Peephole optimization of assembly language using regular expression pattern matching.

Bill Mahoney

PEEPHOLE OPTIMIZATION OF ASSEMBLY LANGUAGE USING REGULAR EXPRESSION PATTERN MATCHING

A Thesis

Presented to the

Department of Mathematics and Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

University of Nebraska at Omaha

by

Bill Mahoney

December, 1990

Abstract

Producing compilers for high level computer languages is difficult. The early phases of the compiler, syntactical and lexical analysis, are well understood and have been automated. Code generation is more ad-hoc in both design and implementation. The code generators in use may be sufficiently complex that the output could benefit from additional optimization.

This thesis describes current efforts to produce peephole optimizers which perform optimizations on the compiler output. A case is presented for basing optimizations on human written rules. A peephole optimizer generator is presented which reads a rule language which includes regular expressions and produces an executible program which operates on the assembly language output from a compiler. The resultant program, named OP, implements the guidelines specified in the original specification. Pattern matching is used within OP to track both register assignments and the matching of rules. Therefore, regular expression notation is sufficient for generating peephole optimizers.

Contents

For Joan, no longer the "Thesis Widow".

1. Introduction

In recent years there has been a great deal of effort placed in automating the construction of compilers for high level computer languages. Some of these efforts have paid off in the form of program generators which take language specifications and construct some of the various parts of a compiler. Unfortunately there has been no universal consensus as to the best way to automate the final step in a compiler: the conversion into a machine language that the hardware understands.

This last phase of the compiler, the code generation phase, is still largely written by hand. Typical code generators are extremely complex, and rewriting this phase to generate software for a different machine is a considerable task. For these reasons automating code generation is a high priority. Several methods are currently used to augment the writing of code generators. One area receiving less research is concerned with improving the machine language which is produced by the compiler.

These improvements are performed by peephole **optimizers**, so named because they examine only a narrow section of the source code at any one time [19]. Efforts have been made to automate the generation of peephole optimizers as well. These efforts, along with some experimental results, are the topic of this thesis.

A brief outline of the components which make up a compiler is first presented. This introduces many of the terms used throughout the remainder of the text. Section three is concerned with code generation itself. A discussion of intermediate forms which languages may take is followed by the introduction of code generation and some of the more accepted methods for conversion to machine codes. Chapter four outlines several past attempts, mostly successful, at obtaining a peephole optimizer from machine descriptions or rule languages. Which method is preferable is an argument which is discussed in the next section. A case is built for preferring the rule method. Supporting evidence is shown in section six, where a simple rule based optimizer is described.

2. Language Compilers

Interactions with both humans and computers is effectively carried out with language. In the case of computers, a programming language provides a mechanism for communicating ideas, data, and procedural steps which allow the machine to accomplish work. These programming languages enhance the development and maintainability of software. A good programming language will bridge the gap between languages in use by humans, and languages understood by the computer hardware.

If the language is close to the problem to be solved, the solution will be easier. A programming language which contains constructs similar to the way the computer hardware operates are generally not as similar to the problem as, say, an algorithm which can be used effectively and is written in English. So there is a natural tendency to desire languages which have terminology more closely akin to the way a human would like to solve a problem. Such languages are termed "high level", as opposed to the "low level" constructs of the computer.

High level languages offer many advantages. They are easier to learn, a programmer does not need to be concerned with machine specific details, there are a variety of control structures to make the language match the algorithm, high level languages are typically easier to debug, and high level languages permit a more modular resolution of the problem.

Furthermore, there is a desire to have the solution to a problem be independent of the actual underlying circuitry. That is, when the solution is found and written, it should be easily moved from one type of hardware to another different type of hardware. This can not be achieved if the language used to solve the problem is dependent on details of the machine itself. For these reasons, high level languages are abundant in computer science.

2.1. Language Conversion Methods

Unfortunately, as computer hardware does not typically understand high level languages, some methodology must be used to convert one type of language into another type of language. This is the job of a **compiler** program; compilers translate a **source** language into a **target** language. Initially this appears to be an almost overwhelming difficulty. There are thousands of source languages ranging from traditional high level languages such as FORTRAN and Algol to specialized languages for particular applications such as reservation systems. Unfortunately the variety of target languages is equally broad. The target language could be another programming language, or instructions for driving a lathe. Further, compilers are classified as "single pass", "load and go", and others, depending on how they perform the translation. Regardless of the apparent complexity, much of the process of creating compilers has been automated, and the techniques for writing compilers have become well known.

A typical language compiler in computer science is a tool which converts the high level constructs of a programming language into **machine language** or **assembly language.** For example, a compiler might convert a FORTRAN program into the native machine language used by a personal computer, which consists of long sequences of numbers. Assembly language is at only a slightly higher level than machine language; a special compiler called an **assembler** is used to translate the assembly language into the raw numbers of machine language. There is generally a one to one correspondence between one statement in assembler and one machine instruction in the hardware.

2.2. Lexical Analysis

Internally, there are several components to a high level language compiler. First, the input language must be broken into **tokens** representing the words of the language. These words are groupings of the alphabet defined for the language. As an example, the high level language Pascal permits statements such as:

```
        diagonal:=sqrt( a*a + b*b );
```

The alphabet for the language Pascal contains the letters A through Z, digits, plus several special characters including parenthesis. Different arrangements of characters from the alphabet of a language form different types of tokens. In the above example, some of the tokens which would be valid in Pascal are:

```
        diagonal,
        :=
        sqrt
        (
        a
```

et cetera.

Note that spacing is not a sufficient indicator of where the tokens are to be found in the input stream. Instead, some clue of what combinations of symbols from the alphabet should be formed together is needed. Linear analysis provides these clues. When linear analysis is used in a compiler it is termed lexical analysis [1].

The lexical analyzer is the first major part of a compiler. It is responsible for reading a sequence of characters in the alphabet of the source language and grouping these characters into tokens which are meaningful. Usually the lexical analyzer consists of some type of function which is called by other parts of the compiler software. It is the job of the subroutine to read the next few input characters until a valid token is found, and to return that token. This makes the overall structure of the compiler simpler by isolating the lexical analysis from the remainder of the program.

Since the lexical analysis phase of the compiler reads the input language, it may optionally support other functions of the compiler, such as producing listings of the input source, removing commentary, or handling the inclusion of other source files.

Frequently, the lexical analysis phase not only returns the token itself, such as "diagonal" in the above example, but also some indication of the type of the token. "Diagonal" might be of type "variable", for example, while the Pascal token "while" might be of type "keyword".

Placing the lexical analysis phase separately in the compiler has several benefits. Since it is independent of the remainder of the compiler, it is easier to maintain. Changes to the analyzer can be made at will as long as the interface with other sections remains the same. In addition, much of the software that makes up the lexical analysis phase is similar for different high level languages, so a large amount of the program can be reused for different compilers. Lastly, since the interface to the remainder of the compiler is well defined, the lexical analysis can provide the input tokens regardless of what the input media happens to be. Details involved in reading a tape, for example, can be contained in this phase.

It is also interesting to note that lexical analysis of the input typically produces few errors, since the overall language of the program is not being considered. Again in Pascal, one might accidentally write:

```
          total := vector( 5 );
```

where "vector" is really an array, and "[5]" should be used. If "vector" was instead a function name, this would be a legal Pascal statement; however the lexical analysis phase has no knowledge of what "vector" represents, and can not make a decision beyond returning "vector" with a type of "variable".

A complicating issue for lexical analysis is that many characters may have to be scanned prior to a determination of which token to return. A variable named "beginning" is similar to a keyword "begin", and the difference can not be detected until the sixth character in the input stream is read. Frequently this requires that input be saved, so that input which is read in advance can be reprocessed for the next input token. This "push back" of the input character stream can be extreme in cases where the high level language does not use reserved keywords, since the analyzer must examine the context surrounding the token in order to decide which type it belongs to.

2.3. Lexical Analyzer Generators

When the first high level languages were being developed, the lexical analysis phase was most often written by hand. A lookup table of possible input tokens was scanned, or a series of nested if-then-else statements selected which type of input token was found. This hand written approach frequently produces lexical analyzers which are very fast, but difficult to maintain. The maintenance is a large factor when constructing a new language, since it is nice to be able to simply add new language constructs, keywords, and the like without having to completely rework the lexical analyzer.

The more modern approach utilizes a lexical analyzer generator program to take a list of keywords or identifiers that need to be scanned from the input, and produces a working analyzer based on these descriptions.

One of the more popular lexical analyzer generators is a program named Lex, written by Lesk and Schmidt [17]. The compiler writer gives a set of patterns to Lex, which constructs a program in the language C. This program is the lexical analyzer. More specifically, one can generate a Lex program that, when processed by Lex, produces a C program named "lex.yy.c". This file can then be compiled with a standard C language compiler, and a complete lexical analyzer is generated.

There is a fairly straightforward way to interface Lex with other parts of the compiler. For example, as Lex constructs the latest token from the input stream, it places this string into a variable with a known name ("yytext"), which can be used externally, or within the Lex program itself. In addition, the Lex program specifies for each matched pattern what to return as the type of the token. Thus it is a simple matter to associate the token type as well as the token value to the caller of the lexical analysis function. This meets the requirements outlined above.

Lex is only one lexical analyzer generator available to computer scientists. However, it is so popular and ubiquitous a tool that numerous look-alike programs exist in the public domain. In this regard we can safely say that the methods for producing a quality, working lexical analyzer are well known, and that sufficient easy-to-use tools exist.

To a certain extent, syntax analysis is also well understood, and tools for constructing syntax analyzers also abound.

2.4. Language Grammars

In order to define a programming language, we need to define the alphabet, as discussed above for lexical analysis, the set of all programs which are correct, and the meaning of all programs. The last case is the most difficult.

What exactly constitutes a language? In early attempts at high level computer languages, a particular manufacturer's compiler was the definition of the language. If the compiler allowed a given programming construct, then that construct was in the language. A modern analogy is to have a machine called an acceptor which decides whether a particular grouping of tokens is a part of the language [24]. If we pass a group of tokens to the acceptor and the acceptor indicates "yes", then this grouping is acceptable. But an approach which is more popular is to have a grammar that dictates what is legal and illegal in the language.

A grammar is a finite set of **production rules**, or just productions, which specify the exact syntax of what is and what is not acceptable as a sentence in a given language. The grammar of a language, then, allows us to decide whether a given grouping of tokens is acceptable. The difference between this and an acceptor machine is subtle; the grammar imposes some additional structure on what is legal and what is not.

If the grammar consists of a set of rules, the input tokens are matched against these rules and the legality of the input can be determined. This is done by replacing grammar rules with simpler forms. For example, the following grammar could be used to decide if a set of tokens is a simple assignment statement in a programming language:

```
        assignment is: variable equals number
```

If the types of tokens returned by the lexical analysis phase of the compiler are "variable", followed by "equals" followed by "number", then the grammar rule above is said to be **matched** and the combination represents a legal language construct: "assignment".

The assignment construct can itself be a part of a higher level language feature, such as a statement. This is because assignment may be only one of several possibilities. If, say, reading a value into a variable is a statement, then one of the grammar rules might be:

```
        statement is:   assignment
                  or:   read_stmt
                  or:   .
                        .
```

with the construct "read_stmt" defined as:

```
        read_stmt is: read variable
```

In this way, a hierarchy of grammar rules is built up.

For example, a program might consist of procedures. These might consist of statements. Each statement can be an assignment, a read_stmt, and so on.

This hierarchical analysis is referred to as **parsing**. This involves grouping the types of the tokens passed by the lexical analysis section into meaningful rules defined in the grammar, and then replacing the token types with the rule type. As an example, the statement:

```
var = 10
```

would return the types "variable", "equals", and "number", which would then be parsed and replaced by "assignment". Eventually, "assignment" might be replaced by "statement", and "statement" replaced by "procedure", until an initial grammar rule is finally reached.

2.5. Grammar Rule Languages

These grammar rules are defined in a **metalanguage**. A metalanguage is simply a language that is used to describe another language. For example, a course in high school Spanish might be taught in English, a metalanguage for Spanish. The instructor uses English as one level "higher up" to describe the constructs of the Spanish language. Metalanguages are important in compilers because there are parser-generators, just as there are lexical analyzer generators. A metalanguage describes the grammar rules, and the parser generator creates the software that actually checks sentences against the grammar. A popular parser generator program is YACC, written by Johnson in 1975 [13].

Using YACC, the compiler author specifies the available grammatical rules for the language, and YACC produces a parser that matches tokens against the rules. In the case of a successful match, where the hierarchy has progressed all the way to the first rule of the grammar, the parser indicates that the language matches the specification and returns "yes". Note the production of another language, the reason for having a compiler, is not satisfied by YACC alone. The programmer puts certain actions in the YACC grammar rules that produce the new language output as a side effect.

3. Code Generation

Now that we have automated tools for constructing the lexical analysis phase and the parser for a compiler, it remains only to generate the new language which we are attempting to translate into. In this paper, we will consider only assembly language. Machine language is similar, but the techniques used here will apply to assembler language more easily than to machine language or object code.

In the 1960's and 1970's, much progress was made in the analysis and formalization of procedures for producing high level language compilers. A motivating factor behind research in this area has been the "M times N" problem; if there are N high level languages, and M computer CPUs upon which the language must run, there must be MxN compilers to support the languages. Today, new machine architectures are being produced at a rapid rate, and this makes the problem even more acute.

Early compilers were typically targeted at one individual CPU type. For example, McClure describes an implementaion of FORTAN for an IBM machine [18]. His comments show that the intermediate form was almost a direct translation into certain addressing modes available on the System/360. Thus, if the compiler were to be moved to a different processor, the intermediate representation of the program is imperfect.

As we have seen, most of the research in the automatic generation of compilers has been in the lexical analysis and semantic analysis phases of the compiler. These sections are easily described in precise terms, and many lexical analyzer generators such as Lex, as well as "syntax compilers" such as YACC, are now available. Unfortunately not as much interest has focused on actually taking the semantic information in a high level language and producing optimal machine code.

3.1. Automating the Generation

Although technological advances in hardware make the production of the fastest possible object code less of a concern than it was in the past, it is still necessary to produce good object code from the high level language. Additionally, computer languages are getting more complex, and with the complexity of the language comes complexity of the computer object code needed to support the language. The problem is that the code generation and optimization phase of the compiler is still frequently written by hand in order to take full advantage of special cases in the instruction set of the computer system.

Each CPU is different. Some have instructions containing one operand address, some have two or three address instructions, and so on. Certain registers may have restrictions on how they can be used, and some operations may only work correctly if the operands are in specific hardware registers. This makes it difficult for the compiler author to produce the best possible object code, since extensive case analysis must be used to match the object code to the semantics of the program.

Within approximately the last ten years, efforts have been made to automate the production of the code generation and optimization phase of the compiler as well. Those efforts, which have been successful, have allowed a compiler for a relatively high level language to be retargeted to a new processor in only a few days.

A brief examination of the current technology in the actual code generation phase of compilers is in order.

3.2. Intermediate Languages

Generally, high level languages are not directly translated into assembly language, but into an intermediate form. There are two reasons for this; an intermediate form makes it easier to retarget the compiler to different hardware, and an intermediate representation also makes optimization easier, since the optimization of intermediate code is machine independent. Intermediate code can be a tree structure, postfix notation, instructions for a pseudo-machine, or any of a number of other representations.

Postfix notation is a common choice. Postfix would represent the assignment statements:

```
var := i * ( j + k );
sum := total + a + b * c;
```

as

```
var i j k + * :=
sum total a + b c * + :=
```

Postfix has the advantage of being easily readable, and easily translated into valid assembler source code. Unfortunately the source code is frequently inefficient unless the intermediate form is optimized during the compilation. Tanenbaum has shown that excellent results can be achieved with this method if the intermediate optimization is applied [23].

Another method used for intermediate code is to transform the high level language into three address form. Three address form is a sequence of binary operations with two source variables and a third resultant variable. The above assignments might be converted into these three address forms:

```
temp1 := j + k;
var := i * temp1;
```

and

```
temp1 := total + a;
temp2 := b * c;
sum := temp1 + temp2;
```

Control statements are unraveled in the same way. This three address instruction format is more easily converted to assembly language in many cases because it is already similar to the form that will need to be generated. The temporary values above could be hardware registers, for instance, or the target machine may directly support three-address instructions. Also the use of explicit names in the three address scheme allows the reshuffling of operations to alter the resultant code. In the second example above, the first two lines of the three address code could be reversed with no loss of validity.

Typically, three address statements are stored in a structure with four elements. These are the addresses as shown above, plus the operation to be performed. In cases where the operation is unary and not binary (assigning the negative of some variable to another variable, as an example) one of the fields is simply not used. The three address statements that represent the imtermediate code are then translated into machine language one at a time.

A discussion of intermediate code would definately not be complete without commenting on UNCOL. UNCOL, the UNiversal Computer Oriented Language, was an attempt to categorize the hardware platforms prevalent at the time such that an UNCOL to machine language conversion was simple to perform. HLL compilers were written to produce UNCOL as their output instead of machine code. In essence, the intermediate representation of the compiler was the target language.

The advantage of this technique is that instead of MxN combinations of compilers and machines outlined above, there is only M+N. Each of the N high level language compilers transforms the source language into an intermediate form, the UNCOL. Each of the M targets has a translation program to convert the UNCOL intermediate language into the required machine or assembler language.

A decision needs to be made concerning the proper format that an UNCOL should take. For example, should the UNCOL intermediate language (IL) endeavor to encompass all instructions which might be available on all machines, or should it cover only those features which are common? Another way to put this is to ask what set of allowable operators makes up the target machine language? Two approaches are called the **union** and **intersection** methods.

3.3. Union Machines

Davidson has coined the term union machine to represent certain abstract machines which have instruction sets which are collections of features from many target instruction sets [3]. As an example, if a certain machine (such as the VAX) includes three address instructions, and if it desirable to use them where possible, it is better if the intermediate language also supports three address instructions. This is because it is difficult for a code generator to exploit three address instructions if only one address or two address instructions are present in the intermediate language.

The result is that if the intermediate language is to support a certain machine which has an unusual feature, that unusual feature is easier to generate if it is also present in the UNCOL. Unfortunately, the UNCOL quickly becomes unwieldy, because of the tremendous number of machine features which it must support. Since a translation for each UNCOL statement must be present, the translation phase must be complex. In addition, since there is a far greater number of UNCOL statements than there are assembler mnemonics on a typical platform, not all of the UNCOL statements are trivial to convert into native code.

This is not the only problem with the union approach to intermediate code. Since the intermediate code is more complex, it is not just the translation into assembler that is complicated. The compiler software which produces the intermediate code are more complex and must contain many more special cases.

Davidson notes that the resultant machine code is not of good quality in any case, because of inefficiencies in the intermediate representation. He notes that not many intermediate languages support the concept of a condition code, and as such statements which may be adjacent in the original high level language often end up causing variables to be tested several times in successive assembly language statements. In actual practice, segments like:

```
i := j;
if ( i = 0 ) then
```

should not cause the value of the variable "i" to be tested again if the condition code was set in the assignment statement already.

Finally, consider the union approach from a global perspective. If a certain machine has a new type of instruction, for example if a certain architecture has instructions for dealing with queues, the union approach implies that queue instructions now need to be implemented in the UNCOL. The intermediate code generation phase on all of the N high level language compilers will now need to be changed as well.

The result is that the union approach to the specification of an UNCOL generally does not work.

3.4. Intersection Machines

Given that a union methodology is not the best approach, some developers have attempted the opposite track. Instead of putting each feature available on the M target architectures into the intermediate representation, the intersection approach places only those instructions which are common to all machines into the UNCOL. Thus, only the most general operations which are required to support the language are included.

As an example, not all machines have an increment instruction. In this case, the abstract machine representation, the UNCOL, would not have an operation representing increment. Instead the combination of load, add, and store would be present, since this is presumably available on all machines.

Construction of an UNCOL to machine language translator is now a simpler task, since there are fewer UNCOL instructions which need to be translated. Thus, the code generator is smaller and the translator is smaller, so their implementation is easier. This may be at the expense of more complex lexical analysis and parsing phases.

Unfortunately, not everything works out as well. Even though a particular machine might have a certain function, it is not used, since not all machines have that function. So the result is that the actual assembly language output is not as efficient as it could be.

The balance between an intersection approach and a union approach is delicate. The design of a good UNCOL, trading off simplicity for features, was the "northwest passage" in compiler construction for many years. Everyone thought that it existed, but no one could find the right path. Recently, Tanenbaum has shown that UNCOL is still not spiritless, by demonstrating a portable compiler that uses a simple UNCOL approach to represent the intermediate language in postfix notation [22].

3.5. Conversion to Assembly Language

Once the intermediate form is decided upon, it remains to convert this internal form to a format which can be understood either by an assembler program or by the hardware itself. This is the final code generation phase.

Generation of the best possible code from an intermediate representation is a difficult problem. It must be sufficient to utilize heuristic techniques to generate good assembly language. Additionally, the restrictions imposed on the code generator are tight. It must be fast, but must produce code of high quality. The programmer writing in a high level language has a certain trust placed on the part of the compiler that it will faithfully reproduce the intent of the program, correct or not.

Unlike the input to the compiler, the compiler writer may have some freedom in the format of the output that the compiler produces. There are basically three different outputs that are often used. The compiler can produce absolute machine code - machine code that is ready to be loaded into the memory of the computer and executed. Typically, student compilers are of this type, since they can rapidly compile single unit programs which are immediately run. Another type of output is machine code which can be relocated. This is more often referred to as object code. Object code files can be hitched together with a linking program and executed, which assists in constructing large systems in pieces which are later bound together. Lastly, the output can be assembly language, which is translated into machine code using an assembler. Assembly language has the small advantage that facilities of the assembler, such as macros and the like, are available "free", and the much more profound advantage that the symbols and variables used by the program can be passed on to the assembler instead of being taken care of in the compiler. Another consideration is that there may be no need to duplicate the actions of the assembler inside the code generation phase anyway.

3.6. Peephole Optimization

Regardless of what form the compiler output is to be in, the combination of internal representation and code generation often produce machine instructions that are not as efficient as they could be. This is often the case when successive statements are compiled into the output, but because they are independent, the compiler does not take advantage of reusable variables, registers and the like. To be machine independent, parsing and global optimization preceeds the code generation. But to be simple and fast, the code generator usually operate locally. So the code generator may produce locally optimal code fragments which, when juxtaposed, yield unexpected results:

```
total := total + 1;
avg := total / 10;
```

may produce assembler language which stores the calculation for total, then immediately re-loads the value for the calculation of average:

```
load  total
add   1
store total
```

```
load  total
div   10
store avg
```

Good optimization techniques are commonly found in production compilers that will catch this type of juxtaposition and remove the extra load instruction. However, suppose the processor has an instruction to increment a variable:

```
incr  total
```

Is it now more efficient to increment the variable total and then load it, or is the first version of the output acceptable? This is another example of the extensive case analysis needed by the code generator.

Unfortunately, this example is realistic. Some of the efforts of modern compiler code generators produce machine or assembly language code which still has slight idiosyncrasies which should be "cleaned up". This cleaning up of the compiler output is performed by peephole optimizers.

Kessler [14] has identified three classes of instructions for which peephole optimization is useful. He refers to the new, replacement instructions as "idioms", and lists the "set idiom", the "binding idiom" and the "composite idiom" as his three replacement types. The set idiom refers to special purpose instructions which can be used if the operands are a member of a particular set. As an example, the increment by one is a set idiom for the more general add instruction. The binding idioms are instruction sets in which two or more operands refer to the same location in the machine. A More efficient instruction may be able to take advantage of this by using specialized addressing modes. Lastly there is the composite idiom, an instruction that performs the same actions as some larger sequence of instructions. Examples include special purpose looping constructs. The large complexity of producing good assembly language from a high level language causes some of these idiosyncrasies to slip through.

Some additional work has been done to produce peephole optimizers automatically, just as other phases of the compiler are. In some cases these optimizers are an integral part of the code generation phase of the compiler. In others, the peephole optimization phase is an extra, separate section of software.

These efforts are the subject of this thesis. It is possible to use the same technology available with regular expression lexical analysis to create peephole optimizors which are easy to write and maintain. These optimizers operate on assembly language, seperate from the actual compiler which produced their input.

4. Recent Work

Peephole optimization is designed to locate "equivalence-preserving transformations whose application yields an immediate and obvious improvement in the program" [20]. In other words, we desire to replace some instances of instructions with other instances of instructions which are somehow better, but do the same job. The best candidates for optimization are "those which have a certain generality and can be grouped together in bunches which can be applied in one scan of the argument program."

Peephole optimization typically restricts us to those types of simplifications which depend on the actual form of the program itself; these are sections which can be detected and proven to lead to an equivalent program, with no awareness of what exactly the program is supposed to do. Primarily, these simplifications can be done syntactically rather than semantically. Syntactic improvements make up a mere fraction of the enhancements that a qualified human programmer can detect, but the ability to perform them with a program makes the improvement desireable. Syntactic replacement of instruction groups is also much simpler to perform.

Typical peephole optimizers are tied to the back end of the compiler [11]. The optimizer can thus have access to information which is lost once the transformation to assembley language is made. In addition, much of the scanning required by the optimizer can be done with little extra cost, since the required source is already in memory and ready to process. These are reasons to include the peephole optimization phase in the compiler. As we shall see, it is not an absolute requirement, but if the optimization is in a seperate program, additional processing is needed in addition to merely replacing instruction groups.

For these reasons the focus of research in peephole optimizers has been contained within research on code generation. Much of the investigation into automatically generating optimizers has been accomplished by attempting to automatically create correct instances of assembly language constructs through symbolic machine descriptions.

4.1. Symbolic Machine Descriptions

A symbolic machine description is a notation for indicating not only the syntax of an assembly language construct, but the operation that that instruction performs. For example, it is apparent to a human reader that "ADD" would indicate the addition of some operands with a resultant number.

A typical example of a symbolic representation, and one which is used in work outlined below, is Instruction Set Processor notation, or ISP [2].

ISP descriptions are of the general form:

```
condition -> action/sequence
```

where action/sequence is the result of performing the instruction or condition on the left hand side of the rule. An example will make this clear:

```
exchange -> (A <- B; B <- A)
```

would indicate that an exchange operation causes the contents of A and B to be swapped. Note that the actions contained within the parenthesis happen concurrently. Additional notation is commonly used, such as using M to denote memory, AC for an accumulator, and notably <5:8> to indicate bit positions five through eight. Memory and processor registers are considered to be large arrays of values.

For example, the PDP instruction "CLR @Rd" would be:

```
CLR @Rd -> M[R[d]] <- 0; N <- 0; Z <- 1
```

Here, N and Z refer to codition codes within the processor. The instruction has the effect of taking the contents of register d, and clearing the memory at the address contained within d.

4.2. Peep

Many attempts at peephole optimization attempt to combine adjacent instructions by examining their representative ISP actions and combining the results of the instruction. Then a table lookup is performed to attempt to find an instruction with the same effect, but which has a lower cost associated with it.

Typical of this approach is work by Robert Kessler [15]. Kessler has concentrated much of his work on peephole optimizers on pre-compiling tables of optimizable instructions, based on architectural descriptions of the target CPU. His work represents one of the early attempts to combine machine description languages at the time that the optimizer is built, rather than at optimizer run time.

In several aspects, his optimizer performs more than simple peephole optimization. For example, the optimizer, known as Peep, relaxes the dependency that optimizable instructions be adjacent in the code. This allows optimization that spans basic blocks in the assembly language.

Various versions of Peep exist, but all have PTG in common. PTG is the Peep Table Generator, a program which takes as its input a Lisp-based machine description and produces as output a compilation of all pairs of instructions which could be replaced by a single one. The convenience of using Lisp as the implementation language is that it allows **escapes** into the implementation language to be easily be incorporated as a part of the rules for matching instructions. This assists the peephole optimizer author by allowing him or her to simply call functions written in Lisp from within the architectural descriptions themselves.

The main section of the PTG software begins when it laboriously scans for pairs of instructions which can be combined into one. The scanning is done by taking the Lisp-based notation for what the instruction accomplishes, combining it with all other instructions, and searching each combination to see whether it is in the table. If the second instruction of each pair uses a data item that was defined in the first, it substitutes the reference for the data into the second instruction prior to scanning for the replacement. In this way, each of the N*N combinations is checked. Since this takes place at compiler-generation time, however, it is not a hindrance to the operation of the optimizer itself.

If an instruction with the same actions is found, the triple (original two, plus replacement) is saved in the table.

If the first instruction has no operands in common with the second instruction, Kessler's software concatenates the two instructions together and searches for an instruction that performs both operations in one operation.

It is difficult to assess the quality of enhancements performed by Peep, since much of the optimizer deals with flow analysis, rather than with peephole optimization. For example, Peep keeps a list of "equivalent resources" for each operand used by the instruction; these are used to replace operand fetches from memory when an equivalent value is currently in a register. We can note, though, several interesting points about Peep.

Since it is designed to test only pairs of instructions, three-instruction sequences which could be optimized into more efficient code will be missed. On the bright side, the flow analysis allows "logical adjacency" to be used instead of lexical adjancy. In other words, the instructions do not necessarily need to be next to each other in the assembly language source in order to be optimized. This lets the optimizer compress instructions which are not necessarily obvious, such as a jump instruction which goes to another jump instruction.

Kessler notes that the addition of Peep causes a 40% slowdown in compile time, and a savings of about 20% in the size of the code emitted.

4.3. LGN

Lamb has described another symbolic peeohole optimizer used in conjunction with an Ada compiler [16]. Carnegie-Mellon University began work on a preliminary Ada system, named the Charette compiler, in 1980. The Charette Ada compiler translated a subset of the full DoD Ada language into VAX-11 assembly language. The primary function of the project was not to produce a good implementation of Ada, but rather to explore what potential implementation problems a full Ada language compiler would have to overcome.

About two thirds of the way through the implementation it was decided that the resultant assembly language for even small Ada programs was so large that inspection of the output was not sufficient to determine what the original program was attempting to carry out. Some form of peephole optimization was needed to shorten the output. David Lamb constructed and described the resultant optimizer in his paper.

The optimizer takes as its input a doubly linked list of assembly language statements including labels. (The type of the node is either "opcode" or "label".) An additional phase following the optimizer actually emits the assembly language, so the optimizer itself merely transforms the list from a long format to a (hopefully) shorter one. Each of the nodes can be represented in what Lamb refers to as LGN, for linear graph notation. LGN is used simply to allow a human readable format for the internal structure, in the sense of ISP above.

An example of LGN is given for the VAX instructions:

```
loop: addl2 (sp)+,4(r0)
beql loop
```

The corresponding LGN is:

```
L1: label (name "loop") (prev L0:) (next L2:)
L2: object (opcode addl2) (operand L3: l4:)
    (prev L0:) (next L5:)
L3: address (mode autoincrement) (operand sp)
L4: address (mode displacement) (operand r0 4)
L5: object (opcode beql) (operand L1:) (prev L2:)
    (next L6:)
```

Note the correspondence between the operand nodes in the LGN and the operands in the correspondent assembly language. There is not a one to one matching between the various L1 to L5 and the label "loop" in the source code, since each of the operands is itself a node in the list. The operands of the "(prev" and "(next" constructs form the links for the list.

A pattern language consisting of a match portion and a replacement portion is used. If the match part of a rule agrees with the nodes currently being examined, the replacement section is substituted in its place. The pattern language bears little resemblance to the LGN outlined above, because the patterns are compiled into the implementation language before being checked against the LGN input.

A sample pattern is:

```
BEGIN pushal 0(&2)
      subl2 $&3,(sp)
-->
      pushal &negate(&3)(&2)
END
```

Symbols that have the form &N for an integer N are called pattern variables and are replaced with the operands that correspond to it in the assembly language representation. In this case, the first two lines will match the opcodes "pushal" and "subl2", possibly filling in the pattern variables &2 and &3 in the process.

Assuming the matching portion of the rule is actually in the input, it is replaced with the section of the rule after the "-->" symbol, with the pattern variables &2 and &3 replaced with the original operands.

Unfortunately, patterns are not always enough. In these cases, Lamb also uses escapes into the language in which the optimizer is implemented. These escapes normally take the form of functions which are called when the rule is matched; they are defined along with the rule. The term "&negate" is one example, above. A better way to show the need for escapes might be:

```
        BEGIN jlbs &1,&2: | IsOdd(&1)
        -->
              jbr &2
        END
```

matches "jump if lower bit set" and automatically replaces it with a simple jump if the operand was known to be odd at compile time.

An interesting note is that the pattern matching proceeds backwards rather than forewords. The reasoning is simple, but also elegant. The result of a match-replacement may very well allow another match and another replacement. In Tanenbaum's work outlined below a pattern is matched and the replacement is made, the system must back up a certain distance to try again. The distance that must be backed up is equal to the longest of the rules in the system. Lamb avoids this entirely by scanning the assembly language in reverse, which effectively results in the same thing.

Lamb translated the patterns from the format above into the language BLISS-10 using a SNOBOL program. Primarily, the reasoning behind this methodology is that it makes the patterns very easy to understand. The approach required a separate pattern for each storage size (byte operands, word operands, etc.) and he notes that a more formal notation of the action performed by each instruction would have helped. However, the closeness to assembly language makes it easy to add additional patterns based on observed output of the compiler.

In any case, the original scheme to reduce the complexity of the compiled Ada was successful. A typical reduction of 40-50% in the size of the output was possible in most cases. Lamb gives no indication in his notes of how fast the optimizer runs.

4.4. The Amsterdam Compiler Kit

The Amsterdam Compiler Kit (ACK) is a compiler generator package outlined in [22]. The kit requires the user to write a front end which compiles a high level language into an intermediate code. The intermediate language can then either be interpreted, or converted into assembler language. The ACK, then, is an integrated set of tools for the compiler author.

The interesting note about ACK is it's implementation. The various front ends of the system convert any of Algol, Basic, C, etc. into an intermediate language named EM. EM is the intersection UNCOL mentioned in chapter one. A complete list of the EM operations can be found in [23]. The various phases of the compilers are implemented in a manner analogous to pipes (a string of programs in a line): the output of one phase serves as the input to the next phase. In all, there are seven phases, of which two are peephole optimization.

In the case of EM, the front ends of the compiler emit even less than the intersection UNCOL supported by the code translation phase. The optimization of EM replaces some of the simpler sequences of operations with additional EM instructions which are not directly emitted. In other words, the code generator phase of the compiler produces a subset of EM, which is then peephole optimized into "smarter" EM. The traditional example of peephole optimization, replacing an add with an increment, provides an illustration. The EM sequence:

```
LOC 1           ; pushes 1 onto the RPN stack
ADD             ; adds top two stack elements
```

would be replaced by the smaller EM instruction:

```
INC
```

Because the compilers in ACK use peephole optimization twice, a generalized methodology was developed for specifying the way the optimizer works. Patterns are used to replace instances of instructions with shorter combinations. Like Lamb's work, the patterns are generated by hand, but are compiled into the peephole optimizer.

The input and output are expressed in a compact code, rather than ASCII, so that the overhead is reduced. In the case of EM, there are about 400 patterns which can be matched and replaced with simpler EM code. Each of the pattern replacement pairs is described similarly to:

```
LOL LOC ADI STL ($1=$4) and ($2=1) and ($3=2) ==>
INL $1
```

The part prior to the "==>" symbol is the pattern, and the part after it is the replacement.

In this example, if the code sequence on the left is found, and if the qualifications such as "($1=$4)" are met, the entire pattern is replaced with that which is on the right. LOL loads a local variable on to the stack, LOC loads a constant, ADI is add integers, and STL is store back into a local. If the local variables are the same one (the "($1=$4)") and the increment is one (the "($2=1)"), and the integers are two bytes, replace the four operations with an INL (increment local) operation. EM instructions can only have one operand, so the $1, $2,... are sufficient to denote the operands of the first operation, second operation, etc.

ACK's pattern/replacement method generally performs the following types of optimization: constant folding, strength reduction (replaces multiply by shift, for example), reordering (-N/5 is the same as N/-5, eliminating the need to negate N), and elimination of null instructions such as adding zero.

A frequent occurrence of the pattern replacement strategy is that the result of a replacement can itself be the source of another replacement. To handle this, the optimizer reads an entire procedure into an array, where it is held until optimization is complete. Whenever a match is found with a pattern, the replacement is made. At that point, the position in the array is moved backwards an amount equal to the largest of the patterns. In this way, very large replacements are possible even though only small patterns are in the replacement table.

There are advantages and disadvantages to this approach. The main advantage is speed. Since the patterns are constructed by hand well in advance of the actual use of the optimizer, they can be compiled in. Methods that "discover" optimizations during the actual run are considerably slower. Similar to Lamb's work, the patterns are constructed by the simple method of examining the compiler output (both EM for the intermediate optimizer, and actual machine code for the final pass) and writing appropriate patterns. Thus, there is the possibility of missing some optimizations which could be made, merely because a particular sequence of instructions did not occur in the output of the test programs.

4.5. PO and HOP

By far, the majority of research concerning automatic generation of peephole optimizers has been done by Davidson. Originally, Christopher Fraser designed and implemented a peephole optimizer in a 180 line LISP program named PO [10].

Fraser and (primarily) Davidson then continued research into PO for another decade, producing several papers on their progress.

Given an assembly language program and a machine description for the processor in question, the original PO program simulated adjacent pairs of instructions by combining their descriptions. The combined description was then used to search for another, single instruction which could replace the original two. PO makes one pass to outline the machine description of each instruction, and another pass to reduce pairs. A final pass replaces each pair with a cheaper alternative, where possible.

The machine description language Fraser used is similar to ISP. PO combines lexically adjacent instructions by combining their ISP notations. Fraser gives this example:

```
ADD #-2,R3 ; R3 is the destination in this case
CLR @R3
```

which, in ISP is:

```
R[3] <- R[3] + -2; N <- R[3]+(-2) < 0;
Z <- R[3]+(-2) = 0

M[R[3]] <- 0; N <- 0; Z <- 1
```

PO then combines these to obtain:

```
R[3] <- R[3] + -2; M[R[3]] <- 0; N <- 0; Z <- 1
```

which is the PDP instruction "CLR -(R3)".

Since the original incarnation of PO was simply to prove the concept, the actual implementation was unbearably slow, optimizing 2.5 instructions per second on his PDP-11. However, the conclusions were valid: that PO was feasible to do in a real system. With this in mind, development continued.

One early addition was the combination of triples instead of pairs of instructions. This is necessary because some machines provide simplified instructions for load/operate/store sequences, but few offer instructions to combine load/operate and operate/store combinations. Thus, it is necessary to check for triples in some cases [7]. PO was also reimplemented in SNOBOL at about this time, by a preprocessor that converts ISP into SNOBOL.

In about 1984, Davidson used PO for the basis of another experimental peephole optimizer, HOP [8][11]. The idea behind HOP is to use the patterns derived from PO to generate templates at compile-compile time. These templates are the matches which were discovered by PO. HOP patterns are encoded as text with embedded pattern variables:

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
```

specifies that register transfers such as:

```
r[2] = m[X]
r[2] = r[2] - m[Y]
```

could be replaced by

```
r[2] = m[x] - m[y]
```

HOP patterns are generated by running PO on what Davidson calls a "training set". The patterns are then put into a hash table used by HOP at compile time.

HOP matches patterns as the strings are read in, without actually using string manipulation routines. The pattern retrieved from the operation code is used to locate applicable patterns in the hash table. The patterns in the table have a replacement "skeleton" which is used for the generated code. The variable parts in the replacement skeleton are filled in with the operands from the original instruction.

The design of HOP makes it easiest to use at the "tail end" of a compiler. Davidson does note that if assembly language is first translated into the register transfer language, assembly language can be optimized as well.

Like ACK, there are both advantages and disadvantages.

The primary disadvantage to machine description methods is their speed. Machine-directed optimization uses no patterns, so they are thorough, but slow. Their thoroughness allows a very naive code generator, but the verbose code emitted by it makes the speed even more critical.

Later revisions in the PO system by Davidson sped up the throughput by a factor of five. But Davidson sites the system processing 10 instructions per second on a VAX 11/780, whereas the native compiler processes 40. After the implementation of HOP, however, the peephole optimization rules were generated and stored in advance; this caused the HOP process to perform replacements at a rate of approximately 100 per second on the same machine. (Note that this is replacements, whereas the figure 40 is apparently the rate at which instructions were processed from the compiler.)

This incarnation of PO used three sections: Cacher, Combiner, and Assigner [6]. The cacher program executes the internal representation symbolocally and records the values that they compute. When the repetition of an existing value is found, a search is made to locate the most recent occurrence and an attempt is made to reuse the value computed earlier. These rules allow cacher to intercept and eliminate redundent register loads, common subexpressions in the intermediate language, and the like. The combiner phase permits instruction reordering and replacement to take place so that the output from the assigner phase, which translates the RTL to assembly language, produces optimal code. It is the combiner phase which is typical of earlier work by Davidson, where symbolic instructions are put together and replaced with singletons where appropriate.

The PO system was next converted into a full code generator operating on RTL [4]. Register transfer language is a notation similar to ISP. Any RTL is machine specific, but the general format of RTL is machine independent. Davidson used an intermediate representation which was then translated into naive but nonetheless correct object code for the target hardware. The code was inefficient, but simple to produce. The object code is then passed to PO for optimization.

The large addition at this phase of the development is the integration of reordering in the optimizer. Davidson used a well known algorithm for discovering optimal evaluation ordering of expressions. The RTL notation is similar to that which is used for the internal representation of high level language expressions; as such he was able to insert many optimizations which are normally considered global into the optimization of the object code. An example is elimination of common subexpressions, and the movement of blocks of object code to reduce the number of register loads and stores required by the object code.

The advantage of the machine description method is that nothing is missed, and the optimizer is retargeted by simply writing a new machine description for a new CPU. Davidson has retargeted to a new CPU in as little as five person-days [5]. This is contrasted with classical peephole optimizers in which a few, hand written patterns must suffice. However, more recent work [25] indicates that a methodology more akin to expert systems might be an easier way to retarget.

5. Rule-Based Versus Architecture-Based Optimizers

In the previous chapter, there were two basic research camps. One of the groups believes that peephole optimization should be done with a machine description which is fed to an optimizer generator. In this scenario, the possible combinations of instructions are simulated, and tables are constructed for converting two instruction sequences into singletons. The majority of Davidson's and Fraser's work has been done in this area. The other camp, led by the likes of Tanenbaum, Lamb and others, believes that a human, examining code output from the compiler, should construct a rule set to perform the optimizations.

There are, of course, advantages and disadvantages to both approaches. For instance, automatically generated optimizers have these advantages:

- No instruction combination is missed, since every combination is attempted when the optimizer is generated. Unfortunately, no information specific to a given program can be exploited.
- The representation of an instruction is very specific; that is, the ISP (or other notation) for an instruction indicates exactly what that instruction does. This allows the section of software which symbolically combines instructions to work well regardless of the

processor that the optimizer is being constructed for. This is contrasted with rule methods, where the instruction itself must serve as the only indication to the author of what the instruction does.

The approach has the following disadvantages, as well:

- The action performed by each and every instruction must be specified, even if that instruction is never used in any replacements.
- Since the current trend in code generation is to use architecture descriptions, the peephole optimizer tends to "sneak into" the code generation phase. This is not as clean an implementation as making the optimizer a separate pass, akin to a "pipe".
- Many implementations of architectural description driven optimizers operate on a peephole window which is only two instructions long. This does not allow for many of the more common instruction sequences of length three.

Rule-based optimizers, transforming compiler output according to human-generated specifications, have the following advantages:

- It is easy to see inefficiencies in the generated assembly code and to construct patterns to fix them.

- The separation of patterns, generator and optimizer makes the maintenance of the rules themselves easy.
- Rule based peephole optimizers, when used on intermediate code in addition to final code, allow the same optimizer to be reused for many different high level language front ends. This may not be true if different languages map to different internal representations.
- Perhaps the biggest advantage is that the rules are easy to construct and understand. This is because the notation used to describe the pattern transformations is almost identical to the assembly language programs upon which the transformations are taking place.

Unfortunately, these optimizers have disadvantages as well:

- Depending on the format of the rule language, it may be necessary to have several rules that apply to the same type of instruction. Lamb encountered this when optimizing for the VAX architecture, which has slightly different operation mnemonics depending on the size of the operands involved [16]. He chose to include a different rule for each operand size.
- The ordering of the rules is sometimes important. If a given rule changes a move of zero into a clear, for

example, another larger rule may be disqualified because it is expecting the move instruction. Thus, it is important to order the rules according to the internal replacement policy used by the optimizer.

An interesting note is that there are a few drawbacks that are common to both approaches. Most prominent is that even well-constructed peephole optimizers may inadvertently cause working code to fail. For example, in an open letter Steven Pemberton [21] points out that it is not always a safe assumption that addition is associative. Specifically, on a digital machine it is not always true that A+(B+C) is the same value as (A+B)+C, because one may generate an overflow condition within the processor while the other does not. Therefore should it be legal to arbitrarily change the associativity of an expression? If the programmer has explicitly ordered an expression to prevent overflow, reordering the expression inside of the compiler will cause some confusion.

Another problem with the implementation of both methods is that intervening instructions may cause a rule or replacement match to fail. In other words, a three instruction group may be eligible for replacement but not be replaced because there is an extra instruction included amid the three. Davidson and others have made efforts to locate and handle these by matching near adjacent or semantically adjacent instructions instead of lexically adjacent ones. This is a difficult problem to overcome with rule-based optimizers, as it is not easy to check all combinations of "local" instructions against all patterns.

Which technique is better is an ongoing debate.

Some recent research uses an expert systems approach which is basically a restatement of the rule-based method. Warfield and Bauer [25] describe a technique using a combination of different descriptions. The descriptions and the rule set are both written in a LISP-like notation named MRS, and include notation for the instructions themselves, their addressing modes, and the replacement rule set. Using an expert systems "engine", the optimizer rules return a list of every possible optimization, and what conditions must be true in order for the replacement to take place. These rule lists form the optimizer itself. Note that unlike Davidson's method, the optimizer does not need a "training set" of programs. MRS is a combination approach, where a machine description discovers the rules, and the rules are later used for the optimizer. This may prove to be a good method, since it combines the best of both worlds, but the technique needs further study.

A surprising indication that rule-based optimizers might turn out to be the superior method is given by

Davidson [9] in his description of VPCC (the Very Portable C Compiler). This is the latest article from a succession of ten years of research, and signals that he has switched from the architectural method to the rule method:

> "Although abstract machine modeling simplifies the construction of a retargetable compiler, the emitted code is usually substantially inferior to the code produced by a compiler constructed using a sophisticated code generator. This paper describes a portable C compiler that uses abstract machine modeling for portability, and a simple rule-directed peephole optimizer to produce reasonable object code. The interesting aspect of this compiler is that for all four machines no more than 40 peephole optimization rules were required to produce code that was comparable to the code emitted by compilers that used sophisticated code generators."

This is intriguing in that Davidson indicates that rule based optimizers may be superior. Furthermore, Davidson uses a stack based architecture, translating the stack operations into machine instructions and optimizing the machine instructions. This is quite similar to the approach by Tanenbaum, van Staveren, and Stevenson. In fact, Davidson is now using rules such as:

```
moval %1,r%2
movl (r%2),%3
=>
movl %1,%3
```

This rule replaces a load address and a load indirect instruction with the shorter equivalent move. Davidson allows a limited set of regular expressions to be used in the "%i" notation. For instance "%i[bwld]" would match one of the single characters within the brackets. Regular expression notation is outlined in the next section.

In Davidson's new rule method, the equivalent numbered "%i" expressions must match - so the "r%2" in the first instruction must match with the same string in the second instruction. There is also the provision for calling functions in the implementation language (initially an interpreter and finally C) as pioneered by Lamb. The emitted assembly language is scanned in reverse, as before, with match and replacements changing the contents of a linked list.

One new idea put forward by this new research is that the patterns are compiled in such a way that not every pattern need be checked. In the above example, suppose the first instruction in the attempted match did not begin with the string "mov". All other patterns which begin with "mov" can also be ignored, since this is a fixed string. Other research has simply attempted each pattern in a set until either a successful match is located or the rules are exhausted.

The results cited by this research show that the approach yields code which is equivalent to the VAX C compiler. Only 39 rules were used to optimize the object code to this level, and approximately this same number were required to optimize for the Motorola 68020, the Intel 8086, and the Sun 3/75 processors.

Davidson also notes:

> "Lamb describes a peephole optimizer similar to the one described here. This optimizer was used to improve the code emitted by a prototype Ada compiler that generated code for the VAX-11."
>
> "Although a production version of Lamb's optimizer was never constructed, it was felt that the optimizer could be made to run quite fast. Our experiment confirms this."

For these reasons, the following research has been based on optimizations which are using a rule-based approach.

6. Optimizing from Assembly Language

Much of the previous work summarized above optimizes machine language prior to its being emitted from the back end of the compiler. Davidson notes that assembly language can be used with his optimizer, PO, if the language is first translated into his intermediate representation, ISP. However, others, including Lamb, are operating more directly on assembler language as strings instead of as machine descriptions. Even Lamb's work, though, dealt with a notation written in assembler but translated into LGN for the actual optimization. In addition, Lamb's work dealt with only those transformations which had no side effects such as eliminating redundant loads to registers. Is it possible to perform peephole optimization directly on assembler language with nothing to tie in with the original compiler?

In this chapter we will discuss an optimizer which does. Given a set of replacement rules, the optimizer compiler generates an executable program to transform assembly language programs using these rules. The optimizer is named OP, since it is an opposite approach from Davidson's work with PO. Specifications are written in a rough language named OPSPEC; a program reads in the OPSPEC source and generates an executable program as described above. The resultant program will read in assembly language and convert it according to the rule set outlined in the original OPSPEC program.

OPSPEC specifications are written using regular expressions. The lexical analysis generator program Lex, described in chapter one, uses regular expressions in a similar manner. The example Lex program did not contain any instances of complex regular expressions, but we could have simplified it by including some:

6.1. Regular Expression Notation in OPSPEC

The string "[A-C]" is an example of a regular expression. In the simple case, a string of letters in the alphabet of the language constitutes a regular expression. The string "112233" would match any instance of "112233" in the input, in the same way that plain "a" matched in the example.

Strictly speaking, a regular expression is a formula that is used to describe strings which either pass or fail when matched against the formula. Elements in a regular expression serve to represent the concepts of union, intersection, and closure of sets of characters. These special elements are metacharacters, and the metacharcters usually provided in regular expressions are also provided by the OPSPEC language:

X Single characters represent themselves. The letter

	X would successfully match if "X" was in the input.
AB	Two regular expressions, when they are adjacent, require the input string to have matching adjacency. The expressions are concatenated. The input string "A" followed by "B" would match, but "A" followed by "C" would not.
.	The period character represents any single character in the input stream.
+ and *	The plus and asterisk represent the closure operation. Closures can be either Kleene closure, "*", which matches zero or more if the preceding regular expressions, or positive closure, "+", matching one or more of the preceding expressions.
^	The circumflex is used to anchor the regular expression to the beginning of the line. "^AB" would match only if the input string "AB" was the first thing on the current line of input. The circumflex is not normally used in specifying expressions in OPSPEC, but is prepended to input patterns internally.
$	Similarly, the dollar sign anchors the regular expression at the end of a line of input. Although this is valid in OPSPEC terms, it is not normally used.
[A-F]	Brackets are used to indicate a set of characters

	that are allowed at this position in the input. In this case, the class would match any single character "A" or "B" or ... up to "F". If the first character inside the left brace is the circumflex character "^", then the input must have any character except one of the set in the input.
X{2,5}	Indicates that the preceding regular expression X must be present at least two consecutive times, and matches up to and including five occurrences.
(exp)	When a regular expression is contained within parenthesis, the resultant matching characters are saved in a variable.
\\	The backslash character, when doubled, represents an escape which can be used to negate the meaning of any of the above characters. The string "\\$" would be used to match the dollar sign instead of the end of an input line. This is most frequently used to escape the parenthesis characters.

These regular expression characters can be combined to make complex forms. The expression "[A-Za-z?][A-Za-z0-9?@]*" might be used to represent variable names, which typically must start with a letter and can be optionally followed by letters, digits, and some special characters. The string "max5" is an example of a matching string. Note the use of "*" to indicate closure; zero or more occurrences of the following letters and digits are allowed.

6.2. The OPSPEC Rule Language

In OPSPEC, regular expressions are used as the basis of the match and replacement rules defined for the language. Keywords serve to differentiate the sections. An example rule and replacement might be:

```
when  ADD #-2,R3
      CLR @R3
use   CLR -(R3)
```

This is not a realistic case, as it only matches register three; but it serves to illustrate the rule structure itself. There are one or more lines of input pattern following the keyword "when". There are one or more replacement strings after the keyword "use". If there is an instance of the two adjacent instructions:

```
ADD #-2,R3
CLR @R3
```

the sequence is replaced with the one instruction following the "use" keyword. This rule matching and replacement proceeds backwards, as described by Davidson's work earlier, and as such the newly inserted software may itself be replaced by this or other rules.

Obviously, it is undesirable to require the user to specify the above rule for every one of the available registers on the machine. Instead, using regular expressions helps:

```
when   ADD #-2,R[0-7]
       CLR @R[0-7]
use    CLR -(R3)
```

This is a good idea, but there is one shortcoming. In the case where we match with register five ("R5") register three is still used to clear the memory. Also, if the "ADD" instruction indicates register five, but the "CLR" instruction is for register two, the instruction and replacement shouldn't even apply.

This is solved in two ways. First, certain portions of the instruction can be copied from the input into temporary space. Second, the temporary spaces can be used in an optional "if and only if" clause for the rule.

```
when   ADD #-2,(R[0-7])$0
       CLR @(R[0-7])$1
use    CLR -($1)
iff    {
       return( ! strcmp( $0, $1 ) );
       }
```

Suppose the input matches the above rule. The special variable "$0" is instantiated with a string containing the register used in the "ADD" instruction. The special variable "$1" is filled in with the register from the second variable. If the two variables are the same, we wish to replace the two instructions with the single "CLR" instruction, filling in the appropriate register. Using "$1" (or "$0" as well) causes the "use" section of the rule to fill in with the correct register number. Thus, the rule mow can match any of "R0" through "R7".

A special "iff" rule is supplied in this example. The "iff" rule is an "escape" in the same sense as other research previously discussed. In the case of OPSPEC, the host language is C, and the only requirement of the "iff" rule is that it returns either true or false, dependent on whether the rule is to be replaced. The escape is compiled into the resultant optimizer. In our case, we use "strcmp()" to determine whether or not the operands in each instruction specify the same register. Any support routines needed by the "iff" clauses can be included at the bottom of the OPSPEC source file, after a special token.

The OPSPEC language allows for comments, starting with "#" in column one, and the definition of shorthand notation. A keyword "define" is used to cause textual replacement of one string with another in the source file. Typically this is used for the definition of labels, registers, and the like:

```
# Assembler language labels
define LABEL    [A-Za-z?][A-Za-z0-9?@]*
# Number will also match hex
define NUMBER   [0-9a-fA-F]+[hH]{0,1}
```

which indicates that whenever the string "LABEL" is encountered in the OPSPEC source, it is to be replaced with the regular expression on the right side of the definition. Note that the specification of "NUMBER" will also match hexadecimal values such as "12acH", used by some assembly languages.

The "define" directive is a textual replacement only. Some care must be exercised to assure that larger definitions occur first in order to avoid conflicts. The fact that replacements happen at arbitrary points in the input source is useful in some instances:

```
define NUMBER  [0-9a-fA-F]+[hH]{0,1}
define IREG    I[XY]
define INDEXED \\(IREG[+-]NUMBER\\)
```

causes the token "INDEXED" to now be replaced with the pattern "\\(I[XY][+-][0-9a-fA-F]+[hH]{0,1}\\)". This matches such strings as:

```
(IX+12H)
(IY-100)
```

With the combination of "define" and replacement rules, quite a bit of processing on input source files can be performed. Regardless, other optimizations can not be performed with these constructs alone. Consider this rule for an arbitrary processor:

```
define REG      R[0-7]
define LABEL    [A-Za-z?][A-Za-z0-9?@]*

# Operands are destination,source
when  LOAD      (REG)$0,\\((LABEL)$1\\)
      ADD       (REG)$2,(NUMBER)$3
      STORE     \\((LABEL)$4\\),(REG)$5
use   ADD       ($1),$3
iff   {
      return( ! strcmp( $0, $2 ) &&
              ! strcmp( $2, $5 ) &&
              ! strcmp( $1, $4 ) );
      }
```

Note the presence of a rather interesting regular expression "\\((LABEL)$1\\)". This matches input strings like "(Lab1)" and extracts the string "Lab1" into special variable "$1" without the parenthesis. The double backslash indicates that not all of the parentheses are part of the regular expression.

The rule replaces sequences similar to:

```
LOAD     R4,(Lab1)
ADD      R4,12
STORE    (Lab1),R4
```

with:

```
ADD      (Lab1),12
```

This initially appears to be an excellent example of replacement. However, if subsequent instructions make use of the value currently in register four, the replacement causes the software to no longer function. Obviously this is not desirable, and some method must be available to determine if the value is needed later. If not, the variable is said to be "dead", and the replacement is safe. If the value is needed for succeeding instructions the register is "live".

To solve this problem, we need discover whether the variables are live or dead. Since we are progressing from the bottom to the top of the input source, a means to detect instructions which cause a variable to change from live to dead or vice versa would suffice. Two keywords "kill" and "used" match input statements and cause the state of variables to change. Kill patterns extract special variables and move them from a list of live variables onto a list of dead variables. Used patterns place a special variable onto the list of live variables, and remove them from a list of dead variables.

Again, using our hypothetical instruction set:

```
define REG      R[0-7]
define LABEL    [A-Za-z?][A-Za-z0-9?@]*

kill  LOAD (REG)$0,
kill  LOAD \\((LABEL)$0\\),
used  ADD (REG)$0,(REG)$1
used  ADD \\((LABEL)$0\\),(REG)$1
```

Obviously, this example is not complete. But it illustrates what happens with the following instruction sequence. The dead and live lists are extracted from a run of OPSPEC on the above input to generate OP, and then running OP on the following assembly language source. A run-time flag causes the dead lists to be dumped, as an aid to the OPSPEC author.

```
                    K  dead -> [R1, VARIABLE] used -> [R2]
LOAD    R1,23
                    KU dead -> [VARIABLE] used -> [R1, R2]
LOAD    (VARIABLE),R1
                          U used -> [R1, R2, VARIABLE]
ADD     R1,R2
                            U used -> [R2, VARIABLE]
ADD     (VARIABLE),R2
```

This should logically be read from the bottom to the top. The last "ADD" instruction uses both "R2" and the contents of "VARIABLE", so both are placed onto the used list. Next the contents of "R2" are added to "R1", so "R1" is also in use. But when "LOAD" is discovered, the implication is that the contents of "VARIABLE" are destroyed by the instruction. Anything above this instruction can safely rewrite source code such that references to "VARIABLE" are eliminated. The lists propagate backwards from instruction to instruction, and a special function named "dead" is available in "iff" clauses to indicate weather a given string is present on the dead list.

One exception is needed when an instruction implies that a register or variable is in use, even though it is not apparent in the instruction. An example is a processor which does mathematical operation only on the accumulator, a special register, and the accumulator is not mentioned in the instruction per se. The "implied" keyword can be used in these cases:

```
ADD  R5       implied R0
```

This adds "R0" to the used list and eliminates it from the dead list, even though the instruction itself makes no mention of "R0".

Now we can replace the above rule with:

```
define REG      R[0-7]
define LABEL    [A-Za-z?][A-Za-z0-9?@]*

# Operands are destination,source
when  LOAD      (REG)$0,\\((LABEL)$1\\)
      ADD       (REG)$2,(NUMBER)$3
      STORE     \\((LABEL)$4\\),(REG)$5
use   ADD       ($1),$3
iff   {
      return( ! strcmp( $0, $2 ) && /* same reg */
              ! strcmp( $2, $5 ) && /* same reg */
              ! strcmp( $1, $4 ) && /* same label */
              dead( $0 ) );         /* reg is dead */
      }
```

At this point the replacement will only occur if the register indicated is not needed in the next group of instructions.

The remaining OPSPEC constructs are used to define what exactly the "next group of instructions" is. OP will only optimize within basic blocks of the assembly language source. A basic block in this sense is defined to be the group of input which occurs between instructions which cause a transfer of control. Jump, call, and branching instructions are transfer instructions. This requirement assures that a register that is on the dead list is genuinely assigned to (made dead) prior to jumping to a different section of the software. If this requirement were not made, the input source would need to be completely analyzed with regard to flow, rather than just lexically analyzed as it is now. The "trans" keyword gives a regular expression that causes a transfer of control within the source:

```
trans  CALL    LABEL
trans  CALL    [CMNCZPEO],LABEL
trans  JUMP    LABEL
trans  JUMP    [CMNCZPEO],LABEL
trans  RET[IN]{0,1}
trans  RET     [CMNCZPEO]
```

6.3 OP - The Executable Optimizer

There are several steps which need to be accomplished in order to create a new optimizer from scratch.

Normally, an author familiar with the assembly language if the machine being targeted would start by using "define" to supply more legible expressions for the more commonly encountered substrings in the assembly source code. Examples of these are the registers, labels, and such used in the language. One of the OPSPEC compiler options allows the author to dump the regular expressions which will be replaced by these definitions, so nested "define" statements can be checked.

Next, the "trans" patterns are written. These can be checked by compiling the OPSPEC source, and enabling an option that displays the dead variable lists. Whenever a "trans", "used", or "kill" pattern is encountered, a "T", "U", or "K" is printed as a part of the variable list. Testing on some sample compiler output enables the author to verify that the ends of basic blocks are being detected.

The third step is to write the "used" and "kill" patterns so that the dead variable lists can be maintained. In the trivial case, a complete list of all possible opcodes could be used with patterns taking the place of the operands, but this is obviously cumbersome. Instead, the simplest method is to add patterns to one or the other lists, regenerate the optimizer, and check the dead lists with checking enabled. Then it is necessary to manually inspect the combination of assembly language and lists to determine which changes or additions need to be made. One slight complication is that the source is emitted in reverse when this checking is enabled, since it is being scanned from the bottom up. An extra "reversing" program is used to present the data in a more nominal form.

In the current implementation, the patterns are simply scanned with linear search, so the order that the patterns are listed can have some impact (and, in fact make the job easier).

Once the dead lists are known to be correct, the last step is simply inspecting the code and authoring rules as required. Assembling the resultant code and verifying that the results are still correct is obviously an easy method to determine if incorrect rules or changes have been made. For this reason it is easiest to add a rule, verify several test programs, add the next rule, et cetera. Although this ad hoc method appears both cumbersome and error-prone, it is actually rather simple in practice.

Several additional options are available in the OPSPEC compiler itself to aid in detecting patterns which look good but do not work as desired. Occasionally a manual inspection of the regular expression programs that are created is necessary. Since the generated optimizer acts as a filter, though, simple checks can often be made by just running the program and typing in simple tests.

6.4. Experimental Results and Possible Improvements

Typically, a few days are sufficient if a good understanding of the machine assembly language is present before starting. One of the test systems utilized in this research took several weeks, but this was primarily due to debugging both the specification itself and the OPSPEC processor.

Currently the system has been tested on two different types of assembly language input.

One of the difficulties in testing OP has been obtaining a front end compiler that produces assembly language which is not very good! It was desirable to restrict the research to the optimization phase only, and not to "reinvent the wheel" by having to produce a complete compiler. Naturally, commercial compilers are judged almost exclusively on the quality of the code they produce; it was first necessary to locate compilers which do not generate extremely optimal code. One way this was done was to locate a compiler for a microprocessor which has been supplemented by an upwards compatible model. The output from the compiler, therefore, will definitely not generate optimal code for the superset processor, since not all of the instructions were available when the compiler was completed. This approach was used by producing assembler software from a C language to Z-80 microprocessor compiler. This was then optimized to run on the newer Z-280 machine, which contains both more numerous and more powerful instructions [26].

The other testing was performed using a common public domain C compiler known as Tiny-C.

The testing on the Z-280 processor was performed primarily by compiling software which implements the data encryption standard. This was a useful standard because there is significant processing, but little input or output. Additionally, the results are either correct or not, so judging the validity of the replacement specifications was easy: add one, and test the output to make sure it is the same. An additional nicety of this test is that there are numerous shift and multiply operations which can be optimized.

Since unoptimized source was not available, it is difficult to compare the results of peephole optimization which were obtained by others with that done in this research. However, Tanenbaum has noted an approximate 10-15% improvement in the number of instructions, and this is comparable with the results obtained on the Z-280 processor. Only seven rules were utilized, but the savings was 11% in instruction count. Tanenbaum's results are on unoptimized source, while the input to OP was, in theory, good code to begin with. The OPSPEC input for this test is shown in the appendix.

In the second case, the public domain compiler was not capable of compiling the encryption tests. Instead, Davidson's test function named "ctoi" was compiled to 8086 assembly language [12]. The results for this test were similar but not as dramatic. Five rules yielded a savings of approximately seven percent. This figure is possibly less than the first test because the input to OP is significantly shorter than when the full encryption test is run.

The current implementation of OP was written to show the feasibility of the system, and as such it is not written with efficiency in mind. For example, the dead, used, and trans keywords, along with the matching rules, are all scanned linearly. This makes the execution slow, but allows a certain number of tricks to be done in the OPSPEC input, since very generic patterns can be used. Currently the test loop for patterns terminates early when a match is found, so adding rules in a very specific order is necessary if it is not desirable to code every instruction operation code.

Future implementations would and should use lexical rules, just as Lex does, to scan the input and immediately match the appropriate trans, used, and dead patterns. This would be a requirement if the system were to be used in a production environment. One way to achieve this aim would be to reformat the OPSPEC input into separate files that are valid programs for Lex, then to have Lex make individual functions for each type. These would be combined into the OP program itself, resulting in a practical implementation.

Another implementation problem would need to be changed in a production environment as well. Currently, labels in the assembly source must be on their own line in order for successful matches to take place on the first instruction in the basic block. However, the method in place at this time uses a preprocessing step (stream editor) to put each label on it's own line if it was not already.

One area not addressed by the current system is block move instructions. There is no provision for adding the destination of block moves or indirect stores onto the list of dead variables. Moreover since the destination can not necessarily be known at optimization time it is doubtful if there is an easy solution to the problem. Currently we store the destination of indirect stores by adding the indirect notation itself onto the dead list. An example using our generic assembly language would be:

```
        store R1,@A5    ; contents of R1 go to memory
                        ; addressed by register A5
```

which would cause the string "@A5" to be included in the dead list. However, the contents of "A5" can not be determined at optimization time. Currently, taking the safe approach of adding it to the dead list is sufficient if care is used in disallowing certain matches, but this is obviously a hindrance.

Lastly, there is currently little provision made for assembler input which includes the "$" symbol on valid input. The VAX architecture is one such system. An easy way around this problem would be to allow the OPSPEC language to include a directive to change the variable designator from "$" to some arbitrary symbol defined by the programmer. It is not currently implemented because of the time involved to change the internals of the regular expression functions.

There is room for future research, also. One area that might bear fruit is to collect the dead and live lists at the top of each basic block of code. This information could be copied to any transfer instructions which jump to that block. This would carry over the dead register information, and might make additional optimization possible. Currently, any transfer of control causes the lists to be reset to empty states. An obvious problem would be looping constructs embedded within the assembly language source code.

In any case, using regular expressions and rules in combination does produce a workable peephole optimizer. No knowledge of the compiler producing the optimizer input is needed.

Appendix

Sample OPSPEC Input

```
# Simple-minded description for Z280 assembler output
# from the Computer Design Solutions C compiler operating
# in Z80 only mode. We can convert into quite a few of
# the z280 mnemonics this way.

# Registers, constants, etc.
#
# Nothing from the compiler is in octal, so we omit it.
# The compiler tends to put out either positive hex
# constants which we call NUMBER or signed decimal
# constants which we call SIGNED_DEC. The definition
# of REG8 is technically incorrect as it includes 'G',
# 'J', and 'K', but these are not encountered as labels
# always have '?' appended.

define  REG8            [A-L]
define  LABEL           [A-Za-z?][A-Za-z0-9?@]*
define  NUMBER          [0-9a-fA-F]+[hH]{0,1}
define  SIGNED_DEC      [-+]{0,1}[0-9]+
define  IREG            I[XY]
define  INDEXED         IREG[-+]NUMBER
define  GENERIC         [A-Za-z0-9?_()@]+

# Transfer instructions...

trans   CALL    LABEL
trans   CALL    [CMNCZPEO],LABEL
trans   J[PR]   LABEL
trans   J[PR]   [CMNCZPEO],LABEL
trans   JP      \\(HL\\)
trans   JP      \\(IX\\)
trans   JP      \\(IY\\)
trans   DJNZ    LABEL
trans   RET[IN]{0,1}
trans   RET     [CMNCZPEO]
trans   RST

# Data register and variable kill patterns...
# Ordering is critical in this issuance of opspec.

kill    IN      (REG8)$0,
kill    LD      (\\(REG8REG8\\))$0,
kill    LD      (\\(LABEL\\))$0,
kill    LD      \\((NUMBER)$0\\),
kill    LD      \\((INDEXED)$0\\),
kill    LD      (\\(LABEL[-+]NUMBER\\))$0,
kill    LD      (REG8)$0,
kill    LD        (REG8)$0(REG8)$1,
kill    POP       (REG8)$0(REG8)$1

# Operand used specification patterns...
# Some of these are too general, but simple.
# Again, ordering is all-important.

used    DEC       (REG8)$0(REG8)$1
used    DECW      (\\(LABEL\\))$0
used    INC       (REG8)$0(REG8)$1
used    INCW      (\\(LABEL\\))$0
used    EX        DE,HL   implied D E H L
used    LDIR?     implied B C D E H L
used    LDDR?     implied B C D E H L
used    MULTW     (REG8)$0(REG8)$1,(REG8)$2(REG8)$3
used    OTDR?     implied B C H L
used    OTIR?     implied B C H L
used    OUT[DI]   implied B C H L
used    PUSH      (REG8)$0(REG8)$1
used    [A-Z]+    REG8REG8,(REG8)$0(REG8)$1
used    [A-Z]+    GENERIC,(REG8)$0(REG8)$1
used    [A-Z]+    GENERIC,(LABEL)$0
used    [A-Z]+    GENERIC,(\\(LABEL\\))$0
used    [A-Z]+    GENERIC,(\\(INDEXED\\))$0
used    [A-Z]+    \\(LABEL\\),(REG8)$0(REG8)$1
used    OR        (REG8)$0

# Rule 0          When setting a variable to zero, they
#                 load HL with 0 and store HL into the
#                 variable. On the Z80 we can just set
#                 it directly.
#                 LD      HL,0
#                 LD      (R?1?),HL

when    LD        (REG8)$0(REG8)$1,0
        LD        \\((LABEL)$2\\),(REG8)$3(REG8)$4
use     LDW       ($2),0
iff     {
        return(   ( $0[0] == $3[0] ) &&
                  ( $1[0] == $4[0] ) &&
                  ( dead( $0 ) ) &&
                  ( dead( $1 ) ) );
        }

# Rule 1          This is a very popular thing to emit for
#                 a post-increment on a register variable,
#                 even if HL is not dead. This is always
#                 done with HL, but we'll be general here.
#                 LD      HL,(r?2?)
#                 INC     HL
#                 LD      (r?2?),HL
#                   DEC     HL

when    LD          (REG8)$0(REG8)$1,\\((LABEL)$2\\)
        INC         (REG8)$3(REG8)$4
        LD          \\((LABEL)$5\\),(REG8)$6(REG8)$7
        DEC         (REG8)$8(REG8)$9
use     LD          $0$1,($2)
        INCW        ($2)
iff     {
        return( ( $0[ 0 ] == $3[ 0 ] ) && /* same reg16 */
                ( $1[ 0 ] == $4[ 0 ] ) &&
                ( $0[ 0 ] == $6[ 0 ] ) && /* same reg16 */
                ( $1[ 0 ] == $7[ 0 ] ) &&
                ( $0[ 0 ] == $8[ 0 ] ) && /* same reg16 */
                ( $1[ 0 ] == $9[ 0 ] ) &&
                ( ! strcmp( $2, $5 ) ) ); /* Same var'l */
        }

# Rule 2            The above rule makes the safe assumption
#                   that the hardware register STILL needs
#                   to be loaded. But if it is dead, we can
#                   eliminate that, too. This rule "cleans
#                   up" the above rule.

when    LD          (REG8)$0(REG8)$1,\\(LABEL\\)
use
iff     {
        return( ( dead( $0 ) ) &&
                ( dead( $1 ) ) );
        }

# Rule 3            On the old machine a 16 bit load off of
#                   an index variable has to be done in two
#                   halves. Here we can just load all 16
#                   bits directly.
#                   LD      L,(IX+6)
#                   LD      H,(IX+7)
#                   Assume that the offset is base 10.

when    LD          (REG8)$0,\\(IREG[-+](NUMBER)$1\\)
        LD          (REG8)$2,\\(IREG[-+](NUMBER)$3\\)
use     LDW         $2$0,(IX+$1)
iff     {
        /* We must have a register pair like HL, DE,... */
        /* AND we must have adjacent integer offsets.   */
        return( ( ispair( $2[ 0 ], $0[ 0 ] ) ) &&
                ( atoi( $1 ) + 1 == atoi( $3 ) ) );
        }

# Rule 4            Replace library multiplication (signed,
#                   unsigned) This is something that a true
#                   280 compiler should be doing anyway,
#                   but they call a library function.
#                   This rule does not save instruction
#                   counts but does save time.

when    CALL    ?smult
use     MULTW   HL,DE

# Rule 5        No need to load 23 into a register to
#               do an add to HL.
#               LD DE,23
#               ADD HL,DE

when    LD      (REG8)$0(REG8)$1,(SIGNED_DEC)$2
        ADD     (REG8)$3(REG8)$4,(REG8)$5(REG8)$6
use     ADDW    $3$4,$2
iff     {
        return( ( $0[0] == $5[0] ) &&
                ( $1[0] == $6[0] ) &&
                ( dead( $5 )     ) &&
                ( dead( $6 )     ) );
        }

# Rule 6        Should be LD DE,(R?2?); ADD HL,DE and not:
#               EX      DE,HL
#               LD      HL,(R?2?)
#               ADD     HL,DE
#               Note that the only possible "EX" operands
#               are DE,HL in that order, so we don't bother
#               looking for generic registers.

when    EX      DE,HL
        LD      HL,\\((LABEL)$0\\)
        ADD     HL,DE
use     ADDW    HL,($0)
iff     {
        return( ( dead( "D" ) ) &&
                ( dead( "E" ) ) );
        }

# Anything after the %% is copied intact to the output.
# The following are user-supplied functions...

%%

/* Return true if the two strings      */
/* together make a register pair such  */
/* as HL, DE, or BC.                   */

int ispair( a, b )
char    a, b;
```

```
{
return( ( a == 'H' && b == 'L' ) ||
        ( a == 'D' && b == 'E' ) ||
        ( a == 'B' && b == 'C' ) );
} /* ispair */
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers Principles, Techniques, and Tools. Addison-Wesley Publishing, Reading, Massachusetts, 1986.

[2] C. G. Bell, and A. Newell. Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971.

[3] J. W. Davidson. Simplifying Code Generation through Peephole Optimization. Ph.D. thesis, Department of Computer Science, University of Arizona. December, 1981.

[4] J. W. Davidson. "A Retargetable Instruction Reorganizer". SIGPLAN Notices, 21(7):234-241, July 1986.

[5] J. W. Davidson, and C. W. Fraser. "Code Selection through Object Code Optimization". ACM Transactions on Programming Languages and Systems, 6(4):505-526,October 1984.

[6] J. W. Davidson, and C. W. Fraser. "Register Allocation and Exhaustive Peephole Optimization". Software Practice and Experience, 14(9):857-866, September, 1984.

[7] J. W. Davidson, and C. W. Fraser. "The Design and Application of a Retargetable Peephole Optimizer". ACM Transactions on Programming Languages and Systems, 2(2):191-202, 1980.

[8] J. W. Davidson, and C. W. Fraser. "Automatic Generation of Peephole Optimizations". SIGPLAN Notices, 19(6):111-116, June 1984.

[9] J. W. Davidson and D. B. Whalley. "Quick Compilers Using Peephole Optimization". Software Practice and Experience, 19(1):79-97, January, 1989.

[10] C. W. Fraser. "A Compact Machine-Independent Peephole Optimizer". Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, 1979.

[11] C. W. Fraser, and A. L Wendt. "Integrating Code Generation and Optimization". Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction. SIGPLAN Notices, 21(7):242-248, July 1986.

[12] intel. iAPX 86, 88 User's Manual. Santa Clara, California, 1981.

[13] S. C. Johnson. "Yacc: Yet Another Compiler Compiler". Computing Services Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[14] P. B. Kessler. "Discovering Machine-Specific Code Improvements". Proceedings of the ACM SIGPLAN 1986 Symposium on Compiler Construction. SIGPLAN Notices, 21(7):249-254, July 1986.

[15] R. R. Kessler. "Peep - An Architectural Description Driven Peephole Optimizer". Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction. SIGPLAN Notices, 19(6):106-110, June, 1984.

[16] D. A. Lamb. "Construction of a Peephole Optimizer". Software Practice and Experience, 11:639-647, 1981.

[17] M. E. Lesk. "Lex - a lexical analyzer generator". Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[18] E. S. Lowry, and C. W. Medlock. "Object Code Optimization". Communications of the ACM, 12(1):13-22, January 1969.

[19] W. M. McKeeman. "Peephole Optimization". Communications of the ACM, 8(7):443-444, July 1965.

[20] J. Nievergelt. "On the Automatic Simplification of Computer Programs". Communications of the ACM, 8(6):366-370, June 1965.

[21] S. Pemberton. "On Tanenbaum, van Staveren, and Stevenson's 'Using Peephole Optimization on Intermediate Code'". ACM Transactions on Programming Languages and Systems (letters) 5(3):499, July, 1983.

[22] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. "A Practical Tool Kit for Making Portable Compilers". Communications of the ACM, 26(9):654-660, September 1983.

[23] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. "Using Peephole Optimization on Intermediate Code". ACM Transactions on Programming Languages and Systems, 4(1):21-36, January 1982.

[24] J. P. Tremblay, P. G. Sorenson. The Theory and Prac-

tice of Compiler Writing. McGraw-Hill Book Company, New York, 1985.

[25] J. W. Warfield, and H. R. Bauer, III. "An Expert System for a Retargetable Peephole Optimizer". SIG-PLAN Notices, 23(10):123-130, October 1988.

[26] Zilog. Z280 MPU Microprocessor Unit Preliminary Technical Manual. Campbell, California, 1987.