avoid this, the expander emits two transfers. The first one saves i in a new register. The second performs the addition in the new register. If there are subsequent uses of i, it is already in a register, and the redundant loads can be deleted. On the other hand, if there are no subsequent uses of i, the transfer of the contents of pseudo-register 10 to pseudo-register 12 is unnecessary and will be deleted by PO.

This guideline is irrelevant for three-address machines, since instructions operate on values without destroying them. If this guideline is not followed for two-address machines, code quality suffers; it may contain unnecessary loads.

Writing an expander is an ad hoc process. For traditional expanders, the quality of the final code depends on the ingenuity and expertise of the person writing the code expander. This is much less true for expanders written for use with PO. PO corrects inefficiencies, turning poor code into good code. Generally, it takes a day or so to implement an expander using the techniques described above. For example, the author implemented a code expander for the CDC Cyber series machines in five hours.

Chapter 5
The Optimizer - PO

There are two types of conventional peephole optimization. One type tracks values held in registers and tries to reuse their values whenever possible. The other combines logically adjacent instructions, replacing several instructions with one. PO performs both these optimizations in a machine-independent way. The part of PO that tracks values is called Cacher; the part that combines instructions is called Combiner. These two programs along with a third called Assigner, which assigns registers and translates RTLs to assembly language, form PO. The names Combiner and Assigner are derived from their functions, while the name Cacher suggests its implementation.

5.1 Cacher

Cacher eliminates redundant loads. In addition, it eliminates common subexpressions within a block of code bounded by labels, identifies dead variables, and defines the size of the peephole for Combiner.

Cacher extends the work of Freiburghouse [FREI74] and Gries [GRIE71]. The algorithm is similar to Freiburghouse's, but it is simpler and more thorough. It is simpler because it is just peephole optimization. It is more thorough because it detects and eliminates redundant computations missed by Freiburghouse's algorithm.

5.1.1 The Algorithm

Cacher reads a 'block' of code at a time, processes it, and then emits the simplified code. A block is a section of code bounded by labels. It builds sets of symbolic expressions (c-exprs) it has encountered as it scans the input. Formally, these sets form equivalence classes based on the following equivalence relation.

Let R be the set of all symbolic intermediate results produced by the bounded block. Let $a,b \in R$. Then the equivalence relation $a \approx b$ is true at a given point in the block if and only if the block sets values so that a and b have the same value at that point.

For example, after the register transfers

```
r[1] ← m[a]
r[1] ← r[1] + m[b]
```

r[1] ≈ m[a] + m[b].

Two relations are defined on the elements of R. Let $e,f,g,h \in R$. Let $>_t$ be an order relation where $e >_t f$ means that e was produced first in the code sequence, i.e. e is older than f. Let $<_c$ be an order relation where $g <_c h$ means that g is cheaper than h. Usually cheaper means faster, i.e. g is a faster reference to the machine's store than h. For instance, register references are the fastest references to a register machine's store. To optimize a program for size instead of speed, cheaper would mean smaller.

As Cacher scans the input, it partitions the c-exprs produced by the block into equivalence classes. As new c-exprs are read, Cacher searches the equivalence class for a set whose members are equivalent to the new c-expr. If one is found, Cacher substitutes the cheapest member of the set for the c-expr, eliminating a redundant load. The intermediate result guideline of Chapter 4 ensures that c-exprs previously loaded or computed are not destroyed and are available for reuse.

The input to Cacher is a list of RTLs. Each register transfer of each RTL is split into its destination and source. These are referred to as *dst* and *src*. If a RTL contains more than one register transfer, the operations outlined below are performed on each pair of *dsts* and *srcs*. This initial presentation of the algorithm ignores dead-variable analysis, redundant computation elimination, annotation, and window definition. A formal presentation of the complete algorithm appears in Appendix F.

The first two steps of the algorithm produce canonical forms for *dst* and *src*. Having a unique representation for each symbolic intermediate result simplifies identification of equivalent values—two symbolic results are equivalent if and only if their canonical representations are identical.

1. *Dst* is put into canonical form by first stripping off the outermost name and any brackets, exposing the address calculation. This string is called *adst*. For each register that is a substring of *adst*, the register's equivalence class is found and the register is replaced with the oldest member of that set. The original name and outermost brackets are then replaced. The resulting string is called *cdst*.

Cacher only replaces address calculations of destinations because replacing full destinations would remove stores to memory.

2. The second step puts *src* into canonical from. For each register that is a substring of *src*, the register's equivalence class is found and the register is replaced with the oldest member of that set. The resulting string is called *csrc*.

The relation $>_t$ selects replacement values. Its use ensures that each *dst* and *src* can be transformed to a unique string. The relation $<_c$ cannot be used to select replacement values because several equivalent expressions for the same value may be equally cheap.

3. This step searches for the 'cheapest' replacement for *src* and the cheapest replacement for the address calculation in *dst*. If a set is found in which *csrc* is a member, the cheapest member of that set replaces *src* in the RTL. If a set is found in which *adst* is a member, the cheapest member of that set replaces the address calculation of *dst*.

The fourth step places *cdst* and *csrc* in the appropriate equivalence class, and removes from all equivalence classes any members with which *cdst* interferes. The current register transfer is changing the value of *dst*, so any c-expr that depends on *dst* is also changed, and it may no longer belong in its current set.

4. First, *csrc*'s equivalence class is found. If no set contains *csrc*, one is created. Next, any set members with which *cdst* interferes are removed from their sets. Finally, *cdst* is added to the set found for *csrc* above.

Cdst's new set is identified before the deletions because it cannot be found later if *csrc* is among the deletions. Interference is determined by a machine-independent function described in the implementation section.

Though relatively simple, Cacher implements several optimizations at once. These are described in the following sections.

5.1.2 Redundant Load Elimination

Consider a machine that has a set of registers denoted by r[*n*], where *n* is a register number. For example,

```
a = b
c = b + 1
```

might compile into code that begins

```
1    r[10] ← m[b];
2    m[a] ← r[10];
3    r[11] ← m[b];
     ...
```

Applying the algorithm to the first two RTLs produces the equivalence class

```
{ {m[b], r[10], m[a]} }
```

Processing the third RTL, *dst* and *src* are r[11] and m[b], and *cdst* and *csrc* are r[11] and m[b]. There is a set that contains m[b], and the cheapest member of this set is r[10], so the third RTL is changed to

```
r[11] ← r[10];
```

eliminating a redundant load of memory location *a*. The equivalence class becomes

```
{ {m[b], r[10], m[a], r[11]} }
```

This example exposes one of Cacher's assumptions. In theory, a machine might have instructions for loading from memory, but not for inter-register transfers like the one above. Cacher should use PO's recognizer to verify the legality of its changes. In practice, most machines allow inter-register transfers, so instruction checking has not proven necessary. In general, Cacher assumes that a c-expr can always be replaced by a register.

5.1.3 Common Subexpression Elimination

Cacher maintains *use lists* for each c-expr. A use list contains pointers to each RTL that uses the c-expr. When an instruction is first encountered, it is added to the use lists of each c-expr it uses. When an instruction is changed to use a cheaper reference, it is removed from the use lists of the c-exprs it had used, and it is added to the use lists of the new c-exprs that it now uses. When a register's use list becomes empty, the instruction that loaded that register is deleted and removed from the use lists of the c-exprs it had used. This may trigger further deletions and removals, recursively. By deleting unused instructions Cacher eliminates common subexpressions. For example, consider the code

```
a = b + 3
c = b + 3
```

For the simple machine defined earlier, the RTLs generated are

```
1   r[10] ← m[b];
2   r[11] ← 3;
3   r[10] ← r[10] + r[11];
4   m[a] ← r[10];
5   r[12] ← m[b];
6   r[13] ← 3;
7   r[12] ← r[12] + r[13];
8   m[c] ← r[12];
```

When Cacher processes RTL 6, it detects that 3 was previously loaded into r[11], and rewrites it as

```
r[13] ← r[11];
```

It then adds RTL 6 to r[11]'s use list. r[11]'s use list now contains RTLs 3 and 6. It then processes RTL 7 and discovers that the sum m[b] + 3 has been computed and is in r[10]. It changes RTL 7 to

```
r[12] ← r[10];
```

adds it to r[10]'s use list, and removes it from the use lists for r[12] and r[13]. This empties these lists, so Cacher deletes RTLs 5 and 6. It then processes RTL 8, replacing r[12] with r[10] (older c-exprs are the cheaper of equals). RTL 7 is now unused and is deleted. The final register transfers are:

```
1   r[10] ← m[b];
2   r[11] ← 3;
3   r[10] ← r[10] + r[11];
4   m[a] ← r[10];
8   m[c] ← r[10];
```

Common subexpressions are usually eliminated at a higher level. There is, however, an advantage to doing common subexpression elimination ('CSE') on register transfers. At this level *all* values that are computed by the code sequence are exposed. For example, expanding address calculations often creates redundant machine-independent subexpressions that cannot be eliminated earlier because they did not exist earlier. In addition, machine idiosyncrasies hamper high-level CSE algorithms. For example, some instructions have extra effects (a division may yield both a quotient and a remainder) that cannot be detected or used by high-level CSE algorithms. Some machines compare two values by computing the difference and comparing it with zero. If a later calculation needs the difference between the two values, Cacher detects that it was already computed and uses the previously computed value, which higher-level machine-independent CSE algorithms cannot.

5.1.4 Dead-Variable Identification

Combiner can do a better job of combining instructions if it knows where variables die. A variable is dead if there are no further uses of it or its contents are no longer valid. Cacher identifies dead variables by noting where c-exprs are last used. Identifying dead variables at the machine level provides a bonus similar to that of doing CSE at a low level—all values computed by the code are subjected to dead-variable analysis. For example, many calling sequences return function values in a fixed register. After the function return, the value is moved to another register in case the special register is needed again. Sometimes the move is unnecessary as there are no further uses of the special register. Conventional compilers identify such avoidable moves by case analysis deep in a machine-dependent code generator. Cacher avoids such unnecessary moves through the general operation of dead-variable identification. The last example is shown with the dead variables in parentheses to the right of the RTLs where the variables died.

```
1   r[10] ← m[b];
2   r[11] ← 3;
3   r[10] ← r[10] + r[11];   (r[11])
4   m[a] ← r[10];
8   m[c] ← r[10];            (r[10])
```

5.1.5 Window Definition

Earlier versions of PO only combined lexically adjacent instructions. Many obvious optimizations were missed. For example, PO could not collapse an otherwise-reducible pair separated by a third, uncombinable instruction. This lack of context is addressed by Cacher. By noting where c-exprs are set and used, it does a simple flow analysis of the program. It outputs this information as links between register transfer lists. Use of a c-expr is linked back to the instruction that set it, provided no previous instruction has a link to that instruction.

These links allow Combiner to consider instructions separated by an arbitrary number of instructions. Combiner is no longer looking at the instructions through a small peephole, but through a window whose opening is constantly being adjusted. Experiments show a 20% reduction in code size by an optimizer that uses such window information as compared to one that uses a fixed window. The following is the last example with the links added in braces.

```
1          r[10] ← m[b];
2          r[11] ← 3;
3   {2}    r[10] ← r[10] + r[11];   (r[11])
4   {3}    m[a] ← r[10];
8          m[c] ← r[10];            (r[10])
```

Note that although RTL 8 uses r[10], there is no link between it and RTL 3. This is because of the intervening use at RTL 4. If there were a link, Combiner would attempt to collapse the pair, possibly rearranging the code so that the effect of RTL 3 occurs after RTL 4. This would change, erroneously, the effect of the program.

5.1.6 Annotation

Annotation, introduced in Chapter 4, controls which values are held in registers. C-exprs from annotated register transfers cannot be used to replace the source of a register transfer. This ensures that annotated values do not persist in the machine's registers. This does not mean that annotated register transfers cannot be deleted. If an annotated register transfer provides input to a larger computation that is redundant, when that computation is replaced all RTLs that are inputs to the computation (annotated or not) may be deleted. For the previous simple machine, if it was decided that constants should not be saved in registers, the RTLs generated would be:

```
1   r[10] ← m[b];
2   *r[11] ← 3;
3   r[10] ← r[10] + r[11];
4   m[a] ← r[10];
5   r[12] ← m[b];
6   *r[13] ← 3;
7   r[12] ← r[12] + r[13];
8   m[c] ← r[12];
```

When processing RTL 6, the algorithm would not replace the 3 with r[11]. It would, when processing RTL 7, detect the redundant computation, and replace the source with r[10] and delete RTLs 5 and 6.

The complete algorithm with dead-variable analysis, annotation, redundant expression elimination, and window definition appears in Appendix F.

5.1.7 Implementation

Cacher uses an associative memory or cache to maintain the equivalence classes. When an intermediate result is presented to the cache, its equivalence class is returned. Each set is a linked list of its members ordered according to the relation, $>_t$.

Cacher is retargeted by rewriting the function that determines cost, $<_c$, and supplying patterns that identify registers. The cost function accepts two c-exprs and returns the cheaper one. C-exprs matching a register pattern are preferred to those matching simple memory references, which are preferred to more expensive memory references. Typically, only a few machine-dependent patterns must be changed.

To avoid machine-dependencies in the interference function, the front end of the Y compiler identifies for Cacher conflicts due to aliasing [Aho77]. It can do so without the machine-dependent patterns that Cacher would require to recognize register transfer patterns that result in aliasing (e.g. addressing for array elements, parameters, or globals). Aliasing information is passed by the front end to Cacher through the use of special intermediate language opcodes (see Appendix A). Cacher's interference function thus accepts two arguments and reports a conflict only if the first explicitly appears in the second.

Each c-expr has two lists associated with it. One marks the RTLs that use the c-expr, and the other marks the register transfers that set it. These lists maintain the information needed to produce the dead variable and window definition information.

As Cacher processes each RTL, it computes the window pointers for that RTL. This is accomplished by identifying all cells that are used in the current RTL and locating the RTLs that set them. If no other RTLs have links to the RTL that set the cell, the RTLs are linked.

When Cacher encounters a label, it removes all c-exprs. As each c-expr is removed, it adds the c-expr to the dead-variable list of the RTL that last used the c-expr.

Function and subroutine calls require special attention by Cacher. It knows equivalent values only by simulating the effects of register transfers. Subroutines and functions can change values without Cacher being aware of the effects. It must remove any values from its associative memory that could be modified as a side effect of a subroutine call. It does this by scanning the cache at each subroutine call and removing all references to global variables and to arrays that are arguments of the call. These are identified by the compiler and passed to Cacher using the same intermediate language opcodes to pass aliasing information.

Cacher is written in C and runs on a PDP-11/70 under UNIX. This implementation is 1100 lines of code and processes about 100 instructions per second. Because information is not maintained across labels, it does not require much memory. Experiments show that Cacher produces a five to ten percent improvement in code size, and it allows Combiner to reduce it by another twenty percent.

5.2 Combiner

Combiner symbolically simulates pairs of instructions that have been linked by Cacher's flow analysis and, where possible, replaces them with equivalent single instructions.

5.2.1 Pairs

Combiner determines the effect of a pair by combining their independent effects, substituting values assigned to variables in the first register transfer list for instances of those variables in the second, and removing effects that set dead variables. For example, the DEC-10 instructions

```
setz     1
movem    1,loc
```

are represented in ISP as

```
r[1] ← 0;
m[loc] ← r[1];
```

The combined effect of these instructions is

```
m[loc] ← 0; r[1] ← 0;
```

To determine if the resulting RTL is a legal instruction, Combiner uses the recognizer described in Chapter 3. The register transfer above represents the instruction

```
setzb  1,loc
```

Combiner considers an instruction and its logical predecessors. By combining instructions in reverse order, Combiner avoids backing up to consider new possibilities when a replacement is made. For example, the ISP for the CDC Cyber instructions

```
sb4    b3+b2
sa3    b4
```

is (with dead variables and window pointers)

```
5   {4}     b[4] ← b[3]+b[2];               (b[3],b[2])
6   {5,3}   x[3] ← m[b[4]]; a[3] ← b[4];    (b[4])
```

Combiner first attempts to combine instruction 5 with its logical predecessor, instruction 4 (not shown). Assuming it fails, Combiner advances to instruction 6 and considers it with its logical predecessor, instruction 5. The combined effect of these instructions is

```
x[3] ← m[b[3]+b[2]]; a[3] ← b[3]+b[2];
```

Since the resulting RTL is a legal instruction, Combiner replaces the RTL 6 with the result and deletes RTL 5. In the above example, the RTL represents the instruction

```
sa3     b3+b2
```

After each replacement, Combiner creates the dead-variable list for the new RTL. The new list is formed by merging the dead-variable lists of the RTLs that formed the new instruction. Further, any pseudo-registers used in the first RTL and dead before the second are moved from their old dead-variable list to the new one. If a hardware register used in the first RTL appears on a dead-variable list that lies between the RTLs that formed the new instruction, the combination is illegal and the replacement is not made. The back-pointer list for the new RTL is formed by merging the lists of the RTLs that formed the new instruction. For the above example, the result with merged lists is

```
6   {4,3}   x[3] ← m[b[3]+b[2]]; a[3] ← b[3]+b[2];    (b[4],b[3],b[2])
```

After successfully replacing a pair of instructions, Combiner considers pairs that end with the new instruction. In the example above, Combiner would now consider instruction 6 with its logical predecessors, instructions 4 and 3. When all possible combinations of the current instruction with previous instructions have been considered, Combiner advances to the next instruction. Through the use of window pointers and by combining instructions in reverse order, Combiner reduces *all* possible linked pairs without backing up after each replacement.

5.2.2 Triples

If the result of combining a pair is not a legal instruction, Combiner simulates a triple when possible. If the pointer list of the first RTL of the pair is not empty, it combines its logical predecessor with the result of combining the pair. If the result is a valid instruction, Combiner replaces the last RTL, and deletes the first two. As with pairs, it creates a new dead-variable list and pointer list from all three RTLs. An example clarifies how Combiner handles triples. The following DEC-10 code increments the local variable i.

```
move     10,i(15)
addi     10,1
movem    10,i(15)
```

The corresponding register transfers with window pointers and dead variables are

```
1          r[10] ← m[r[15] + i]];
2   {1}    r[10] ← r[10] @ 1;
3   {2}    m[r[15] + i] ← r[10];      (r[10])
```

@ is the 36-bit addition operator. First, Combiner composes RTLs 1 and 2. This yields the RTL

```
r[10] ← m[r[15] + i] @ 1;
```

which is not a legal DEC-10 instruction. Because list 1 has no pointers, Combiner advances to RTL 3. It combines RTLs 2 and 3 yielding

```
m[r[15] + i] ← r[10] @ 1; (r[10])
```

which, again, is not a legal instruction. RTL 2, however, has a pointer to RTL 1 so Combiner simulates the composition of RTLs 2 and 3 with RTL 1. This yields

```
m[r[15] + i] ← m[r[15] + i] @ 1; (r[10])
```

which is the instruction

```
        aos     i(15)
```

Combiner replaces RTL 3 with the result, and deletes RTLs 1 and 2.

Combiner simulates triples because many machines offer some one-instruction replacements for load/operate/store sequences, but few offer replacements for load/operate or operate/store sequences. Combiner must consider all three register transfers to reduce them to one. Simulating triples slows Combiner, but it does not make it more complex as it uses the machinery that combines pairs to do triples. Fortunately, no special need has been seen for a more complex replacement strategy (e.g., quadruples).

5.2.3 Labels

Labels prevent the consideration of some pairs. Combining pairs whose second instruction is labeled might change, erroneously, the effect of the program. Branches to the label would cause the effects of both instructions to be performed. To do the best job possible, Combiner removes any labels it can. As it reads the program, Combiner constructs a symbol table of labels. The symbol table contains a reference count and a pointer to where the label was defined. Any labels that have a zero reference count are immediately removed. As optimization progresses, labels whose reference counts become zero are removed.

When Combiner removes the last reference to a label that it has passed, it should back up to reconsider the instructions the label separated; optimizations between the previously separated pair may now be possible. This reconsideration is needed only for labels referenced following their definition. When optimizing code generated from a program with 'structured' control flow, the only such labels are loop and subroutine heads, and peephole optimizers seldom remove these labels. This particular form of backup, though easily implemented and theoretically necessary, was discarded as ineffective.

Programs that are too large to fit into Combiner's memory are optimized in segments. When optimizing in segments, Combiner cannot delete labels because parts of the program not yet optimized may reference the labels.

5.2.4 Branches

Pairs that begin with branches need special treatment. A branch instruction is always linked to its lexical predecessor. The condition on which the branch depends must be inverted and added to the register transfers of the second instruction before combining effects. Consider, for example, the PDP-11 code

```
        beq   L1
        br    L2
    L1:
        ...
```

This is represented in register transfers as

```
    3 {2}      PC ← if NZ = 0 then L1 else PC;
    4 {3}      PC ← L2;
    5      L1:
               ...
```

These combine to

```
        PC ← if NZ = 0 then L1 else PC; PC ← if ~(NZ = 0) then L2 else PC;
    L1:
```

A symbolic simplifier improves awkward relationals and removes redundant assignments, yielding

```
        PC ← if NZ ≠ 0 then L2 else PC;
    L1:
```

which is the instruction

```
        bne   L2
    L1:
```

Unconditional branches depend on the constant condition true; the symbolic simplifier deletes register transfers depending on its inverse, removing unreachable code. The following code for the DEC-10 provides an example.

```
    jumpe 3,L2
    ...
    jrst     L3
    jrst     L2
```

has the effect

```
    PC ← if r[3] = 0 then L2 else PC;
    ...
    PC ← L3;
    PC ← L2;
```

The second jrst is linked to the first and they combine to

```
    PC ← L3; PC ← if false then L2 else PC;
```

which simplifies to

```
    PC ← L3;
```

Combiner replaces the pair with the singleton yielding the instruction

```
    jumpe 3,L2
    ...
    jrst     L2
```

removing the second unreachable jump. The next section shows how code can become unreachable.

5.2.5 Branch Chains

Combiner collapses branch chains by treating a branch and its target as a pair. It locates the target of the branch by using the label symbol table. If the pair collapses to a legal instruction, only the branch is replaced; the labeled instruction is not modified. For example, the PDP-10 code sequence

```
           jumpe 3,L1
           ...
           jrst     L3
    L1:    jrst     L2
```

has the effect

```
           PC ← if r[3] = 0 then L1 else PC;
           ...
           PC ← L3;
    L1:    PC ← L2;
```

The jumpe and its target combine to

```
    PC ← if r[3] = 0 then L2 else PC;
```

which is the instruction

```
        jumpe   3,L2
```

Combiner replaces the first instruction with the result, yielding

```
              jumpe 3,L2
              ...
              jrst    L3
        L1:   jrst    L2
```

Notice that a reference to label L1 has been removed. If this is the last reference, the label can be removed making the second jrst unreachable.

Before performing branch chaining, Combiner must make certain that the branch chain does not contain a cycle. Branch chains containing cycles can cause Combiner to perform a infinite sequence of optimizations. For example, the following code for the PDP-11 contains a cycle between labels L1 and L2.

```
              br  L1
              ...
        L1:   br  L2
              ...
        L2:   br  L1
```

Without cycle detection, Combiner would follow the first branch chain and rewrite the first br to branch to L2. It would then follow the new branch chain and rewrite the first br to branch to L1. The code is now back to its original form. Combiner would continue to follow one branch chain and then the other.

Combiner detects cycles by following branch chains and marking branch instructions that it has visited. If it encounters a branch instruction that it has visited before, the chain contains a cycle and branch chaining is not performed.

5.2.6 Simplifications

Combiner has a section of code that is specially designed to simplify RTLs. It removes double negations, jumps to the next instruction, unnecessary additions and subtractions of zero, and conditionals that depend on the constant condition false. It also replaces awkward relationals such as $\sim (x = y)$ with $x \neq y$.

Simplifications can also be machine dependent. For example, the PDP-11 machine description describes the auto-increment register addressing mode as

```
        m[r[RN]++]
```

A simplification is included that rewrites RTLs such as

```
        m[r[2]] ← 0; r[2] ← r[2] + 2;
```

as

```
        m[r[2]++] ← 0;
```

A similar simplification is included for auto-decrement. These are the only machine-dependent simplifications currently found to be necessary.

Simplifications can be added or removed to Combiner as dictated by the target machine. In practice, simplifications added are seldom removed when retargeting PO for a new machine unless the simplification causes poor code to be produced.

5.2.7 Implementation

Combiner's first phase reads the input. The input consists of the output from Cacher: register transfer lists, dead-variable lists, and window pointers. If the entire program will not fit in memory, it reads in as much as possible, optimizing the program in segments. As it reads the input, it builds a doubly-linked list of RTLs and a symbol table of labels.

After the input is read, Combiner deletes all labels that are known to have a zero reference count and begins simulating instructions. When the end of the linked list of RTLs is reached, it outputs the register transfers along with the dead-variable lists. The dead-variable lists are used by Assigner to assign registers.

Combiner is written in C and runs on a PDP-11/70 under UNIX. This implementation is 1600 lines of code and processes approximately 60 instructions per second. Combiner produces a 40 to 50% improvement in code size. A better code generator would reduce this figure.

5.3 Assigner

Assigner maps pseudo-registers to hardware registers and translates RTLs to assembly language.

5.3.1 Register Assignment

When the RTLs are generated, pseudo-registers are used instead of real registers. As it reads RTLs, Assigner associates a hardware register with each pseudo-register and replaces each use of a pseudo-register with the associated hardware register. Assigner frees the associated hardware register when the pseudo-register dies.

There are two aspects of register allocation: value retention and register demand [FREI74]. Value retention is concerned with making sure that a value is held in a register no longer than necessary. Cacher's dead-variable analysis provides an optimal solution to the value retention problem for bounded blocks [FREI74].

Register demand is concerned with minimizing the number of loads and stores when excess demand for registers forces a register to be stored in a temporary location. The optimal algorithm for bounded blocks is given by Belady [BELA66]. It requires looking ahead in the input stream and picking the register whose next reference is farthest away. Freiburghouse compared three techniques for solving the register demand problem to the optimal solution. Of the three methods, usage counts produced the best results. Least-recently-used and least-recently-loaded were second and third. Although Cacher develops usage counts (through the use lists), Assigner uses the LRU technique because Combiner invalidates those counts by collapsing instructions.

For example, suppose the simple machine used in the first part of this chapter had only two hardware registers, 0 and 1. The code

```
1   r[10] ← m[a];
2   r[11] ← 3;
3   r[10] ← r[10] + r[11];    (r[11])
4   m[a] ← r[10];             (r[10])
5   r[12] ← m[b];
6   r[13] ← 4;
7   r[12] ← r[12] + r[13];    (r[13])
8   m[e] ← r[12];             (r[12])
```

would be converted to

```
1    r[0] ← m[a];
2    r[1] ← 3;
3    r[0] ← r[0] + r[1];
4    m[a] ← r[0];
5    r[1] ← m[b];
6    r[0] ← 4;
7    r[1] ← r[1] + r[0];
8    m[e] ← r[1];
```

Pseudo-registers 10 and 13 are assigned hardware register 0, and 11 and 12 are assigned hardware register 1.

When the demand for hardware registers exceeds the supply, Assigner allocates a temporary, generates code to save the contents of the hardware register in the temporary, and frees the register for use. This is called a register spill. The pseudo-register that was associated with the freed hardware register is now associated with a temporary location. When Assigner encounters the pseudo-register again, it allocates a hardware register, generates code to load the register from the temporary location, and frees the temporary.

One flaw in this organization is that instructions to load and store registers from temporary locations are added after Combiner's peephole optimization phase. Optimizations may now be possible between the new instructions and the old ones. Fortunately, this seldom occurs. The 3500-line Y compiler is compiled using only 42 register spills for the PDP-11 (with three allocated registers) and none for the DEC-10 (with twelve).

Assigner also makes certain optimizations. Because of the code generation guideline of retaining temporary values (intermediate result guideline, Chapter 4), RTLs of the form

```
r[11] ← r[10]; (r[10])
```

often appear for 2-address machines. Assigner deletes such RTLs, and associates the hardware register that was associated with pseudo-register 10 with pseudo-register 11.

Thus, for the simple machine introduced at the beginning of this chapter, the following register transfers

```
1    r[10] ← m[b];
3    r[12] ← r[10];        (r[10])
4    r[12] ← r[12] + 3;
5    m[a] ← r[12];
8    m[e] ← r[12];         (r[12])
```

are transformed to

```
1    r[0] ← m[b];
4    r[0] ← r[0] + 3;
5    m[a] ← r[0];
8    m[e] ← r[0];
```

5.3.2 Assembly

After the pseudo-registers of a register transfer list are mapped to real registers, Assigner translates the register transfer list to the assembly language of the target machine and outputs it. This translation is performed by the transducer described in Chapter 3.

For example, Assigner translates the PDP-11 RTL

```
m[r[5] + i] ← m[r[5] + i] << 1; (NZ)
```

to

```
asl i(r5)
```

The CDC Cyber series RTL

```
x[3] ← 0;
```

is translated to

```
mx3   0
```

This last example illustrates one of the functions of Assigner—picking the cheapest instruction that does the job. On the CDC Cyber series machines there are several ways to clear a X register. A zero can be loaded via a set X instruction, zero can be loaded from register B0 which always is zero, or the mask instruction can be used to build a zero mask. The fastest way to clear a X register is to use the mask instruction. Because instructions in the machine description are ordered according to speed, Assigner can pick the fastest instruction that performs the register transfers requested.

5.3.3 Implementation

Assigner is generated from the machine description of the target machine and from a machine-independent register assigner. The register assigner is retargeted by supplying information about the machine's registers. For each type of register, Assigner must know how many there are, their sizes, which ones are available for assignment, and code templates for storing them into and loading them from temporaries. Approximately twenty lines of code must be changed to retarget Assigner's register assignment module for a new machine.

Assigner is written in C and runs on a PDP-11/70 under UNIX. This implementation is 1400 lines of code and processes 90 instructions per second.

5.4 Results

PO performs all of the following optimizations.

- common subexpression removal within blocks
- allocation and assignment of registers
- special case instruction usage
- exotic addressing mode usage
- branching chaining
- unreachable code removal

In addition, PO is complete in the sense that upon completion no instruction, and no pair or triple of logically adjacent instructions, can be replaced with a cheaper single instruction that has the same effect.

Chapter 6
Results and Conclusions

The previous chapters have outlined a technique for producing retargetable compilers. Y cross-compilers for several machines have been constructed using these techniques. One of these compilers, the Y PDP-11 compiler, has been in use for over a year and a half. Compilers constructed using this technique compare favorably to existing compilers for other high-level languages.

6.1 Comparison

The effectiveness of this technique for producing retargetable compilers can best be measured by comparing a compiler built using these techniques to existing compilers. There are four areas of comparison: the quality of the generated code, the speed of the compiler, machine applicability, and the effort required to retarget the compiler for a new machine.

6.1.1 Code Quality

Code quality will be illustrated using some of the many routines compiled by the Y cross-compilers for the PDP-11, DEC-10, and CDC Cyber 175. The same routines have also been compiled on those machines using existing compilers.

One such routine is ctoi [KERN76, KERN81] which converts an ASCII string to a number. It skips over leading tabs and blanks and stops at the first character that is not a digit. This routine has been written in C, Pascal, SIMULA [DAHL66], Ratfor [KERN76], and Y. The Y version of ctoi is

```
integer ctoi(in)
char in[ ]
int i, sum

    i = 1;
    while (in[i] == ' ' | in[i] == '\t')
        i = i + 1
    sum = 0
    while (in[i] >= '0' & in[i] <= '9') {
        sum = sum * 10 + in[i] - '0'
        i = i + 1
        }
    return (sum)
end
```

Below is the code produced by the PDP-11 Y compiler and the PDP-11 Unix C compiler. They have been edited to improve their readability. The important differences are summarized following the code.

```
      Y Compiler                        C Compiler
ctoi:  jsr    r5,ysv           ctoi:  jsr    r5,csv
       sub    $150,sp                 sub    $4,sp
       mov    $1,i(r5)                mov    $1,i(r5)
L1:    mov    i(r5),r2                jbr    L4
       add    in(r5),r2        L10:   inc    i(r5)
       movb   -1(r2),r3        L4:    mov    in(r5),r0
       cmp    r3,$' '                 add    i(r5),r0
       jeq    L4                      cmpb   $' ',(r0)
       cmp    r3,$'\t'                jeq    L10
       jne    L3                      mov    in(r5),r0
L4:    inc    i(r5)                   add    i(r5),r0
       jbr    L1                      cmpb   $'\t',(r0)
L3:    clr    sum(r5)                 jeq    L10
L5:    mov    i(r5),r2                clr    sum(r5)
       mov    r2,r3                   jbr    L6
       add    4(r5),r3         L20:   mov    in(r5),r0
       movb   -1(r3),r4               add    i(r5),r0
       cmp    r4,$'0'                 cmpb   $'9',(r0)
       jlt    L7                      jlt    L7
       cmp    r4,$'9'                 mov    sum(r5),r1
       jgt    L7                      mul    $12,r1
       mov    $12,r1                  mov    in(r5),r2
       mul    sum(r5),r1              add    i(r5),r2
       mov    r1,r3                   movb   (r2),r2
       add    r4,r3                   add    r2,r1
       sub    $60,r3                  add    $-60,r1
       mov    r3,sum(r5)              mov    r1,sum(r5)
       inc    r2                      inc    -10(r5)
       mov    r2,i(r5)         L6:    mov    in(r5),r0
       jbr    L5                      add    i(r5),r0
L7:    mov    sum(r5),r0              cmpb   $'0',(r0)
       jmp    yret                    jle    L20
                               L7:    mov    sum(r5),r0
                                      jmp    cret
```

The Y compiler's code is only two instructions shorter than the code produced by the Unix C compiler. It occupies, however, ten percent fewer bytes. The Y code is shorter because the expression in[i] is kept in a register within a basic block. For instance in the loop that computes sum, the Y compiler loaded in[i] into register 4 and avoided two redundant loads. C programmers can put variables in registers, but not expressions such as in[i]. Johnson's Portable C Compiler for the PDP-11 produces exactly the same code as the Unix C compiler. A comparison of the execution speed of the generated code appears at the end of this section.

The Vrije Pascal compiler generates code for the PDP-11. This compiler produces code for an abstract machine called EM-1. Before EM-1 is translated to assembly language, it is optimized by a peephole optimizer. The Y compiler generates 110 bytes of code for ctoi. The VU Pascal compiler generates 162 bytes.

There are several reasons for this large difference. The VU code for loading in[i] is longer. The code generated by the two compilers for loading in[i] is

```
   Y Compiler             VU Compiler
 mov    i(r5),r2        mov   i(r4),r0
 add    in(r5),r2       dec   r0
 movb   -(r2),r3        add   (r4),r0
                        clr   r2
                        bisb  *r0,r2
```

Because of Pascal's character convention, a character is loaded by clearing a register and or-ing in the character. The code could still be improved. For example, the decrement instruction could be combined with the bisb forming the instruction

```
bisb  -1(r0),r2
```

This code was not produced because optimization was performed before code expansion. Consequently, the peephole optimizer could not detect that parts of the address calculation could be combined. This is a specific instance of the more general problem of applying optimizations at a high-level. Certain optimizations are missed because information about the machine's instruction set is unavailable.

The other difference between the Y compiler's code and the VU Pascal compiler's code is the elimination of redundant loads. The VU Pascal compiler could save forty-two bytes of code if it eliminated redundant loads.

There is a Pascal compiler developed at the Swiss Federal Institute of Technology (ETH) for the CDC Cyber series machines. Although the compiler is one-pass, it generates very good code [AMMA77]. Below is comparison of the code generated by a Y compiler targeted for the CDC machines and the code generated by the ETH Zurich Pascal compiler. The program compiled was an integer version of ctoi (see Appendix G).

	Y Compiler			ETH Pascal		
1	ctoi.	rj	ysav	ctoi	rj	xp.pen
2		sb7	b7+28		sx6	b1
3		sx6	b1		sa6	b5+i
4		sa6	b6+i.	L1	sa1	b5+in
5	L1	sa1	b6+i.		sa2	b5+i
6		sa2	b6+in.		ix0	x1+x2
7		sb2	x2		sa3	x0−1
8		sb3	x1+b2		sx0	55b
9		sa1	b3+−1d		ix3	x3−x0
10		sx2	32d		ix4	x1+x2
11		ix3	x1−x2		sx5	11b
12		zr	x3,L4		sa4	x4−1
13		sx2	9d		ix4	x4−x5
14		ix3	x1−x2		bx6	x6−x6
15		nz	x3,L3		ix7	x6−x3
16	L4	sa1	b6+i.		bx7	−x7−x3
17		sx2	b1		ix3	x6−x4
18		ix6	x1+x2		bx3	−x3−x4
19		sa6	a1		bx4	x7+x3
20		eq	L1		pl	x4,L2
21	L3	mx6	0		sx3	b1
22		sa6	b6+sum.		ix7	x2+x3
23	L5	sa1	b6+i.		sa7	a2
24		sa2	b6+in.		eq	L1
25		sb2	x2		sa6	a2−b1
26		sb3	x1+b2	L2	sa1	b5+in
27		sa2	b3+−1d		sa2	b5+i
28		sx3	48d		ix0	x1+x2
29		ix4	x2−x3		sa3	x0−1
30		ng	x4,L7		sx0	33b
31		sx4	57d		ix3	x3−x0
32		ix5	x4−x2		ix4	x1+x2
33		ng	x5,L7		sa4	x4−1
34		sa5	b6+sum.		sx5	b0+44b
35		sx4	10d		ix4	x5−x4
36		ix5	x5*x4		bx6	x3+x4
37		ix4	x5+x2		ng	x6,L3
38		ix6	x4−x3		sa3	a2−b1
39		sa6	a5		lx4	b1,x3
40		sx2	b1		bx6	x4
41		ix6	x1+x2		lx6	2
42		sa6	a1		ix6	x6+x4
43		eq	L5		ix4	x1+x2
44	L7	sa1	b6+sum.		sa4	x4−1
45		bx0	x1		ix4	x6+x4
46		eq	yret		ix6	x4−x0
47					sa6	a3
48					sx3	b1
49					ix7	x2+x3
50					sa7	a2
51					eq	L2
52				L3	sa3	a2−b1
53					bx6	x3
54					sa6	a1+b1
55					bx6	x3
56					eq	xp.pex

The Y compiler's code is ten instructions shorter than the ETH compiler, but the sequences occupy the same number of words. The ETH compiler is tailored for model 6400 CDC machines. For example, multiplies by a constant that can be expressed as a sum or difference of powers of two are performed using faster shifts and adds instead of a multiply instruction. While this would be faster on a 6400, it is slower on the newer model 175 and 176's. The compiler uses this technique to accomplish the multiply by ten at lines 39 through 42. The ETH code sequence would be shorter if they had used a regular multiply instruction..

The ETH compiler also attempts to preserve values in registers that may be used later. For example, addresses are preserved in address registers as long as possible to avoid recomputing addresses. This also allows smaller fifteen-bit load and store instructions to be used where possible. The ETH Pascal compiler uses this technique to load and store sum. For example at line 25, it stores a zero into memory location sum. It has remembered that X6 contains zero and that a2 contains the address of i which is the address of sum plus one.

The initial implementation effort for the ETH compiler was fourteen months. The Cyber Y compiler took the author five days, and this included developing a skeletal runtime system. Despite this large difference in implementation effort and the careful tailoring of the ETH compiler to one class of machines, the Y compiler generally generates code of roughly equal quality.

Glanville's dissertation displays code produced by his code generator for the PDP-11. One of the routines that he used to compare code with the Unix C compiler is shown below rewritten in Y. It reads an integer from the input.

```
int ch
int readn()
    int answer, lval, base

    while (ch == ' ')
        ch = getc()
    if (ch <= '9' & ch >= '0') {
        if (ch == '0')
            base = 8
        else
            base = 10
        lval = 0
        repeat {
            lval = lval * base + ch - '0';
            ch = getc()
            } until (ch < '0' | (ch - '0') > base)
        answer = lval
        }
    else
        answer = -1
    return (answer)
end
```

The code produced by the Y PDP-11 compiler and Glanville's code generator is shown below.

```
               Y Compiler                          Glanville's Code Generator
 1   L1:   cmp   ch,$' '                     L1:   cmp   ch,$' '
 2         jne   L3                                jne   L2
 3         jsr   pc,getchar                        jsr   pc,getchar
 4         mov   r0,ch                             mov   r0,ch
 5         jbr   L1                                jbr   L1
 6   L3:   mov   ch,r2                             cmp   ch,$'9'
 7         cmp   r2,$'9'                           jgt   L3
 8         jgt   L4                                cmp   ch,$'0'
 9         cmp   r2,$'0'                           jlt   L3
10         jlt   L4                                cmp   ch,$'0'
11         cmp   r2,$'0'                           jne   L5
12         jne   L6                                mov   $10,base(r5)
13         mov   $10,base(r5)                      jbr   L6
14         jbr   L7                          L5:   mov   $12,base(r5)
15   L6:   mov   $12,base(r5)                L6:   clr   lval(r5)
16   L7:   clr   lval(r5)                    L7:   mov   base(r5),r0
17   L8:   mov   lval(r5),r1                       mul   -4(r5),r0
18         mul   base(r5),r1                       add   ch,r0
19         add   ch,r1                             sub   $60,r0
20         sub   $60,r1                            mov   r0,lval(r5)
21         mov   r1,lval(r5)                       jsr   pc,getchar
22         jsr   pc,getchar                        mov   r0,ch
23         mov   r0,ch                             cmp   ch,$'0'
24         mov   ch,r2                             jlt   L8
24         cmp   r2,$'0'                           mov   ch,r0
25         jlt   L11                               sub   $60,r0
26         sub   $60,r2                            cmp   r0,base(r5)
27         cmp   r2,base(r5)                       jlt   L7
28         jle   L8                          L8:   mov   lval(r5),answer(r5)
29   L11:  mov   lval(r5),answer(r5)               jbr   L4
30         jbr   L5                          L3:   mov   $-1,answer(r5)
31   L4:   mov   $-1,answer(r5)              L4:   mov   answer(r5),r0
32   L5:   mov   answer(r5),r0
```

Although the code produced by the Y compiler is one instruction longer than the code produced by Glanville's code generator, it is actually two bytes shorter.[†]

Similar comparisons with code produced by Ganapathi's code generator show that Y compilers constructed using PO generally produce better code.

Comparisons of code on other machines with other language processors show that code generated using PO is generally of better quality except for language processors that perform optimizations such as global register allocation and code motion. Large gains in both the size and speed of the generated code are obtained when optimizations such as assignment of loop indices to registers are performed. The timings of the CDC and DEC-10 Fortran compilers are evidence of this.

In addition to comparing code size, the execution speed of the code produced by retargetable Y compilers is compared to the execution speed of code produced by existing compilers. To accomplish this, three programs are presented as benchmarks. A program that solves the eight-queens problem (called 8q) was chosen to test recursion. An integer version of ctoi (called ictoi) was chosen because it involves integer arithmetic and manipulation of integer arrays. Ctoi was chosen because it involves characters and character arrays. Listings of these programs appear in Appendix G.

To minimize drift and error in the timings, each routine was surrounded by an outer loop that executed the routine many times. Ctoi and ictoi were called thirty thousand times, while 8q was called one

† Glanville's code appears to contain a subtle error. Sixteen-bit multiplication should be performed in an odd register; see line 17.

hundred times. Timings were taken on the PDP-11, the DEC-10, and the CDC Cyber using a retargetable Y compiler and various existing compilers. Each program was compiled using all optimizations available for the particular compiler. The execution times are in seconds.

On the PDP-11, the programs were compiled using the Y compiler, the Unix C compiler, and the VU Pascal compiler. The timings are as follows.

	Y Compiler	C Compiler	VU Pascal
8q	37.0	37.3	62.2
ctoi	19.5	21.5	26.2
ictoi	39.5	45.9	69.6

The timings for the Y compiler are better than those for the UNIX C compiler and the VU Pascal compiler, mainly because the Y compiler eliminates redundant expressions. Notice that the timings for the character versions are substantially better than the integer versions. This is because the integer versions require a shift instruction to convert an array index to a word index.

On the CDC Cyber machine, timings were taken using the ETH Pascal compiler, Ratfor, and the Y compiler. A Ratfor version of 8q could not be run because it is recursive. A Ratfor version of ctoi also could not be run because the version of Fortran used did not support characters. The timings follow.

	Y Compiler	Ratfor	ETH Pascal
8q	17.6	n/a	16.6
ctoi	18.1	n/a	26.3
ictoi	15.5	8.7	14.0

The timings show that the Cyber Y compiler is more efficient than the ETH Pascal compiler in handling characters while the ETH compiler is slightly better at handling integers. The Ratfor version of ictoi is much faster than the other two versions. Most of this difference is due to the differences in the calling sequences. Instrumenting the Y code showed that approximately thirty percent of the execution time in the procedure prologue and epilogue code. The other differences come from global optimizations that the Fortran compiler performs.

On the DEC-10, the benchmark programs were run using four compilers. They were a SIMULA compiler, the DEC-10 Ratfor compiler, a Y compiler specific to the DEC-10, and the retargetable Y compiler.

	Y Compiler	DEC-10 Y	Ratfor	SIMULA
8q	29.0	28.7	n/a	66.0
ctoi	37.7	55.3	n/a	82.6
ictoi	29.7	26.8	20.5	84.6

The Y compiler produces better code for ctoi than the DEC-10 specific Y compiler. This is because of a different code generation strategy for handling characters. This is one of the advantages of an easily retargetable compiler. Different code generation strategies can be explored to find the most efficient.

The Ratfor compiler's code for ctoi runs faster because global register optimization assigns i and sum to registers. The code produced by the SIMULA compiler is much slower than that of the other compilers because of the calling sequence and the poor code generated.

6.1.2 Compiler Speed

The retargetable Y compilers currently only run on the PDP-11/70, so the only basis of comparison is between compilers that run on the PDP-11/70. These are the Unix C compiler, the Portable C Compiler, and the VU Pascal compiler. The following table compares the speed of the PDP-11 Y compiler to these three compilers.

	Y Compiler	Port. C	Unix C	VU Pascal
instructions/sec	12	57	74	50
source lines/sec	6	22	33	10

The current implementation of Y is six times slower than the Unix C compiler. Much of this difference is due to the organization of the Y compiler. In order to overcome the address limitations of the PDP-11/70, the compiler was split into five filters [KERN79], each of which produces output that is human readable. While this encoding is an asset while developing and debugging the compiler, it is not necessary in a production environment. Consequently a large percentage of the compiler's time is wasted doing I/O and rebuilding data structures. Of the five filters only the first, the compiler front end, has been extensively optimized. Speeding up the compiler is one area for further research. It should be possible to reduce the execution time by at least a factor of two.

6.1.3 Machine Applicability

Building production-quality compilers using a retargetable peephole optimizer is applicable to a wide range of machines. Y cross-compilers have been developed for four machines: the CDC Cyber 175, the DEC-10, the PDP-11, and Intel's 8080 microprocessor. These machines represent a broad range of architectures. The CDC Cyber 175 and the DEC-10 are two very different types of large-scale mainframes. The PDP-11 is a sixteen-bit minicomputer, while the 8080 is an eight-bit microprocessor. As described below, it even appears possible to use PO to build compilers for unconventional architectures such as as array processors and pipelined machines.

6.1.4 Implementation Effort

Retargeting the compiler for a new machine requires the following steps.

1. Rewrite the code expander for the target machine. It can generate very naive code.
2. Write patterns for Cacher that identify registers and rank addressing modes according to speed.
3. Write a description of the target machine's architecture.
4. Provide Assigner with information about the machine's register set; how many registers of each type and which can be assigned.
5. Provide a skeletal run-time support system.

Each of the above tasks requires usually less than a day to accomplish. The author implemented a cross compiler and a skeleton run-time system for the CDC Cyber machines in five man-days. A cross compiler for Intel's 8080 required the author three days.

6.2 Areas for Further Research

One area for further research is to make the compilers built using PO run faster. There are several obvious changes that would make the compiler more efficient. As mentioned above, a more compact representation of the intermediate files would produce a large savings. A even larger savings could be made if the four filters comprising the back end of the compiler were merged into one program. Such a compiler is planned for a VAX-11/780 implementation. This will remove most encoding, decoding, I/O, and memory management overhead.

As mentioned in Chapter 4, the speed of PO hinges on the speed of the recognizer and transducer built from the machine description. The current implementation uses Lex to generate the recognizer and transducer. Further research is needed in the area of encoding and recognizing valid machine instructions.

PO currently only performs local optimizations. As the previous timings for various compilers show, a large payoff may be gained by performing global optimizations. Many optimizations traditionally performed at a high level, such as register allocation and common subexpression removal within basic blocks, are now accomplished by PO. PO may also be able to do certain global optimizations.

PO currently allocates registers to loop indices, but leaves the load at the top of the loop and the store at the bottom. By moving the load and store out of the loop, it may be possible to achieve a measure of global register allocation. Other optimizations such as code motion and strength reduction may also be possible and useful at this low level. For example, changing a loop index to count by two or four on a byte-addressable machine is properly a machine-dependent optimization.

Generating code for unconventional architectures such as array processors and machines with a high degree of parallelism is difficult. By supplying PO with additional information about the processor through the machine description, it may be possible to generate code for the processor in the conventional way. PO would apply transformations to the naive code, producing code that takes advantage of the unconventional architecture. For example, on a machine with multiple arithmetic processors, code can be generated that uses only one processor with each instruction. PO can replace pairs of instructions that use different processors with a singleton that uses both.

On the CDC Cyber machines with multiple functional units, it is advantageous to use the results as far as possible from their computation. This allows the processor to proceed without having to wait for a result to be computed. PO could accomplish such instruction scheduling by reordering code. PO, when eliminating redundant code, could construct lists of the intermediate steps of computations. By applying scheduling algorithms to such lists, PO could rearrange the steps of the computation to avoid processor waits.

6.3 Contributions

This dissertation has presented a technique for the rapid implementation of production-quality compilers. This has been accomplished through the use of a machine-independent retargetable peephole optimizer, PO. PO simplifies many of the tasks typically associated with developing compilers, in particular retargetable compilers.

The design of abstract machines is simplified. Instead of using large complex 'union' abstract machines, simple 'intersection' machines can be used. The abstract machine mode no longer needs elements from the set of target architectures. Intersection machines simplify code generation by eliminating most case analysis typically performed by the compiler or code expander.

Intersection machines also simplify code expansion. Code expanders are smaller because the abstract machine is smaller. In addition, they are conceptually simpler. Tasks typically performed by code expanders such as register allocation and register assignment are performed by PO. With very naive code expanders that perform a simple macro replacement of abstract opcodes, production-quality code is still obtained.

A notation for describing machine architectures has been developed that is flexible yet simple. Using the notation, someone familiar with the target machine can write a description in a day. Several machines with different architectures have been described using the notation. The machine description drives the retargetable peephole optimizer PO.

PO simplified the optimization phase of the compiler by collecting several disparate optimizations and generalizing them as peephole optimizations. Register allocation and assignment, removal of unreachable code, removal of common subexpressions, and exploitation of special case instructions and addressing modes are all performed by PO.

Traditional compilers perform as many optimizations as they can before code generation. Optimization is performed as far as possible from the target machine to avoid machine-dependent complications. PO shows that optimization can be performed at a low-level, and that it is beneficial to do so. Traditional optimizations may be done more thoroughly and completely when information about the target machine is available.

Appendix A
The Y Intermediate Language

In the following description of the Y postfix operators, CGS refers to the code generator's stack and RTS refers to the runtime stack. References to the CGS refer to the cell or register that simulates that CGS location. For instance, on the PDP-11 if the top of the CGS stack is register 1, the operation

```
        ADDA 1
```

would cause the register transfer

```
        r[1] ← r[1] + 1;
```

to be emitted.

The increment (++) and decrement (−−) operators are similiar to those defined for C, and are used to manipulate the CGS and RTS stacks. These stacks are assumed to grow up in memory. If this is not convenient for the particular machine environment (particularly for the RTS), the usage of the increment and decrement operators may be changed to suit the environment. MEMC, MEMI, and MEMR are character, integer, and real memories.

ADDA *n*
  add a constant *n* to an address: *n* + CGS[top] → CGS[top].

ADDCA *n*
  add a constant *n* to a character address of a string: *n* + CGS[top] → CGS[top].

ADDI
  add two integers: CGS[top−−] + CGS[top−−] → CGS[++top].

ADDR
  add two reals: CGS[top−−] + CGS[top−−] → CGS[++top].

ADDRESS *n*
  generate a cell containing the address of label *n*.

AND
  and two integers: CGS[top−−] & CGS[top−−] → CGS[++top].

ARGA
  push an address argument onto the runtime stack: CGS[top−−] → RTS[++top].

ARGCA
  push a character address argument onto the runtime stack: CGS[top−−] → RTS[++top].

ARGC
  push a character argument onto the runtime stack: CGS[top−−] → RTS[++top].

ARGI
  push an integer argument onto the runtime stack: CGS[top−−] → RTS[++top].

ARGR
  push a real argument onto the runtime stack: CGS[top−−] → RTS[++top].

BCMPIE

compare two integers and generate TRUE if they are equal, otherwise generate FALSE: CGS[top−−] = CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPIGE

compare two integers and generate TRUE if the first is greater than or equal to the second, otherwise generate FALSE: CGS[top−−] ≥ CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPIGT

compare two integers and generate TRUE if the first is greater than the second, CGS[top−−] > CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPILE

compare two integers and generate TRUE if the first is less than or equal to the second, otherwise generate FALSE: CGS[top−−] ≤ CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPILT

compare two integers and generate TRUE if the first is less than the second, otherwise generate FALSE: CGS[top−−] < CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPIN

compare two integers and generate TRUE if they are not equal, otherwise generate FALSE: CGS[top−−] ≠ CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRE

compare two reals and generate TRUE if they are equal, otherwise generate FALSE: CGS[top−−] = CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRGE

compare two reals and generate TRUE if the first is greater than or equal to the second, otherwise generate FALSE: CGS[top−−] ≥ CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRGT

compare two reals and generate TRUE if the first is greater than the second, otherwise generate FALSE: CGS[top−−] > CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRLE

compare two reals and generate TRUE if the first is less than or equal to the second, otherwise generate FALSE: CGS[top−−] ≤ CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRLT

compare two reals and generate TRUE if the first is less than the second, otherwise generate FALSE: CGS[top−−] < CGS[top−−] then TRUE else FALSE → CGS[++top].

BCMPRNE

compare two reals and generate TRUE if they are not equal, otherwise generate FALSE: CGS[top−−] ≠ CGS[top−−] then TRUE else FALSE → CGS[++top].

BEG *id*

beginning of module *id*.

BSS *n*

reserve *n* units of storage. An unit of storage is particular to each machine.

CALLIF *id,n,size*

call an integer function *id* with *n* arguments totaling *size* units of storage.

CALLRF *id,n,size*

call a real function *id* with *n* arguments totaling *size* units of storage.

CALLP *id,n,size*
call a procedure *id* with *n* arguments totaling *size* units of storage.

COM
logical complement of an integer: ~CGS[top−−] → CGS[++top].

DIVI
divide two integers: CGS[top−−] / CGS[top−−] → CGS[++top].

DIVR
divide two reals: CGS[top−−] / CGS[top−−] → CGS[++top].

END
end of module.

ENDEXPR
end of expression; can be used in conventional code expanders to free resources such as registers and temporaries.

ENT *id*
exported identifier; *id* is accessible from other modules.

EPDEF
end of definition of procedure prolog.

EPROC
end of a procedure.

FPARM *id,size*
declare a formal parameter *id* occupying *size* units of storage.

EXT *id*
imported identifier; *id* is defined in another module.

FIX
convert from real to integer: FIX(CGS[top−−]) → CGS[++top].

FLOAT
convert from integer to real: FLOAT(CGS[top−−]) → CGS[++top].

GLBL *id*
declare global identifier *id*.

INDEXC *size*
compute the character address of an element in a character array given a character address and an index; each element of the array occupies *size* units of storage.

INDEX *size*
compute the address of an element in a array given an address and an index; each element of the array occupies *size* units of storage.

INT *n*
reserve storage for an integer and initialize it to *n*.

JCMPIE *n*
compare two integers and branch to label *n* if they are equal:
CGS[top−−] = CGS[top−−] then *n* else PC + 1 → PC.

JCMPIGE *n*
compare two integers and branch to label *n* if the first is greater than or equal to the second:
CGS[top−−] ≥ CGS[top−−] then *n* else PC + 1 → PC.

JCMPIGT *n*
compare two integers and branch to label *n* if the first is greater than the second:
CGS[top−−] > CGS[top−−] then *n* else PC + 1 → PC.

JCMPILE *n*
compare two integers and branch to label *n* if the first is less than or equal to the second:
CGS[top−−] ≤ CGS[top−−] then *n* else PC + 1 → PC.

JCMPILT *n*
compare two integers and branch to label *n* if the first is less than the second:
CGS[top−−] < CGS[top−−] then *n* else PC + 1 → PC.

JCMPIN *n*
compare two integers and branch to label *n* if they are not equal:
CGS[top−−] ≠ CGS[top−−] then *n* else PC + 1 → PC.

JCMPRE *n*
compare two reals and branch to label *n* if they are equal:
CGS[top−−] = CGS[top−−] then *n* else PC + 1 → PC.

JCMPRGE *n*
compare two reals and branch to label *n* if the first is greater than or equal to the second:
CGS[top−−] ≥ CGS[top−−] then *n* else PC + 1 → PC.

JCMPRGT *n*
compare two reals and branch to label *n* if the first is greater than the second:
CGS[top−−] > CGS[top−−] then *n* else PC + 1 → PC.

JCMPRLE *n*
compare two reals and branch to label *n* if the first is less than or equal to the second:
CGS[top−−] ≤ CGS[top−−] then *n* else PC + 1 → PC.

JCMPRLT *n*
compare two reals and branch to label *n* if the first is less than the second:
CGS[top−−] < CGS[top−−] then *n* else PC + 1 → PC.

JCMPRN *n*
compare two reals and branch to label *n* if they are not equal:
CGS[top−−] ≠ CGS[top−−] then *n* else PC + 1 → PC.

JMP *n*
jump to label *n*.

LLAB *n*
generate local label *n*.

LCL *id,size*
declare local identifier *id* occupying *size* units of storage.

MOD
generate modulus of two integers: CGS[top−−] % CGS[top−−] → CGS[++top].

MULI
multiply two integers: CGS[top−−] * CGS[top−−] → CGS[++top].

MULR
multiply two reals: CGS[top−−] * CGS[top−−] → CGS[++top].

NEGI
negate an integer: −CGS[top] → CGS[top].

NEGR
negate a real: −CGS[top] → CGS[top].

OR
or two integers: CGS[top−−] | CGS[top−−] → CGS[++top].

POPC
store a character: CGS[top] → MEMC(CGS[top−1]); CGS[top−−] → CGS[top].

POPI
store an integer: CGS[top] → MEMI(CGS[top−1]); CGS[top−−] → CGS[top].

POPR
store a real: CGS[top] → MEMR(CGS[top−1]); CGS[top−−] → CGS[top].

PROC *id*
begin procedure or function *id*.

PUSHGA *id*
push the address of a global: *addressof(id)* → CGS[++top].

PUSHGCA *id*
push the character address of a global: *addressof(id)* → CGS[++top].

PUSHLA *id*
push the address of a local: *addressof(id)* → CGS[++top].

PUSHLCA *id*
push the character address of a local: *addressof(id)* → CGS[++top].

PUSHCCA *c1,...,cn*
push the character address of character constant comprised of ASCII characters *c1,...,cn*: *addressof(c1,...,cn)* → CGS[++top].

PUSHIC *n*
push an integer constant: *n* → CGS[++top].

PUSHRC *n*
push a real constant: *n* → CGS[++top].

PUSHC
push a character: MEMC(CGS[top]) → CGS[top].

PUSHI
push an integer: MEMI(CGS[top]) → CGS[top].

PUSHR
push a real: MEMR(CGS[top]) → CGS[top].

PUSHRP *id*
push the address of a reference parameter: *addressof(id)* → CGS[++top].

PUSHRPC *id*
push the character address of a reference parameter: *addressof(id)* → CGS[++top].

PUSHVP *id*
push the address of a value parameter: *addressof(id)* → CGS[++top].

REAL *x*
reserve storage for a real and initialize it to *x*.

RET
return from a procedure.

RETI
return an integer.

RETR
return a real.

SEG *n*
switch to segment *n*.

SETC *n*
begin calling sequence for *n* arguments.

SHIFTL
logical left shift: CGS[top--] << CGS[top--] → CGS[++top].

SHIFTR
logical right shift: CGS[top--] >> CGS[top--] → CGS[++top].

STMTBGN *n*
begin statement *n*; can be used to generate debugging code if desired.

STMTEND *n*
end statement *n*; can be used to generate debugging code if desired.

STR *n,c1,...,cm*
reserve storage for a string of *n* characters and initialize it to *c1,...,cm*

SUBI
subtract two integers: CGS[top--] - CGS[top--] → CGS[++top].

SUBR
subtract two reals: CGS[top--] - CGS[top--] → CGS[++top].

SWITCHC *lb,ub,n*
case switch: jump to label *n* if switch value on the top of the CGS is between *lb* and *ub*.

SWITCHJ *lb,ub,n*
jump switch: if switch value on the top of the CGS is between *lb* and *ub* jump through dispatch table that immediately follows the instruction; otherwise, jump to default label *n*.

TRASHG
used by compiler to pass information to Cacher about which globals to remove from the cache.

TRASHL
used by compiler to pass information to Cacher about which locals to remove from the cache.

TRASHP
used by compiler to pass information to Cacher about which parameters to remove from the cache.

Notes

For case statements, the Y compiler decides whether to generate code that simulates an else-if chain or a branch table. The SWITCHJ opcode is used to generate branch table code; SWITCHCs are used to generate else-if code.

The EQI and EQR opcodes along with the other relationals that compute a value should actually not be included in an 'intersection' machine. For certain machines, however, it is possible to generate very efficient code when comparing values. At this time, PO is not able to convert naive code to this efficient code for some machines. Consequently, these operations have been included.

Appendix B
PDP-11 Machine Description

The following is a complete PDP-11 description except for floating-point operations.

```
RN              [0-7]+
XDNT            ((("_"|"L")[A-Za-z0-9_]+)|(-?[0-9]+))
IDNT            {XDNT}(" "[-+]" "{XDNT})*
LABEL           "L"[L0-9]+
%%
RG      :=      r[RN]                   :=      rRN

        :=      0                       :=      $0

        :=      1                       :=      $1

LB      :=      LABEL                   :=      LABEL

ID      :=      IDNT                    :=      $IDNT

WD      :=      m[IDNT]                 :=      IDNT
                m[r[RN] + IDNT]         :=      IDNT(rRN)
                m[r[RN]++]              :=      (rRN)+
                m[--r[RN]]              :=      -(rRN)
                m[r[RN]]                :=      (rRN)
                m[m[r[RN]++]]           :=      *(rRN)+
                m[m[--r[RN]]]           :=      *-(rRN)
                m[m[r[RN] + IDNT]]      :=      *IDNT(rRN)
                m[m[r[RN]]]             :=      *(rRN)
                m[m[IDNT]]              :=      *IDNT

BT      :=      b[IDNT]                 :=      IDNT
                b[r[RN] + IDNT]         :=      IDNT(rRN)
                b[r[RN]++]              :=      (rRN)+
                b[--r[RN]]              :=      -(rRN)
                b[r[RN]]                :=      (rRN)
                b[m[r[RN]++]]           :=      *(rRN)+
                b[m[--r[RN]]]           :=      *-(rRN)
                b[m[r[RN] + IDNT]]      :=      *IDNT(rRN)
                b[m[r[RN]]]             :=      *(rRN)
                b[m[IDNT]]              :=      *IDNT

SO      :=      <<

IMP     :=      ->

NZ      :=      NZ

PC      :=      PC
```

```
YRT     :=      yret

SXT     :=      SXT

RL      :=      ==                              :=      eq
                !=                              :=      ne
                >=                              :=      ge
                <=                              :=      le
                <                               :=      lt
                >                               :=      gt
%%
RG1     :=      RG
RG2     :=      RG
DSTW    :=      RG|WD
SRCW    :=      RG|ID|WD|0|1
DSTB    :=      RG|BT
SRCB    :=      RG|ID|BT|0|1
%%
NZ = DSTW ? 0;                                          :=      tst     DSTW
NZ = DSTB ? 0;                                          :=      tstb    DSTB
NZ = DSTW ? SRCW;                                       :=      cmp     DSTW,SRCW
NZ = DSTB ? SRCB;                                       :=      cmpb    DSTB,SRCB
DSTW = SXT;                                             :=      sxt     DSTW
DSTW = 0;NZ = 0 ? 0;                                    :=      clr     DSTW
DSTB = 0;NZ = 0 ? 0;                                    :=      clrb    DSTB
DSTW = SRCW;NZ = SRCW ? 0;                              :=      mov     SRCW,DSTW
DSTB = SRCB;NZ = SRCB ? 0;                              :=      movb    SRCB,DSTB
DSTW = DSTW + 1;NZ = DSTW + 1 ? 0;                      :=      inc     DSTW
DSTW = DSTW - 1;NZ = DSTW - 1 ? 0;                      :=      dec     DSTW
DSTW = DSTW + SRCW;NZ = DSTW + SRCW ? 0;                :=      add     DSTW,SRCW
DSTW = DSTW - SRCW;NZ = DSTW - SRCW ? 0;                :=      sub     DSTW,SRCW
DSTW = DSTW & ~SRCW;NZ = DSTW & ~SRCW ? 0;              :=      bic     DSTW,SRCW
DSTW = DSTW $ SRCW;NZ = DSTW $ SRCW ? 0;                :=      bis     DSTW,SRCW
DSTW = -DSTW;NZ = -DSTW ? 0;                            :=      neg     DSTW
DSTW = ~DSTW;NZ = ~DSTW ? 0;                            :=      com     DSTW
PC = LB;                                                :=      jbr     LB
PC = YRT;                                               :=      jmp     YRT
PC = ID(RG);                                            :=      jmp     *ID(RG)
PC = NZ RL 0 IMP LB | PC;                               :=      jRL     LB
DSTW = DSTW SO 1;NZ = DSTW SO 1 ? 0;                    :=      asl     DSTW
RG = RG SO SRCW;NZ = RG SO SRCW ? 0;                    :=      {
  !strcmp(SRCW, "-1"): asr RG
  ash SRCW,RG
  }
DSTW = DSTW SO ID;NZ = DSTW SO ID ? 0;                  :=      {
  strcmp(ID, "-1"): ABORT
  asr DSTW
  }
RG = RG * SRCW;NZ = RG * SRCW ? 0;                      :=      {
  !odd(RG) || !ispwr2(SRCW): ABORT
  !strcmp(SRCW, "2"): asl RG
  pwr2(SRCW): ash SRCW,RG
  mul SRCW,RG
  }
RG1 = RG1 RG2 / SRCW;RG2 = RG1 RG2 % SRCW;NZ = RG1 RG2 / SRCW ? 0;   := {
  !pair(RG1, RG2): ABORT
  div SRCW,RG1
  }
```

Appendix C
DEC-10 Machine Description

The following is a DEC-10 description except for floating-point operations.

```
IR              [1-7][0-7]*
REG             [0-7]+
XDNT            ((([L%$.][A-Za-z0-9]+)|(-?[0-9]+))
IDNT            {XDNT}(" "[-+]" "{XDNT})*
PM              [-+]
%%
ZRG     :=      r[0]                        :=      0

IR      :=      r[IR]                       :=      IR

MEM     :=      m[IDNT]                     :=      IDNT
                m[r[IR] PM IDNT]            :=      PM IDNT(IR)
                m[m[IDNT]]                  :=      @IDNT
                m[m[r[IR]]]                 :=      @(IR)
                m[m[r[IR] PM IDNT]]         :=      @PM IDNT(IR)

        :=      0                           :=      0

        :=      1                           :=      1

E       :=      IDNT                        :=      IDNT
                r[IR] PM IDNT               :=      PM IDNT(IR)
                m[IDNT]                     :=      @IDNT
                m[r[IR]]                    :=      @(IR)
                m[r[IR] PM IDNT]            :=      @PM IDNT(IR)

EQ      :=      ==                          :=      e

NE      :=      !=                          :=      n

ORL     :=      >=                          :=      ge
                <=                          :=      le
                <                           :=      l
                >                           :=      g

PUSH    :=      PUSH

LDB     :=      LDB

DPB     :=      DPB

IBP     :=      IBP

PC      :=      PC
```

```
PCP     :=      PC+1

IMP     :=      ->

SF      :=      <<

HRL     :=      HRL

MI      :=      @-
%%
RL      :=      EQ|NE|ORL
IM      :=      E|0|1
R       :=      ZRG|IR
R1      :=      IR
M       :=      ZRG|IR|MEM
AO      :=      +|@
SO      :=      -|MI
LB      :=      E
%%
R = M;                                          :=      move      R,M
M = R;                                          :=      movem     R,M
R = R AO M;                                     :=      add       R,M
R = R SO M;                                     :=      sub       R,M
R = R * M;                                      :=      imul      R,M
R = R & M;                                      :=      and       R,M
R = R $ M;                                      :=      or        R,M
M = R AO M;                                     :=      addm      R,M
M = R SO M;                                     :=      subm      R,M
M = R * M;                                      :=      imulm     R,M
M = R / M;                                      :=      idivm     R,M
M = R & M;                                      :=      andm      R,M
M = R $ M;                                      :=      iorm      R,M
R = R AO M;M = R AO M;                          :=      addb      R,M
R = R SO M;M = R SO M;                          :=      subb      R,M
R = R * M;M = R * M;                            :=      imulb     R,M
R = R & M;M = R & M;                            :=      andb      R,M
R = R $ M;M = R $ M;                            :=      iorb      R,M
R = R AO IM;                                    :=      addi      R,IM
R = R SO IM;                                    :=      subi      R,IM
R = R * IM;                                     :=      imuli     R,IM
R = R & IM;                                     :=      andi      R,IM
R = R $ IM;                                     :=      ori       R,IM
R = 0;                                          :=      setz      R
R = IM;                                         :=      movei     R,IM

R = R / M;M = R / M;R1 = R % M;                 :=      {
  !pair(R, R1): ABORT
  idivb    R,M
  }

R = R / M;R1 = R % M;                           :=      {
  !pair(R, R1): ABORT
  idiv    R,M
  }

R = R / IM;R1 = R % IM;                         :=      {
  !pair(R, R1): ABORT
  idivi    R,M
  }
```

```
PUSH(R) = M;                                          :=    push      R,M
DPB(M) = R;                                           :=    dpb       R,M
R = LDB(M);                                           :=    ldb       R,M
M = IBP(M);                                           :=    ibp       M
R = LDB(IBP(M));                                      :=    ildb      R,M
DPB(IBP(M)) = R;                                      :=    idpb      R,M
R = HRL(IM,R);                                        :=    hrli      R,IM
R = HRL(M,R);                                         :=    hrl       R,IM
M = 0;                                                :=    setzm     M
M = 0;R = 0;                                          :=    setzb     R,M
PC = R RL M IMP PCP| PC;                              :=    camRL     R,M
PC = M RL 0 IMP PCP| PC;                              :=    skipRL    M
PC = M RL 0 IMP PCP| PC;R = M;                        :=    skipRL    R,M
PC = R RL IM IMP PCP| PC;                             :=    caiRL     R,IM
M = M @ 1;                                            :=    aos       M
M = M MI 1;                                           :=    sos       M
PC = LB;                                              :=    jrst      LB
R = R @ 1;PC = LB;                                    :=    aoja      R,LB
R = R MI 1;PC = LB;                                   :=    soja      R,LB
M = M @ 1;R = M @ 1;                                  :=    aos       R,M
M = M MI 1;R = M MI 1;                                :=    sos       R,M
R = R SF IM;                                          :=    lsh       R,IM
R = -M;                                               :=    movn      R,M
R = ~M;                                               :=    setcm     R,M
M = -R;                                               :=    movnm     R,M
M = ~R;                                               :=    setcam    R,M
R = -IM;                                              :=    movni     R,IM
R = ~IM;                                              :=    setcmi    R,IM
M = -M;                                               :=    movns     M
M = ~M;                                               :=    setcmm    M
PC = R & M EQ 0 IMP PCP| PC;                          :=    tdne      R,M
PC = R & M NE 0 IMP PCP| PC;                          :=    tdnn      R,M
R = R & ~M;                                           :=    tdz       R,M
R = R $ M;                                            :=    tdo       R,M
R = R $ M;PC = PCP;                                   :=    tdoa      R,M
R = R & ~M;PC = PCP;                                  :=    tdza      R,M
PC = R RL 0 IMP LB| PC;                               :=    jumpRL    R,LB
PC = ID(R);                                           :=    jrst      @ID(R)
PC = R & M EQ 0 IMP PCP| PC;R = R $ M;                :=    tdoe      R,M
PC = R & M NE 0 IMP PCP| PC;R = R $ M;                :=    tdon      R,M
R = R @ 1;PC = R @ 1 RL 0 IMP LB| PC;                 :=    aojRL     R,LB
R = R MI 1;PC = R MI 1 RL 0 IMP LB| PC;               :=    sojRL     R,LB
M = M @ 1;R1 = M @ 1;PC = M @ 1 RL 0 IMP PCP| PC;     :=    aosRL     R1,M
M = M MI 1;R1 = M MI 1;PC = M MI 1 RL 0 IMP PCP| PC;  :=    sosRL     R1,M
PC = R & M EQ 0 IMP PCP| PC;R = R & ~M;               :=    tdze      R,M
PC = R & M NE 0 IMP PCP| PC;R = R & ~M;               :=    tdzn      R,M
```

Appendix D

Cyber 175 Machine Description

The following is a complete CDC Cyber description with enough floating-point operations to describe integer division.

```
LDR             [67]
R1              [0-7]+
R2              [0-7]+
XDNT            (("L"[0-9]+)|([A-Za-z][A-Za-z0-9_]*[$#%])|(-?[0-9]+"d"))
IDNT            {XDNT}([-+]{XDNT})*
LABEL           "L"[L0-9]+
%%
STN     :=      LDR             :=          LDR

RN      :=      R1              :=          R1

LB      :=      LABEL           :=          LABEL

ID      :=      IDNT            :=          IDNT

ADRS    :=      a[R1]+IDNT      :=          aR1+IDNT
                b[R1]+IDNT      :=          bR1+IDNT
                x[R1]+IDNT      :=          xR1+IDNT
                x[R1]+b[R2]     :=          xR1+bR2
                a[R1]+b[R2]     :=          aR1+bR2
                a[R1]-b[R2]     :=          aR1-bR2
                b[R1]+b[R2]     :=          bR1+bR2
                b[R1]-b[R2]     :=          bR1-bR2

BRL     :=      ==              :=          zr
                !=              :=          nz
                >=              :=          ge
                <=              :=          le
                <               :=          ng

RL      :=      == 0            :=          zr
                != 0            :=          nz
                >= 0            :=          pl
                < 0             :=          ng

UPCK    :=      UPCK

PCK     :=      PCK

NRM     :=      NRM

MASK    :=      MASK

SL      :=      <<

SR      :=      >>
```

```
IMP     :=      ->

PC      :=      PC

YRT     :=      yret

MI      :=      @-

PL      :=      @
%%
RN1     :=      STN|RN
RN2     :=      STN|RN
RN3     :=      STN|RN
RN4     :=      STN|RN
ADDR    :=      LB|ID|ADRS
LAB     :=      LB|YRT
%%
x[RN1] = x[RN2];                                        :=    bxRN1     xRN2
x[RN1] = x[RN2] PL x[RN3];                              :=    ixRN1     xRN2+xRN3
x[RN1] = x[RN2] MI x[RN3];                              :=    ixRN1     xRN2-xRN3
x[RN1] = x[RN2] $ x[RN3];                               :=    bxRN1     xRN2+xRN3
x[RN1] = x[RN2] & x[RN3];                               :=    bxRN1     xRN2*xRN3
x[RN1] = x[RN2] * x[RN3];                               :=    ixRN1     xRN2*xRN3
x[RN1] = x[RN2] / x[RN3];                               :=    fxRN1     xRN2/xRN3
x[RN1] = x[RN2] # x[RN3];                               :=    bxRN1     xRN2-xRN3
x[RN1] = ~x[RN2] & x[RN3];                              :=    bxRN1     -xRN2*xRN3
x[RN1] = ~x[RN2] $ x[RN3];                              :=    bxRN1     -xRN2+xRN3
x[RN1] = ~x[RN2] # x[RN3];                              :=    bxRN1     -xRN2-xRN3
x[RN1] = ~x[RN2];                                       :=    bxRN1     -xRN2
x[RN1] = x[RN1] SL ID;                                  :=    lxRN1     ID
x[RN1] = x[RN1] SR ID;                                  :=    axRN1     ID
x[RN1] = x[RN2] SL b[RN3];                              :=    lxRN1     xRN2,bRN3
x[RN1] = x[RN2] SR b[RN3];                              :=    axRN1     xRN2,bRN3
x[RN1] = MASK(ID);                                      :=    mxRN1     ID
m[ADDR] = x[STN];a[STN] = ADDR;                         :=    saSTN     ADDR
x[RN1] = m[ADDR];a[RN1] = ADDR;                         :=    saRN1     ADDR
m[b[RN1]] = x[STN];a[STN] = b[RN1];                     :=    saSTN     bRN1
m[a[RN1]] = x[STN];a[STN] = a[RN1];                     :=    saSTN     aRN1
m[x[RN1]] = x[STN];a[STN] = x[RN1];                     :=    saSTN     xRN1
x[RN1] = m[b[RN2]];a[RN1] = b[RN2];                     :=    saRN1     bRN2
x[RN1] = m[a[RN2]];a[RN1] = a[RN2];                     :=    saRN1     aRN2
x[RN1] = m[x[RN2]];a[RN1] = x[RN2];                     :=    saRN1     xRN2
b[RN1] = ADDR;                                          :=    sbRN1     ADDR
b[RN1] = b[RN2];                                        :=    sbRN1     bRN2
b[RN1] = a[RN2];                                        :=    sbRN1     aRN2
b[RN1] = x[RN2];                                        :=    sbRN1     xRN2
x[RN1] = b[RN2];                                        :=    sxRN1     bRN2
x[RN1] = a[RN2];                                        :=    sxRN1     aRN2
PC = x[RN1] RL IMP LAB | PC;                            :=    RL        xRN1,LAB
PC = LAB;                                               :=    eq        LAB
PC = LAB(b[RN1]);                                       :=    jp        bRN1+LB
x[RN1] = PCK(x[RN2],b[RN3]);                            :=    pxRN1     xRN2,bRN3
x[RN1] = NRM(x[RN2]);b[RN3] = NRM(x[RN2]);              :=    nxRN1     xRN2,bRN3
x[RN1] = UPCK(x[RN2]);b[RN3] = UPCK(x[RN2]);            :=    uxRN1     xRN2,bRN3
PC = b[RN1] BRL b[RN2] IMP LAB | PC;                    :=    BRL       bRN1,bRN2,LAB
x[RN1] = ADDR;                                          :=    {
  !strcmp(ADDR, "0d"):    mxRN1    0
  !strcmp(ADDR, "1d"):    sxRN1    b1
  sxRN1    ADDR
  }
```

Appendix E
8080 Machine Description

The following is the 8080 description. Many of the eight-bit operations have been omitted because Y does not use them.

```
XDENT          ((([A-Za-z][A-Za-z0-9_]*"$")|(-?[0-9]+))
IDENT          {XDENT}(" "[-+]" "{XDENT})*
LABEL          "L"[0-9]+
%%
LAB     :=     LABEL                        :=    LABEL

BC      :=     r[bc]                        :=    b

DE      :=     r[de]                        :=    d

HL      :=     r[hl]                        :=    h

SP      :=     r[sp]                        :=    sp

AR      :=     r[a]                         :=    a

RG      :=     r[b]                         :=    b
               r[c]                         :=    c
               r[d]                         :=    d
               r[e]                         :=    e
               r[h]                         :=    h
               r[l]                         :=    l

        :=     0                            :=    0

        :=     1                            :=    1

IDT     :=     IDENT                        :=    IDENT

RLC     :=     EQ                           :=    eq
               CMP                          :=    cmp

RL      :=     !=                           :=    ne
               ==                           :=    z
               >=                           :=    p
               <                            :=    m

SHFT    :=     <<                           :=    rlc
               >                            :=    rrc

PUSH    :=     PUSH

POP     :=     POP
```

```
TOP     :=      TOP

ZS      :=      ZS

IMP     :=      ->

PC      :=      PC
%%
ID      :=      IDT|0|1
RG1     :=      AR|RG
RG2     :=      AR|RG
RP      :=      BC|DE|HL|SP
%%
RG1 = RG2;                                  :=      mov     RG1,RG2
RG1 = b[HL];                                :=      mov     RG1,m
b[HL] = RG1;                                :=      mov     m,RG1
RG1 = ID;                                   :=      mvi     RG1,ID
b[HL] = ID;                                 :=      mvi     m,ID
RP = ID;                                    :=      lxi     RP,ID
AR = b[ID];                                 :=      lda     ID
b[ID] = AR;                                 :=      sta     ID
HL = m[ID];                                 :=      lhld    ID
m[ID] = HL;                                 :=      shld    ID
AR = b[RP];                                 :=      ldax    RP
b[RP] = AR;                                 :=      stax    RP
HL = DE;DE = HL;                            :=      xchg
RG1 = RG1 + 1;ZS = RG1 + 1 ? 0;             :=      inr     RG1
RG1 = RG1 - 1;ZS = RG1 - 1 ? 0;             :=      dcr     RG1
b[HL] = b[HL] + 1;ZS = b[HL] + 1 ? 0;       :=      inr     M
b[HL] = b[HL] - 1;ZS = b[HL] - 1 ? 0;       :=      dcr     M
RP = RP + 1;                                :=      inx     RP
RP = RP - 1;                                :=      dcx     RP
HL = HL + RP;                               :=      dad     RP
PC = LAB;                                   :=      jmp     LAB
PC = ZS RL 0 IMP LAB | PC;                  :=      jRL     LAB
PC = HL;                                    :=      pchl
SP = PUSH(RP);                              :=      push    RP
SP = POP(SP);RP = TOP(SP);                  :=      pop     RP
SP = HL;                                    :=      sphl
HL = HL & DE;                               :=      call    and.
HL = RP * RP;                               :=      call    mul.
HL = ~HL;                                   :=      call    cma.
DE = -DE;                                   :=      call    tcma.
HL = DE / HL;                               :=      call    div.
DE = DE % HL;                               :=      call    mod.
HL = DE SHFT HL;                            :=      call    SHFT.
ZS = RLC(DE,HL);                            :=      call    RLC.
HL = RLC(DE,HL);                            :=      call    RLC.
```

Appendix F
The Caching Algorithm

The steps below are performed for each bounded block of register transfer lists (RTLs). A bounded block is a section of code with only one entry. C is the set of equivalence classes. At the start of each block, the set C is assumed to be empty. *Dst* and *src* are the destination and source of each RTL as it is processed. At the end of a block, the modified list of RTLs is output. The modified list includes identification of dead variables as well as linkage information denoting where cells were set and then used.

1. *Dst* is put into canonical form by first stripping off the outermost name and any brackets. Call this string *adst*. For each register that is a substring of *adst*, do the following. Call the register r. For each $A \in C$, if $r \in A$ then substitute b, where $b \in A$, and for all other $z \in A$, $b <_t z$. Add the RTL to r's use list. Replace the original name and the outermost brackets. Call the resulting string *cdst*. Add the RTL to *dst*'s set list.
2. For each register that is a substring of *src*, do the following. Call the register r. For each $A \in C$, if $r \in A$ then substitute b, where $b \in A$ and for all other $z \in A$, $b <_t z$. Add the RTL to r's use list. Call the resulting string *csrc*.
3. For each $A \in C$, if $csrc \in A$, then find an $a \in A$ such that $a <_c b$ for all other $b \in A$. Substitute a for *src* in the current RTL. Add the current RTL to a's use list. Apply the function *del* to *src*. *Del* is defined below.
4. For each $A \in C$, if $adst \in A$, then find an $a \in A$ such that $a <_c b$ for all other $b \in A$. Substitute a for the address calculation of *dst*. Substitute this new string for *dst* in the current RTL. Add the current RTL to a's use list. Apply the function *del* to the address calculation of *dst*.
5. For the possibly new RTL just created by the above two steps do the following. For each register that is a substring of the RTL, locate the RTL that set it. Call it S. If no previous RTLs have links to S, then link the current RTL to S.
6. For each $A \in C$ and for each $a \in A$, if $csrc \approx a$, then remember the set A. If no equivalence class contains *csrc*, create one. Call this set E. For each $A \in C$ and for each $a \in A$, if *cdst* interferes with a, then set A to $A - \{ a \}$. Finally set E to $E \cup \{ cdst \}$, and set C to $C \cup E$.
7. For each string s on the dead-variable list, for each $A \in C$ and for each $a \in A$, if s interferes with a, then set A to $A - \{ a \}$.

The function *del* is defined as follows. For each register r that is a substring of *del*'s argument do the following. Remove the last RTL on r's use list. If the use list becomes empty, the RTL that set r is no longer necessary. Call that RTL S. Apply *del* to the source of S and then delete S.

When the end of the block is reached, do the following.

1. For each $A \in C$ and for each $a \in A$, add a to the dead-variable list of the RTL at the end of a's use list.
2. Output the RTL's with window information and dead-variable information appended.

Appendix G

Benchmark Programs

The following are the benchmark programs 8q, ctoi, and ctoi used in Chapter 6.

8q

```
import putc from "/usr/include/ylib.d"
integer up[15], down[15], rows[8], x[8]
main( )
integer i, k
    for (k = 1; k <= 100; k = k + 1) {
        queens(1);
        putc(012);
        }
end
queens(c)
integer r, c

    for (r = 1; r <= 8; r = r + 1)
        if (rows[r] == 0 & up[r-c+8] == 0 & down[r+c-1] == 0) {
            rows[r] = up[r-c+8] = down[r+c-1] = 1
            x[c] = r
            if (c == 8)
                print( )
            else
                queens(c + 1)
            rows[r] = up[r-c+8] = down[r+c-1] = 0
            }
end
print( )
integer k

    for (k = 1; k <= 8; k = k + 1) {
        putc(' ')
        putc('0' + x[k])
        }
    putc(012)
end
```

```
import printf from "/usr/include/ylib.d"
main( )
integer i, j, k, ctoi( )

    for (i = 1; i <= 30; i = i + 1)
        for (j = 1; j <= 10000; j = j + 1)
            k = ctoi("   3567")
        printf("k is %d\n", k)
end

# ctoi - convert string to integer
integer ctoi(in)
char in[ ]
integer i, sum

    i = 1;
    while (in[i] == ' ' | in[i] == '\t')
        i = i + 1
    sum = 0
    while (in[i] >= '0' & in[i] <= '9') {
        sum = sum * 10 + in[i] - '0'
        i = i + 1
        }
    return (sum)
end
```

```
import printf from "/usr/include/ylib.d"
main( )
int i, j, k, ctoi( )
int str[10]

    str[1] = str[2] = str[3] = ' '
    str[4] = '3'
    str[5] = '5'
    str[6] = '6'
    str[7] = '7'
    str[8] = 0
    for (i = 1; i <= 30; i = i + 1)
        for (j = 1; j <= 10000; j = j + 1)
            k = ctoi(str)
    printf("k is %d\n", k)
end

# ctoi - convert string to integer
integer ctoi(in)
integer in[ ]
integer i, sum

    i = 1;
    while (in[i] == ' ' | in[i] == '\t')
        i = i + 1
    sum = 0
    while (in[i] >= '0' & in[i] <= '9') {
        sum = sum * 10 + in[i] - '0'
        i = i + 1
        }
    return (sum)
end
```

List of References

AHO72 Aho, A. V. and Ullman, J. D. *The theory of parsing, translation, and compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

AHO77 Aho, A. V. and Ullman, J. D. *Principles of compiler design*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

AHO80 Aho, A. V. "Translator writing systems: where do they now stand?," *IEEE Computer* **13**, 8 (August 1980), 9-14.

AMMA77 Ammann, U. "On code generation in a PASCAL compiler," *Software—Practice & Experience* 7, 3 (June 1977), 391-423.

BELA66 Belády, L. A. "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal* **5**, 2 (April 1966), 78-101.

BELL71 Bell, C. G., and Newell, A. *Computer structures: readings and examples*, McGraw-Hill, New York, NY, 1971.

BERR78 Berry, R. E. "Experience with the Pascal P-compiler," *Software—Practice & Experience* **8**, 5 (September 1978), 617-627.

CATT78 Cattell, R. G. G. "Formalization and automatic derivation of code generators," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, April 1978.

CATT79 Cattell, R. G. G., Newcomer, J. M., and Leverett, B. W. "Code generation in a machine-independent compiler," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 65-75.

CATT80 Cattell, R. G. G. "Automatic derivation of code generators from machine descriptions," *ACM Transactions on Programming Languages and Systems*, **2**, 2 (April 1980), 173-190.

DAHL66 Dahl, O. and Nygaard, K. "SIMULA—an ALGOL-based simulation language," *Communications of the ACM* **9**, 9 (September 1966), 671-678.

DART70 Dartmouth College, *BASIC*, 5th edition, Dartmouth College, Hanover, NH, 1970.

DAVI80 Davidson, J. W., and Fraser, C. W. "The design and application of a retargetable peephole optimizer," *ACM Transactions on Programming Languages and Systems*, **2**, 2 (April 1980), 191-202.

DEWA77 Dewar, R. B. K. and McCann, A. P. "MACRO SPITBOL—a SNOBOL4 compiler," *Software—Practice & Experience* **7**, 1 (January 1977), 95-113.

DONE73 Donegan, M. K. "An approach to the automatic generation of code generators," Ph.D. dissertation, Rice University, Houston, TX, 1973.

DONE79 Donegan, M. K., Noonan, R. E., and Feyock, S. "A code generator generator language," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 58-64.

ERNS69 Ernst, G. W. and Newell, A. *GPS: a case study in generality and problem solving*, Academic Press, New York, NY, 1969.

FRAS77 Fraser, C. W. "Automatic generation of code generators," Ph.D. dissertation, Yale University, New Haven, CT, 1977.

FRAS79 Fraser, C. W. "A compact machine independent peephole optimizer," in *Conference Record of the 6th Annual Symposium on Principles of Programming Languages*, January 1979, pp. 1-6.

FREI74 Freiburghouse, R. A. "Register allocation via usage counts," *Communications of the ACM* **17**, 11 (November 1974), 638-642.

GANA80 Ganapathi, M. "Retargetable code generation and optimization using attribute grammars," Ph.D. dissertation, University of Wisconsin, Madison WI, 1980.

GIMP73 Gimpel, J. F. *SITBOL; version 3.0*, Technical Report S4D30b, Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1973.

GLAN77 Glanville, R. S. "A machine independent algorithm for code generation and its use in retargetable compilers," Ph.D. dissertation, University of California, Berkeley, CA, 1977.

GLAN78 Glanville, R. S., and Graham, S. L. "A new method for compiler code generation," in *Conference Record of the 5th Annual Symposium on Principles of Programming Languages*, January 1978, pp. 231-240.

GRAH80 Graham, S. L. "Table-driven code generation," *IEEE Computer* **13**, 8 (August 1980), 25-34.

GRIE71 Gries, D. *Compiler construction for digital computers*, John Wiley & Sons, New York, NY, 1971.

GRIS71 Griswold, R. E., Poage, J. F., and Polonsky, I. P. *The SNOBOL4 programming language*, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ, 1971.

GRIS72 Griswold, R. E. *The macro implementation of SNOBOL4*, W. H. Freeman and Co., San Fransico, CA, 1972.

GRIS77 Griswold, R. E. "Benchmarks of DEC-10 SNOBOL4 processors," Department of Computer Science, The University of Arizona, (June 1977).

HADD78 Haddon, B. K. and Waite, W. M. "Experience with the universal intermediate language Janus," *Software—Practice & Experience* **8**, 5 (September 1978), 601-627.

HANS81 Hanson, D. R. "The Y programming language," *SIGPLAN Notices* **16**, 2 (February 1981), 59-68.

ICHB79 Ichbiah, J. D., Heliard, J. C., Roubine, O., Barnes, J. G. P., Kreig-Brueckner, B., and Wichmann, B. A. "Reference manual for the Ada programming language," *SIGPLAN Notices* **14**, 6 (June 1979).

JOHN78 Johnson, S. C. "A portable compiler: theory and practice," in *Conference Record of the 5th Annual Symposium on Principles of Programming Languages*, January 1978, pp. 97-104.

JOHN80 Johnson, S. C. "Language development tools on the UNIX system," *IEEE Computer* **13**, 8 (August 1980), 16-21.

KERN76 Kernighan, B. W., and Plauger, P. J. *Software tools*, Addison-Wesley, Reading, MA, 1976.

KERN78 Kernighan, B. W., and Ritchie, D. M. *The C programming language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

KERN80 Kernighan, B. W., and Mashey, J. R. "The UNIX programming environment," *Software—Practice & Experience* **9**, 1 (January 1980), 1-15.

KERN81 Kernighan, B. W., and Plauger, P. J. *Software tools in Pascal*, Addison-Wesley, Reading, MA, 1981.

KNUT68 Knuth, D. E. "Semantics of context-free languages," *Math Systems and Theory* **2**, 2 (June 1968), 127-145.

KORN80 Kornerup, P., Kristen, B. B., and Madsen, O. L. "Interpretation and code generation based on intermediate languages," *Software—Practice & Experience* **10**, 8 (August 1980), 635-658.

LAMB81 Lamb, D. A. "Construction of a peephole optimizer," *Software—Practice & Experience* **11**, 6 (June 1981), 639-647.

LESK79	Lesk, M. E. "Lex—a lexical analyzer generator," *UNIX Programmer's Manual 2*, Section 20, January 1979.
LEVE80	Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R., and Wulf, W. A. "An overview of the production quality compiler-compiler project." *IEEE Computer* **13**, 8 (August 1980), 38-49
McKE65	McKeeman, W. M. "Peephole optimization," *Communications of the ACM* **8**, 7 (July 1965), 443-444.
NELS79	Nelson, P. A. "A comparison of PASCAL intermediate languages," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 208-213.
NEWE72	Newey, M. C., Poole, P. C., and Waite, W. M. "Abstract machine modelling to produce portable software—a review and evaluation," *Software—Practice & Experience* **2**, 2 (April 1972), 107-136.
NORI81	Nori, K. V., Ammann, U., Jensen, K., Nageli, H. H., and Jacobi, C. H. "Pascal-P implementation notes," in *Pascal—The Language and its Implementation*, D. W. Barron, Ed., Wiley-Interscience, Chichester, UK, 1981.
PERK79	Perkins, D. R. and Sites, R. L. "Machine-independent Pascal code optimization," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 201-207.
RICH71	Richards, M. "The portability of the BCPL compiler," *Software—Practice & Experience* **1**, 2 (April 1971), 135-146.
RICH77	Richards, M. "The implementation of BCPL," in *Software Portability*, P. J. Brown, Ed., Cambridge University Press, Cambridge, UK, 1977.
RITC74	Ritchie, D. M., and Thompson, K. "The UNIX time-sharing system," *Communications of the ACM* **17**, 7 (July 1974), 365-375.
SHIL78	Shillington K. A. and Ackland, G. M. *UCSD Pascal version 1.5*, Institute for Information Systems, University of California, San Diego, CA, 1978.
SNYD74	Snyder, A. "A portable compiler for the language C," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974.
STRO58	Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., Steel, T. "The problem of programming communication with changing machines: a proposed solution," *Communications of the ACM* **1**, 8 (August 1958), 12-18.
STEE61	Steel, T. B. "A first version of UNCOL," in *Western Joint Compter Conference Proceedings*, May 1961, pp. 371-378.
TANE80	Tanenbaum, A. S., Stevenson, J. W., and van Staveven, H. "Description of an experimental machine architecture for use with block-structured languages," Informatic Rapport IR-54, Vrije Universteit, Amsterdam, The Netherlands, 1980.
WAIT73	Waite, W. M. *Implementing software for non-numeric applications*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
WIRT75	Wirth, N., and Jensen, K. *Pascal user manual and report*, Springer-Verlag, New York, NY, 1975.
WULF71	Wulf, W. A., Russell, D. B., and Haberman, A. N. "BLISS: a language for systems programming," *Communications of the ACM* **14**, 12 (December 1971), 780-790.
WULF75	Wulf, W. A., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. *The design of an optimizing compiler*, Elsevier North-Holland, New York, NY, 1975, 107-125.
WULF81	Wulf, W. A. "Compilers and computer architecture," *IEEE Computer* **14**, 7 (July 1981), 41-47.