MIT/LCS/TR-199

THE SPECIFICATION OF CODE GENERATION ALGORITHMS

Christopher Jay Terman

January 1978

MIT/LCS/TR-199

THE SPECIFICATION OF CODE GENERATION ALGORITHMS

by

Christopher Jay Terman

January, 1978

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

THE SPECIFICATION OF CODE GENERATION ALGORITHMS

by

Christopher Jay Terman

Submitted to the Department of Electrical Engineering and Computer Science on January 20, 1978 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

This thesis addresses the problem of automatically constructing the code generation phase of a compiler from a specification of the source language and target machine. A framework for such a specification is presented in which information about language- and machine-dependent semantics is incorporated as a set of transformations on an internal representation of the source language program. The intermediate language which serves as the internal representation, and the metalanguage in which the transformations are written are discussed in detail.

The major goal of this approach is to separate machine- and language-dependent knowledge (as embodied in a transformation catalogue) from general knowledge about code generation. This general knowledge is supplied by the third component of the framework: a metainterpreter incorporating a fairly complete repertoire of language- and machine-independent optimization algorithms for intermediate language programs. The metainterpreter is also capable of selecting and applying transformations from the transformation catalogue. The three-component framework described in the thesis provides a specification that can easily be tailored to new languages and machine architectures without compromising the ability to generate optimal code.

THESIS SUPERVISOR: Stephen A. Ward
TITLE: Assistant Professor of Electrical Engineering and Computer Science

Key Words and Phrases:
machine-independent code generation, compiler metalanguages

ACKNOWLEDGMENTS

First and foremost, I am indebted to my thesis advisor, Steve Ward, for his encouragement and support during the long gestation period of this thesis. His continuing interest both in my research topic and me personally was largely responsible for the final completion of this work. Without his insight and willingness to engage in discussion, I would still be vacillating between competing approaches.

I would also like to thank my compatriots in the DSSR group of the Laboratory of Computer Science for their friendship through the ups and downs of preparing this thesis. Bert Halstead and Clark Baker were uncomplaining sounding boards for many of my early ideas on code generation and I would be remiss if I did not mention the excellent facilities for text preparation developed by Jim Gula, John Pershing, Tom Teixeira, and Terry Hayes.

I would like to acknowledge the financial support of the Electrical Engineering and Computer Science Department and the Laboratory for Computer Science thus far during my graduate career here at MIT.

And of course I must thank my family for their support through many years of academe -- I am sure they must have wondered what I was doing for all those years at MIT.

TABLE OF CONTENTS

This empty page was substituted for a blank page in the original document.

CHAPTER ONE

§1.1 Introduction

The creation of a compiler for a specific language and target machine is an arduous process. It is not uncommon to invest several years in the production of an acceptable compiler; the excellent compilers available for PL/I on MULTICS and System 370 evolved over a decade or more. With the rapid development of new computing hardware and the proliferation of high-level languages, such an investment is no longer practical, especially if there is little carry-over from one implementation to the next.

Compiler writers currently suffer from the same malady as the shoemaker's children: they seem to be the last to benefit from the improvements in compiler language technology that problem-oriented language processors have incorporated. The current research has been directed towards providing the compiler writer with the same high-level tools that he provides for others. In an effort to automate compiler production, systems have been developed to automatically generate those portions of the compiler which translate the source language program into an internal form suitable for code generation. These systems have enhanced portability and extensibility of the resultant compiler without a significant degradation in its performance. The final phases of a compiler, those concerned with code generation, are now coming under a similar scrutiny. Many different approaches are possible (see §1.4); this thesis addresses the issue of providing a *specification* of a code generator. Such a specification is constructed by the code generator designer within a framework provided by an *intermediate language* (IL) and a *metainterpreter*. The intermediate language is used as the internal representation of the code generator — the initial input (provided by the first phase of the compiler) is a source language program expressed as an IL program; the final output is the IL representation of the target machine program. The metainterpreter has a detailed understanding of the semantics of IL programs and is capable of performing many transformations and optimizations on those programs. The semantics of IL are limited to concepts common to many languages and machines: flow of control and the management of names and values are the only primitive concepts. Specification of machine- and language-dependent semantics (e.g., the semantics of individual operators) are provided by the designer in the form of a *transformation catalogue*. In essence, the semantics of IL serve as common ground on which the designer (through the transformation catalogue) "explains" the source language and target machine to the metainterpreter which then performs the appropriate translation. This "explanation" is in terms of a step-by-step syntactic manipulation of the IL program; each transformation accumulates additional information for the metainterpreter or provides possible translations for IL statements which are not yet target machine instructions. Since the metainterpreter incorporates many of the optimizations commonly performed by compilers, the specification need not supply detailed implementation descriptions of these operations.

One can envision several distinct uses for such a specification:

- as a convenient way of replacing English descriptions of an algorithm (much the same way a BNF documents syntactically legal programs);
- as a program which, along with a specific input string, can be interpreted to produce an acceptable translation (e.g., syntax directed translation based on a parse of the input string); or
- as an input to a system which automatically constructs a code generator (similar to the various specifications fed to a compiler-compiler).

Each successive use requires a more thorough understanding of the specification but repays this investment with a corresponding increase in the degree of automation achieved. The increase is based for the most part on a better understanding of the interaction between components of the specification. Automatic creation of a code generator from a specification would require extensive analysis of these interactions, a capability only now just emerging from artificial intelligence research on program synthesis [Barstow]. Fortunately most of the analytical mechanism required is in addition to the facilities provided by the metainterpreter and intermediate language -- it is reasonable to expect that future research will be able to extend the framework described in the preceding paragraph to allow automatic construction of a code generator. This thesis concentrates on developing the framework to the point where it can be used interpretively (as suggested by the second use): implemented in a straightforward fashion, the metainterpreter can perform the translation by alternately applying transformations from the catalogue and optimizing the updated IL program. While this approach is admittedly less efficient than current code generators, it represents a significant step towards separating machine and language dependencies in a declarative form (the transformation catalogue) from general knowledge about code generation (embodied in the metainterpreter).

The following section provides a brief overview of the tasks confronting a code generator. §1.3 presents a summary of the salient features of IL, the transformation catalogue, and the metainterpreter. In §1.4, related work is discussed with an eye towards providing a genealogy for the research reported here. Finally, §1.5 outlines the organization of the remainder of the thesis.

§1.2 Setting the stage

Before embarking on a discussion of the proposed formalism, let us first characterize the nature of the task we wish to describe:

> code generation is the translation of a representation (in some intermediate language) of the computations specified in the original source language program into a sequence of instructions to be directly executed by the target machine.

The idea, of course, is that by executing the resulting sequence of machine instructions the target machine will carry out the specified computation. The remainder of this section outlines the tasks confronting a code generator; our objective is to sketch the variety of knowledge needed for making decisions during code generation and how current code generators embody this knowledge.

An optimizing code generator is organized around three main tasks:

machine-independent optimization
↓
translation to target machine instructions
↓
machine-dependent optimization.

Machine-independent optimizations include global flow analysis, constant propagation, common subexpression and redundant computation elimination, etc. – these transformations modify the semantic tree, producing a new tree which is strictly equivalent (i.e., equivalent regardless of the choice of target machine). Certain of these transformations do make general assumptions about the target machine architecture; for instance, constant propagation assumes that it is more efficient to access a constant than a variable. The more sophisticated code generators [Wulf] do not actually modify the semantic tree – they maintain a list of alternatives for each node in the tree†, postponing the choice of transformation

† They do not, however, list all possible alternatives as this would result in the combinatorial growth of the semantic tree. Searching the full tree for the optimal program accounts for the NP-completeness of the code generation problem [Aho77].

until the translation phase.

The translation to target machine instructions takes place in several stages:

(i) Storage is allocated for variables and constants used in the source program. The semantics of the program often require specific allocation strategies (e.g., stacks).

(ii) Algorithms which implement the required computations (FOR-loops, subroutine calls, etc.) are chosen.

(iii) The order in which computations are to be performed is determined. Through the detection of redundant computations, it is often possible to permute the evaluation order and realize savings of both time and space in the resulting code while maintaining the correctness of the computation.

(iv) Actual target machine instructions are generated. Machine-dependent considerations (such as locations of operands for particular operations, the lack of symmetrical operations, etc.) enter at this level.

From the many possible transformations applicable to a particular source program, an optimizing code generator chooses some subset to produce the "best" translation. These transformations are interdependent and an *a priori* determination of their combined effect is difficult.

Machine-dependent (peephole) optimization [McKeeman, Wulf: Chapter 6] of instruction sequences can be used to improve the generated code — just how much improvement can be made depends on the sophistication of the translation phase. The goal is to substitute more efficient instruction sequences for small portions of the code. Examples: elimination of jumps to other jumps and code following unconditional jumps, use of short-address jumps (limited in how far they can jump), elimination of redundant store-load sequences, etc. This phase is iterated until no more improvements can be made. Before the reader dismisses this final phase as "trivial," he should consider this comment from [Wulf, pg. 124]:

> ... all the fancy optimization in the world is not nearly as important as careful and thorough exploitation of the target machine. ... It is difficult to determine to what extent [this final phase] would be needed if more complete algorithms, rather than heuristics, existed in

> earlier phases of the compiler. However, since some of the operations of [this final phase] exist simply because the requisite information does not exist earlier, we suspect that there will always be a role for a [similar module] ...

It should be noted that relatively few of the transformations mentioned above are uniformly applicable. Unfortunately, the conventional control structures upon which extant code generators are based preclude a trial-and-error approach to optimization. The programmer, using his knowledge of the target machine architecture, must, out of necessity, incorporate in the code generator either some subset of the applicable transformations or *heuristics* to select the "best" transformation at specific points in the code generation process. These heuristics base their decisions on a local examination of the tree; more far-reaching consequences are difficult to determine – thus, most heuristics "work" for only a subset (albeit large) of the possible programs. Although the compromises inherent in heuristics serve primarily to reduce the amount of computation needed to complete the translation, they also embody knowledge helpful in the generation of code. Some of these transformations are of general use in that they are independent of both the intermediate representation and the target machine; these transformations form a nucleus of knowledge for the portable code generation system.

§1.3 Introduction to IL/ML

The framework for the specification of code generators provided by the IL/ML system has three basic components:

- an *intermediate language* (IL) which serves as the internal representation for all stages of the translation. At any given moment, the IL program embodies all the text, symbol table, and state information accumulated by the code generator up to that point in the translation.

- a *transformation catalogue* whose component transformations are expressed in a context-sensitive pattern-matching metalanguage (ML) as pattern/replacement pairs. The pattern specifies the context of the transformation as an IL program fragment; the replacement is another fragment to be substituted for the matched fragment.

- a *metainterpreter* incorporating a fairly complete repertoire of machine- and language-independent optimization algorithms for IL programs. The metainterpreter is also capable of selecting and applying transformations from the transformation catalogue.

Within this framework, code generation may be viewed as follows†: the transformation catalogue is searched by the metainterpreter until a pattern is found that matches some fragment of the current IL program, then the corresponding replacement is substituted for the matched fragment creating an updated version of the IL program. Next, the metainterpreter optimizes the new version of the program utilizing new information and opportunities presented by the transformation. This cycle is repeated until no further matches can be found, at which point the translation is completed. The simplicity of the mechanism, along with the modularity of the transformation data base, make this an attractive basis for a code generator specification.

Only concepts common to most machine and source language programs have been incorporated into IL and the metainterpreter – concepts specific to a machine or language are introduced by the designer through the transformation catalogue. Many of these new concepts need never be related to the primitives of IL: they can be introduced into the IL program as *attributes* of some component of the IL program where they can be referenced by transformations. The semantics of these attributes are established by the role they play in various transformations;

† This description is only a conceptual model; in a code generator constructed from the specification, the decisions inherent in choosing and applying a transformation would have been ordered by the metacompiler and incorporated in the organization of the code generator (actual searching would be seldom be done). Some decisions would be made during the construction of the code generator, others would be embodied as decision trees and heuristics. Other distinctions between interpretation and compilation of the specification are ignored until Chapter 5.

for example, the concept of an addition operator need only be related to the integer and floating-point addition instructions of the target machine – neither IL nor the metainterpreter have to support addition as a primitive operation. The ability to express source language semantics in terms of other, simpler operations and, ultimately, in terms of target machine instructions without recourse to some fixed semantics allows great flexibility without any attendant complexity in the intermediate language or metainterpreter.

But isn't it "cheating" to require the designer to spell out source language semantics in terms of target machine instruction sequences? Doesn't that raise the objection to conventional code generators, viz. that a large investment is necessary to redo the translations when another target machine or source language is to be accommodated? No, not really. There is no "magic" provided by the IL/ML system – the semantics of the source language and target machine must always be described by the designer in any truly language- and machine-independent system. However, their most natural (and useful) description is in terms of one another – after all, the designer in theory fully understands both and the simplicity of the IL/ML system minimizes the need for expertise in any other language/interpreter. Moreover, since the metainterpreter incorporates the necessary knowledge about general optimization techniques, the overhead of the description is small compared to coding a conventional code generator. It is true that a more highly specified intermediate language semantics might be more appropriate for a specific source language and target machine, but such constraints impede the transition to other languages and target machines (see description of abstract machines in §1.4). Since IL/ML is to be a *general purpose* code generation system, such constraints have been avoided.

§1.3.1 A syntactic model of code generation

One of the most useful discoveries of artificial intelligence research is that complicated semantic manipulations can be accomplished with step-by-step syntactic manipulation of an appropriately chosen data base (see, for example, [Hewitt]). This section explores the application of this approach to the process of code generation. The objective of this exploration is to provide a different perspective of the IL/ML system – hopefully this will lead to a better designed transformation catalogue.

One can characterize code generation as a consecutive sequence of transformations chosen from the transformation catalogue and applied to an intermediate language input string:

$$s_{intermediate} \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow s_{target\ machine}.$$

$s_{target\ machine}$ is not necessarily unique; thus, the code generation algorithm may have to choose among many translations. If the translation uses an abstract machine then we will have

$$s_{intermediate} \rightarrow s_1 \rightarrow \ldots \rightarrow s_{k-1} \rightarrow s_{AM} \rightarrow s_{k+1} \rightarrow \ldots \rightarrow s_n \rightarrow s_{target\ machine}.$$

The transformations leading to s_{AM} are independent of the target machine; the transformations following s_{AM} are machine dependent. If we group transformations according to the code generation steps they describe (e.g., storage allocation, register assignment, etc), each group describes the translation of programs for a particular abstract machine into programs for another. By defining a hierarchy of abstract machines, the designer can limit the impact of a particular feature of the target machine to a few transformations. This type of organization of the transformation catalogue leads to a highly modular specification.

As was mentioned above, the resulting machine language program is not always unique – in order to be able to decide among competing translations, it is necessary to introduce some measure (m) of a program's cost:

$$m: s \rightarrow R \cup \infty.$$

This totally ordered measure is to reflect the optimality of the translation; the smaller the measure, the more optimal the translation. Note that the measure is not defined ($m(s') = \infty$) for intermediate language strings (s') that do not represent a completed translation. Typically this measure is computed from the values of attributes of the statements in the final program: it is up to the designer to ensure that each statement is assigned these attributes – if some statement does not have the appropriate attributes defined, the measure for that IL program will be undefined. The final choice for a given input string s and measure m is the set of "optimal" translations given by

$$O_m(s) \equiv \{ s' \mid s \overset{*}{\rightarrow} s' \text{ and for all } s'' [s \overset{*}{\rightarrow} s'' \text{ implies } m(s') \leq m(s'')] \}.$$

Note that we restrict our notion of optimality to those strings which can be actually derived from the initial program (s) by repeated applications of transformations from the transformation catalogue (i.e., $s \overset{*}{\rightarrow} s', s''$). It is possible that semantically equivalent strings exist which are more optimal but which may not be discovered because of some inadequacy in the transformation catalogue. In some sense this inadequacy is intrinsic since the semantic equivalence problem is in general unsolvable [Aho70].

In our syntactic view of code generation, we have set forth two tasks for the code generator. First, it must produce a set of translations for the given input string that meet certain basic criteria: e.g., they must be well-formed machine-language programs (only these should have the correct attributes needed to compute the measure). Second, it must select one of these translations as *the* translation. This selection is based on the optimality of the translation as well as other constraints the user may supply at compile time (e.g., upper bounds on space and/or execution speed). The filtering process is an expensive one as it means discarding completed translations – the more restrictive the compile-time constraints, the more programs may have to be discarded before a satisfactory translation is encountered. An alternative approach is to include these criteria as part of transformations in the catalogue, using contextual information to disqualify transformations which result in a violation. Thus unproductive translations are aborted before the effort is expended to complete them. The decision to include essentially all constraints as transformations allows a parsimonious description of acceptable translations at the cost of additional transformations. Experiments with automatic creation of a code generator from a set of transformations may prompt us to change our minds.

Let us take a moment to outline the advantages and disadvantages of this approach to code generation. By modeling code generation as a series of simple syntactic transformations, we have removed the onus of specifying the order in which the transformations are to be done – we have removed the control structure of the code generator. In its place we require that the designer specify enough context for each transformation to guarantee it will be used only when appropriate. The merits of this tradeoff are difficult to judge. For small sets of transformations it is simpler to omit the control structure as it is possible to foresee undesirable interactions between the various transformations and head them off at the pass. As the number of transformations increases, it becomes increasingly difficult to account for the global effect of an additional transformation. Adopting a modular organization for the transformations alleviates this problem – the use of a hierarchy of transformations (with little overlap between levels) supplies an implicit context for the transformations on a given level. There are many syntactic mechanisms for enforcing this modularity; several are presented in later examples. The greatest advantage of a syntactic view of code generation is that the designer is not encumbered with the details of programming but is able to deal at a higher, more natural level in describing code generation. The principal disadvantage is the current lack of a simple technique for realizing a code generator from the transformations. To actually implement a code generator, we will have to make explicit the implicit control structure supplied by the context of each transformation. Until this problem is solved, it looms as the largest barrier to accepting the syntactic view of code generation.

§1.3.2 The transformation catalogue and metainterpreter

Since the emphasis in a specification is on describing *what* the code generator is to do rather than *how* it is to be done, an effort has been made to distinguish *strategy* from *mechanism*. The strategic decisions made by a code generator are embodied in the transformation catalogue and fall roughly into three categories:

(1) expansion of a high-level IL statement into a series of more elementary statements;

(2) simplification or elimination of IL statements whose operations can be performed at compile time;

(3) transformations on sequences of IL statements, e.g., code motion in loops, permutation of evaluation order to achieve better register usage, peephole optimizations, etc.

The applicability of a transformation to a particular IL statement depends on the context in which that statement appears. In traditional code generators the context of an operation is established by two interdependent computations:

- flow analysis to determine available expressions, use-definition chaining, and live variables;

- compile-time computation of values for variables and intermediate results.

In a IL/ML specification, these computations have been incorporated as part of the context matching performed before a transformation is applied – the designer never explicitly invokes the underlying mechanism. Instead he may deal directly with values of variables, execution order of IL statements, etc. as part of an ML pattern.

The adequacy of IL/ML as the basis for a code generator specification hinges on the ability of the pattern matching mechanism to express the desired context. The pattern primitives provided by ML are based on standard data flow analysis techniques [Ullman, Kildall] and do not require extensions to the state of the art. Fortunately, these standard techniques easily compute the information required by many common optimizations. Combined with modest symbolic computation abilities (arithmetic on integers, canonicalization of expressions, etc.) the bulk of a code generators' task can be easily described without further mechanism. Ideally, it would be nice to stop here and rely on sequences of transformations to implement the more exotic transformations (such as induction variable elimination or register allocation) which are not currently incorporated in the metainterpreter. Unfortunately, this is an unreasonable attitude in light of the complexity of current algorithms for performing these transformations; the resulting set of transformations, if possible to construct at all, would be so large as to intimidate even the most dedicated reader of the specification. Two alternatives are

(1) to express the kernel of the algorithm as a simple transformation (such as assigning a compiler temporary a free register name) and rely on a combinatoric search to try all the possible alternatives. A clever metacompiler might be able to recognize these transformations for what they are and substitute one of several heuristics in the resulting code generator.

(2) to include built-in predicates (in the case of induction variables) or functions (for register allocation) that provide enough information for a simple transformation to perform the desired translation. To ensure that the specification does not build in certain heuristics this scenario requires algorithms that always "work" (i.e., produce complete or

optimal results); for many of the transformations in question no such algorithm currently exists.

Neither alternative is completely satisfactory and further research is needed to reach a conclusion. It seems reasonable to expect an eventual resolution of this issue and there is some evidence [Harrison] that many such optimizations may be ignored without significantly degrading the usability of the specification. In this spirit, the remainder of the thesis concentrates on the specification of code generation techniques which have a basis in flow analysis and its extensions.

§1.4 Relation to previous work

Until recently, research had focused on two approaches for the specification of code generators: the development of high-level languages better adapted to the writing of code generators and the introduction of an "abstract machine" to further simplify the code generation process. The new high-level languages [Young] provide as primitives many of the elementary operations used in code generation such as storage and register allocation and automatic management of internal data bases (e.g., the symbol table). The actual process of code generation typically fills in a user-provided code template with sundry parameters such as the actual location of the operands, etc. Local optimization is accomplished by special constructs within the template which allow testing for given attributes of the parameters. Modularity of the code generator is improved and much of the machine-dependent information is in descriptive form. Of course, the portions of the code generation algorithm and the optimization mechanism which depend on the semantics of the source language or target machine must still be coded into procedural form. The encoding of this information (usually as special cases) represents a large portion of many optimizing compilers [Carter].

The apparent dichotomy between descriptions of the intermediate language and the target machine led to disparate mechanisms for describing each. The use of an abstract machine (AM) capitalizes on this dichotomy. The operations of the AM are a set of low-level instructions based on some simple architecture. A code generator based on an AM [Poole] performs two translations: first the parse tree is translated into a sequence of AM operations and then each AM operation is, in turn, expanded into a sequence of target instructions. The optimality of the resultant code is largely a function of how closely the AM and the target machine correspond and how much work is expended on the expansion.

The first AM was UNCOL (*un*iversal *c*omputer *o*riented *l*anguage) [Steel], introduced to solve the "m×n translator" problem. Its proponents hoped that the use of a common base language would reduce the number of modules needed to translate m languages to n machines from m×n to m+n; they would translate a program in one of the m languages to UNCOL and then translate the UNCOL program to one of the n machines. The "UN" in "UNCOL" was their undoing as it proved exceedingly difficult to incorporate all the features of existing languages and machines into the primitives of a single language. By limiting the scope of the AM to a class of languages and machines [Coleman, Waite], it was possible to achieve truly portable software with a minimum of effort. Current implementations fall into two categories:

(a) The expansion is guided by a description of the target machine [Miller, Snyder]. The code generator may be easily modified to accommodate a different machine; however, due to the loss of information during the translation to AM operations, it is difficult to use special features of the target hardware to advantage. The description language is generally tailored for a specific class of machines and cannot easily be augmented.

(b) The expansion is done by a program designed to produce highly optimized code for a specific target machine [Richards]. This end is achieved via a "simulation" of the AM operations to gather sufficient information about the original program to allow more then local optimization. As a result, this phase can become quite costly to implement.

Thus, the designer had to choose between achieving a limited machine independence at the cost of poor optimization or producing optimized code and investing a substantial effort for each new target machine.

In an effort to accommodate a wider class of machines than encompassed by a single AM, some researchers [Bunza, Wick] have used a more general machine-description facility such as that provided by ISP [Bell]. An ISP description provides a low-level (i.e., register transfer), highly detailed description of the target machine which is amenable to mechanical interpretation to simulate the described processor. If an ISP description of source language operations is also available, a sophisticated code generator would have sufficient information to complete a translation. Despite the success of ISP in describing processors [Barbacci], it is not really suitable for describing the semantics of a high-level language: the level of detail required by ISP would require complex descriptions for many of the operators and data types of the language. In addition, reducing the semantics of the target machine and source language to their lowest common denominator results in the loss (or obscuring) of information used by many optimization strategies.

The introduction of *attribute grammars* [Knuth, Lewis] has coupled recent research with the formal systems developed for the parsing phase of compilation. In an attribute grammar, the underlying grammar is augmented by the addition of attributes associated with the nonterminal symbols of the parse. These attributes correspond to the "meaning" of their associated symbol; this naturally leads to two categories: inherited and synthesized attributes. Inherited attributes describe the context in which the nonterminal symbol appears; synthesized attributes describe those properties of the nonterminal symbol which derive from its component parts.

The relationship between the attributes of one symbol and another is specified by "semantic rules" associated with each production defining the synthesized attributes for the nonterminal symbol on the left-hand side of the production and the inherited attributes for the nonterminal symbols on the right-hand side of the production. [Neel] presents several production systems augmented with attributes that describe information commonly collected in the course of optimization (block numbers, whether a statement can be reached during execution, etc.). The principal advantage of such production systems is that there are no dependencies in the formalism on specific language or machine semantics — attribute grammars provide a general mechanism for accumulating contextual information during the first phase of compilation. However, optimizations that require other than a local examination of context are hard to accommodate: constructing the appropriate attributes can be nontrivial (cf. lambda calculus example in [Knuth]). Finally, except in trivial cases, translation into a target machine program (with the attendant optimizations) still requires another phase — one which is highly machine dependent.

Attributes have been adopted by Newcomer in his work on generalizing the optimization strategies employed by the BLISS/11 compiler [Newcomer]. In performing the expansion into PDP-11 code, this compiler depends heavily on tables which contain hand-compiled information on the best choices for each expansion. Newcomer attempts to automate the production of these tables by examining a description of the target machine. He uses a GPS-like search technique based on a difference operator to exhaustively search possible instruction sequences — from this search (guided by a preferred attribute set initially specified in the machine description) he collects the information needed to construct the tables. The machine description is a set of context-sensitive transformations where the appropriate context is established through the use of attributes. Although the results of this work do not establish the viability of automatically constructing a compiler in this manner, the notion of context-sensitive transformations as the basis of a machine description is a valuable contribution to the IL/ML system.

Perhaps the most successful attempt to date at constructing a modular code generation scheme incorporating a fairly complete optimization repertoire is the General Purpose Optimizing (GPO) compiler developed at IBM [Harrison]. The structure of the GPO compiler is similar to that proposed by this thesis: there is an intermediate language schema used as the internal representation, a set of defining procedures that serve as the basis for translating/expanding programs into pseudo-machine language, and a program which modifies the internal representation as optimizations and expansions are applied. The expansions and optimizations are iterated until the translation is complete; a final phase translates the resultant program into machine language, performing register assignments, etc. The GPO compiler is oriented towards PL/I-like programs -- the primitives provided in the intermediate language directly support block structure, PL/I pointer semantics, etc. The set of defining procedures allow tailoring of code dependent on attributes of the operands. The main differences between the GPO compiler and IL/ML are

- the lack of sophisticated name management (e.g., overlaying, aliasing) on the part of the GPO compiler.
- the syntax of defining procedures of the GPO compiler are best suited for PL/I-like programs.
- there is no notion of combining adjacent statements into a single operation (as in peephole optimization). Although Harrison talks of compiling past the machine interface, optimizations take place on a statement-by-statement basis (i.e., there is no general pattern matching facility).
- in the GPO compiler, attributes are treated like any other variable -- optimizations such as constant propagation are relied upon to make the attribute information available throughout the program. IL/ML provides a separate semantics for attributes thereby eliminating

certain situations where the optimizations would not be able to unravel a complicated sequence of statements.

The complexity of the GPO compiler is greatly reduced from that of current PL/I optimizing compilers. [Carter] has hand-simulated the expansion of test cases using a set of simple defining procedures for the substring operator of PL/I, producing code which equals or betters that of the IBM optimizing compiler (which includes some 8000 statements to treat special cases of substring). The inclusion of more sophisticated optimizations in the processor (cf. [Schatz]) should further improve these statistics. Encouragingly, many of these results seem applicable to the formalism proposed in this thesis -- the increased generality of IL/ML should not reduce its performance in this area.

§1.5 Outline of remaining chapters

Chapter 2 is a detailed description of the intermediate language IL: the syntax of IL is defined and the representation of data is discussed. The semantics of each IL construct is described and related to the needs of ML and the metainterpreter. The chapter concludes with a brief introduction to the compile-time calculation of values.

Chapter 3 discusses the construction of a transformation from ML templates that specify its context and effect. The syntax of a template (description of an IL program fragment) is described emphasizing the utility of wild cards and built-in functions. Rules for applying the transformation and updating the IL program are given. The final section describes a few sample transformations.

Chapter 4 presents a set of sample transformations and simulates their application by the metainterpreter to a sample IL program. This detailed example is aimed at demonstrating the ease of constructing a transformation catalogue and feasibility of performing code generation using the IL/ML system.

The final chapter briefly discusses the metainterpreter and the facilities it should provide then summarizes the results of this work and suggests directions for further research.

CHAPTER TWO

§2.1 The intermediate language: IL

The intermediate language described in this chapter serves as foundation for a specification constructed as outlined in §1.3. IL supports a skeletal semantics common to all programs from source to machine language; this includes primitives to describe the flow of control and the managing of names and values within an IL program. In addition, IL includes a mechanism for accumulating information on particular operations and storage cells for later use by the transformation catalogue and the metainterpreter. The remainder of the semantics of an IL program (e.g., the meaning of operations) reside in the transformation catalogue and are made available when these transformations are applied by the metainterpreter. By relegating the language and machine dependence to the transformation catalogue and providing a general syntactic mechanism for accumulating information, IL becomes a suitable intermediate language for the entire translation process. In order to allow common code generation operations (flow analysis, compile-time calculation of values) to be subsumed by the metainterpreter, separate fields are provided in each IL statement for the information required by the metainterpreter in performing its analysis.

Although IL in its most general form has a rather skeletal semantics and is a suitable intermediate language for a wide variety of source languages, certain conventions are established below for use in examples in later sections. Most of these conventions were inspired by conventional sequential, algebraic languages such as ALGOL, BLISS, or even CLU that are amenable to efficient interpretation by conventional machine architectures (i.e., those traditionally thought of as compiled languages). These conventions will be inappropriate in part for compiled languages that are not related to ALGOL (e.g., LISP); in many cases these can be easily accommodated by relatively simple changes. No direct attention has been paid to the special problems associated with the translation of those languages whose control structure differs substantially from that of ALGOL (e.g., SNOBOL, DYNAMO, SIMULA, etc.); this omission reflects the bias of this research towards the specification of conventional code generators. Hopefully, further work will fill this gap.

The most common form of intermediate representation is a flow graph of *basic blocks* where each basic block is described by a directed acyclic graph or *dag* (see, for example, Chapter 12 of [Aho77b]). IL is a linearization of this graphic representation with several additional restrictions to allow easy modeling of conventional languages. An IL statement may specify one of two actions: the conditional transfer of control to another statement (these correspond to the arcs of the flow graph); or the application of an operator to its operands (these correspond to the interior nodes of a dag), optionally saving the result in a named *cell*. Similarly, an IL statement may have one of two effects: transfer of control or the change in the value of one or more cells. As we will see below, it is easy to determine the exact effect of a statement from its syntactic form; targets of transfers of control and the set of cells changed by a statement (its *kill set*) are syntactically distinguishable from other portions of an IL statement.

As mentioned above, IL provides a schematic representation which is flexible enough to be used for programs varying in level from source to machine language. To encompass such a variety of programs, IL could not (and does not) have much in the way of built-in semantics. The following list summarizes the primitive concepts of IL:

- *conditional transfer of control* to another IL statement. In the absence of a transfer of control, execution proceeds sequentially through the IL program.

- *application* of an operator to its operands. There are no built-in operations supported by IL – the designer must ensure that each operator can be interpreted by the target machine or further expanded in the transformation catalogue.

- *value storage* provided by named cells. The scope of a cell name and the extent of its storage cover the entire IL program. Note that there is no distinction between program variables and compiler temporaries – all requirements for value storage must be met by using cells. Cell references have an lvalue/rvalue semantics similar to BCPL or BLISS. The name of a cell serves as its lvalue; applying the contents operator to the lvalue of a cell (i.e., <*lvalue*>) yields the rvalue of that cell. Aggregate data such as arrays or structures may be modeled by structuring the lvalue and rvalue of a cell.

- *attributes* for both lvalues and rvalues provide a syntactic mechanism for accumulating "declared" information that is unaffected by subsequent IL operations. A third type of attribute provides the same capability for each statement in an IL program.

- *literals* fill the dual role of reserved words (operators, attribute names, etc.) and constant rvalues (numbers, character strings, etc.). The meaning of a literal is "self-contained," one need go no further than the statement in which it appears to establish its meaning. Note that there is no such thing as a literal lvalue, i.e., an lvalue whose meaning can be established independently of the context in which it appears – thus it is not legal to apply the contents operator to a literal.

The following sections describe each of these areas in more detail, discussing how popular concepts such as block structure, data types, etc. are handled by IL.

§2.2 Data in IL

All data storage in IL is provided by named cells - program variables, intermediate results, etc. are represented in an IL program by a cell. Each cell has three components:

(1) an lvalue (name) which unambiguously identifies the cell. The scope of the lvalue covers the entire IL program. An lvalue can be structured for modeling arrays, structures, etc.

(2) an rvalue (written <lvalue>) which is modified whenever the cell named *lvalue* is used to hold the result of an operation. Any quantity associated with the cell that can be modified by an IL operation is considered to be part of the rvalue; if more than one such quantity exists, both the rvalue and the lvalue must be structured.

(3) a set of attributes associated with either the lvalue or rvalue. Attributes are used for declarative information that, once established, is unaffected by subsequent IL operations – attributes are sort of a manifest rvalue.

Note that no automatic translation is provided by the metainterpreter for cells; the designer is responsible for realizing each cell utilized in the IL program (by incorporating appropriate transformations in the transformation catalogue). This may include allocating main storage (for program variables), assigning registers (for short-lived intermediate results), or subsuming them completely (for intermediate results computed at compile time or internally by the target machine – e.g., indexed addressing).

Although an lvalue unambiguously identifies a cell, it is not necessarily unique. A given cell may come to have more than one name through redundant expression elimination or the ALIAS pseudo-operation (see §2.3.3). From then on either name may be used interchangeably. The ALIAS pseudo-operation may also be used to implement the overlaying of storage, an operation provided in many source languages by allowing the equivalencing of names. Unlike FORTRAN, however, each alias must be made explicitly – this is explored further in §2.2.2. Note that an lvalue may be used as an operand and that, as an operand, it will require declaration of attributes similar to those for an rvalue (type, length, value, etc.) – care must be taken so as not to confuse lvalue attributes with rvalue attributes and vice versa.

There is no separate provision for the scoping of lvalues (block structure). Through a declaration of a variable of the same name in an inner block, scoping allows shielding of a cell from use inside that block. In practice, however, procedure calls and pointers allow access to cells which are not directly accessible as operands. Thus the original cell cannot be "forgotten" completely while processing the inner block: a mechanism must be provided for referencing both the new cell and shielded cell when describing the effect of computations within the inner block. The other information provided by scope rules – lifetime information – is more accurately determined by live variable analysis performed by the metainterpreter. The additional cells provided for by scope rules can be created by choosing different cell names for each new declaration of the variable (perhaps by suffixing the linear block number to the variable name).

IL does not directly support data types (not even bit strings!): rvalues are simply objects. If the source language has declared types, these may be incorporated as attributes of the rvalue (for tagged data types the type information is another component of the rvalue). Transformations can utilize these attributes to tailor the generated code (see Figure 2.2). Similar conventions suffice for other properties of rvalues: their size, precision, etc. In theory, data types provide additional information in strongly typed languages. For example, assignment through an integer pointer should affect only cells whose rvalues have type integer. In practice, aliasing (see above), lack of type checking in computing pointer values, and (legal) inconsistencies between actual and formal procedure parameters conspire to prevent the designer from taking advantage of this additional information. In other words, just because the pointer has been declared as integer pointer does not guarantee that it points to only cells of type integer. It is worth noting here that the metainterpreter does know about certain classes of objects, such as numbers, allowing transformations to manipulate certain rvalues at compile time.

§2.2.1 Attributes

Attributes provide a general mechanism for associating information with components (cells and statements) of an IL program. Attributes associated with the lvalue or rvalue of a cell provide information which is unaffected by IL operations, e.g., its type, storage class, size, etc. This information is initially provided by the first phase of the compiler or added during translation by transformations as it is "discovered." Once established, cell attributes are available from any point in the IL program – dynamic information that is context dependent (e.g., which register contains the current value of the cell) *cannot* be stored as an attribute†. Attributes are the work horse of a specification: they provide a symbol table facility for each declared variable and intermediate result, model synthesized and inherited attributes used for passing contextual information about the operation tree, and so on *ad infinitum*.

Statement attributes allow information not relevant to the result cell to be associated with each statement. This includes properties of the operator (e.g., commutativity, size of a target machine instruction), effects on the global state of the interpreter (e.g., which condition codes are changed by a target machine operator), progress made in translating the statement (useful for communication between a set of transformations), etc. By incorporating these pieces of information as attributes, transformations can tailor the IL program taking into consideration machine- and language-dependent features without building machine and language dependencies into the metainterpreter.

---

† Dynamic information may be stored as part of the rvalue of a cell; in many cases compile-time computation of rvalues will propagate this information as effectively as if it were an attribute. Moreover, much of this type of information is used for optimizations which are already incorporated in the metainterpreter.

Attributes are referenced in an IL program as follows:

"*attribute_name*" for statement attributes;
"*lvalue:attribute_name*" for lvalue attributes;
"*<lvalue>:attribute_name*" for rvalue attributes.

Each attribute has a value (always a literal) established in some IL statement by including an assignment to the attribute name in the attribute field of that statement. For example, the following IL program statement illustrates the attributes which might be associated with the declaration of a real variable "Z" in a PASCAL program:

Label	Operator Operands	Attributes
Z	declaration	Z:type=unsigned_integer Z:size=2 Z:level=2 Z:offset=14 <Z>:type=real, <Z>:size=8

The first line indicates that the address (lvalue) of Z is a two byte unsigned integer – this information will be needed for type checking performed by some transformation if Z enters into a pointer calculation. The second line gives the lexical level and stack frame offset† assigned to Z (either by the first phase of the compiler or a transformation applied earlier); a transformation could be included in the transformation catalogue to compute the actual address of Z from this information. Finally, the third line indicates that the value of Z occupies 8 bytes and has type real. Note that the "declaration" operator has no special significance in IL; any semantics associated with this operator (e.g., allocation of storage or the initialization of Z's rvalue) will be captured in the transformation catalogue. The same is true for each of the attributes described in this paragraph: in IL, their values are simply literals – the interpretation ascribed to them in the explanation

---

† These were arbitrarily chosen to be lvalue attributes: general attributes of a cell may be associated with either the lvalue or rvalue – a convention is chosen here so that the transformations "know where to look" for the information.

reflects the role they play in transformations applied by the metainterpreter.

§2.2.2 Structuring of cell names and values

The ability to structure lvalues (and their corresponding rvalues) simplifies the modeling of aggregate data and operations which affect one or more components. Each component is, in effect, a separate lvalue; its type, size, and other attributes can be maintained separately from those of other components. It is also possible to perform operations on the aggregate data as a whole, changing all components in one operation. A component's lvalue is constructed by appending the appropriate selector to the lvalue of the aggregate, like so: *aggregate_name.selector*. For example, if A were an array dimensioned from 1 to 10 then

rvalue	*refers to*
<A>	the entire array
<A>.2	A[2], the second component of A
<A>.<I>	A[I], the Ith component of A
<A>.*	all components of A (A.1 through A.10)

Note that <*aggregate_name.selector*> is equivalent to <*aggregate_name*>.*selector* – either form may be used interchangeably. In the last line, "*" was introduced as a convenient abbreviation for "all possible component names." Of course, "*" is never actually expanded but rather serves as a wild card when resolving attribute references to components of an aggregate cell. For example, <A>.* would be used when referring collectively to elements of the array, as when declaring the type of the elements (assuming A is homogeneous). Thus, if a program contained the definition <A>.*:type=boolean then the attribute reference <A>.3:type could be resolved to "boolean." <A>.* used as the prefix of an attribute reference is *not* equivalent to <A>: attributes for an aggregate are maintained separately from those of its components. The following IL statement illustrates the attributes which might be associated with a declaration of the above array:

Label	Operator	Operands	Attributes
A	declaration		A:type=unsigned_integer A:size=2 A:dimensions=1 A:ubound=10 A:lbound=1 A:lbound:type=integer A:lbound:size=2 <A>:type=array <A>:size=10 <A>.*:type=boolean <A>.*:size=1

Note that the example specifies that the rvalue of A is an array 10 bytes long and that the lvalue of A is a 2-byte unsigned integer (just like any other address!). The third line is included since A:lbound is likely to be used as an operand in subscript calculations and therefore needs the appropriate attributes. The final line indicates the type and size of the components of the array. In choosing the attributes to be included in this array declaration, every effort has been made to ensure that each quantity which might appear as an operand in subsequent operations has the required attributes. This eliminates the need for any special casing — a multiply operation performed during a subscript calculation receives the same treatment as any multiply operation.

In many cases the "*" notation is more powerful than the corresponding expansion. For example, consider the declaration given above and the attribute reference <A>.<I>:type (the type of the Ith component of A). The last line of the declaration indicates that the type of any component is "boolean" and so <A>.<I>:type can be resolved to "boolean" without further ado. If, on the other hand, separate type definitions had been provided for each component — i.e., <A>.1:type=boolean, etc. — resolution of <A>.<I>:type could not proceed without more knowledge of <I> (the value of the subscript). Even though bounds checking may be desirable, it is better accomplished explicitly at run time rather than implicitly during compile-time type checking. Another solution would be to endow the metainterpreter with special knowledge concerning attributes of array subscripts, but this leads to undesirable language dependencies in the metainterpreter. All in all, the "*" notation comes much closer to the semantics common to most aggregate data and leads to a simple mechanization of attribute resolution.

Operations which affect the rvalue of an aggregate cell (e.g., an array assignment to <A>) are understood to change the rvalues of the components (e.g., <A>.1, <A>.2, ..., <A>.10). The converse is also true: a change in a component's rvalue changes the rvalue of the aggregate. Both cases are based on the premise that the rvalue of an aggregate is the "sum" of its components – i.e., that the rvalue of an aggregate is *not* maintained separately from the rvalues of its components. Thus <A> is equivalent to <A>.* (when speaking of rvalues – this differs from the conclusion reached above for the managing of attributes). The effect of this reasoning (see discussion in §2.8.1 on augmentation of kill sets) coincides with common practice: a change in <A>.3 should invalidate any temporary copies of the whole array (<A>) but should not affect temporary copies of other components (e.g., <A>.7); on the other hand, changes in the whole array should invalidate temporary copies of any component.

As a final example of a structured cell, consider the following series of IL statements (see §2.8.3 for a detailed description of the ALIAS pseudo-operation):

Label	Operator	Operands	Attributes
X	declaration		X:type=unsigned_integer X:size=2 <X>:type=long <X>:size=4
I	ALIAS	X.1	I:type=unsigned_integer I:size=2 <I>:type=integer <I>:size=2
J	ALIAS	X.2	J:type=unsigned_integer J:size=2 <J>:type=integer <J>:size=2

In this example, the rvalues of I and J overlay the rvalue of X (the designer has the responsibility for making the storage allocated for I and J overlay the storage for X in the final translation by adding appropriate transformations to the catalogue). Note that although X is not explicitly declared to have any components, aliasing I and J to X.1 and X.2 has caused them to become components of X. Thus, using the reasoning of the preceding paragraph:

(1) changes to the rvalue of X invalidate the rvalues of I and J;

(2) changes to the rvalue of I invalidates the rvalue of X, but does not affect the rvalue of J; and

(3) changes to the rvalue of J invalidates the rvalue of X, but does not affect the rvalue of I.

The final two conditions show that I and J are understood to be disjoint. These three conditions are just the semantics one associates with overlayed storage†.

§2.3 The syntax of IL

An IL program is a sequence of statements made up of tokens classed as literals, lvalues (the name of a cell), or rvalues (the application of the contents operator to an lvalue). Depending on where a token appears in an IL statement, it is further classified as a label, operator, operand, or attribute. Label tokens must be lvalues; operator and attribute tokens are always literals; operand tokens may be any flavor. Beyond the semantics associated with these four classes of tokens, IL provides no further interpretation of ordinary tokens. In this sense, IL is similar to a BNF: neither provides any interpretation of the symbols of the language. Special tokens are provided to indicate transfers of control and their corresponding targets within an IL program. These tokens are used in data flow analysis and are seldom referenced directly by the user. An IL statement has the following form:

† No provision has been made to show how to compute new values of I and J from a new value of X (and vice versa). The details of this computation depend on storage allocation and machine representations and so should be relegated to the transformation catalogue. Such transformations can be generated at compile-time from the ALIAS statement through the use of transformation macros (see §3.7).

Label	Operator	Operands	Attributes
label	*operator*	*operand...*	*attribute...*
...	...	...	...

where the components are described below.

label — This field names the cells whose rvalues might be changed by this statement. Two labels, → and ●, have a special meaning to the system (see Section 2.3).

operator — This field indicates the operation performed by this statement.

operand... — Zero or more operands used as arguments to the preceding operation.

attribute... — A set of zero or more "name=value" pairs further describing the context and semantics of the statement.

Figure 2.1 shows the initial IL representation of the following program:

```
integer X,Y,Z;
if X>Y then { X=2; Y=3 } else { X=3; Y=2 };
Z = X+Y;
```

There is no single IL representation for a given program; e.g., one could eliminate the definition of C1 and C2 entirely from Figure 2.1 and use the literals "2" and "3" directly. Choices as to the number of levels of indirection, etc. are not dictated by IL and can be made on the basis of compatibility with the transformation catalogue, appropriateness for the target machine, etc. Note that in Figure 2.1 attributes have only been given for the declaration portion of the program – the remainder will be filled in by the metainterpreter as it applies transformations. The initial attributes are similar to those that might be provided by the first phase of the compiler. Attributes are described in more detail in §2.2.1.

In the description which follows, it will be useful characterize tokens as either literals or *references* (either an lvalue or rvalue). By way of example, consider the following two lines from Figure 2.2:

Label	Operator	Operands	Attributes
X	declaration		X:type=integer X:size=2 <X>:type=integer <X>:size=2
Y	declaration		Y:type=integer Y:size=2 <Y>:type=integer <Y>:size=2
Z	declaration		Z:type=integer Z:size=2 <Z>:type=integer <Z>:size=2
C1	constant	"2"	<C1>:type=integer
C2	constant	"3"	<C2>:type=integer
T1	greater_than	<X> <Y>	
→	if_goto	<T1> L2 L1	
•	label	L1	
X	store	<C2>	
Y	store	<C1>	
→	goto	L3	
•	label	L2	
X	store	<C1>	
Y	store	<C2>	
•	label	L3	
T2	add	<X> <Y>	
Z	store	<T2>	

Figure 2.1: Initial IL representation

Label	Operator	Operands	Attributes
T100	*equal*	<X>:type *"integer"*	
...			
T1	*add*	<X> <Y>	

The italicized tokens are literals; the rest, references. In IL, literals are nothing more than character strings – interpretation of these strings is provided by the transformation catalogue and the metainterpreter. References "refer" to values established by other statements – they provide a level of indirection. The principal difference between literals and references is that the meaning of a literal can be established at compile time whereas references often refer to values that are not known until execution time. Literals are of central importance during optimization since their fixed semantics provide opportunities for compile time evaluation of operations. Some references (e.g., <X>:type) may, depending on the context in which they appear, refer to literals; in these cases it is advantageous to remove the unnecessary level of indirection at compile time. For those references that cannot be resolved into literals at compile time (e.g., <X>), it will be necessary to produce code which actually performs the indirection specified in the IL program (e.g., by performing a fetch from the storage location used to hold the desired value).

§2.3.1 The label field

The label field of an IL statement lists the cells which are affected by execution of that statement. A statement may affect a cell in two ways:

> a cell is *killed* by a statement if execution of the statement *might* cause the rvalue of the cell to change; the set of killed cells is called the *kill set*.
>
> a cell is *defined* by a statement if execution of the statement *always* changes the rvalue of the cell; the set of defined cells is called the *defined set* of the statement. Note that the defined set $\subseteq$ the kill set for any statement.

When a cell is killed, its rvalue can no longer be used for calculating common subexpressions (assuming that the cell had not been killed previously). If a cell is defined by a statement, it will always contain the value calculated by the statement after the statement's execution. Therefore, if a statement executed subsequently is identified as performing the same computation, it can be replaced by a reference to the defined cell. Moreover, if the defined value is a literal, subsequent references to the rvalue of the defined cell can be resolved to that literal. By convention, the *lvalue* of each affected cell is listed in the label field; the implicit contents operator is omitted for the sake of brevity. The label field is used by the metainterpreter in two important optimizations: redundant computation elimination and use-definition chaining (compile-time evaluation of statements).

With one exception, the kill set provides all the information needed to perform these optimizations. This suggests two formats for the label field: "K" and

"K,D" where K is the kill set of the statement and D the corresponding defined set ($D \subseteq K$). When the abbreviated first format is used, D is calculated as follows:

> *case 1.* If K is empty ($|K| = 0$) then $D = \phi$.
> *case 2.* If K has a single element ($|K| = 1$) then $D = K$.
> *case 3.* If $|K| > 1$ then $D = \phi$.
> *case 4.* If $K = \{*\}$ then $D = \phi$.

Considering only statements that affect at most one cell (all the statements in Figure 2.1 fall into this category), there is a natural interpretation for each of the above cases. Statements affecting no cells (e.g., transfers of control) are covered by case 1. Statements whose operators have an applicative semantics (add, multiply, etc.) fall under case 2; the single element of the kill set is the lvalue of the cell where the result is stored. The specified cell is *always* changed by executing the statement, so $D = K$. This is also the case for assignment statements which always change the same cell (i.e., they do not compute its lvalue) – in these statements the label is essentially another operand. Case 3 covers assignment statements that compute the lvalue of the cell in which the result is to be placed, e.g., assignments through pointers or to array elements with non-constant subscripts. Here, each cell in K has been killed (its previous rvalue *may* have been changed, thus it can no longer be assumed that it is available) however no cell in K has been defined (no single cell is certain to have been changed) hence $D = \phi$. In the final case, a label of "*" indicates that all cells *might* be affected by executing the statement. For essentially the same reasons given in §2.2, no provision has been made for specializing "*" by specific cell attributes (e.g., type): in almost every language there exist loopholes which make attribute information unreliable†. This label is used when the statement has

† This attribute information *will* be used in the expansion of operations in the IL program. Despite the suspect nature of attribute values, this is the semantics provided by many languages and relied upon by programmers to circumvent certain language restrictions. However, this information cannot be used as a basis for

unfathomable side-effects, for example, when the label field contains too complex an expression (e.g., deeply nested contents operators) – when an lvalue subexpression has become unwieldly it is always legal to assume its value is "*" and proceed from there. This overly conservative interpretation may result in missed optimization opportunities but never in an incorrect translation.

Procedure calls have the potential of affecting many cells and so do not fall into the categories discussed above. The sequence of statements which form the body of the procedure may kill and define cells – taken in the aggregate it is possible that $K \supset D \neq \phi$. In addition, procedures that return a value add yet another element to D (the cell containing the returned value). The second label format, "K,D", is used for procedure calls. While it is theoretically possible to compute the appropriate label by examining the body of the procedure, this calculation quickly becomes unwieldly. A reasonable alternative is to assign procedure calls the label "*,R" where R is the lvalue of the cell in which the returned value (if any) is stored. Thus the semantics of a procedure call is reduced to invalidating previously calculated values for all cells except the one containing the return value.

As was outlined in §2.2.2, it is occasionally necessary to augment the kill set of a statement to account for the semantics of aggregate cells. Although the size of the kill set may be increased, the defined set calculated above remains unchanged – essentially no new cells are being added to the kill set, but only other lvalues for the affected rvalue(s). The objective of augmenting the kill set is to explicitly include the lvalue of every cell which is affected by the statement; this reduces the amount of computation performed by the metainterpreter when using

---

optimizations, as is would lead to incorrectly transformed programs – only the programmer is allowed to play havoc with his program!

the kill set.

The following algorithm constructs an augmented kill set K' from the original kill set K. K' will include all lvalues ALIASed to lvalues in K as well as the lvalues of aggregates which subsume lvalues in K. In constructing K', a distinction is made between an aggregate and its components: if an aggregate name appears in K', it refers to the aggregate treated as a single value (i.e., any temporary copies of the entire aggregate should be invalidated); if temporary copies of an aggregate's components should also be invalidated, the "*" notation is used. For example, "A" would invalidate any copies of the array A but leave its components unaffected; "A.*" would invalidate any components (and subcomponents, etc.) of A. The algorithm is

1. Initially K' = K.

2. For each structured lvalue α in K, add α.* to K'. An lvalue is structured if any attributes have been defined for any lvalue or rvalue components of the lvalue or if ALIASes have been made to any lvalue components. This step ensures that if an entire aggregate value was in the original kill set, all of its components will also be invalidated in the augmented kill set.

3. For each lvalue α in K', add any aliases declared for α to K'.

4. For each component lvalue α.β in K', add α to K'. The intent here is to add all the prefixes for each component lvalue, e.g., if A.1.2.3 were an element of K', this step would add A.1.2, A.1, and A to K'.

5. Repeat steps 3 and 4 until no more additions are made to K'.

The final result for K' is the augmented kill set for the statement. The following series of examples should clarify the workings of the algorithms. For purposes of exhibition, duplicate lvalues (e.g., X.* and X.1) have been removed from the kill sets. The examples assume the declaration given in examples in §2.2.2.

original kill set (K)	*augmented kill set (K')*
{A}	{A A.*}
{A.3 A.4}	{A.3 A.4 A}

{A.*}	{A.* A}
{I}	{I X.1 X}
{X}	{X X.* I J}

Note that the augmented kill sets agree with the desiderata outlined in §2.2.2.

§2.3.2 The operator and operand fields

No particular semantics is attached to the operator field of a statement. The meaning of an operator is established by transformations which expand it into other IL or target machine operations. A useful analogy for an IL operator is a macro -- the body of the macro defines the effect of an operator in terms of other, usually simpler, operations. If the effect of the macro can be accomplished directly by the target machine no further refinement of the operation is necessary; the translation of the statement is complete. Otherwise, the body of the macro (in this case a sequence of IL operations) should be substituted for the operation, making the appropriate substitutions of actual operands for formal parameters of the macro. If each expansion is subject to later optimization, it is possible to use general definitions for each macro operation, i.e., definitions such as one would find in an interpreter. Special cases that hinge on particular values of the operands would be explicitly tested for in the substituted sequence; later optimization would eliminate those operations which could be performed at compile time. For example (see Figure 2.2), the expansion of the addition operator might test the type of its operands and then perform an integer or floating point addition as appropriate. If the type of the operands could be established at compile time, this test would be subsumed during optimization. Although it is not necessary, use of general definitions greatly simplifies the top level of a specification as there will be only one transformation for an operation rather than one for each special case.

Label	Operator	Operands	Attributes
X	declaration		<X>:type=integer
Y	declaration		<Y>:type=real
T1	plus	<X> <Y>	

Figure 2.2a: Original IL program

Label	Operator	Operands	Attributes
X	declaration		<X>:type=integer
Y	declaration		<Y>:type=real
T100	equal	<X>:type "integer"	
→	if_goto	<T100> L1 L4	
●	label	L1	
T101	equal	<Y>:type "integer"	
→	if_goto	<T101> L2 L3	
●	label	L2	
T1	add	<X> <Y>	
→	goto	L7	
●	label	L3	
T102	float	<X>	
T1	addf	<T102> <Y>	
→	goto	L7	
●	label	L4	
T103	equal	<Y>:type "real"	
→	if_goto	<T103> L5 L6	
●	label	L5	
T1	addf	<X> <Y>	
→	goto	L7	
●	label	L6	
T104	float	<Y>	
T1	addf	<X> <T104>	
●	label	L7	

Figure 2.2b: IL program with expanded definition of plus

Label	Operator	Operands	Attributes
X	declaration		<X>:type=integer
Y	declaration		<Y>:type=real
T102	float	<X>	
T1	addf	<T102> <Y>	

Figure 2.2c: "Optimized" IL program

Operands serve as arguments to the preceding operation and may be any of the following:

> a *literal*. Literals are enclosed in quotes when they appear in the operand field so that they may be distinguished from lvalues.
>
> an attribute reference. Note that it is possible to references attributes of attributes, etc. All attribute references should be able to be resolved at compile time (i.e., there should be an appropriate definition generated at some point in the expansion of the IL program). If no such definition exists then the attribute reference is illegal.
>
> a reference expression: a simple lvalue if the "address" of the cell is needed; otherwise, an rvalue expression (which may be nested) is used.

There is no a *priori* restriction on the complexity of a reference expression, but more than one level of indirection (contents operator) will likely have to be calculated in a separate statement. By convention, at most a single level of indirection is used in an operand.

§2.3.3 The END and ALIAS pseudo-operations

Pseudo-operations provide a mechanism for informing the metainterpreter about information difficult (or impossible) to derive from the IL program. IL statements with pseudo-operators are "visible" to the transformations which may transform them into ordinary IL statements, etc. but they become "invisible" in the final translation (i.e., they are not output in the resulting target machine program). The names chosen for pseudo-operations are reserved and should not be used for other purposes by the designer; in this thesis, pseudo-operators will be displayed in upper case and all other operators displayed in lower case.

The statement in which the END pseudo-operation appears marks the logical end of an IL statement sequence – flow analysis for that sequence will not proceed past this statement. Statements following this statement up to the next target statement (see §2.4) are considered inaccessible and will be removed by the metainterpreter. The END pseudo-operation is intended for use at the end of the IL program and for marking the end of procedure bodies within the IL program; presumably some transformation will translate it into a exit or return as appropriate. This operation makes no use of the label, operand, or attribute fields and so may be used as the operator of a target statement.

The ALIAS pseudo-operation provides the capability of defining equivalence classes of lvalues — any member of an equivalence class refers to the same rvalue (although each member may have different attributes associated with it). This operation is used to indicate sharing of rvalues (overlaying of storage) as declared by the source language program (e.g., with the FORTRAN EQUIVALENCE statement) or as determined in some transformation (e.g., when used to indicate that two cells hold the same value; this typically occurs during optimization when a sequence of statements boils down to a move from one cell to a temporary — the ALIAS operation would indicate that the temporary is aliased with the original cell). In the latter case, the ALIAS operation provides a renaming capability to the transformation designer. The form of the ALIAS statement is

Label	Operator	Operands	Attributes
$lvalue_1$	ALIAS	$lvalue_2$	*attributes...*

which causes the metainterpreter to place $lvalue_1$ in the same name equivalence class as $lvalue_2$. Note that, by definition, ALIAS is a transitive operation. Typically $lvalue_1$ is the new lvalue to be defined and *attributes...* are its initial attributes.

§2.4 Flow of control in an IL program

In the previous sections, the syntax and semantics of a single IL statement were described; this section describes the semantics of a sequence of IL statements. IL statements are executed sequentially, modulo explicit transfers of control. This classic control structure was chosen because of its compatibility with the control structure provided by most target machines — the operations primitive to IL are similar to those provided at the machine level. Sequential execution is also compatible with a wide variety of languages, especially those that have relatively severe ordering constraints (e.g., ALGOL, which specifies strict left-to-right evaluation of expressions). This control structure is more constraining than the one provided by the dags on which IL was modeled: the only constraint imposed by dags is that the sons (operands) of an interior node (operation) must be evaluated before the node can be evaluated. Some languages (e.g., BLISS) take advantage of this flexibility in expression evaluation by only imposing evaluation order constraints on certain operators (such as BEGIN..END). Such flexibility is not inherent in an IL program and must be provided by the transformation catalogue and the metainterpreter: transformations can change the order of statements in an IL program†.

As was mentioned at the beginning of this chapter, the syntactic conventions discussed below are not particularly appropriate for languages whose control structure differs substantially from that above. SNOBOL, for example, requires a "transfer of control" with every statement — the difficulty in accommodating this construct in IL reflects the difficulties in producing a SNOBOL compiler for conventional machines; perhaps when the latter problem has been solved, the solution can be incorporated in IL.

---

† In general these transformations only change the evaluation order to achieve some goal, for example, a reduction in the number of registers required to evaluate the operator. In this way the conditions under which evaluation order can be modified and what metrics are used to judge the result are made explicit in the transformation catalogue. This information would be useful during the analysis phase of a metacompiler attempting to construct a code generator from the specification.

IL statements which cause a transfer of control (transfer statements) are readily identified: they have a "→" in their label field. Note that this use of the label field prevents the statement from also computing (and saving) a value, it can only effect a transfer of control. Procedure calls are handled differently: since control returns to the statement following the procedure call, they are similar to ordinary statements except for the possible side effects of the procedure body. In §2.3.1, a convention for the label field for procedure calls was established (listing the side effects of the procedure); thus, no transfer is explicitly indicated. Procedures are treated as "complex" operations in so far as this section is concerned. Note that a transfer statement always transfers control; if execution can conditionally continue with the next statement, it must be provided for explicitly by adding an additional label statement.

IL statements which are targets for a transfer of control (target statements) are identified by placing a "@" in their label field. As for transfer statements, target statements cannot compute (and save) a value since their label field has been preempted. The following convention is used by the metainterpreter for determining which target statements are possible targets for a given transfer statement:

> a target statement is a target for a given transfer statement iff the same lvalue appears as some operand of both the target statement and the transfer statement.

This convention allows additional arguments to transfer and target statements which can be used by the operator of these statements. The following example (extracted from Figure 2.2b) illustrates the convention more clearly:

Label	Operator	Operands	Attributes
→	if_goto	<T100> L1 L4	
•	label	L1	
...			
•	label	L4	
...			

The first line is a transfer statement (has → in its label field) which can transfer control to either of the target statements. The second statement is recognized as a possible target since the lvalue L1 is an operand of both the first and second statement; similar reasoning holds for the last target statement. It is not possible to tell from the above program the circumstances under which either label is chosen as that depends on the semantics of the if_goto operation (and presumably the value of T100), information that only exists in the transformation catalogue.

This information is used by the metainterpreter to construct a "maximal" flow graph for the IL program. The flow graph is maximal in the sense that all possible targets are considered for each transfer statement, even those which may be ruled out by the semantics of the operator of the transfer statement. This graph serves as the basis for the flow analysis performed by the metainterpreter and is updated whenever a transformation changes or eliminates a transfer statement.

§2.5 Compile-time calculation of rvalues

One of the goals for the syntax of IL is to allow the compile-time calculation of rvalues. This section briefly touches on the resolution of rvalues (and lvalues) using the notation developed in earlier sections. In this section, set notation is used to indicate possible values for a reference expression, e.g., if the rvalue of I is known to be either 3 or 4 then we write <I> = {3 4}. If the value of a reference expression is unknown (i.e., it could be any possible value) then we write {*}.

Occasionally, it is possible to further resolve a particular reference expression. If <I> = {3 4} then

<A>.<I> = <A>.{3 4} = {<A>.3 <A>.4}.

If, on the other hand, the value of I is unknown (<I> = {*}) then

<A>.<I> = <A>.{*} = <A>.*.

IL recognizes the alternative forms in each example as equivalent: in effect, such resolution is performed automatically. Even in the absence of knowledge about the rvalue of I, a reasonable interpretation of rvalues incorporating <I> is possible, erring only in that it is likely to be an overly conservative interpretation. In the second example above, the distinction between "*" as an abbreviation for all possible component names and {*} as the representation for all possible values has been deliberately blurred. The intent behind assigning numeric selectors for the components of the array A is to allow this sort of felicitous confusion.

As a rule of thumb, the utility of the compile-time computation of a cell's rvalue is inversely proportional to the size of the value set. There are several contributing factors: as the size of the value set increases, it becomes increasingly unlikely that any significant optimizations will be possible for rvalue operations on that cell. In addition, uncertainty in one cell's rvalue tends to propagate to other cells whenever the first cell is used as an operand (the value set of an operation is proportional to the product of the value sets of the operands). Such "dilution" of compile-time information is not unexpected — it would be unreasonable to expect to perform all computations at compile time! However, the prognosis at this point is not encouraging: it would appear that large amounts of compile-time information could be collected with little prospect of a corresponding gain in the optimality of the resulting translation.

The preceding paragraph prompts two observations: compile-time calculation of rvalues is subject to the law of diminishing returns, and therefore rvalues are not suitable for cell attributes that do not change with each operation on the cell. The first observation serves as further motivation for the introduction of {*} for rvalue sets which have grown too cumbersome. The second suggests that attributes are a useful addition to the semantics of a cell because they provide a mechanism for circumventing the vagaries of rvalue computations.

CHAPTER THREE

§3.1 The transformation catalogue

A major design goal for the IL/ML system was to keep knowledge about the source language and target machine separate from general knowledge about code generation. This was accomplished by providing for a separate description of machine- and language-dependent semantics – the embodiment of this description is the transformation catalogue. Each piece of language- or machine-specific information is expressed as a syntactic transformation of an IL program fragment; after the transformation has been applied, the updated program will have been modified to incorporate this new information in terms the metainterpreter understands: as attributes or a new sequence of IL statements. The metainterpreter provides the remainder of the framework needed to finish the task of code generation: whenever it exhausts its analysis of the current program it returns to the transformation catalogue to gather additional information (in the form of a "new" IL program to analyze). This cycle of analysis and transformation repeats until the translation is complete.

This chapter discusses the transformation catalogue and the language which serves as its basis: a metalanguage (ML) for describing IL program fragments. Using ML, the designer can write templates which describe the class of IL statements in which he is interested. This class can be quite large (e.g., "all IL statements which have commutative operators") or quite small (e.g., "only statements which apply the sine operator to the argument 3.14159") depending on the application the designer has in mind. Members of the class of IL fragments described by a template are said to *match* the template. §3.2 presents a detailed description of the syntax of ML.

Two templates are incorporated in each transformation: one as a *pattern*, the other as a *replacement*. The pattern specifies the context of the transformation as a set of program fragments on which the transformation can operate†. IL statement(s) which match the pattern become candidates for the modifications specified by the replacement. The replacement, perhaps using statements or components matched by the pattern, tells how to construct a new IL program fragment to be substituted for the matched fragment.

The use of transformations is a well-established technique for embodying knowledge for later use in a mechanized fashion (see §1.3.1). If all the contextual information in the data base (in this case, the IL program) is available in syntactic form, patterns provide a concise description of where the piece of information captured by the transformation is applicable. Using the transformation catalogue is reduced to finding a transformation which matches the given IL statement (or any IL statement, if the metainterpreter has no specific goal in mind); alternatively, the replacement (which is also a pattern) can be examined to determine if it accomplishes the desired effect. The ability to use transformations from either end enhances their utility as the basis for knowledge representation.

§3.3 describes how transformations are constructed and how they are used by the metainterpreter. The final section of this chapter presents a series of annotated example transformations.

† This context can be further modified by a set of *conditions* specifying constraints which are not expressible in terms of the syntax of the IL program (see §3.3.1).

§3.2 ML: a language for describing IL program fragments

ML is similar to other metalanguages -- its syntax subsumes that of IL (i.e., an IL statement is a legal ML statement) and, in addition, it allows certain metasymbols to replace IL components or statements. The metasymbols come in two flavors: wild cards that act as "don't cares" in the matching process, and calls to built-in functions that allow access to some of the metainterpreter's knowledge of IL program semantics. Use of these metasymbols permits the designer to write generalized IL program fragments; these fragments are more general than an IL program fragment because the designer has constrained only those statement components in which he is interested (using wild cards to specify the remaining components).

However, the designer can only generalize along certain dimensions as his only access to the meaning of an IL statement is its syntactic form and whatever built-in functions are available (see §3.2.2). Since the separate fields for kill sets and attributes in an IL statement seem to be as far as one can go towards making the syntactic form of an IL statement reflect the statement's semantics without limiting the generality of IL, the limiting factors are the capabilities of the built-in functions. The designer can determine whether two literals are the same but may not be able to find out, for example, whether the square root of a literal is an integer. These restrictions on the abilities of built-in functions are the most severe limitation of ML: building in language- and machine-specific predicates into ML is ruled out as this effects the generality of the system and, unfortunately, it would be impossible to include all the generally useful functions. Lest we be accused of making a mountain out of a molehill, it should be pointed out that the result of these limitations is missed optimization opportunities. Presumably all the computations specified in the IL program *could* be done at execution time; the computational facilities provided by ML are intended to allow special tailoring of the transformations and not to be an essential component of the transformations. ML takes the middle road by providing built-in functions for manipulation of literals and for interpreting literals as numeric quantities — other functions must be constructed from these by including the appropriate transformations in the catalogue. These additions to the catalogue are sufficient for most purposes — for example, the catalogue may contain transformations for simplifying the application of the transcendental functions to certain arguments (π, π/2, etc.) but would translate all other applications to a run-time call of the appropriate function.

§3.2.1 describes wild cards; §3.2.2 enumerates some example built-in functions. Example ML statements can be found in the last section of the chapter as patterns and replacements in transformations.

§3.2.1 Wild cards

Wild card metasymbols are used as components of an ML statement wherever a specific IL component would be too restrictive — the wild card will match any IL component(s). The discussion below describes the meaning of ML statements when used in a pattern; to a large degree the semantics of a replacement are similar (differences are described in §3.3.2). There are four forms of wild card:

wild card	*will match*
?name	a single IL component
$name	a single IL statement
*?*name*	a sequence of IL components
*$*name*	a sequence of IL statements

name is an optional identifier which is used to distinguish between multiple wild cards used in a single pattern or replacement. These names are also used in the replacement to refer to components or statements matched in the pattern. If a given wild card appears more than once in a pattern or replacement (i.e., two or more wild cards with the same form and name) they are understood to represent the same IL component; if this duplication occurs within a pattern then all the copies must match IL components with the same representation.

The ? and $ wild cards match a single, non-null component or statement respectively, i.e., for each ? ($) there must be a corresponding IL component (statement) in the IL program fragment which is being matched. Note that when describing an IL statement, all of its components (with the exception of attributes, see §3.3.2) must be accounted for in the ML statement – either explicitly or as wild cards – or the match will fail. Thus, if only the label field is to be constrained in the pattern, wild card components must be used for the contents of the operator (use ? wild card) and operand (use ?* wild card) fields.

The ?* wild card matches any sequence of zero or more IL components within a single field – what components are matched usually depends on the components on either side of the ?* wild card in the ML statement. If these adjacent components constrain the match for the ?* wild card to a single sequence, the ?* wild card is said to be *unambiguous*. In general, if more than one ?* wild card is used in a single field, they may be ambiguous; this is always the case if two ?* wild cards are adjacent or separated by any number of ? wild cards. Even if specific IL components are interposed, duplication of this component in the IL field can cause the ?* wild cards to be ambiguous. For example, consider the sequence of components "A B C C D". There are two ways in which components can be assigned to the ML expression "?x ?*y C ?*z":

?x="A" ?*y="B" ?*z="C D" or ?x="A" ?*y="B C" ?*z="D".

Ambiguous wild cards are useful for matching a specific IL component anywhere in a field; e.g., the following ML statement matches any add statement which has at least one "0" operand:

Label	Operator	Operands	Attributes
?label	add	?*ops1 "0" ?*ops2	?*attributes

If "add" is a binary operator, one of ?*ops1 and ?*ops2 will be assigned no components during the match. The ?*attributes wild card shows the more traditional use of unambiguous wild cards to match a whole field for later replication in the replacement.

The $* wild card matches a sequence of zero or more IL statements. Unlike ?* however, the sequence is not determined by lexical juxtaposition in the IL program but by flow of control: statements are considered adjacent in the process of matching if one might follow the other in execution. Branches and joins in the flow of control often result in more than one possible sequence of statements that could match a $* wild card. For example, consider the IL program given in Figure 2.1 and the following sequence of ML statements:

Label	Operator	Operands	Attributes
Z	declaration		
	$*A		
Z	?op	?*opnds	

Figure 3.1 shows the two possible sequences of IL statements that could be matched by $*A. In such cases, both sequences are saved as possible values for $*A. The most common use of $* wild cards (and the sets of statement sequences that they match) is to establish the context of a transformation — there exist built-in functions that test these sequences for simple properties (e.g., presence of a given lvalue in the label field of at least one statement in one of the sequences).

Label	Operator	Operands	Attributes
C1	constant	"2"	<C1>:type=integer
C2	constant	"3"	<C2>:type=integer
T1	greater_than	<X> <Y>	
→	if_goto	<T1> L2 L1	
●	label	L1	
X	store	<C2>	
Y	store	<C1>	
→	goto	L3	
●	label	L3	
T2	add	<X> <Y>	

Label	Operator	Operands	Attributes
C1	constant	"2"	<C1>:type=integer
C2	constant	"3"	<C2>:type=integer
T1	greater_than	<X> <Y>	
→	if_goto	<T1> L2 L1	
●	label	L2	
X	store	<C1>	
Y	store	<C2>	
●	label	L3	
T2	add	<X> <Y>	

Figure 3.1: Matches for $*A from Figure 2.1

§3.2.2 Built-in functions

Built-in functions are used in ML statements to perform operations that require more power than simply rearranging an IL statement. A call on a built-in function has the following form:

$$function[argument_1,\ldots,argument_n]$$

The use of square brackets distinguishes built-in function calls from ordinary IL components (which are restricted to the use of parentheses). All functions return a result (no side effects are possible); this result can be used as the argument to another built-in function or, if the call was part of a replacement, become part of an IL program. The arguments to a function may be written as either IL or ML components but they must be able to be resolved by the metainterpreter to a particular IL component (or IL statement sequence for certain functions). In the process of applying the function to its arguments, the function may *abort* causing the application of the transformation to fail regardless of the location of the function call (pattern, replacement, or conditions). The main reason for aborting a function is an inappropriate argument, e.g., the argument has the wrong type, cannot be resolved to a literal, etc. For instance, the add function aborts if both operands are not literals that can be interpreted as numeric quantities.

By way of example, several functions are described below; this list is not meant to be complete -- only a sampling of each category of function have been described. It is expected that an implementation would expand the list; the only criterion for including a function is that it not cater to a specific language or machine. The following argument types are used in describing functions:

component	Any IL component is an acceptable argument.
literal	The argument must be an IL literal (i.e., an operator, attribute reference, or operand enclosed in quotes).
number	The argument must be an IL literal which can be interpreted as a number (i.e., it contains only digits, a decimal point, and a sign).
boolean	The argument must be one of the IL literals "true" or "false".
sequence	The argument must be the result of a $* wild card match (i.e., a set of IL statement sequences).

If the supplied argument does not have the correct type, the metainterpreter will abort the application of the function and hence the application of the transformation in which it appears.

and[*boolean*,*boolean*]
or[*boolean*,*boolean*]

not[*boolean*]

the standard boolean functions evaluating to the literals "true" or "false" as appropriate. These are used most often in conjunction with other functions to form more complicated expressions.

equal[*literal*,*literal*]

compares two literals to see if they have the same representation; evaluates to "true" if they do, "false" otherwise. Note that equal cannot be used to compare two arbitrary IL components -- this can usually be accomplished directly in the pattern by using the same wild card name in both component locations.

constant[*component*]

evaluates to "true" if the argument is a *literal*, "false" otherwise.

lvalue[*component*]

evaluates to true if the argument represent a valid lvalue.

label[*label*,*sequence*]

evaluates to "true" if any member of the augmented kill set represented by *label* appears in the label field of a statement contained in the set of IL statement sequences *sequence*. This function determines whether a cell(s) has been modified in an IL statement sequence. The label function is representative of functions that search IL statement sequences for simple properties; other functions that test for properties in every sequence and search other statement fields should be included.

add[*number*,*number*]
subtract[*number*,*number*]
multiply[*number*,*number*]
divide[*number*,*number*]

the standard arithmetic functions returning the appropriate numeric literal. In order to avoid representation problems, a precision limit may be set by the implementation.

power_of_two[*number*]

evaluates to "true" if the argument is a numeric literal which is a power of two, "false" otherwise. This function is useful for determining when to change multiplications and divisions into shifts. This example represents the tip of the iceberg when it comes to useful arithmetic functions -- a reasonable subset might be to include only operations on binary representations (binary log, logical and arithmetic shifts, etc.).

Choices of the domain (arguments for which the function will not abort) for the predicates described above have been made arbitrarily. All that really matters is that the choices are consistent with the use of the functions in the transformation catalogue.

§3.3 Transformations and pattern matching

A transformation is made up of three components: a pattern, a replacement, and a set of conditions. The pattern (an IL program fragment) and the conditions (a set of predicates) establish the context of the transformation by identifying those IL program fragments on which the transformation can operate. A contiguous group of statements within the pattern is designated as the *target* – these statements must be contiguous as they will be replaced in their entirety by the new IL program fragment constructed from the replacement once the context has been verified†.

The following criteria must be met before a transformation can be applied:

(1) all components of the pattern must match some component in the IL program fragment (and vice versa). Duplicated wild cards must have matched IL components with the same representation.

(2) each of the conditions must evaluate to true. If any condition aborts (see §3.2.2), the application of the transformation fails. Note that conditions may use named wild cards from the pattern as part of an argument; these wild cards will be replaced by the IL component(s) they matched during (1) before evaluation of the function.

(3) the target must be a contiguous group of statements from the matched IL program fragment.

(4) the replacement must be successfully constructed – each in-line built-in function call must be evaluated without aborting.

If all these criteria are met, the newly constructed replacement is substituted for the target, completing the application of the transformation.

The following section describes the syntax of a transformation in more detail; §3.3.2 outlines how the replacement is constructed.

† Statement sequences matched by $* wild cards cannot, in general, be used in a target since they do not necessarily contain lexically adjacent statements. For similar reasons, $* wild cards are seldom used in the specification of a replacement.

§3.3.1 The syntax of a transformation

A transformation has the following form:

Label	Operator Operands	Attributes
	... pattern goes here ...	
	... replacement goes here ...	
	... conditions go here ...	

The first section contains the ML program fragment which serves as the pattern, the second section contains the replacement (also an ML program fragment), and the final section contains a set of conditions (if no conditions are needed, the final section may be omitted). Target statements within the pattern are indicated by a double vertical bar to their left. For example:

Label	Operator	Operands	Attributes
→	beq	?dest1 ?next	location=?branch_pc
●	label	?next	
→	jmp	?dest2	
●	label	?dest1	
	$*		
●	label	?dest2	location=?dest_pc
→	bne	?dest2 ?dest1	location=?branch_pc
conditions: less_than[subtract[?dest_pc,?branch_pc],"255"]			

In this transformation the first three statements of the pattern are the target and will be replaced by the single statement replacement when the transformation is applied. The remaining statements matched by the pattern (two labels and the intervening statements) will be unchanged. The intent of the transformation is to use the short address form for the jump-if-not-equal construct formed by the first three statements if the ultimate destination (?dest2) is not too far away (less than 255 bytes). This transformation only handles forward jumps — another transformation would be needed to accommodate jumps in the other direction. Other points to note: the use of duplicate wild cards to specify that the same IL component must appear in more than one place; the first and last statement of the matched fragment must have location attributes.

With one exception, each component of the matched IL program fragment must be subsumed by some component of the pattern. The contents of the attribute field are exempt from this condition -- attributes in the IL fragment that are not named in the pattern do not enter into the matching process. The use of a ?* wild card to capture the unspecified attributes for later replication in the replacement is not necessary as there are special rules concerning them in construction of the replacement (see §3.3.2). Thus, attributes are largely transparent to a transformation; the information they contain is automatically copied to the updated program wherever necessary. New attributes may be added to any statement or call by simply including the appropriate assignment in the replacement. In the example above, a location attribute is defined for the new "bne" statement with the same value as the location attribute for the original "beq" statement.

A new rvalue for a call may be indicated to the metainterpreter by including an assignment to the rvalue (similar to the definition of an attribute) in the attribute field of the appropriate statement in the replacement. For example, the following transformation replaces the addition of two constants with a store operation, indicating that the destination of the store has acquired a new value which is the sum of the constants.

Label	Operator	Operands	Attributes
?dest	plus	?op1 ?op2	
?dest	store	add[?op1,?op2]	<?dest>=add[?op1,?op2]
conditions: and(number(?op1),number(?op2))			

The result from the call to add in the operand field of the replacement will be automatically surrounded by quotes (to indicate that the new operand is a literal). The number built-in function returns "true" if its argument is a numeric literal; the condition could be omitted entirely as add aborts if its arguments are not numeric literals, causing the transformation to fail. Note that the rules mentioned in the previous paragraph will ensure that any attributes defined for ?dest in the original statement will be added to the attribute field for the store statement. Finally, it is worth pointing out that ?op1 and ?op2 do not need to be literals in the original program – ?op1 and ?op2 need only be able to be resolved to literals when the transformation is applied. For example, the statement "add <X> <Y>" would match the pattern if <X> and <Y> were both known to have constant values. These values would have been established in previous statements by including assignments to <X> and <Y> in the attribute fields of those statements.

§3.3.2 Constructing the replacement

Two capabilities are provided by the replacement that have not been discussed previously: the generation of new symbols unused elsewhere in the program and the automatic handling of attributes. The ability to generate an unused symbol is necessary when the transformation expands a single statement into a series of new statements as temporary cells used by the new statements need to be supplied names that are not used elsewhere in the program. Automatic handling of attributes enables the designer to ignore attributes with which he is not directly concerned and guarantees that no attribute information will be lost through an oversight in composing the transformation.

When expanding the specification of the replacement to arrive at the new program fragment all wild cards must be eliminated. If the wild card has the same form and name as one which appeared in the pattern, the IL component matched by that wild card serves as its value in the replacement. For instance, applying the last transformation in the previous section to

Label	Operator	Operands	Attributes
A.1	add	"54" "40"	

would result in the replacement

Label	Operator	Operands	Attributes
A.1	store	"94"	<A.1>=94

If a ? wild card in the replacement does not correspond to some wild card in the pattern (i.e., its name is different from any used in the pattern), a new lvalue is created to be used as its value. The new lvalue is guaranteed to be different from any used in the remainder of the IL program. Note that the designer must include any attributes to be associated with the new lvalue as part of the transformation. If there are no wild cards in the pattern that correspond to $, ?*, and $* wild cards in the replacement, the transformation is illegal and will never be applied.

As an example of generated lvalues consider the following transformation concerned with the expansion of the subscript operator:

Label	Operator	Operands	Attributes
?ptr	subscript	?array ?index	?ptr:class=temporary
?t1	convert	?index	?t1:class=temporary <?t1>:type=integer
?t2	subtract	<?t1> ?array:lower_bound	?t2:class=temporary <?t2>:type=integer
?t3	multiply	<?t2> <?array>.*:size	?t3:class=temporary <?t3>:type=integer
?ptr	add	<?t3> ?array	<<?ptr>>:type=<?array>.*:type

The convert operator in the first line of the replacement will coerce the value of the index to type "integer" (see §3.4 for a sample definition of convert). ?t1, ?t2, and ?t3 are all new cells which will be named when this transformation is applied; ?ptr, ?array, and ?index will be taken from the subscript statement matched by the pattern. Note that pertinent attributes for the new cells have been defined in the transformation. The attribute defined in the last line of the replacement indicates that the type of the value pointed to by ?ptr is the same as the type of an element in the array being subscripted.

The following rules are used in establishing attributes for statements in the replacement:

1. Every attribute definition in the target statements will be copied to the attribute field of some statement in the replacement by the metainterpreter when it applies the transformation. Where possible the statement chosen in the replacement field will have the same label as the defining statement in the target — this does not make any difference as far as defining the attribute is concerned, but it improves the documentation value of the definition. If there are no statements in the replacement (the target is being completely eliminated), some other statement in the updated program is chosen to receive the definitions.

2. If applying a transformation would result in a conflicting attribute definition (i.e., two or more definitions of the same attribute with different values), the transformation fails.

3. Statement attributes are never copied to the replacement; only cell attributes are updated.

Rule 2 ensures that once defined, attributes can be counted on to maintain their original value (i.e., attribute definitions are conserved).

§3.4 Example transformations

The first example is a transformation which expands the coercion operator used in the sample expansion of subscript in the previous section. The convert operator coerces its argument to have the type of destination cell; it assumes that types are constrained to be one of "integer" or "real".

Label	Operator	Operands	Attributes
?result	convert	?arg	
→	if_goto	equal[?result:type,?arg:type] ?L1 ?L2	
●	label	?L1	
?result	store	?arg	
→	goto	?L5	
●	label	?L2	
→	if_goto	equal[?result:type,"integer"] ?L3 ?L4	
●	label	?L3	
?result	real_to_int	?arg	
→	goto	?L5	
●	label	?L4	
?result	int_to_real	?arg	
●	label	?L5	

It is expected that all the testing and branches can be done at compile time. For example, if ?arg:type=integer and ?result:type=real then the replacement can be reduced to a single statement by elimination of dead code and compile-time evaluation of the if_goto operations. Although this transformation is lengthy due to the lack of any sugaring in ML for dispatching on the values of attributes, it was straightforward to construct. Note that this transformation cannot be applied if either ?arg:type or ?result:type is undefined (equal will abort). Through the use of conditions, it would be possible to rewrite the single transformation above as three separate transformations, one for each of the cases treated; the amount of optimization required to achieve the same result as above would be considerably reduced.

The following series of transformations deal with the expansion of the store operator. Unlike the transformation above, these expansions must be done in separate transformations because of the use of the ALIAS operator†. The first transformation handles the case where the store operation can be eliminated completely because the destination is a newly defined temporary and the value

† The ALIAS operator, like attributes, provides information which is independent of the flow of control; branches cannot prevent "execution" of the alias operation. Thus, the strategy used for expanding the convert operator cannot be used.

being stored is already contained in an accessible cell. In this case, all that needs to be done is alias the temporary to the cell already containing the value (effectively renaming all occurrences of the temporary to use the cell name).

Label	Operator	Operands	Attributes
?dest	store	<?source>	?dest:class=temporary
?dest	ALIAS	?source	
conditions:	and[equal[<?dest>:type,<?source>:type],lvalue[?source]]		

The next two transformations translate the store instruction to the appropriate machine instruction, depending on the type of the destination.

Label	Operator	Operands	Attributes
?dest	store	?source	
?dest	mov	?source	
conditions:	equal[<?dest>:type,"integer"] equal[<?source>:type,"integer"]		

Label	Operator	Operands	Attributes
?dest	store	?source	
?dest	movf	?source	
conditions:	equal[<?dest>:type,"real"] equal[<?source>:type,"real"]		

These two transformations "overlap" the first – program fragments matched by the first transformation will also be matched by one of the other two transformations. It is up to the metainterpreter to decide which of the applicable transformations to apply; presumably the first transformation will be used whenever possible because of the reduced cost of the resulting code. The final transformation accommodates store statements whose source and destination have different types.

Label	Operator	Operands	Attributes
?dest	store	?source	
?t1	convert	?source	?t1:class=temporary <?t1>:type=<?dest>:type
?dest	store	<?t1>	
conditions:	not[equal[<?dest>:type,?source:type]]		

CHAPTER FOUR

§4.1 Example: a mini-translator

As an example of the IL/ML system in action, this chapter presents a catalogue of patterns describing the translation of a simple block-structured language to a PDP11-like assembly language. The initial IL program is the program to be translated, for example:

Label	Operator	Operands	Attributes
	begin	PROG	
A	declaration		A:type=automatic <A>:type=integer <A>:size=2
B	declaration		B:type=external <B>:type=integer <B>:size=2
C	declaration		C:type=automatic <C>:type=integer <C>:size=2
A	assign	"1"	
B	assign	"2"	
T1	plus	<A> <B>	T1:type=temporary <T1>:type=integer
T2	plus	<T1> "[illegible]"	T2:type=temporary <T2>:type=integer
C	assign	<T2>	
	end	PROG	

The final output of the IL/ML system is an IL assembly language program which implements the initial high-level (?) program. Starting with the program above, one possible outcome might be:

```
global  B          ;declare B to be external
mov     sp,r5      ;initialize local frame pointer
sub     #4,sp      ;allocate automatic storage
mov     #1,(r5)    ;perform assignment to A
mov     #2,B       ;and then assignment to B
mov     #[illegible],2(r5)  ;and do C [illegible]
add     #4,sp      ;finally deallocate storage
```

The toy language used in this example is very rudimentary: the only operations are addition and assignment; the order of expression evaluation is constrained to be left-to-right (no reordering is allowed); all quantities are 16-bit two's complement integers (the same for both the source and machine language). In examining the assembly language program, it is apparent that certain conventions have been used in the translation: r5 is used as the local stack frame pointer, external variables are referenced by name, local (automatic) storage for blocks is allocated from the stack and referenced using the local stack frame pointer, and so on. These conventions are established originally by the designer and implemented by transformations in a straightforward fashion.

Although it is possible to interpretively apply the transformations and derive a translation, the reader should be reminded that the main goal of the transformations is to be *descriptive*. Many of the transformations below employ attributes and conditions that represent a reasonable description of the information and constraints involved in a transformation — these transformations are not the most elegant expression of the necessary syntactic transformation. In the final analysis, a transformation should be judged on the information it conveys and not how close it comes to "the way it should *really* be done."

The approach adopted for the organization of the transformations is as follows: the initial IL program is first translated into instructions for a stack architecture, then the updated program is translated into target machine instructions. Optimizations exist for each level of intermediate program — sample high-level optimizations are described in §4.3, stack optimizations in §4.1, and peephole machine optimizations in §4.2.

The first group of transformations describes the process of storage allocation. An "offset" attribute is introduced for each automatic variable declared in the block, giving the variable's offset from the base of the local stack frame; the highest offset assigned is used in calculating the storage to be allocated for the block when it is entered.

Label	Operator Operands	Attributes
	begin ?name	
	enter ?name:storage comment	 offset=0

In this transformation, the "begin" statement is translated to instructions that allocate a stack frame of the appropriate size -- the size (?name:storage) is known to be a constant but its value has not yet been determined. The last statement in the replacement initializes the offset for later transformations -- its initial value indicates that storage is allocated anew for each block. The comment operator is ignored by assembler and will be used in the transformations as an operator in statements where only the attribute fields are used. Comment statements could be eliminated altogether and their associated attribute definitions placed in attribute fields of other statements; they are used here to improve the readability of the IL program examples.

Label	Operator Operands	Attributes
  ?name	comment $*stat declaration	offset=?off ?name:type=automatic
	comment	?name:offset=?off offset=max[?off,<?name>:size]
conditions; not[attribute["offset=?",$*stat]] not[operand["declaration",$*stat]]		

Label	Operator Operands	Attributes
?name	declaration	?name:type=external
	global ?name	

The two transformations above handle declaration processing -- automatic variables are assigned offsets, external variables are declared global. In the first transformation, offsets are propagated with the aid of a comment statement that

Label	Operator	Operands	Attributes
	enter	PROG:storage	
	comment		offset=0 registers={R0 R1 R2 R3 R4}
	comment		A:type=automatic A:offset=0 <A>:type=integer <A>:size=2 offset=2
	global	B	B:type=external <B>:type=integer <B>:size=2
	comment		C:type=automatic C:offset=2 <C>:type=integer <C>:size=2 offset=4
A	assign	"1"	
B	assign	"2"	
T1	plus	<A> <B>	T1:type=temporary <T1>:type=integer
T2	plus	<T1> "0"	T2:type=temporary <T2>:type=integer
C	assign	<T2>	
	exit	PROG:storage	
	comment		PROG:storage=#4

Figure 4.1: Sample program after declaration transformations

gives the current offset. The $*stat wild card will match only statement sequences that do not contain an "offset" attribute definition or "declaration" operator in any statement (this restriction is embodied in the condition). Note that attributes defined for the declared variables will be automatically copied over to some replacement statement (in these cases, there is only one).

Label	Operator	Operands	Attributes
	comment $*stat end	 ?name	offset=?off
	exit comment	?name:storage	 ?name:storage=#?off
conditions: not[attribute["offset=?",$*stat]] not[operator["declaration",$*stat]]			

This transformation handles block exit after all declarations have been processed, deallocating storage for the block and defining the storage size attribute (?name:storage) for use during block entry. The condition is similar to that for automatic variable declarations. Figure 4.1 shows the IL program after these

Label	Operator	Operands	Attributes
	enter	PROG:storage	
	comment		offset=0
			registers={R0 R1 R2 R3 R4}
	comment		A:type=automatic A:offset=0
			<A>:type=integer <A>:size=2
			offset=2
	global	B	B:type=external
			<B>:type=integer <B>:size=2
	comment		C:type=automatic C:offset=2
			<C>:type=integer <C>:size=2
			offset=4
	push	"1"	
	pop	<A>	
	push	"2"	
	pop	<B>	
	push	<A>	
	push	<B>	
	add		T1:type=temporary <T1>:type=integer
	push	"6"	
	add		
	pop	<C>	T2:type=temporary <T2>:type=integer
	exit	PROG:storage	
	comment		PROG:storage=#4

Figure 4.2: Sample program after translation to stack machine

transformations have been applied.

The next two transformations translate "plus" and "assign" to stack operations. The information in the label field is incorporated into the operand field of the new instructions and the three-address "plus" operation is expanded into a series of one-address stack operations. Type considerations are ignored; in this case, propagation of the "integer" type attributes would not change the code generated.

Label	Operator	Operands	Attributes
?dest	assign	?source	
	push	?source	
	pop	<?dest>	

Label	Operator	Operands	Attributes
?dest	plus	?op1 ?op2	
	push	?op1	
	push	?op2	
	add		
	pop	<?dest>	

The following two transformations perform simple optimizations on the stack machine code generated so far. Both transformations improve on pop/push instruction pairs that have identical operands: the first transformation eliminates pairs whose arguments are temporaries; the second transformation converts pairs whose arguments are variables to a copy from the top of the stack. Since temporaries were generated by the compiler and do not represent user-visible quantities, they may be eliminated during optimization. Figure 4.2 shows the example IL program after translation to stack instructions.

Label	Operator	Operands	Attributes
	pop	?arg	
	push	?arg	
conditions: equal[?arg:type,"temporary"]			

Label	Operator	Operands	Attributes
	pop	?arg	
	push	?arg	
	copy	?arg	
conditions: not[equal[?arg:type,"temporary"]]			

§4.2 Compiling past the machine interface

In this section, we deal with translating stack machine programs to target machine programs. The first set of transformations are a straightforward translation of "push", "pop", "copy", and "add" to PDP11-like instructions. The size in bytes and number of storage references required for each machine instruction are indicated by the "size" and "refs" attributes respectively.

Label	Operator	Operands	Attributes
	push	?arg	
	mov	?arg -(sp)	size=2 refs=2

Label	Operator	Operands	Attributes
	pop	?arg	
	mov	(sp)+ ?arg	size=2 refs=2

Label	Operator	Operands	Attributes
	copy	?arg	
	mov	(sp) ?arg	size=2 refs=2

Label	Operator	Operands	Attributes
	add		
	add	(sp)+ (sp)	size=2 refs=3

Label	Operator	Operands	Attributes
	enter	?size	
	mov sub	sp r5 ?size sp	size=2 refs=1 size=4 refs=2

Label	Operator	Operands	Attributes
	exit	?size	
	add	?size sp	size=4 refs=2

Initial values for the "size" and "refs" attributes do not take operands into account – the operand's contributions will be included when they are translated to legal assembly language construts.

The next group of transformations translates individual operands into the appropriate machine addresses. Recall that r5 is used as the base of frame pointer and that external operands are addressed by name.

Label	Operator	Operands	Attributes
	?rator	?"before <?rand> ?"after	size=?size refs=?refs
	?rator	?"before ?rand:offset(r5) ?"after	size=add[?size,"2"] refs=add[?refs,"2"]
conditions: equal[?rand:type,"automatic"]			

Label	Operator	Operands	Attributes
	?rator	?*before <?rand> ?*after	size=?size refs=?refs
	?rator	?*before ?rand ?*after	size=add[?size,"2"] refs=add[?refs,"2"]
conditions: equal[?rand:type,"external"]			

Label	Operator	Operands	Attributes
	?rator	?rand ?dest	size=?size refs=?refs
	?rator	#?rand ?dest	size=add[?size,"2"] refs=add[?refs,"1"]
conditions: constant[?rand]			

?*before and ?*after are ambiguous wild cards used to select any component in the operand field that has the correct form (specified by the remaining component in the pattern's operand field). Note that the specification of "size" and "refs" attributes in the patterns ensures that the transformations will only be applied to machine instructions. Figure 4.3 shows the IL program after application of these transformations (unused attributes have been eliminated for brevity).

The most obvious optimization opportunity involves a push onto the stack (a "mov" instruction with a second argument of "-(sp)") followed by an instruction that pops the stack to get its source operand (an instruction with a first argument of "(sp)+"). Since an "add" can take the same source operands as a "mov" instruction, the push/pop sequences can be reduced to a single instruction:

Label	Operator	Operands	Attributes
	mov ?op	?source -(sp) (sp)+ ?dest	size=?size1 refs=?refs1 size=?size2 refs=?refs2
	?op	?source ?dest	size=subtract[add[?size1,?size2],"1"] refs=subtract[add[?refs1,?refs2],"2]

Figure 4.4 shows the effect of this single optimization.

Many other machine level optimizations are possible at this point; several optimizing transformations are listed below. These include removing superfluous zeroes in index expressions, eliminating additions with a zero operand, and

Label	Operator	Operands	Attributes
	mov	sp r5	size=2 refs=1
	sub	PROG.storage sp	size=4 refs=2
	global	B	
	mov	#1 -(sp)	size=4 refs=3
	mov	(sp)+ 0(r5)	size=4 refs=4
	mov	#2 -(sp)	size=4 refs=3
	mov	(sp)+ B	size=4 refs=4
	mov	0(r5) -(sp)	size=4 refs=4
	mov	B -(sp)	size=4 refs=4
	add	(sp)+ (sp)	size=2 refs=3
	mov	#0 -(sp)	size=4 refs=3
	add	(sp)+ (sp)	size=2 refs=3
	mov	(sp)+ 2(r5)	size=4 refs=4
	add	PROG.storage sp	size=4 refs=2
	comment		PROG.storage=#4

Figure 4.3: Sample program after translation to machine instructions

Label	Operator	Operands	Attributes
	mov	sp r5	size=2 refs=1
	sub	PROG.storage sp	size=4 refs=2
	global	B	
	mov	#1 0(r5)	size=6 refs=4
	mov	#2 B	size=6 refs=4
	mov	0(r5) -(sp)	size=4 refs=4
	add	B (sp)	size=4 refs=4
	add	#0 (sp)	size=4 refs=3
	mov	(sp)+ 2(r5)	size=4 refs=4
	add	PROG.storage sp	size=4 refs=2
	comment		PROG.storage=#4

Figure 4.4: Sample program after push/pop optimization

eliminating unnecessary moves.

Label	Operator	Operands	Attributes
	?rator	?*before 0(r5) ?*after	size=?size refs=?refs
	?rator	?*before (r5) ?*after	size=subtract[?size,"2"] refs=subtract[?refs,"1"]

Label	Operator	Operands	Attributes
	add	#0 ?dest	size=?size refs=?refs

Label	Operator	Operands	Attributes
	mov	<?source> <?source>	size=?size refs=?refs

Figure 4.6 shows the IL program after application of these final transformations – comment and attributes have been omitted and attribute references resolved. Obviously, additional transformations would be needed to handle optimization opportunities that arise from the translation of other programs; however, the bulk of the translation can be accomplished with these few transformations.

§4.3 Interacting with the metainterpreter

The transformations in the previous section dealt with the translation of the input program to a target machine program with little attention to the semantics of the initial IL program. For the most part, the metainterpreter had only to choose which transformations to apply – this task was made fairly simple for, in almost every case, if the transformation's pattern and conditions were met, it was appropriate to apply the transformation. This section explores how the capabilities of the metainterpreter can be called into play to improve the quality of the resulting translation.

The first example exploits the metainterpreter's ability to perform certain computations at compile time. Consider the addition of the following transformations to the catalogue:

Label	Operator	Operands	Attributes
	mov	sp r5	size=2 refs=1
	sub	#4 sp	size=4 refs=2
	global	S	
	mov	#1 (r5)	size=4 refs=3
	mov	#2 S	size=6 refs=4
	mov	(r5) -(sp)	size=2 refs=3
	add	S (sp)	size=4 refs=4
	mov	(sp)+ 2(r5)	size=4 refs=4
	add	#4 sp	size=4 refs=2

Figure 4.5: Sample program after final optimizations

Label	Operator	Operands	Attributes
?dest	assign	?source	
?dest	assign	?source	<?dest>=?source

Label	Operator	Operands	Attributes
?dest	plus	?op1 ?op2	
?dest	assign	add[?op1,?op2]	<?dest>=add[?op1,?op2]

These transformations tell how the rvalue of the result cell is affected by the "assign" and "plus" operators. Using the definition of "add" given in §3.2.2, the second transformation will only succeed if ?op1 and ?op2 are numeric literals. By extending the metainterpreter to support symbolic computation, both the transformations above would be useful even for non-literal operands (although the second transformation should not eliminate the explicit plus operation unless the add would succeed at compile time). The primary benefit of such an extension would be a corresponding extension in the metainterpreter's ability to detect redundant computations.

Applying these transformations to the sample program in the first section, the metainterpreter can acquire the following rvalue information:

<A>="1" <B>="2" <T1>=<C>="3".

As a result of this new information, the initial program can be modified as shown in Figure 4.6 (update of Figure 4.2). By adding a transformation to eliminate assigns

Label	Operator	Operands	Attributes
	enter	PROG:storage	
	comment		offset=0
	comment		A:type=automatic A:offset=0
			<A>:type=integer <A>:size=2
			offset=2
	global	B	B:type=external
			<B>:type=integer <B>:size=2
	comment		C:type=automatic C:offset=2
			<C>:type=integer <C>:size=2
			offset=4
A	assign	"1"	
B	assign	"2"	
T1	assign	"3"	T1:type=temporary <T1>:type=integer
T2	assign	"3"	T2:type=temporary <T2>:type=integer
C	assign	"3"	
	exit	PROG:storage	
	comment		PROG:storage=#4

Figure 4.6: Sample program after declaration transformations

Label	Operator	Operands	Attributes
	mov	sp r5	size=2 refs=1
	sub	PROG:storage sp	size=4 refs=2
	global	B	
	mov	#1 (r5)	size=4 refs=3
	mov	#2 B	size=6 refs=4
	mov	#3 2(r5)	size=6 refs=4
	add	PROG:storage sp	size=4 refs=2
	comment		PROG:storage=#4

Figure 4.7: Sample program after optimizations of §4.3

to subsequently unused temporaries, the transformations of §4.2 can produce a program identical to the assembly language program given in §4.1 (see Figure 4.7).

CHAPTER FIVE

§5.1 Summary

The emphasis of this thesis has been on developing a framework that can be used in the specification of a code generator. The major design goal for this framework was to segregate language and machine dependencies from the remainder of the code generation process while maintaining the ability to produce optimized code. A three component system was developed that makes a significant step towards reaching this goal. Although many features provided by the system are in need of polishing to remove their rough edges, the specification that emerges seems to satisfy the initial design goal. The proposed system is simple compared to many of the available alternatives; there are no severe restrictions on the class of languages or machines that can be accommodated.

Chapter 2 describes a general purpose intermediate language based on a semantics common to a wide class of programs: the only primitives concern flow of control and management of names and values. The syntax has been designed to place information important to optimization algorithms in separate fields so at to be accessible to the metainterpreter without a detailed analysis of the actual operation performed by each statement. Information about the flow of control and the effect of each statement on the values of variables can be easily determined from the label field of that statement. In addition, attributes provide a general mechanism for accumulating declarative information about each variable and statement. Attributes can be used to supply a symbol table facility for variables and contextual information for statements. Moreover, the form of this information allows it to be referenced by the transformations, permitting the translation of statements to be tailored in response to special properties of the operands or opportunities presented by the context.

In Chapter 3, the transformation catalogue is discussed and the metalanguage in which the individual transformations are written is presented. The metalanguage provides the ability to describe classes of IL program fragments, leaving statements and components unspecified through the use of wild cards. Each transformation contains two ML program fragments (templates): a pattern that, along with a set of conditions, specifies the IL program fragments to which the transformation can be applied, and a replacement that tells how to construct an updated IL program. Built-in functions that allow access to some of the metainterpreter's knowledge about IL programs and perform some simple computations on literals are provided — these functions are used in constructing the replacement and conditions. The conditions associated with a transformation specify contextual constraints that are not related to the syntactic form of the matched fragment. The wide range of information available to a transformation enables the semantics of code generation to be expressed as step-by-step syntactic transformations of the intermediate language program.

Chapter 4 presents a set of example transformations as a specification for translating a rudimentary source language to PDP11-like assembly language. As suggested in §1.3.1, the transformations are organized about the use of an abstract machine (in this case, with a stack architecture). The initial translation to stack machine instructions allows several optimizations to be accomplished that would have otherwise been difficult (e.g., the removal of unnecessary temporaries inserted by the first phase of the compiler). Several transformations that allow the metainterpreter to infer the run time values of the variables and subsequently optimize the resulting code are included in the catalogue, performing several operations at compile time that had previously appeared in the final assembly program. Although the example is fairly simple, it demonstrates the feasibility of the proposed approach to code generation.

The final component of the proposed framework – the metainterpreter – has only been alluded to in the previous chapters. The next section provides a brief overview of the capabilities the metainterpreter needs to supply. The final section of this chapter points out several directions in which subsequent research might head.

The most important contribution made by this research is the design of an intermediate language that caters to the need for the careful data flow analysis that is the foundation of many optimizations. In contrast, many so called portable code generation schemes sacrifice the ability to do any substantial processing during code generation to achieve their portability. The type of processing needed to be done by an optimizing code generator is becoming more systematized: recent advances in the theory of code generation have provided algorithms where only heuristics existed previously and there is promise that these advances can be extended to the problems of real languages. The approach to code generation proposed by this thesis would allow these advances to be incorporated without extensive editing of existing specifications.

§5.2 An overview of the metainterpreter

Throughout earlier portions of this thesis the metainterpreter has been assigned tasks whenever they can be divorced from the semantics of the source language and target machine; this section summarizes these tasks. The responsibilities of the metainterpreter fall in two main areas: bookkeeping and flow analysis. Bookkeeping tasks are performed whenever possible and include

- translation of attribute references to their corresponding values wherever possible. If any unresolved attribute references remain after completion of the transformation process, the metainterpreter should abort, indicating an inconsistent IL program.

- evaluation of built-in functions. If a function application aborts (e.g., because of domain errors), it is saved for reevaluation later in the translation.

- propagation of rvalue information. In combination with data from flow analysis, it is possible to replace rvalue operands with literals representing the known value of the rvalue.

- application of a chosen transformation. Information obtained during the match of the pattern is incorporated in the replacement specification (along with any generated symbols) to create a replacement for the target statements in the pattern. During the construction of the replacement, many of the other bookkeeping functions can be performed then and there, eliminating the need for extra passes over the IL program.

Two other tasks fall in this area: checking for termination conditions and choosing which transformation to apply next.

§1.3.2 outlines how to tell when the translation is complete: a measure of the programs optimality is computed using a formula (in this case, involving the values of attributes associated with every statement) supplied by the user – if the calculation aborts because some statement does not have the appropriate attributes, the application of more transformations is called for; if no more transformations are applicable, backtracking is called for. If the measure can be computed, it is used to remember the best translation found to date and the metainterpreter backtracks to find other translations. Backtracking involves undoing the last successful transformation and applying some other transformation (repeating for another level if all the applicable transformations have been applied at this level). Exhaustive search of the transformation tree can be avoided if the user supplies a "trigger" value for the measure – any program whose measure is less than the trigger value is considered an acceptable translation and becomes the final output. Often the transformations are constructed in such a way that the first successful translation will be acceptable (an infinite bigger value).

There are many ways in which to choose the next transformation to apply: the simplest is to search the transformation catalogue for a transformation that is applicable to some portion of the current IL program. A more satisfactory scheme involves completing the translation of the beginning portions of the IL program before moving on to later portions in the hope that optimizations will eliminate the need to translate some parts of the program. Cycles in the flow graph may require translation (at least in part) of portions of the program that will eventually be eliminated. Sophisticated use of the transformation catalogue requires a good understanding of the effect of each transformation, an understanding that may be difficult to achieve (see the comments on metacompilation at the end of §5.8).

Flow analysis is necessary for many of the optimizations incorporated in the metainterpreter and is doubly important as these optimizations form the basis for replacing the manual analysis conventionally applied to determine special code generation cases. It is common for transformations to do a "sloppy" job of translation, incorporating explicit tests in the expanded code rather than iterating transformations with different contexts. The optimizations listed below are capable of improving such code to the quality of code produced by human programmers writing in low-level languages [Carter]. The optimizations include

- constant propagation. This optimization assumes lesser importance in the IL/ML system since attributes provide much of the information commonly embodied as constants in other general purpose optimizing compilers.

- dead code elimination — code that can no longer be reached during execution can be removed from the IL program (remembering to save any attribute definitions somewhere else).

- redundant expression elimination. Simple determination of the redundancy of a statement can be accomplished by a straightforward lexical comparison of statements known to precede (in execution) the statement of interest, keeping in mind the possible redefinition of variables used in the expression. More complicated detection is possible when rvalue information is considered.

There are other related optimizations requiring the same data flow information.

The required flow analysis could be done anew at the completion of every transformation application but this would be incredibly inefficient -- prohibitive for large programs. The bit vector methods outlined by [Schatz] and [Ullman] offer an efficient representation of the data flow information that can be incrementally updated as long as the underlying flow graph is not changed (except to add/delete more straight-line code or loops completely contained in the added code). Thus, the more time consuming iterative calculation required when the flow graph is not known need only be performed when a transformation affects the branches and joins of the graph. A large percentage of transformations do not affect the graph itself -- all of the transformations in Chapter 3 could be accommodated by incremental analysis.

In a different vein, code motion out of loops, elimination of induction variables, etc. (see [Aho77b] for a large sample) represent other optimizations that could be incorporated in the metainterpreter. As algorithms are developed for register allocation and optimal ordering of expression execution, these will also be prime candidates for inclusion. Our shopping list can easily grow must faster than our ability to implement the algorithms effectively within the framework provided by the metainterpreter. Fortunately, some transformations are much more important that others; the list given under flow analysis is a good start towards an excellent code generator.

§5.3 Directions for future research

Two avenues of research are natural extensions of the work reported here. The examples of Chapters 3 and 4 indicate that much improvement could be made to the usability of the metalanguage. Many operations commonly performed during code generation (allocation of storage, register assignment, etc.) could benefit from direct support in ML, eliminating the need for hard-to-decipher transformations. Several suggestions are:

- the inclusion of a structure mechanism for laying out storage. Providing a general purpose allocation scheme would allow the metainterpreter to shoulder the responsibility for assigning successive offsets and ensuring the storage requirements. If the syntax [illegible] provided by [illegible] languages for variable declarations, there would be provision for variable size components and [illegible].

- support for the notion of temporary locations. Many machines have some special storage facilities that are particularly appropriate for use as temporary locations — registers, stacks, etc. In machines with no such facilities, temporary locations could be assigned from a pool of ordinary storage [illegible] by the metainterpreter. It should be possible to separately describe the allocation and deallocation of temporary locations; a distinct syntax could then be used in the ML program to bring this mechanism into play.

- adding the ability to expand user-defined procedures in ML. This capability is related to the current inability in ML to deal with groups of statements (procedures, [illegible], etc.) that [illegible] follow one another during execution. The ability to perform compile time substitution of actual arguments for formal parameters is also lacking — such an addition would allow partial evaluation of procedures at compile time.

Accumulating a variety of such optional features would greatly enhance the capabilities of ML without restricting its generality.

The second major area for future research is the implementation of a metainterpreter and/or metacompiler. The Experimental Compiler Group at the IBM Thomas J. Watson Research Center [Harrison] has implemented, with some success, a compiler (the GPO compiler) based on a similar, although more restricted, scheme. The GPO compiler alternately applies transformations to remaining high-level operations and optimizes the current intermediate program. A similar straightforward implementation effort for the IL/ML system would have similar prospects for success. Enhancing the optimization capabilities of the underlying program could lead to a very competent compiler that is easily maintained and modified to produce code for different target machines.

Many other implementation approaches lie further off the beaten path. One of the most interesting is the prospect of creating a "compiled" code generator based on an analysis of the specification. Such compilation would require extensive information on the interaction between components of the specification; the metacompiler would have to "understand" the effect of each transformation in a much more fundamental way than is needed from an interpretive approach. Compiling the specification would eliminate much of the searching and backtracking described in the beginning of §5.2 with the result of a vast improvement in the performance of the code generator. The metacompilation phase will almost certainly be necessary if the performance of our code generator is to approach that of conventional ad hoc code generators.

Metacompilation is closely related to current work in the field of automatic program synthesis. The specification provided by the IL/ML system has many of the same characteristics as descriptions used in these synthesis systems [Green]: a pattern-based transformation system is used as the knowledge base by both systems. This commonality promises to allow many of the same techniques to be used in the analysis of the specification. This area of research is still virgin territory with the same promises of success and failure offered by any frontier.

BIBLIOGRAPHY

1. Aho A. V., R. Sethi and J. D. Ullman. "A Formal Approach to Code Optimization," *SIGPLAN Notices* 5:7 (July 1970), pp. 86-100.

2. Aho, A. V., S. C. Johnson, and J. D. Ullman. "Code Generation for Expressions with Common Subexpressions," *JACM* 24:1 (January 1977), pp. 146-160.

3. Aho, A. V. and J. D. Ullman. *Principles of Compiler Design*, Addison-Wesley, 1977.

4. Barbacci, M. R. and D. P. Siewiorek. *Some Aspects of the Symbolic Manipulation of Computer Descriptions*, Dept. of Computer Science, Carnegie-Mellon University, 1974.

5. Barstow, D. and E. Kant. "Observations on the Interaction between Coding and Efficiency Knowledge in the PSI Program Synthesis System," *Proc. 2nd International Conference on Software Engineering*, October 1976.

6. Bell, C. G. and A. Newell. *Computer Structures: Readings and Examples*, McGraw Hill, New York, 1971.

7. Bunza, G. Master Thesis in progress, Digital Systems Laboratory, Massachusetts Institute of Technology.

8. Carter, J. L. *A Case Study of a New Compiling Code Generation Technique*, RC 5666, IBM Thomas J. Watson Research Center, October 1976.

9. Coleman S. S., P. C. Poole, and W. M. Waite. "The Mobile Programming System, JANUS," *Software Practice and Experience* 4:1 (1974), pp. 5-23.

10. Green, C. "The Design of the PSI Program Synthesis System," *Proc. 2nd International Conference on Software Engineering*, October 1976, pp. 4-18.

11. Harrison, W. "A New Strategy for Code Generation – the General Purpose Optimizing Compiler," *Proc. of Fourth ACM Symposium on the Principles of Programming Languages*, 1977, pp. 29-37.

12. Hewitt, C. *Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot*, Massachusetts Institute of Technology Artificial Intelligence Laboratory Technical Report AI TR-258, 1972.

13. Kildall, G. A. "A Unified Approach to Global Program Optimization," *ACM Symposium on Principles of Programming Languages*, October 1973, pp. 194-206.

14. Knuth, D. E. *Examples of Formal Semantics*, Stanford Artificial Intelligence Project Memo AIM-126, 1970.

15. Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns, "Attributed Translations," *Proc. of Fifth Annual ACM Symposium on Theory of Computing*, 1973, pp. 160-171.

16. McKeeman, W. M. "Peephole Optimization," *CACM* 8:7 (July 1965), pp. 443-444.

17. Miller, P. L. *Automatic Creation of a Code Generator from a Machine Description*, Massachusetts Institute of Technology Project MAC Technical Report TR-85, 1971.

18. Neel, D. and M Amirchahy, "Semantic Attributes and Improvement of Generated Code," *Proc. ACM Annual Conference, San Diego*, 1974, vol. 1, pp. 1-10.

19. Newcomer, J. M. *Machine-Independent Generation of Optimal Local Code*, Dept. of Computer Science, Carnegie-Mellon University, 1975.

20. Poole, P. C. and W. M. Waite. "Machine-Independent Software," *Proc. ACM Second Symposium on Operating Systems Principles*, October 1969.

21. Richards, M. "The Portability of the BCPL Compiler," *Software Programming and Experience* 1:2 (1971), pp. 135-146.

22. Schatz, B. R. *Algorithms for Optimizing Transformations in a General Purpose Compiler: Propgagation and Renaming*, RC 6232, IBM Thomas J. Watson Research Center, October 1976.

23. Snyder, A. *A Portable Compiler for the Language C*, Massachusetts Institute of Technology Project MAC Technical Report TR-149, 1974.

24. Steel, T. B. "A First Version of UNCOL," *Proc. of the Western Joint Computer Conference*, 1961, pp. 371-377.

25. Ullman, J. D. "A Survey of Data Flow Analysis Techniques," *2nd USA-Japan Computer Conference Proceedings*, AFIPS, August 1975.

26. Waite, W. M. "The Mobile Programming System: STAGE-2," *CACM* 13:7 (July 1970), pp. 415-421.

27. Wick, J. D. *Automatic Generation of Assemblers*, Dept. of Computer Science, Yale University, Research Report #50, 1975.

28. Wulf, W., *et al.* *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

29. Young, R. P. *The Coder: A Program Module for Code Generation in High-level Language Compilers*, Dept. of Computer Science, University of Illinois at Urbana-Champain, UIUCDCS-R-74-666, 1974.