Palo Alto Research Center

Code Generation and Machine Descriptions

By R. G. G. Cattell

XEROX

Code Generation and Machine Descriptions

by R. G. G. Cattell

CSL-79-8 October 1979

Abstract:

This is a collection of three papers covering a Ph.D. dissertation, "Formalization and Automatic Derivation of Code Generators," Cattell[1978]. The papers are revised reprints of versions appearing in the literature. The papers describe, respectively,

1. A code generator generator (this paper includes an overview of the thesis),
2. The model of machines (instruction set processors), and
3. The table-driven code generator and portions of the compiler in which it operates.

The tables for the code generator (3) are derived by the code generator generator (1) from the machine description (2). Thus the three papers describe the three main parts of the thesis work.

CR Categories: 4.12

Key words and phrases: Code Generator Generator, Compiler-compiler, Machine Description, Optimizing Compiler, Code Generation, Compiler Generator, Computer Description

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Acknowledgements

I would like to thank Bill Wulf, Mario Barbacci, Allen Newell, Alice Parker, Joe Newcomer, John Oakley, Steve Saunders, Susan Graham, Chris Fraser, Bruce Schatz, Steve Hobbs, Steve Crocker, Bruce Leverett, and Doug Clark for their invaluable comments and conversations concerning my thesis work.

Automatic Derivation of Code Generators from Machine Descriptions

R. G. G. Cattell

Carnegie-Mellon University[1]

Abstract

Work with compiler-compilers has dealt principally with automatic generation of parsers and lexical analyzers. Until recently, little work has been done on formalizing and generating the back end of a compiler, particularly an optimizing compiler. This paper describes formalizations of machines and code generators, and a scheme for the automatic derivation of code generators from machine descriptions. It was possible to separate all machine dependence from the code generation algorithms for a wide range of typical architectures (IBM-360, PDP-11, PDP-10, Intel 8080) while retaining good code quality. Heuristic search methods from work in Artificial Intelligence were found to be both fast and general enough for use in generation of code generators with the machine representation proposed. A scheme is proposed to perform as much analysis as possible at code generator generation time, resulting in a fast pattern-matching code generator. The algorithms and representations were implemented to test their practicality in use.

1. Introduction

In the past decade, there has been increasing interest in reducing the effort to construct compilers. The problem has become more important as good-quality compilers are required for the increasingly numerous machine architectures made possible through microprogramming and LSI technology. Progress has been made in automatic generation of the parsers that translate source language into internal notation. However, it has proven

---

[1]The author's present address is Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract number F44620-74-C-0074 and is monitored by the Air Force Office of Scientific Research.

much more difficult to do the same for the second part of the compilation process: the translation of internal notation into machine code. The work presented here suggests that more general formalizations of machines and code generators are needed to allow the automatic derivation of code generators.

This work necessarily involves relatively disparate areas of computer science: computer architecture, compilers, automatic programming. It is only one of many possible applications of machine descriptions, that include emulation of machines, automatic generation of assemblers (Wick[1975]) and diagnostics (Oakley[1976]), automated hardware design (Barbacci & Siewiorek[1975]), as well as this work, which is part of the Production-Quality Compiler-Compiler (PQCC) project at Carnegie-Mellon University (Leverett et al[1979]).

The PQCC group is interested in simplifying and/or automating the construction of a high-quality compiler generating optimized code. The work is concentrating on the machine-dependent aspects of optimizing compilers, a difficult problem that has received little attention. PQCC is using the multiple-phase structure of the Bliss-11 compiler (Wulf et al[1975]) as a starting point for the research.

Some work has been done in the area of code generation in general (Wilcox[1971], Weingart[1973], Simoneaux[1975]). There have been two classes of approach to simplifying production of code generators. The first is the development of specialized languages for building code generators, with built-in machinery for dealing with the common details; this might be called the *procedural* language approach. Early work in this area was done in compiler writing systems (Feldman[1966], Feldman & Gries[1968], McKeeman et al[1970], White[1973]). Also, Elson & Rake[1970] and Young[1974] have concentrated specifically on code generator specification languages and have been relatively successful. The other extreme is the *descriptive* language approach: automatically building a code generator from a purely structural and behavioral machine description. Miller[1971], Donegan[1973], Weingart[1973], Snyder[1975], and Newcomer[1975] fit the descriptive language category, to varying degrees. A survey of the above work, particularly as it relates to the goal of automating the production of code generators, can be found in Cattell[1977].

More recently, Fraser[1977], Glanville[1977], Ripken[1977], and Johnson[1978] have done related work. All of these are concerned with formalizing the code generation process in the sense of separating the code generation algorithms from machine dependent tables with which they operate, but they differ in the generality of the machine representation and the assistance provided in constructing the tables. Ripken and Johnson propose code generation schemes based on templates mapping program trees onto instructions. Glanville also uses these templates as a machine description, but automatically derives a transition table from them; the resulting table-driven code generator is thereby faster. In this work and in Fraser's, the machine description is more complex and an analysis of the machine is needed to derive the templates. There are therefore two parts to this work, the template-

driven code generator and the template-deriving code generator generator. Fraser takes a human-knowledge-based approach to the problem, as opposed to the formal approach taken in this work; these approaches are complementary, with different strengths, providing an interesting contrast of the use of methods from Artificial Intelligence.

A common objection to general work in the code generation area has been that it has not been practically applicable. In order to demonstrate the feasibility of the ideas, a prototype system of the algorithms and representation proposed here has been implemented.

Figure 1 gives an overview of the problem viewed by this work. Three algorithms and representations are involved:

1. The formal representation of the machine, labelled MD (Machine Description) in the figure, and its extraction from a procedural machine description language such as ISP (Bell & Newell [1971]).

2. The tabular representation of the parse-tree to machine-code translation, labelled MT (Machine Tables) in the figure, and the code generation algorithms that use these tables.

3. The algorithms which derive (2) from (1) by heuristic search for optimal code sequences.

These three problems are discussed in Sections 2 through 4 of this paper, respectively.

2. Formalization of Instruction Set Processors

Before we can deal with code generators or their automatic generation, we must define the class of machines with which we are dealing. This is a crucial step. We want a machine formalization that is sufficiently restrictive to make it possible to generate code with a simple fast algorithm, but general enough to include a wide range of typical computer architectures.

We will assume such a machine consists of an *instruction set processor* that iteratively retrieves instructions from a *primary memory*, and (conditionally) changes the contents of a set of locations termed the *processor state,* as specified by the instruction.

Five main kinds of information are given by the formal machine description:

> *Storage Bases*: The basic storage array(s) of the processor state. Each has a length (the number of words in the array, possibly one), a width (number of bits per word), and a type. The type essentially specifies how the storage base can be used; it may be: General-purpose (locations that may be used to hold values), Temporary

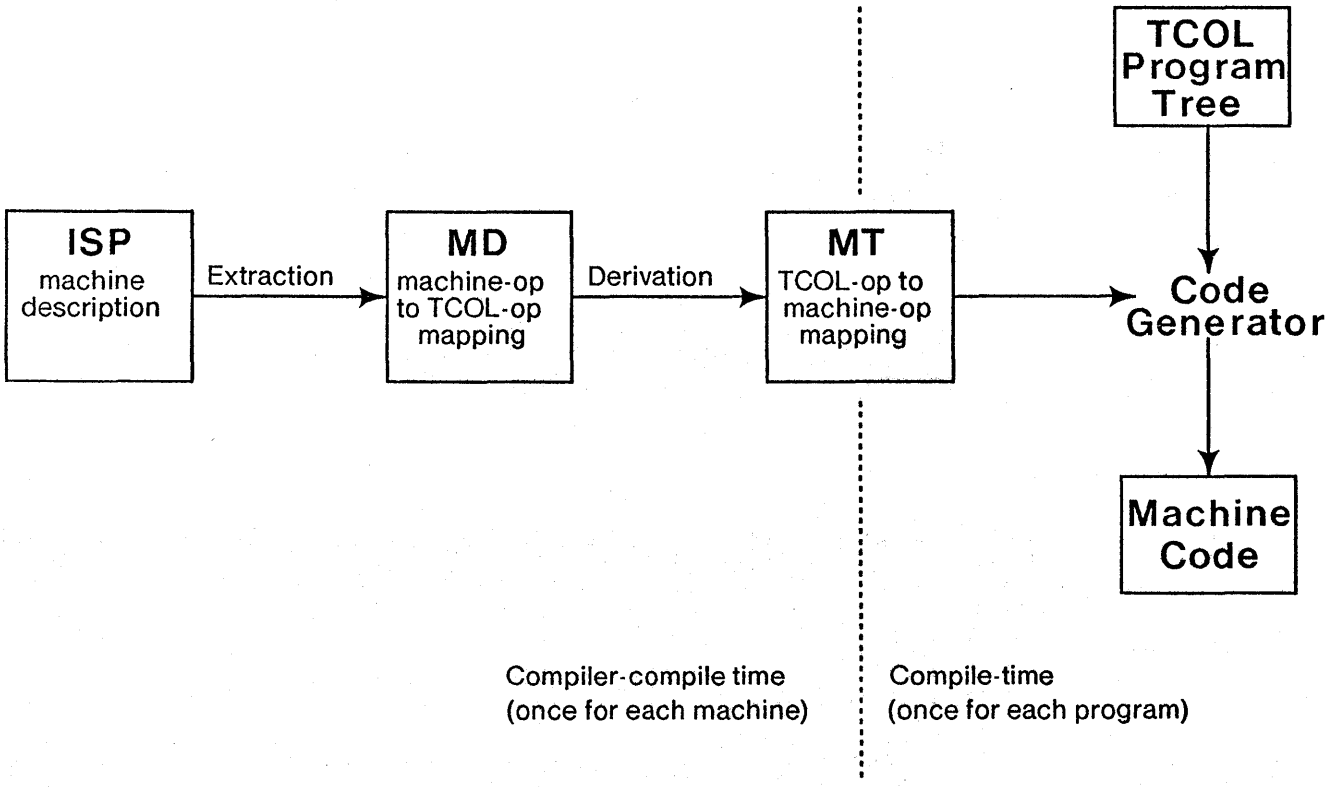

Figure 1: Relationship of the programs and representations proposed. In the horizontal direction, the construction of a code generator from the machine description is shown. The MD is the formal representation of the machine, which could be extracted from a machine description such as ISP. The MT is the formal representation of the code generation process; the code generator is table-driven by this Machine Table. The code generator derivation process constructs the MT from the MD. In the vertical direction, the use of the code generator itself is shown, translating program trees into code for the target machine. The program tree is produced by language-dependent compiler front end that translates source code into the TCOL (Tree-based Common Language) notation. Further processing such as peephole optimization may be performed on the machine code output. Note: in Cattell[1978], the terms MOP and LOP are used for MD and MT, respectively, for historical reasons.

(condition codes), or Reserved (locations such as a stack pointer that may not be used to hold values). In addition, two storage bases are distinguished as special in the machine description: the *Program Counter* (which is type Reserved), and the *Primary Memory* (General-purpose).

Operand addressing: The instructions have particular properties with respect to the kinds of storage base accesses they may make; they differ not only in *which* may be accessed, but in *how*, i.e., the address computation. We define an *Access Mode* for each distinct type of addressing on the machine; for example, indexed off a register, indirect through a memory location, or an immediate constant from an instruction field. The access mode is described by an expression that represents the access in terms of arithmetic operators and accesses to storage bases, for example, $M[C_2 + R[C_1]]$ designates accessing storage base M by indexing off register C_1 by the constant C_2. An operand of an instruction can generally belong to any of a *set* of access modes depending on opcode, mode bits, etc.; for each such set there is an *Operand Class.* For example, an ADD instruction might require a general-purpose register as the operand receiving the result, and allow either an immediate constant or a memory location as the other operand. Formally, an Operand Class is a set of tuples consisting of an access mode, its time/space cost (in this context), and a specification of the corresponding instruction field values (e.g., mode bits or address field). The separation of the operand addressing function from the instructions themselves greatly reduces the number of instruction descriptions necessary for machines with a number of addressing modes, as well as simplifying the generation of good code using address mode computations.

Machine Operations: These represent the actual instructions available. For each we need to know cost (space and time), binary formatting information, and a set of *input/output assertions*. The latter describe the effects of the instruction. Each assertion specifies a destination operand class (the location to be modified), an expression over constants and operand classes (the new value of the location in terms of the previous processor state), and a boolean function (again over the processor state) specifying when the location takes on the new value. For example, an increment-and-skip-on-zero (ISZ) instruction might have two assertions

(1) $R \leftarrow R + 1$ (2) if $(R + 1) = 0$ then $PC \leftarrow PC + 1$

represented in Algol-like form for readability. (The actual assertions are represented as trees for ease in matching to program trees; throughout this paper the details of the representations are suppressed for the sake of a clearer exposition.) The first assertion always holds; the second contains a boolean function specifying when the location (the program counter) takes on a new value. R is an operand class, that might for example represent any one of the machine's general purpose registers.

Note that references to such locations refer to their values before instruction execution, i.e., there is no ordering on the assertions.

Data Types: Machines are normally built around a set of data types. For each one, we need to know the abstract domain (e.g., reals, integers, characters) that it represents and the encoding/decoding function to/from binary bit strings (e.g., 16-bit twos complement). Each arithmetic or logical operator in the instruction assertions specifies the data type(s) on which it operates.

Instruction Fields and Formats: Finally, the machine description must specify the correspondence of the abstract operations and operands to the actual binary encoding. For example, an ADD instruction might have an instruction format with three fields: an opcode, address mode bits, and a displacement field used in the computation of the operand address. The values of these fields are determined by the instruction (e.g., opcode = 17) and its operands (e.g., mode = 1, displacement = 2473) as specified by the binary formatting information associated with machine operations and operand classes. The binary representation description will not be necessary for the purposes of this paper; note that we can ignore this description precisely *because* it has been separated from the rest of the abstract machine.

Note that the machine representation used differs from a procedural machine description language such as ISP due to the structure it requires of the machine and the non-sequential instruction descriptions. It is possible, however, to *derive* our representation from ISP with symbolic simulation and a little help from a human (Oakley[1979]).

Note also that the machine representation does *not* say how to generate code for the machine in any way; it essentially specifies a mapping from machine operations to operations of a common semantic notation (TCOL, described in the next section), and the code generation problem is to invert that mapping.

Space limitations do not allow a complete definition of the components of the machine model here. The interested reader is referred to Cattell[1978] and the second paper in this collection.

3. Formalization of Code Generation

In order to automate the generation of code generators from a machine description, it is first necessary to define what a code generator *is* in a machine-independent manner, and to separate good code generation algorithms from the machine-dependent tables they use.

The code generation scheme is based on a form of templates we will refer to as *tree*

productions, which are collected in the Machine Tables (MT). A given source program is translated into an intermediate parse-tree-like notation (TCOL) by the front end of the compiler. The code generator traverses the program tree, matching each node against *patterns* on the left-hand-sides (LHSs) of the productions in the Machine Tables. When a pattern matches, the right-hand-side (RHS) of the production specifies code to be generated, special compiler actions such as allocation, or further matches to be recursively performed. For example, a simple production might be

R←R+E ⇒ ADD R,E

where R and E are operand classes, and the RHS consists of a single instruction (an add) to be emitted (the actual syntax for productions is more complex; see the examples in Section 4.3). This production might be used in generating code for the TCOL tree X←X+2*Y: the code generator recognizes that X (allocated to a register, say) can be accessed directly by operand class R, and generates a subgoal to make 2*Y accessible via operand class E. The subgoal is of the form L←2*Y, where L is an allocated location compatible with E. Had the expression 2*Y been of a form which happens to conform to E, say the $M[C_2+R[C_1]]$ example given in the previous section (a computation of a contiguous vector element), then just the single ADD instruction would be generated for the entire assignment statement X←X+A[I].

A simple example of a production dealing with a control construct is

IF R=0 THEN S ⇒ BNE R,L1; S; L1: ... .

This illustrates a recursive call of the code generator (on the statement S) and a compiler-generated label (L1), as well as emitting an instruction (Branch if Not Equal). More complex productions are needed to deal with constructs such as loops, for example to take advantage of the ISZ instruction given in the previous section.

Thus we have a simple, machine-independent code generation algorithm, and Machine Tables that specify productions, addressing, and formatting information for the target machine. The strategy used by the basic code generation algorithm (tree traversal and subgoals) is essentially identical to the heuristic search for code sequences used in the code generator generator we will discuss in Section 4.2. The difference is that the code generator need only deal with one kind of mismatch between the source tree and pattern tree: the operands must be assigned or moved to locations compatible with the pattern requirements. This register allocation (and associated operand moves) is as essential to every code generator as the actual selection of instructions.

Register allocation is actually done before the code generation in the PQCC compiler by performing a pseudo-code-generation pass to determine where storage bases of various type are desirable. Peephole optimization and code output is performed after code generation

because certain operations are more conveniently done on a symbolic representation of the object code (e.g., resolving absolute versus short relative addressing, dealing with base registers, and eliminating redundant instructions).

A prototype of the code generator has been built. It appears that code comparable to a good hand-crafted compiler (e.g., Bliss-11) can be generated with proper care; experimental results are discussed in a later section of the paper.

Further details of the code generator, register allocation, and code optimizations have been deferred to the third paper in this collection (Cattell, Newcomer, & Leverett[1979]), so that we may proceed to the central topic of this paper, the automatic derivation of the code generators.

4. Automatic Derivation of Code Generators

In this section, we consider the problem of deriving the Machine Tables, that control the code generation process, from the Machine Description (see Figure 1).

One of the central algorithms used in the *derivation* of code generators is itself a code generator, specifically one which takes as input a machine description and [heuristically generated] TCOL tree. This machine-independent code generator could be used directly as the code generation phase of the compiler (i.e., use MD directly instead of MT in Figure 1). In practice, however, it is preferable to separate *compile*-time from *compiler-compile*-time, to make the code generator as compact and efficient as possible, allowing a thorough analysis of the alternatives at (much less expensive) compiler-compile-time. Thus we introduce this extra level of table for efficiency.

As a result, the code generator derivation process must be broken into two parts: selection of the special cases to be put into the Machine Table for the derived code generator (the LHSs of the productions), and for each of these, finding the best code sequence for the target machine (the RHSs of the productions). The former is performed by a heuristic algorithm we will refer to as Select, the latter by the machine-independent code generator we will refer to as Search. We first discuss the design of Search.

4.1 Tree Equivalence Axioms

The central formalism on which Search is based is a set of axiom schemas that specify semantic equivalences over computations; some examples of these are shown in Figure 2. The axioms express the classical arithmetic and boolean laws, as well as rules about programs and the model of instruction set processors. They will be used to specify the legal tree transformations (programs, instructions, and axioms are represented as trees) in the

Boolean axioms
not not E <=> E
E_1 and E_2 <=> not((not E_1) or (not E_2))
E_1 and E_2 <=> E_2 and E_1
E and E <=> E

Arithmetic axioms
E + 0 <=> E
-(-E) <=> E
-E <=> 0-E
E_1 * E_2 <=> E_2 * E_1
E shift 1 <=> E*2

Relational axioms
not ($E_1 \geq E_2$) <=> ($E_1 < E_2$)
($E_1 < E_2$) or ($E_1 = E_2$) <=> $E_1 \leq E_2$

Fetch/Store decomposition rules
$E_1(E_2)$ <=> S←E_2; E_1(S)
S_1←E <=> S_2←E; S_1←S_2

Side-effect compensation axioms
S; D←E <=> S if D is temporary state
S; D←E <=> Alloc(D); S if D is general purpose

Sequencing Semantics axioms
S_1 <=> S_1; PC←PC + n; S_2<space n>
if E then PC←PC + n; S<space n> <=> if not E then S
PC←E => goto E (unconditional jump)
goto L_1 <=> goto L_1-L_2 + PC; L_2: (relative jump)

Implementation rules
while E do S => L_1: if not E then goto L_2; S; goto L_1; L_2:
if E then S_1 else S_2 => if E then goto L_1; S_2; goto L_2; L_1: S_1; L_2:
if E then S => if (not E) then goto L; S; L:

Notation:
L: location in instruction store;
D: location in data store;
E: combinatorial tree;
S: statement (assignment or conditional) tree;

Figure 2 : Tree Equivalence Rules (Examples)

heuristic search for optimal code sequences.

The advantage to the use of the axioms and the formal search methods we are about to discuss is that these are almost entirely machine-independent, and thus only the machine description itself need be changed to generate different target code. (However, there are exceptions to this statement: the axioms for the machine's particular binary representation of integers must be used, and it may be necessary to add axioms to reflect new data types/operations provided by the machine.)

The arithmetic and boolean axioms are relatively straightforward: they include laws such as commutativity of AND and +, DeMorgan's law, the double-complement rule, and the idempotence of adding zero.

The remaining axioms (see figure) were developed specifically for this work, and require some explanation. Note, for example, the axiom labelled Fetch Decomposition:

$E_1(E_2)$ <=> $S \leftarrow E_2$; $E_1(S)$

This states that an expression E_1 with a sub-expression E_2 can be computed by first computing E_2 and storing the result in a location S, then replacing E_2 with S in the computation of E_1. This essentially says that *storage may be used for temporary results.* The companion axiom Store Decomposition is simply a special case in which E_1 is an assignment statement; this case is treated separately because of the way the search works.

Other axioms deal with *side effects.* If an instruction has more than one assertion, it may be possible to use it for a *subset* of its effects, and ignore or compensate for any undesired side effects on the processor state, depending on the type of storage base involved (see Section 2): *temporaries* such as condition codes may be ignored, *general-purpose* locations such as registers and primary memory may be used if allocated, and *reserved* locations such as the stack pointer and program counter must not be destroyed.

The remaining axioms are concerned with flow of control. Some define higher-level constructs in terms of low-level ones. For example,

if E then S <=> if not E then goto L; S; L:

describes how to implement an IF with a conditional jump. The other flow axioms define the semantics of the program counter and low-level control:

goto E <=> PC←E
PC←PC + n; S<space n> <=> <nil>
$PC \leftarrow L_1$ <=> $PC \leftarrow PC + L_1 - L_2$; L_2:

The first of these simply defines the program counter (PC); the second rule says that

incrementing the PC by n causes the next n units of code to have no effect; and the last allows the use of relative and absolute jumps interchangeably.

4.2 Search

Now consider the machine-independent code generation problem. We are given a machine M with instructions m_1, m_2, ... m_n, and a goal tree G (from the Select algorithm) for which we would like to generate code (in the machine language of M). That is, we would like to find a sequence of instruction tree instantiations that is semantically equivalent to the goal tree G. The axioms presented in the previous section define "equivalent": if a subtree matches one side of an axiom schema, the subtree may be replaced by the instantiation of the other side. In this way, the goal G can be successively transformed into other trees, until eventually we may arrive at a tree that is a sequence of instruction trees:

$$G \Rightarrow G' \Rightarrow G'' \Rightarrow \ldots \Rightarrow m_{i_1}; m_{i_2}; \ldots m_{i_k}.$$

Because more than one axiom may be applicable to a tree at any point, and we can test for the termination condition of a sequence of instructions, we have a classical search problem. That is, starting with G, we may use all applicable axioms to obtain a set of equivalent trees, recursively apply all applicable axioms to *those* trees, and so on, until we have one or more instruction sequences for the goal tree.

Applying this search algorithm literally is undesirable, as the search space is combinatorially large. Note that axioms may be applicable at more than one point in a goal tree, and more than one axiom may be applicable at each one of these points.

To reduce the size of the search space, we use some established methodology from the field of Artificial Intelligence. In fact, we use not one method, but several, allowing the strongest applicable method to be used for each kind of information. For this purpose, the axioms have been divided into three classes:

1. Transformations. These are the axioms concerned with arithmetic and boolean equivalence. Transformations will be used in conjunction with means-ends analysis in the search.

2. Decompositions. These axioms are normally those concerned with control constructs; they decompose constructs into sequences of other constructs, allowing the search to recursively proceed on subgoals. Decompositions will be used in conjunction with a heuristic search.

3. Compensations. These are the axioms concerned with side effects. No search at all will be associated with these axioms; it will be possible to use them in a pre-pass on the Machine Description.

Briefly, the basic Search algorithm, which is applied to each pattern (goal) tree determined by Select, is as follows:

S1. If the goal tree matches an instruction directly (or matches a pseudo-instruction with side effects, as described in the next section), we return that instruction as the code sequence for the goal tree.

S2. If there are any Decomposition axioms applicable to the goal tree, this search algorithm is applied recursively to try each new goal tree resulting from their application. If any of these recursive instantiations succeed in finding code sequences, the alternatives will be returned. Decomposition is used in the example presented subsequently..

S3. Means-ends analysis is applied to the goal tree as follows. A set of instructions are selected whose assertion trees are *semantically close* to the goal tree (see below). For each of these selected instructions, an attempt is made to recursively transform it so that it may be used for the goal tree, by applying the Transformation axiom(s) that reduces the difference between the two. The first example in the next section illustrates the use of this strategy.

All possible code sequences found for a given goal tree are returned by the Search routines, and the best of these is chosen by Select to be entered into the Machine Table. The "best" cost is determined by a user-supplied function of time and space; the time and space cost for instructions are known from the machine description.

A relatively simple heuristic measure of the *semantic closeness* used in S3 was found to work quite well. The measure is based on comparing the *primary operator* of the goal tree and a potential instruction. The primary operator of a tree T, po(T), is defined in terms of the top operator of T, op(T), as follows:

If op(T) is:	po(T) is:
a conditional	op(lhs(T)) [i.e., the conditional expression]
an assignment	op(rhs(T)) [i.e., the expression to compute]
anything else	op(T)

We now define a tree S to be semantically close to a tree T iff po(S) = po(T) or there exists an axiom $P_1 => P_2$ such that $po(P_1) = po(T)$ and $po(P_2) = po(S)$. The net effect of this heuristic measure is to select an instruction tree S if it performs an operation that is identical or arithmetically related to the goal tree. Some further performance improvements can be obtained by some minor refinements of this closeness measure; the reader is referred to Cattell[1978] for further details. Note that instructions/axioms can be *indexed* by their primary operator/operators. As a result the selection of semantically close instructions in S2, the selection of potentially applicable axioms in S3, and the selection of axioms that

reduce differences (defined as the difference in po's) in S2 can be performed in essentially constant time.

It is important to the Search procedure that the three classes of axioms deal with orthogonal types of TCOL constructs. Note in particular steps S2 and S3. The decomposition axioms used in S2 deal primarily with control constructs, while the transformation axioms used in S3 deal with arithmetic and boolean computations. The means-ends analysis used in S3 efficiently handles the large search spaced defined by the arithmetic/boolean axioms by selecting potential instructions to determine the choice of axioms to apply, while the much smaller search space defined by axioms on control constructs can be handled by the [almost] brute force search used in S2. The particular order here of S2 and S3 was not essential to success. The reverse order in fact has some advantages, as the set of axioms used in the brute force step can be expanded to include transformation axioms in the event that the means-ends analysis fails (though empirically this did not occur).

The Search procedure could potentially run forever: it is necessary to restrict the search both in the depth of recursion and in the breadth (the number of semantically close instructions tried in S3). The Select procedure described in Section 4.4 was designed to increase the depth and breadth if a searched failed, although a fixed search actually worked adequately.

4.3 Example

As an example of the use of transformations, consider the problem of loading the accumulator on a simple PDP-8-like machine (there is no load instruction to do this directly). The process can most easily be understood by following the steps of the actual search algorithms; a trace output from the implementation is shown in Figure 3.

Note: in the examples which follow, the comments in italics have been inserted to annotate the output; also, parts have been truncated with "..." for readability. A parenthesized LISP-like form is used for the TCOL trees. For example, (← %ACC (+ %ACC %MP)) means add a memory location (%MP) to the accumulator (%ACC). Parameters, e.g., "$1", are associated with nodes for later reference. Global parameters, e.g., "$$1", are parameters whose scope is over an entire search, as opposed to a single axiom or M-op; they are used to refer to temporaries needed in the code sequence. Access modes are preceded with "%" by convention; operand classes (e.g., Z in the example) are not. For complete and more extensive examples, see Cattell [1978].

```
Search: (← %ACC %MP)                          *Search is passed goal tree
Attempting M-op-match                         *no instructions match goal
Attempting Decompositions
Attempting Transformations                    *...attempt transforming Twos Comp. Add
 Feasible[1]: (← %ACC (+ %ACC $1:Z))          *(TAD) instruction to use for the goal
  Transform: (← %ACC %MP) => (← %ACC (+ %ACC $1:Z))
   Transform: %ACC => %ACC                    *LHS of the "←" matches
   Transform: %MP => (+ %ACC $1:Z)            *but RHS mismatches
    Applying $1 :: (+ 0 $1) to: %MP           *try this axiom to reduce difference
     Transform: (+ 0 %MP) => (+ %ACC $1:Z)    *now " + " node matches
      Transform: 0 => %ACC                    *but 0 still mismatches %ACC
       Applying Fetch Decomposition to: 0     *Acc←0 will fix this mismatch
        Search: (← %ACC 0)                    *and there is a CLRA Machine-op
        Attempting M-op-match                 *(M-op match explained later)
        M-op Match: (; (ALLOC $$2:Z) (EMIT[DCA 1 1 1] 3 $$2:Z)) *See text:      <--
        M-op Match: (EMIT[CLRA 3 1 1] 7 0 20) *there are 2 ways to clear Acc
      Transform: %MP => $1:Z                  *Z is an operand class, and matches %Mp
 Feasible[2]: (← %ACC (+ %ACC 1))             *try other feasible M-ops...
  Transform: (← %ACC %MP) => (← %ACC (+ %ACC 1))
   ...                                                *but no other solutions found
Best Sequence is:
  [Alloc $$1:%ACC]
  CLRA
  TAD %MP
-----------------------
```

Figure 3. The Machine Description has been input, and the top-level search routine is given the goal tree "(← %ACC %MP)", the TCOL representation of the problem of interest.

The first feasible instruction found is the two's complement add (TAD) instruction, whose tree representation is "(← %ACC (+ %ACC $1:Z))"; no other instruction matches the primary operator and also has the appropriate destination (%ACC). The system therefore attempts to transform

(← %ACC %MP) ⇒ (← %ACC (+ %ACC $1:Z)).

The %ACC part matches, but the RHSs mismatch. The program finds the transformation, $1 ⇒ (+ 0 $1), whose root operators match the mismatching subtrees, and it is applied to create the subproblem of transforming

(+ 0 %MP) ⇒ (+ %ACC $1:Z).

The "+"s now match, but the 0 and %ACC mismatch. Fetch decomposition is applied to make these match, by storing 0 into %ACC. Two instructions are found to do this (see next paragraph), the better one being CLRA (clear accumulator). The %MP matches the operand class Z, because Z is defined to allow either a direct or indirect memory reference. We have then completed the match. The search proceeds to try other feasible instructions, but no further code sequences are found. The best code sequence to load %ACC is therefore to clear %ACC and add %MP.

As an example of the use of compensations, note the line in the figure marked with "<--" on the right. The compensation rules tell the search that the accumulator can be cleared by using the deposit and clear instruction, if a memory location is allocated into which the accumulator may be stored, resulting in the sequence

```
[Alloc $$2:%MP]
DCA $$2:%MP
TAD %MP
```

to load the accumulator. This is of higher cost than the best sequence, however, so it is rejected (the reader may be curious as to the case where we already know Acc=0; this optimization is handled in a separate compiler phase (FINAL)).

As an example illustrating the use of *decompositions*, Figure 4 shows the generation of code for "If Acc=0 then Acc←1". Both the definition of IF and skip-decomposition get applied in this derivation, and two alternative code sequences are found depending on which is tried. The better sequence is to do a SKPNE (skip if accumulator non-zero) followed by SET1A (set accumulator to 1).

```
Search: (IF (EQL %ACC 0) (← %ACC 1))
Attempting M-op-match
Attempting Decompositions                    *first Search tries applying defn of IF
Applying (IF $1 $2) :: (; (-> (NOT $1) $3:%MP) $2 (LABEL $3:%MP))
*note: (-> A B) means "if A then goto B", LABEL means emit a label, and ";" means perform
*its arguments in sequence (these three arguments shortly become three subgoals...)
  Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0) *note logical simplifications must be
done
  Search: (; (-> (NEQ %ACC 0) $$1:%MP) (← %ACC 1) (LABEL $$1:%MP))
  Attempting M-op-match
  Attempting Decompositions                  *Search decomposes ";" node from IF
  Applying Sequence-Decomposition            *and treats each subnode as a subgoal
    Search: (-> (NEQ %ACC 0) $$1:%MP)        *1st subgoal (from IF-defn)
    Attempting M-op-match
    Attempting Decompositions
    Applying Skip-Decomposition              *decompose into skip and goto
      Search: (GOTO $$1:%MP)
      Attempting M-op-match
      Attempting Decompositions
      Applying (GOTO $1) :: (← %PC $1)       *the goto is recognized as a store into PC
        Search: (← %PC $$1:%MP)
        Attempting M-op-match
        M-op Match: (EMIT[JMP 2 1 1] 5 $$1:%MP)   *and a JMP is used to satisfy it
```

```
      Attempting Transformations
      Simplifying (NOT (NEQ %ACC 0)) to (EQL %ACC 0)
      Search: (-> (EQL %ACC 0) ( + %PC 1))        *the skip is recognized as SKPE
      Attempting M-op-match
      M-op Match: (EMIT[SKPE 3 1 1] 7 1 5)
        ...
    Breadth Limit Reached(6)
    Search: (← %ACC 1)                            *now do 2nd subgoal (THEN part of IF)
    Attempting M-op-match
    M-op Match: (EMIT[SET1A 3 1 1] 7 0 30)
    Search: (LABEL $$1:%MP)                       *3rd (and last) subgoal: generate label
*now we have one possible code sequence: "SKPE; JMP $$1; SET1A; $$1:".
*Search then tries applying Skip-decomposition to original goal instead of IF defn, i.e.,
*IF X THEN Y =>IF NOT X THEN PC←PC+1; Y (since Y is Acc←Acc+1, takes 1 word)
Applying Skip-Decomposition
  Search: (← %ACC 1)                              *first try THEN-part
  Attempting M-op-match
  M-op Match: (EMIT[SET1A 3 1 1] 7 0 30)          *OK, it takes 1 word
  Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0)
  Search: (-> (NEQ %ACC 0) ( + %PC 1))            *so try conditional skip of 1
  Attempting M-op-match
  M-op Match: (EMIT[SKPNE 3 1 1] 7 1 2)           *success. now have 2 seqs.
Attempting Transformations                        *keeps trying, but no more solutions..
    ...

----------------------
Nodes Examined:  15
Est. Seconds:    .062
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<<<<<
    SKPE
    JMP    $$1:%MP
    SET1A
$$1:
----------or----------
    SKPNE
    SET1A
>>>>>>>>>>>>>>>>>>>>>>>>
Best Sequence is:
    SKPNE
    SET1A
----------------------
```

Figure 4. Generation of code for "If Acc = 0 then Acc←1."

4.4 Select

As explained earlier, the machine-independent code generator (Search) is not used directly in the compiler for performance reasons. Rather, a level of indirection is introduced by running Search only on a preselected set of tree patterns to be inserted into the machine table used by the actual code generator. The preselection is done by the Select algorithm, basically as follows:

S1. Include all instructions in the Machine Table as is. Thus, if a program tree segment matches directly, the instruction will be selected.

S2. For each instruction with *multiple assertions* (i.e., for which two or more locations could be modified), for each of its actions for which the instruction may be used according to the Compensation Axioms, add a new *Pseudo-instruction* for that action alone. The right hand side of this new Pseudo-instruction production may include not only the instruction itself, but some compensation for the other side effects. For example, an increment-and-skip-if-zero instruction could be used for the increment action alone by appending a noop; or an instruction that stores into both memory and a register can be used for the latter action alone by preceding it with an allocation of a dummy memory location.

S3. Insure there is a production for A←B, for every pair of distinct access modes A and B such that A and B are "simple" references to locations of the same size. A "simple" reference is one in which the index into the storage base is a cardinal (as opposed to, say, indirect or relative addressing). If there is already such an entry from steps (1) and (2), no action is taken. Otherwise, Search is called to find the best code sequence for A←B, and a tree production is added whose pattern (LHS) is the A←B tree and whose RHS is the code sequence.

S4. Insure there is at least one production in the Machine Table for every TCOL operator. This is done similarly to the previous step, calling Search for every tree of the form "A←B op C", "A←op B", and "if A op B then goto C". It is unimportant what locations are represented by A, B, and C, since the code generator will make any moves needed to put the data in the required locations. For example, if logical "AND" did not exist as the primary operator of an instruction directly on the machine, a code sequence for it would be derived and the resulting production (LHS is the AND tree, RHS is the derived code sequence) added to the Machine Table. All derived productions are also indexed as if they were machine instructions, for use in further searches.

S5. Finally, add to the Machine Table the productions for control operators. These correspond to the axioms in Figure 2 which define WHILE-DO, IF-THEN-ELSE, etc., in terms of conditional and unconditional jumps.

This algorithm insures that the minimal code generator using the Machine Table will be able to generate code for all TCOL operators, and that if there exists a one-instruction code sequence for a subtree, it will find it (by S1). It does *not* guarantee that if the Search algorithm discussed in the previous section generates optimal code that the code generator using the Machine Table generated therefrom will do so, because the necessary special case combination of TCOL operators may not have been included in the Machine Table. Of course, special cases could be suggested by a human, but interestingly, such help was not found to be necessary: special cases more complex than single TCOL operators (in all contexts) were not needed for the machines tested, except for cases in which instructions

match directly (SELECT handles these in step S1) and cases which are already handled by the peephole optimization phase. (The second example in the previous section is one of the few in the latter case; peephole optimization is not even needed here if the skip decomposition axiom is instead used at compile time, to recognize the one-instruction "THEN"-parts of conditionals.) Nevertheless, an algorithm to guarantee an optimal set of special cases would be desirable (say, by some exhaustive analysis of possible pattern trees). This is an area for future research.

Note also that the search algorithm presented in the previous section does not guarantee optimal code, or any code at all for that matter, because the search may not be deep enough to discover the equivalence. Furthermore, even if we searched to an arbitrary depth, a code sequence might still not be found, because a necessary axiom to determine the sequence's equivalence to the goal tree may not be in Search's repertoire. The search failure implies that the axiom set is not complete; however, *no* set of axioms could form a basis for all equivalences true over all programs (Luckham et al [1970]). This indicates that the goal of this work, i.e., to take an arbitrary machine description and generate code, is unachievable! Fortunately, this result does not have great practical impact: the set of about 50 axioms was adequate for the machines tested.

5. Summary

This work has dealt with: (1) a model of instruction set processors, (2) a code generation algorithm in which machine-dependent information is separated into tabular form, and (3) a scheme for heuristic search for optimal code sequences, based on an axiomatization of tree equivalence.

5.1 Results

The results have been encouraging. The machine representation was general enough to deal with a variety of actual machine architectures (the IBM 360, PDP 10, PDP 11, Intel 8080, Motorola 6800, and PDP-8 are discussed in Cattell[1978]). The code generation algorithm satisfies the goals of tabularizing machine dependence and at the same time remaining flexible and fast enough for use in a production compiler. The last and perhaps most interesting result is that the formal approach of heuristic search for code sequences did not fail to find the optimal code sequences (for the machines tested, within the scope of the data types and operations covered by the axioms).

One might expect the code generator in the compiler to be relatively slow, since it involves a table-driven pattern-matching scheme. However, the prototype implementation on a PDP-10/KL10 is basically I/O bound, generating about 2000 instructions per second from the intermediate TCOL representation. It is written in Bliss (Wulf et al[1975]). The code

generator (and the entire PQCC compiler) cross-compiles rather than compiles. With some care in making the compiler code portable, of course, the compiler could be used to compile itself by the usual bootstrapping procedure to obtain a compiler running on an arbitrary machine. The code itself is quite compact, requiring only 1K 36-bit words, because all the machine-dependent information is in the tables. The tables require considerably more space (the amount being target-machine dependent, but order of 10K words). The prototype system described here is under redesign and integration into the PQCC compiler; the complete compiler will be necessary for an objective evaluation of the code quality, although small examples (see Cattell[1978]) led to code comparable to a hand-coded optimizing compiler.

The code generator generator is also surprisingly fast in comparison to previous results using formal methods (e.g., Newcomer [1975]). The derivations of code sequences for templates typically took about 0.1 seconds (KL10). The generation of the Machine Table itself took about 10 seconds for a typical machine (the PDP-11). The code generator generator uses 40K words plus 10 to 20K data; it is implemented in SAIL (Reiser et al [1976]).

The speed of the code generator generator is not greatly affected by either the number of axioms or the number of instructions on the target machine, because the indexing scheme allows the search routines to go almost directly to the applicable axiom (for a mismatch) or instruction (for a goal tree). Note that the axioms are machine-independent, so that it should only be necessary to add new axioms when a new domain is added, e.g., when TCOL is extended to include a new data type such as character strings.

The machine descriptions used in this work (MD in Figure 1) are in a relatively compact parenthesized text form. The PDP-11/20 description (a fairly basic machine, with no floating point or unusual operations), for example, is about 200 lines. An understanding of the components of the machine model (Section 2) is of course necessary to construct such a description, and a man-week or so is required to write and debug a typical one (the PDP-11). The machine model and description format are under further development in PQCC.

The success with formal methods is probably due to the choice of *representation*. In general, efficient algorithms were straightforward when the problems were expressed in the right way. This principle can be seen to apply in several areas of the work, including the machine representation, code generator representation, and the use of axioms and trees in the search for code sequences. In particular, some important representational issues were:

1. The use of a common notation, TCOL, to represent procedural semantics. Also important is the extensibility of TCOL with respect to new data types and operators (these then require additional axioms).

2. The restricted form of the instruction interpreter, reducing the selection of primitives to sequences of actions represented by input/output assertions.

3. Abstraction of orthogonal properties such as addressing and binary representation from the representation of the abstract operations themselves (the instructions).

Some of the techniques used in this work may be useful to other applications of machine descriptions. For example, automated hardware generation is conceptually analogous to code generation, as it involves decomposing a given algorithm into a set of given primitives (Leive [1977]).

The techniques used here may also be useful in the generation of microcode, although the latter calls for somewhat different algorithms. For example, it is typical rather than exceptional for micro-instructions to have more than one action (see S2 of Section 4.4), so the code generation algorithm might routinely perform a look-ahead in an attempt to use an instruction for several of its actions.

5.2 Limitations and Future Research

The model of machines used in this work is more general but more complex than that used in previous work, in an attempt to allow a wide range of architectures but enable good code generation. Considerably more work in machine formalization is needed, however; success with the current model suggests this research would be profitable. The model summarized in Section 2 did not deal adequately with description of machine data types (e.g., character strings), input/output, and special architectural features such as instruction lookahead, pipelines, and caches (which must be considered for good code). Extensions to the tree notation (TCOL) are needed in conjunction with additional axioms and the specification of data types in order to deal with more complex machine instructions and optimizations, specifically bit field extraction/modification, byte string manipulation, and machine operations tailored to high-level language operations.

A better template selection scheme than the one outlined in Section 4.4 almost surely exists. Alternatively, the performance obtained in the code generator generator suggests that at least some of the axioms could instead be applied at compile-time without unreasonable speed degradation.

A final but important area for future research, particularly for optimizing compilers, is the integration and generation of the other compiler phases: register allocation, the compiler-writer's virtual machine (translation of high-level operations like parameter passing and compound data structure access into primitive TCOL operations), and peephole optimization. Continued research building on the work described here is still in progress in the PQCC

project; a summary of the PQCC work can be found in Leverett et al[1979]. The interaction of the phases of the optimizing compiler are complex compared to any one phase, and centrally important to the generation of good code.

Machine Descriptions for Automatic Derivation of Code Generators

R. G. G. Cattell

Carnegie-Mellon University[1]

Abstract

The requirements of machine descriptions for applications involving automatic generation of software are discussed. The formalization of instruction set processors used in the author's work on automatic derivation of code generators is presented as an example of a machine model from the point of view of this class of application.

1. Automatic Generation of Machine-Dependent Software

Automatic generation of machine-dependent software is a relatively new and exotic application of machine description languages. Examples of applications in this class include automatic generation of assemblers, code generators, peephole optimizers, diagnostics, and run-time support routines. Until quite recently, there had been little success with this class of applications. There are probably two main reasons for more recent success: first, more appropriate representations of instruction set processors have been formalized for these applications; and second, new methodologies, particularly from artificial intelligence, have been applied to the area.

In particular, two groups have been active in this field. At Yale, John Wick[1975] developed a methodology to derive from an ISP description (Bell & Newell[1971]), with relatively little human interaction, an assembly language and an assembler to generate object code from

---

[1]The author's present address is Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.
This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract number F44620-74-C-0074 and is monitored by the Air Force Office of Scientific Research.

that language. His central algorithm was a symbolic execution of the instruction cycle of the machine, to detect which locations and which instruction fields are used in what ways. In the last year, even more ambitious applications have been built. Fraser[1977], also at Yale, designed a code generator driven off of an ISP description. Fraser's scheme is "human knowledge" based; his central algorithm consists of pattern matching common cases which the system "understands". The observation making this approach possible is that most current computer architectures are quite similar in design, and consequently it is possible to base the system on a manageable number of cases (Fraser presents evidence that the amount of new programming knowledge that must be added decreases as new machines of similar architecture are added). In contrast, this paper will discuss work testing the antithesis of this approach, using more formal methods. This approach runs the risk of combinatorial explosion in a search for code sequences; however, it has the potential for more generality and completeness.

Others at Carnegie-Mellon have also been active in this field (Barbacci[1974]). Oakley[1976] has a methodology for automatic generation of diagnostics. The first part of his algorithm consists of a symbolic execution of the instruction cycle, to derive assertions giving the instruction outputs and the control paths to be tested. The second half of the work is concerned with the synthesis of instruction sequences in the target machine language for automatically selected test cases. Hobbs[1976] is concerned with automatic generation of peephole optimizers. Hobb's work and the work described here are part of a project, PQCC (Production-Quality Compiler-Compiler), under the supervision of Prof. W. A. Wulf, to derive optimizing compilers.

2. Instruction Set Processors

It is a thesis of this paper that the various applications concerned with automatic generation of machine-dependent software have similar requirements with respect to the form of the machine description. Furthermore, these requirements are not necessarily similar to those of other applications. For example, for the purpose of driving a machine emulator, a machine description could essentially be any programming language, though normally with specialized features for the purpose of emulation. In contrast, to automatically generate software, we must make assumptions about the structure of the machine. In particular, we will assume such a machine consists of an *instruction set processor*, which iteratively retrieves commands from a *primary memory*, and (conditionally) changes the set of locations termed the *processor state* as specified by the command.

In this section we will describe the components of the machine model used in the author's work. This model is very similar to the view taken by the other work mentioned, so it probably can be taken as representative of this class of applications.

Five main kinds of information must be given by the machine description:

1. **Storage Bases**: The processor state. A storage base is an array of one or more *words*, each word consisting of a fixed number of bits. Each storage base is defined by a *word length*, an *array length*, and a *type*. The *type* is used to identify particular classes of locations. Specifically, we distinguish between: the program counter, the primary memory, reserved locations, temporary locations, and general-purpose locations. There must be exactly one storage base of each of the first two types. Temporary locations are those which it is permissable to destroy in generating code (e.g., condition codes). General-purpose locations (e.g., registers or primary memory) may be used for storing data, while reserved locations (e.g., a stack pointer) may not.

2. **Operand Addressing**: The instructions have particular properties with respect to the classes of storage base accesses they may make; they differ not only in the actual storage bases that may be accessed, but in *how* they are accessed, e.g., the computation of the address. We define an *Access Mode* for each distinct type of addressing on the machine; for example, indexed off a register, indirect through a memory location, or an immediate constant from an instruction field. The access mode is described by an expression (tree) which represents the access in terms of arithmetic operators and accesses to storage bases. As we will see shortly, an operand of an instruction (a location) is permitted to be any of a *set* of access modes depending on opcode, mode bits, etc. These sets of access modes are defined as *Operand Classes* (OCs), which specify a cost and values for the binary instruction fields for each access mode in the set. The separation of the operand addressing functions from the description of the instructions themselves in this way greatly reduces the number of instruction descriptions necessary for machines with a number of addressing modes, and also simplifies the recognition of addressable expressions in the code generator.

3. **Machine Operations** (M-ops): The M-ops represent the actual machine instructions. For each M-op we need to know cost (speed and time), formatting information (specifically, a *field-value list* and an *instruction format*, described shortly), and a set of *input/output assertions*. The input/output assertions specify the new values of processor state locations that are modified by the instruction, in terms of location values prior to instruction execution. Each assertion consists of:

a) an operand class (specifies a location)

b) an arithmetic function over constants and operand classes (locations)

c) a boolean function over operand classes

The location specified by (a) has the new value specified by (b) if the condition specified by (c) is satisfied (if no assertion condition for the location is satisfied, it is unchanged). In programming language terms, we may think of the assertions as "conditional assignment statements" whose variables are locations of the processor state. In fact, for the purpose of the code generation application, the assertions are represented as canonical parse trees whose nodes are operators such as "IF", "←", "+", "AND", etc. This representation of the assertion will be referred to as the *M-op tree*. The usefulness of this representation is that the M-op tree pattern will match intermediate-notation parse trees used by the code generator when the M-op performs the specified action.

4. **Data Types:** A machine data type consists of:

   a) A *length in bits*

   b) An *abstract domain* which the data type represents: for example integers, reals, or characters.

   c) An *encoding function* which, given an object in the abstract domain, gives a bit string that is the representation of the object; and a *decoding function* which is the functional inverse of the encoding function.

   Each arithmetic operator in an M-op tree specifies one of these data types as the type it operates upon. It is essential to notice that the operators have meaning only through their correspondence to data types. Also note that the "data" type is associated with the *operator*, not the *data* or locations as in most programming languages.

5. **Instruction Fields and Formats:** These specify the correspondence between the abstract machine operators/operands and the actual binary or assembly-level instructions. An *instruction field* consists of:

   a) A *bit position*, a non-negative integer giving the bit position relative to the first bit of the instruction word.

   b) A *length in bits*, a positive integer specifying the number of bits in this field.

   c) A *word specifier*, the word position of this field in the instruction (used for

variable-length instructions).

d) A *type*, which specifies the use of this field:

- type O indicates this field is part of the Opcode (determines M-op)
- type C indicates this field is used to control the operand selection (e.g., address mode bits)
- type D indicates this field is used only as data (e.g., an immediate constant)

An *instruction format* is an ordered list of instruction fields and operand classes. The instruction format is used in conjunction with the field-value list of a M-op and operand class to determine the binary instruction representation. Recall that the M-op input/output assertions specify the *effect* of an instruction when it is executed by the instruction interpreter. The field-value list, together with the instruction format, specify the *conditions* under which the instruction is executed, and the correspondence between instruction operands and binary fields. Specifically,

a) The instruction fields in the instruction format are asserted to have the values specified by the corresponding elements of the field-value list. For example, the constant "7" might be associated with the field "OpCode".

b) The *operand classes* in the instruction format *indirectly* assert values for the instruction fields. This process is somewhat complex. A M-op represents a *family* of actions, because the operand classes permit different access modes, and even for one particular access mode (e.g., indexed by a register), different actual addresses may be involved (e.g., the register number or memory location). To allow this field specification to be separated from the M-op descriptions, a *parameter* is associated with each operand class in the M-op input/output assertion. That is, the parameter and operand class occur in corresponding positions of the field-value list and instruction format. The field-value list and format for the *operand class* specify the actual field values: if a field is fixed for that access mode (e.g., addressing mode bit(s)), a constant is specified in the field-value list; if the field has a value dependent on the program tree (e.g., an address field), a parameter associated with a constant in the access mode tree is given.

As mentioned earlier, the machine representation presented here is somewhat different from a "procedural" machine description, which is the flavor of nearly all current machine description languages. However, it is possible to *derive* such a description from a procedural description. Certain human input is required to identify structures such as the

instruction interpretation process and the program counter. Then, by recognizing the relationships between the structures, other information can automatically be discovered, such as the instruction formats and primary memory. Finally, each instruction must be symbolically simulated, by calculating and simplifying the new location values computed by the M-op, to form the input/output assertions (see Oakley[1979]). The reason for the symbolic execution is to put the instruction descriptions in a canonical form, eliminating temporary results, etc.; if the form of these in the machine description is restricted, the simulation is not necessary.

An example of the syntactic representation of the components of an instruction set processor is given in the appendix to this paper, for a somewhat simplified version of the PDP-8. The correspondence to the abstract components is explained in the appendix.

3. Summary

This paper has described a model of instruction set processors from the point of view of automatic generation of software. A comparatively simple model was found that spans nearly all common machine architectures, and is demonstrably useful in the generation of code generators, described in the other papers in this collection. The model of machines proposed also suggests requirements on a machine description language to be used for automatic generation of software and other applications (Parker et al[1979]).

4. Appendix: Example Machine Description

The representation of the instruction set processors used as input to the code generator generator. This particular description is of a computer that is similar to the DEC PDP-8. The six parts of this description (Instruction Fields, Instruction Formats, Storage Bases (SBs), Access Modes (AMs), Operand Classes (OCs), and Machine-operations) are described in the text. For the former three, the items in the parenthesized lists correspond directly to the corresponding components in the text. For the latter three (AMs, OCs, M-ops), some explanation is required. Trees are represented in the form

(operator son_1 son_2 ... son_n).

Access modes consist of a mnemonic and a location description tree; the "<>" pseudo-operator describes a location in the form

(<> storage-base word-index bit-index bit-length).

The "#8" refers to a constant of length 8 bits. The OCs and M-ops are represented in the

form:

pattern tree :: (EMIT[mnemonic format# timecost spacecost] field-value-list)

where the pattern tree describes the input/output assertions for the instructions, or gives the access mode name for OCs. The mnemonic is optional.

```
{I-flds} [                                  {Instruction fields; e.g.,}
(OP 0 3 0 O)                                {"OP" starts at word/bit 0,}
(I.BIT 3 1 0 C)                             {is 3 bits, and type "O"}
(ADR 4 8 0 D)
(IO.BITS 4 8 0 D)
(UBITS 5 7 0 O)
(UCLASS 4 1 0 O) ]

{SBs} [                                     {Storage Bases; for example,}
(Mp 4096 12 M)                              {"Mp" is 4096 12-bit words,}
(PC 1 12 P)                                 {and is type "M"}
(ACC 1 12 G)
(IO.REG 1 8 R)
(L 1 12 R) ]

{AMs} [
%8:       $1:#8                             {8-bit constant}
%Mp:      (<> Mp $1:#8 0 12)                {access to memory}
%@Mp:     (<> MP (<> Mp $1:#8 0 12) 0 12)   {indirect access to memory}
%PC:      (<> PC 1 0 12)                    {access to program counter}
%ACC:     (<> ACC 1 0 12)                   {..to accumulator}
%L:       (<> L 1 0 12)                     {..to link register}
%IO.REG:(<> IO.REG 1 0 8) ]                 {..to input/output register}

{OCs} [

Y: (
%8 :: (EMIT[5 0 0] $1 0)
%Mp :: (EMIT[5 1 0] $1 1) )

Z: (
%Mp :: (EMIT[5 1 0] $1 0)
%@Mp :: (EMIT[5 2 0] $1 1) )

IO: (
%8 :: (EMIT[6 0 0] $1) ) ]

[
{I-fmts}
{FMT 1} (OP Z)                              {1-opnd format}
{FMT 2} (OP Y)                              {jump format}
{FMT 3} (OP UCLASS UBITS)                   {micro format}
{FMT 4} (OP IO)                             {IOT format}

{OC-fmts}
{FMT 5} (ADR I.BIT)                         {Y and Z}
{FMT 6} (IO.BITS) ]                         {IO}

{M-ops}[
```

```
(← %ACC (AND %ACC $1:Z)) ::                          (EMIT[AND 1 1 1] 0 $1)
(← %ACC ( + %ACC $1:Z)) ::                           (EMIT[TAD 1 1 1] 1 $1)
(; (← $1:Z ( + $1:Z 1))
  ( => (EQL $1:Z -1) (← %PC ( + %PC 1)))) ::         (EMIT[ISZ 1 1 1] 2 $1)
(; (← $1:Z %ACC) (← %ACC 0)) ::                      (EMIT[DCA 1 1 1] 3 $1)
(; (← %L %PC) (← %PC $1:Y)) ::                       (EMIT[JMS 2 1 1] 4 $1)
(← %PC $1:Y) ::                                      (EMIT[JMP 2 1 1] 5 $1)
(← %IO.REG IO) ::                                    (EMIT[IOT 4 1 1] 6 $1)
(← %ACC (NOT %ACC)) ::                               (EMIT[COMA 3 1 1] 7 0 40)
(← %ACC 0) ::                                        (EMIT[CLRA 3 1 1] 7 0 20)
(← %ACC ( + %ACC 1)) ::                              (EMIT[INCA 3 1 1] 7 0 10)
(← %ACC (- %ACC 1)) ::                               (EMIT[DECA 3 1 1] 7 0 4)
(← %ACC (↑ %ACC 1)) ::                               (EMIT[SLA 3 1 1] 7 0 1)
(NO.OP) ::                                           (EMIT[NOP 3 1 1] 7 0 0)
(← %ACC 1) ::                                        (EMIT[SET1A 3 1 1] 7 0 30)
(← %PC %L) ::                                        (EMIT[RTS 3 1 1] 7 1 40)
(← %PC %ACC) ::                                      (EMIT[JMPA 3 1 1] 7 1 20)
( => (EQL %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPE 3 1 1] 7 1 5)
( => (NEQ %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPNE 3 1 1] 7 1 2)
( => (LSS %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPL 3 1 1] 7 1 4)
( => (GTR %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPG 3 1 1] 7 1 1)
( => (LEQ %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPLE 3 1 1] 7 1 6)
( => (GEQ %ACC 0) (← %PC ( + %PC 1))) ::             (EMIT[SKPGE 3 1 1] 7 1 3) ]
```

Code Generation in a Machine-independent Compiler[1]

R. G. G. Cattell
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Joseph M. Newcomer, Bruce W. Leverett
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents some code generation issues in the context of the PQCC Production-Quality Compiler-Compiler project (Leverett et al[1979]). The approach taken is unusual in several ways. The machine-dependent information for selection of code sequences, register assignments, etc., has been separated throughout, in tabular form, from the machine-independent algorithms. This not only greatly simplifies the development of code generators for new machines or languages, but paves the way for automatic generation of these tables from formal machine descriptions such as ISP (Bell & Newell[1971]). A parse-tree-like internal program representation is used, facilitating the use of context and data dependency information about expressions. The code generation process has been broken into several phases. This leads to simplification and better understanding of the code generation process, and also allows important improvements in the quality of generated code. The algorithms for preliminary determination of addressing modes, allocation of registers and other locations, and the instruction selection case analysis are discussed. The algorithms described in the paper are being implemented and used in the PQCC compiler.

---

[1]This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract number F44620-74-C-0074 and is monitored by the Air Force Office of Scientific Research.

1. Introduction

There has been some interest in recent years in building machine-independent compilers: compilers in which information dependent on the target machine (the machine on which compiled programs are to run) has been separated in tabular form from the machine-independent algorithms. There has been progress toward *language*-independent compilers, particularly with respect to automatic parser generation. However, much less progress has been made on the more difficult problem of formalizing the *back* end of the compiler, that part which generates machine code from the compiler's intermediate representation of the program. It would be desirable to generate good quality code for a wide range of machines in this way. That is the topic of this paper.

Code generation will be discussed in the framework of the Production-Quality Compiler-Compiler (PQCC) project at Carnegie-Mellon University. Although one of our goals is code comparable to the best hand-written compilers, the PQCC work is not primarily concerned with optimization techniques themselves: to a large extent, the compiler technology has been taken from the Bliss-11 optimizing compiler Wulf et al[1975]. The research to date has concentrated upon:

1. formulating the compilation in such a way that the machine-dependent information is separated in tabular form, while retaining the excellent code quality exhibited by the original Bliss-11 compiler structure.

2. automating or simplifying the generation of the tables from a machine description.

The work described in this paper deals primarily with (1). A more complete discussion of formalization of machines and the automatic generation of code generators can be found in the first two papers in this collection.

Before we can construct a full PQCC system, we first must understand the model of the compilation process used by the target compiler, or PQC. A simplified picture of the compiler structure is shown in Figure 1; in this paper we will be dealing with the phases labelled DELAY, TNBIND, and CODE. A source language program is parsed by the LEXSYN phase into an intermediate parse-tree-like notation which we refer to as TCOL. The FLOW phase then does global data flow analysis to determine possible code transformations to reduce program size or time costs. The DELAY phase then performs a collection of optimizations and analyses, basically arriving at a guess at the "shape" of the eventual code to be generated. TNBIND then uses this guess (at where temporaries will be required) to perform allocation of registers and other locations on the target machine. The FINAL phase can then perform the case analysis to actually generate the machine code in a symbolic form. Finally, FINAL performs peephole optimizations and code motion optimizations resulting in the actual target machine code. These phase names are taken from the BLISS-11 compiler (Wulf et al. [1975]); we use them as convenient groupings of the optimizations. However, we have identified about three dozen distinct "phases", usually defined as single

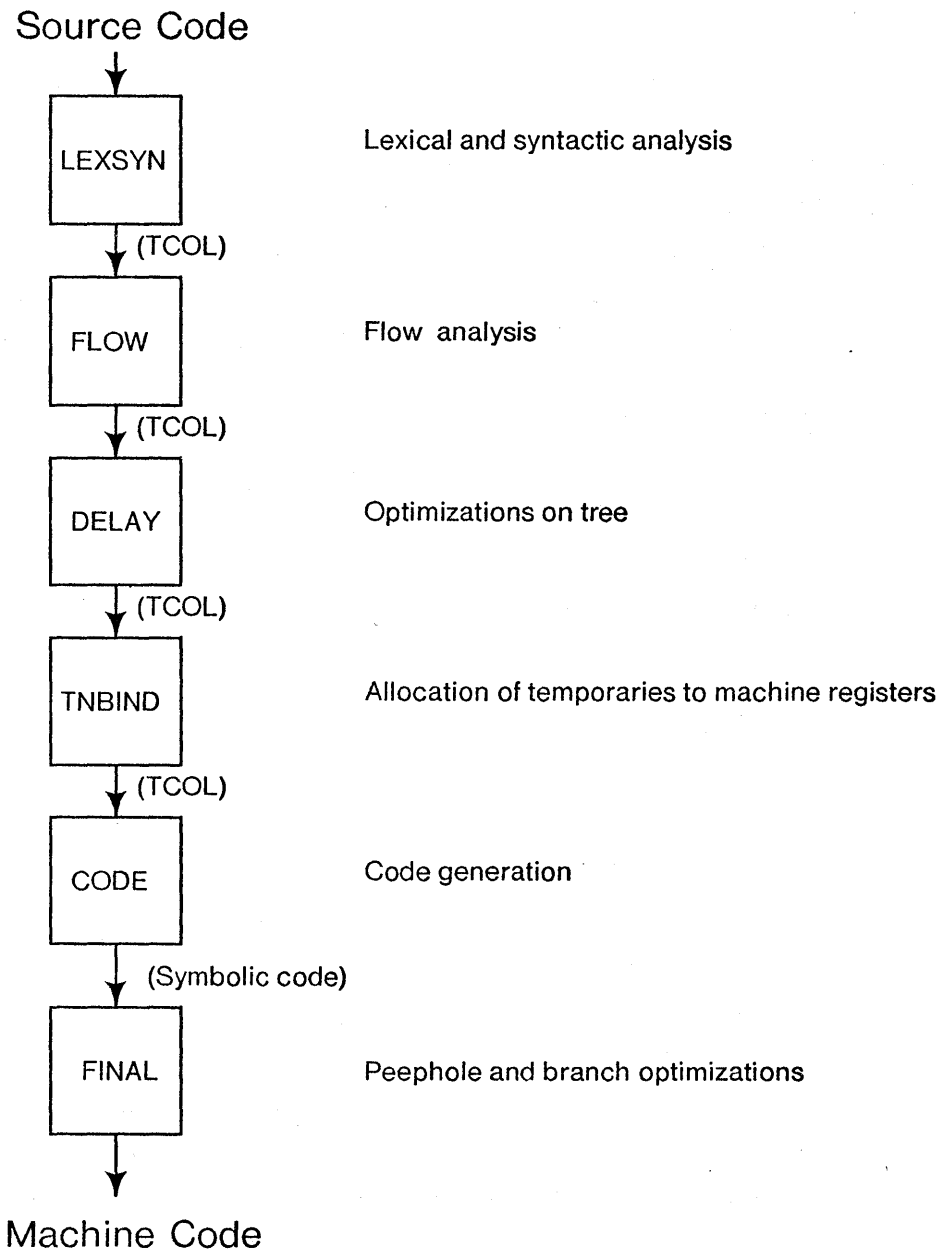

Figure 1. Structure of the Bliss-11 compiler, being used as basis for PQCC.

tree or graph walks which will constitute the actual compiler.

In the next section of this paper, enough of the machine formalization is given to understand the code generation problem. In Sections 3.1 - 3.5, the part of the DELAY phase concerned with guessing the best instructions and address modes for program tree segments is discussed. In Section 3.6, the register allocation algorithm is presented. In Sections 5.2 and 5.3, we discuss the basic code generation algorithm and its extensions to generate code using control flow optimizations, common sub-expressions, and other information. Finally, in Section 6 we summarize the results and discuss the current state of the prototype implementation of the machine-independent compiler.

2. Instruction Set Processor Formalization

We begin with an overview of our model of machines, or *instruction set processors*. We define an instruction set processor in terms of seven components: *machine operations* and *data types*, which specify the operations available on the machine; *storage bases*, *access modes (AMs)*, and *operand classes*, which specify the locations available on the machine and how they may accessed as operands; and *instruction fields and formats*, which specify the binary representation of instructions.

Associated with each instruction are a set of *input/output assertions*, which express the action of the instruction. An *output assertion* specifies the processor state after instruction execution as a function of the processor state prior to instruction execution. Note that since most processor state remains the same, we actually need to express only the state which has changed. Paired with each output assertion is an *input assertion* which specifies a conditional function of the processor state. The output assertion holds, i.e., the state has a new value, only if this input assertion is satisfied (the state is unchanged if the input assertion is not satisfied). The assertions are *represented* as *program tree patterns* which correspond to the action the instruction performs. For example, an ADD instruction might be represented as

$$A \leftarrow A + E$$

where A is an accumulator and E an operand in memory. The utility of representing the assertions as program trees is that we will be able to "match" these as pattern trees against program trees which an instruction could implement.

The actual locations of the processor state are the *Storage Bases* (SBs). The SBs may be simple locations of various sizes, such as an accumulator or condition code, or arrays of locations, such as general register sets or the primary memory.

There are typically several choices for operands of an instruction, corresponding to different modes of addressing on the machine: "indexed by a register", "indirect through a memory

location", "an accumulator", and so on. These will be referred to as *Access Modes* (AMs). The correspondence of these access modes to the instruction operands is specified by *Operand Classes*. Specifically, an operand class defines an operand position of an instruction: it specifies a set of access modes that are valid in that context. Any of the specified set of access modes may be used in fetching/storing the corresponding operand of the instruction. For example, an ADD instruction might require a general-purpose register as the operand receiving the result, and allow either an immediate constant or a memory location as the other operand (in the earlier example, A and E were operand classes).

As we proceed through the next sections, the utility of the three-level description (Storage Base, Access Mode, Operand Class) of a machine's addressing in reducing the size of the machine description and aiding in good code generation should become more apparent.

For some special purposes we find it useful to classify access modes into three categories: atomic, molecular, and compound. An atomic access mode is an immediate-constant type of argument of an instruction, i.e., a constant with some constraints on its possible range of values. A molecular access mode is a direct access to a storage base (the word/bit position(s) by which the storage base is indexed are atomic). In contrast, compound access modes (all other cases) include indirect and indexed accesses to the storage bases. Only molecular access modes are of interest for the purpose of this paper; we will call such an access mode a *storage class* because it represents a particular set of locations within a storage base.

Also associated with an operand class is information specifying, for each applicable access mode, the time and space costs and instruction bits format. Such additional information is also associated with each instruction description. However, the reader can ignore these details here, except to note that they allow us to evaluate costs of code sequences and to generate the actual binary representation of the abstract operations when required.

A detailed discussion of the machine formalization can be found in Cattell[1978] or the second paper in this collection. The preceding should be adequate to continue here. The central components of the formalization for the purposes of Sections 3 and 5 are the access modes and instruction assertions. Both of these are represented as pattern trees, and the main code generation process simply consists of traversing the source program tree, matching these patterns to determine the code sequences and locations to use for the current tree node.

3. Operations Preceding Code Generation

The phases between flow analysis and the code generator perform a number of machine-related optimizations, which will be sketched only briefly here. More complete explanations may be found in Wulf et al[1975], which explains the optimizations as done in the Bliss-11

compiler, and in Leverett et al[1979], which discusses the extensions done for the PQC. The discussions here must be brief, and we include only enough explanation to set the context in which the code generator actually operates.

One major difference between the PQC and a conventional compiler is that the "code generator" does not have to make any decisions about local or global register allocation, allocation of temporaries for results, determination of evaluation order, or any of the tasks conventionally thought of as "code generation". All of these functions are performed, of course, but *not* by the CODE phase that actually determines what code to emit.

3.1 Context Determination

Context determination determines the way in which the result of an operation is used. For example, a "+" node may be used to compute a result to store in a variable or to compute an address to be used for accessing data. If the target machine has an address size which is smaller than a full machine word (e.g., DEC PDP-10 or IBM S/370) it may also have instructions which operate only on addresses; in addition, it may be possible to take advantage of the implicit address calculation hardware of the machine, e.g., indexing, to perform the addition (e.g., using the "load address" instruction on a S/370 to add the values in two registers plus a small constant to produce a 24-bit result). We distinguish these cases as "real" and "address" contexts.

Another context is that used to determine program control flow. Consider a boolean expression "A < B". If this had appeared as the right hand side of an assignment statement, the bit pattern representing "true" or "false" would have to be developed and stored. But if it appears as the boolean part of a conditional, it may be necessary only to cause a conditional change in the flow of control. In this case, we designate the node as being in a "flow" context.

Finally, expressions may produce no result at all; this is important in an expression language such as Bliss where there is no syntactic distinction between, for example, a "conditional statement" and a "conditional expression".

The four classes of result we currently recognize are "void", "real", "address" and "flow". Note that the presence of common subexpression detection makes it possible for a given expression to exist in more than one (perhaps all) contexts.

3.2. Unary Complement/Target Path Determination

Unary complement operations are those which propagate the unary complement operators to higher tree nodes until they are subsumed in other operations or can move no further. A traditional example is to change "(-A * -B)" to "(A * B)" by propagating the minus signs

upward and cancelling them at the "*" node. However, in the PQC a unary complement operator may also change the operator (e.g., "+" to "-") of the parent node. When two temporary locations are involved this also can affect which one is optimal for developing the result (the "target path").

The machine-dependent information required by this phase is quite simple. For example, in the case of optimizing unary negation, it must know the cost of

- Moving a value from memory to a register
- Moving the negation of a value from memory to a register
- Negating a value in a register or memory
- Storing a value from a register to memory
- Storing the negation of a value from a register to memory

Given this cost information and some information on the ability to do memory-to-memory or register-to-memory arithmetic, a guess at the optimal target path will be made and the choice recorded in the program tree representation. Upon completion of this phase, the unary complement optimizations have been done and the desired target path has been determined.

3.3. Evaluation Order

This phase determines the evaluation order for arithmetic expressions in the tree. In the absence of common subexpressions there are known optimal algorithms for this determination; in the presence of common subexpressions or side effects of evaluating an operand, the problem becomes NP-complete and a simple heuristic is applied; details are given in Wulf et al[1975].

3.4. Flow Analysis

Preceding all of the phases described here is a global data-flow analysis phase. It does most of the classical optimizations which are basically source-to-source transformations, e.g., moving constant computations outside loops, moving common computations to the head or tail of forked control constructs, etc. These are discussed in more detail in Leverett et al[1979]. However, the flow analyzer does not actually make any program transformations; it only indicates which ones are feasible. After evaluation order has been decided, another phase will choose which of the feasible optimizations are in fact desirable. When this phase has completed its work, an execution order flow graph has been created for the tree. Although this is not used in the current implementation of the code generator, it will become important in future work, as described in Section 5.2.

3.5. Access Mode Determination

The Access Mode Determination phase searches the machine tables and determines which access modes may be used to evaluate a given node. For example, instructions which perform only address arithmetic (where the address has fewer bits than a full machine word) would not be selected, if a full result would have to be developed (e.g., 24 bit address arithmetic on a S/370). However, elaborate addressing modes such as pre-indexed indirection, post-indexed indirection, double-indexing with or without a constant offset (*such* as most IBM S/370 instructions with base + index + displacement), auto-increment or auto-decrement addressing, and indexing with a scaled index register (LLL S-1 (Hailpern and Hitson[1979]) or DEC VAX-11/780 [DEC 1977]) would all be taken into account. In addition, knowledge about the length of the operands and results can be taken into account; this is useful when the target architecture has the ability to encode small values in a few bits (e.g., the VAX) or perform operations on smaller fields (e.g., address arithmetic on a PDP-10 or S/370). All of this information is derived from the machine description tables. For complex architectures, this determination can be arbitrarily complex, particularly when the presence of common subexpressions is taken into account; examples are given in Leverett et al[1979].

Upon completion of this phase, a good guess at the optimum use of all access modes in the machine has been made. Note that resource restrictions (such as the availability of registers) may not make it possible to actually realize the goals set by this phase.

3.6. Temporary Name Assignment

The usual task of "register allocation" is partitioned in the PQC into several phases. The important concept here is that *all* allocation is handled in a uniform manner: variables and compiler-generated temporary locations are all subjected to the same processing.

The unit of allocation assigned to all of these is the *Temporary Name* or TN. It is first necessary to determine which nodes in the tree require TN's for their evaluation. This apparently requires knowledge of the code to be generated: to know what TN's are required, if any, for an addition, we must know not only what instructions are available for adding pairs of numbers, but also which one will be selected. Our method of addressing this problem of circularity is to have a fake code generation phase: the same traversal of the program tree that is done for code generation, described in later sections, is done in this earlier phase, but instead of generating instructions, this phase simply notes the requirements of the instructions for accumulators or other special types of storage. These requirements are embodied in TN's. If an arithmetic operator may be computed in either of two ways, depending on what kind of storage is chosen for its result, the TN which represents its result is marked so that it may be (later) assigned to either of those kinds of storage. The assumption of "infinite registers" is made: in the absence of knowledge made available by later phases, it is assumed that any type of storage necessary for a particular

operation will be available.

TN's are not only assigned to represent the results of expressions. Sometimes computation of an expression may require a completely temporary location, which is no longer needed when it is finished. For example, comparison of two quantities on some machines requires that one of them be moved to an accumulator, regardless of where the result of the comparison will be stored (if it will be stored anywhere). Also, for a procedure call, each parameter requires a TN; this TN is likely to be restricted, so that it may only be allocated to a particular location on the run-time stack.

Normally, each user-defined variable might also be assigned a single TN. However, the well-known strategy of keeping the value of a variable in a fast register during a loop is sometimes desirable; the recognition that this might be useful is noted by creating another TN to represent the variable within the loop. (Later processing determines whether this optimization is possible; if it is not, the loop TN is simply allocated to the same storage as the variable itself.) Also, since flow analysis information is available, it is possible to determine if the uses of a user-declared variable actually have separate lifetimes; for example:

```
begin
    integer  A;
        A← ...;
            ... ← A; /* last use of
                        A... */
    (other code);
        A← ...;     /* ...until
                        this
                        assignment */
            ... ← A;
end
```

In this case, two TN's might be assigned to the variable A with the result that A may actually be assigned to different locations during each of these disjoint uses!

This allocation of TN's requires a low-level knowledge of the machine; the storage bases and the set of interesting storage classes must be defined for the architecture. A storage class is a particular set of locations within a storage base: for example, the set of all even registers or the set of odd-even register pairs are storage classes which may be of particular interest.

The AMD and TN-assignment phases produce cost information which indicates which storage classes may be used to hold a TN, and the additional cost incurred if a less-desirable storage class must be used. This information is used by the TN packing phase, which is completely machine-independent, to determine the "most profitable" allocation of TN's to storage classes. Although in general this is an example of an NP-complete problem,

a number of approximate algorithms have been suggested in the literature; those which have been used in this style of compiler are discussed in Wulf et al[1975] and Johnsson[1975]. We intend to investigate still more.

At the completion of this phase, the tree contains nearly all of the information classically associated with the problem of "code generation"; it is only necessary now to generate the instructions to perform the computation. We describe this case analysis, which produces as output a symbolic representation of the object code sequence, in Section 5. Before proceeding to this, however, we briefly describe the FINAL phase which follows the case analysis.

4. Operations Following Code Generation

The output of the code generator goes to a final optimization pass which operates on the object code. All of the classical "peephole" optimizations (McKeeman[1965]) are performed here, as well as other optimizations that we touch on briefly in this section. Object code optimizations are desirable even in a compiler with a good-quality code generator, because it is easier to perform these operations after the actual code placement is known. This is because certain code adjacencies occur which are the result of evaluating widely separated tree nodes.

We include this brief discussion of FINAL here because peephole optimizations are frequently thought of as "code generation" activities. In the PQC, code generation, the main topic of this paper, is a separate activity, and does not attempt to perform any local or peephole optimizations.

FINAL performs some optimizations which are also done, generally on a more global scale, by earlier phases of the compiler. Examples are redundant store/load elimination, constant folding (e.g., two instructions which add immediate constants to the same location may be collapsed into one), and dead code elimination. Other optimizations are done which are only relevant to FINAL. An example of one of these, which could be done in the code generator but is much more simple if left to the separate phase, is jump chaining: a jump instruction whose destination is another jump instruction is changed, so that its destination is the ultimate destination.

One feature of FINAL which is important on machines which have both relative and absolute transfer instructions deals with resolving which form of instruction to use. Typical machines are the PDP-10, which has both skip and jump instructions, the PDP-11, which has relative branch (possibly conditional) and absolute jump (unconditional only) instructions, and the 360, which may require loading a base register before doing a control transfer. For the PDP-10 and PDP-11, FINAL will reverse the sense of a test and provide the appropriate conditional when better code may be produced, e.g.,

```
        CAME  r,m                       CAMN r,m
        JRST  label          ⇒          ADD  x,y
        ADD   x,y
label:
```

It is important that FINAL be able to perform these optimizations in a machine-independent manner; although the algorithms are fixed, the particular characteristics of the machine must be described in a table. In FINAL, the table is organized as a set of production rules. For the PDP-10, only thirty such rules are required to obtain optimizations of the quality of BLISS-11.

FINAL performs other optimizations, which are not relevant to this paper; for a summary of them, see Leverett et al[1979].

5. Code Generation

The code generation discussion has three sections: the abstract idea of code generation in our compiler model (Section 5.1); the code generation algorithm itself (Section 5.2); and the extension of this algorithm to use information from the register allocation and flow analysis phases of our compiler (Section 5.3).

5.1 Abstract Code Generation

It is easier to understand our view of code generation if we first present a model of how we think of it. This model is unrealistic in that no "real" compiler would actually do means-ends analysis and goal-directed search during the compilation process. In our implementation, we perform as much of this search as possible during the *generation* of the code generator.

Given a machine description, what the code generator attempts to do is locate, for each operator in the parse tree, a set of instructions which could feasibly evaluate the operator. This involves heuristic search techniques; there may be an instruction that could evaluate the operator if only both of its operands met some criterion (e.g., they were in registers); if they do not, goal-directed search is used to attempt to transform the actual situation into the desired situation.

The goal-directed search involves the use of a set of rules for transforming trees into equivalent trees. These include simple boolean and arithmetic axioms such as

$$\neg \neg E \equiv E$$

$$E + 0 \equiv E$$

$$E_1 * E_2 \equiv E_2 * E_1$$

In addition, there are rules which deal with abstract machine behavior. The *fetch decomposition* rule is expressed as:

$$E_1(E_2) \equiv S \leftarrow E_2;\quad E_1(S)$$

This rule essentially says that storage locations on the machine can be used to hold intermediate results: an expression E_1 with a subexpression E_2 can be computed by first computing E_2 in a location S, and then replacing E_2 with S in the computation of E_1.

Sequencing rules describe the flow of control in a machine with a program counter; in particular, the conventional automatic incrementation of a program counter (PC) is assumed in our model of machines. Some examples of such rules are the following:

goto E ≡ PC ← E

PC←PC + n; S<space n> ≡ <nil>

goto L_1 ≡ goto L_1-L_2 + PC; L_2:

The first rule states that a *goto* is the same as an assignment to the PC. In the machine description, the output assertion for an absolute jump instruction is expressed as an assignment to the PC, so this rule relates a *goto* to a jump. The next rule describes *skip* instructions, i.e., that a block of code of length *n*, expressed as "S<space n>", will be skipped if *n* is added to the PC. The final rule indicates that absolute and relative control transfers may be used interchangeably.

The complete set of rules and their use is described in Cattell[1978].

In practice, it is not reasonable to perform the heuristic search using these rules during the actual compilation process. Therefore, we use a program which performs a great deal of this search to produce a set of tables which involve little or no search strategy to locate the desired instruction sequence. This program is the code-generator generator, or CGG. It is described in detail in Cattell[1978] and the first paper in this collection.

CGG contains heuristics for selecting "interesting" code generation cases (pattern trees) the code generator is likely to encounter. If no instruction in the target machine instruction set exists to solve that case, the search techniques are applied to generate one or more sequences of instructions that will accomplish the task. If more than one alternative is found, the lowest-cost alternative is retained. As a result of these searches by CGG, the only rule which is applied at compile-time at present is fetch decomposition, which is essential to register allocation and composition of the patterns (this is discussed further in the next section).

It is worth observing here that the CGG strategy does not guarantee that an optimal solution,

or indeed any solution, can be found. For example, the depth of the search may hit its limit just before an instruction or instruction sequence is found. It may also be the case that an axiom necessary to determine the code sequence's equivalence to the goal tree may not be in the axiom repertoire. The general case of program equivalence is unsolvable because *no* set of axioms can express the equivalences that are true over all program trees. However, this theoretical result has little practical impact; for "real" machines and "real" programs, a small set of axioms and limited search depth seems to suffice.

5.2. The Code Generator

5.2.1. Overview

The code generator uses the output of CGG as its database of patterns and corresponding code sequences. Its basic operation is to retrieve the set of all patterns which it hypothesizes will apply to the parse tree, and find among these the lowest-cost instruction sequence which actually does apply. In the case where the leaves of the goal tree do not match the leaves of the pattern (e.g., contain an expression where the pattern expects a simple storage unit), fetch/store decomposition rules are applied to obtain a simple storage unit by generating code for the subtree.

Because of the structure of the PQC, many conventional "code generation" tasks simply do not exist. For example, all "register" allocation, which is actually TN allocation, has already been done. The code generator therefore cares little about the actual storage locations used. Since the TN assignment and allocation phase has already examined the machine description for feasible storage classes, it is certain that at least one pattern will match. As a result, the code generation algorithm need not deal with obtaining storage locations in feasible storage classes when none of the patterns match. If it is necessary to move data from one location to another (because the TNs have been assigned to different storage classes or different locations within the same storage class) then this will be done; the cost of this transfer has already been taken into account as part of the cost of the particular TN assignments.

5.2.2. Pattern Matching

Pattern retrieval in the current code generator is very simple. Patterns are indexed within each context (real, flow, void, address) by the primary (root) operator of the pattern tree. The primary operator may be any arithmetic, relational, or boolean operator. The patterns are ordered within each retrieval set by a cost metric (currently code size) in order of decreasing "profit". We treat "profit" as different from "cost", since profit indicates the

value of choosing one pattern over another. Details of cost and profit determination will be given in Section 5.3.

Rather than simply indexing the patterns produced by CGG, one could derive a more efficient representation for a table-driven code generator from them (Glanville[1978]). This essentially avoids re-matching similar parts of a new pattern when a pattern fails to match. We have not as yet found it necessary to use such a scheme to achieve reasonable performance in the prototype code generator.

As a result of the pattern indexing scheme, the code generator will attempt to use the highest-profit pattern first when presented a target tree. If the match is successful, the associated code sequence will be attached to the portion of the tree which matched the pattern. Otherwise, lower-profit patterns are attempted until one matches. There will eventually be a pattern which matches, since CGG generates at least one pattern for every operator. If the repertoire of axioms or depth of search were insufficient to find a code sequence for some operator, the user of CGG would have been requested to extend its knowledge. This might involve extensions to the axiom system, extensions to its case-generator, extensions to the machine description (adding new instructions, for example), or adding new "virtual instructions" (treated as axioms), e.g., defining the floating point arithmetic operations as generating subroutine calls.

5.2.3. Reverse Code Generation

A novel feature of the code generator is that it generates code in *reverse* execution order. A conventional code generator would typically do a left-right, depth-first, recursive tree walk to the leaf nodes, generate code for each leaf if necessary, back up to the operator node and generate code for the operation, and repeat until code had been generated for the entire tree. Because our code generator attempts to exploit the addressing modes of the machine and as much of the instruction set as possible, it starts at the root of the tree and finds the highest-profit pattern for that operator. This pattern will usually be the pattern that spans the largest set of nodes in the tree. By repeating this process for each of the tree nodes which do not match the leaves of the pattern (applying fetch/store decomposition), the minimum number of patterns will normally be used to match the tree. It is not possible, without exhaustive enumeration, to guarantee the actual *lowest* cost code sequence Ripken[1977]. However, this heuristic technique will produce a close approximation to it, and frequently the actual lowest cost sequence, without the computation required for exhaustive search.

In our compiler this matching task is complicated by the presence of common subexpressions (cse's); a pattern cannot cross a cse boundary. The special cases of cse-creation and cse-last-use must also be handled. However, the fact that the destination of the result of a cse is determined by TN assignment, it is simple to generate code to use a

cse result even though the code to compute that result has not yet been generated.

The output of the code generator is a doubly-linked list of instruction sequences; the tree structure along with its decorations (cse links, flow graph, etc.) is discarded.

5.3. Use of Flow Analysis and Temporary Allocation Information

We now discuss our design to deal with program flow optimizations and allocation of temporaries in the algorithm discussed in the previous section. The implementation of these interactions between phases has not been completed at the time of this writing.

In the model shown in Figure 1, the phase TNBIND does the assignment of TNs to storage locations. The component of TNBIND which does this is a phase called PACK, which is attempting to solve a 2-1/2 dimensional bin-packing problem, a problem known to be NP-complete. Because PACK has more global knowledge of the utilization of storage locations, it may choose assignments which violate the "infinite registers" assumption used by DELAY for its estimation of code shape (because, of course, this assumption is false in any real machine). In addition, because it implements a heuristic solution with limited backtracking, it may not be able to satisfy particular desires for TN assignment (note that a "desire" is a weaker constraint than a "requirement"). The result of this global allocation is that the code generator may find that no existing template will match the existing tree, and will have to bring the operands into conformance with a given template.

This does not pose any particular problems in the conceptualization of the model or its implementation. The search required to bring operands into conformance is trivial; it is fetch decomposition. The cost of doing this fetch decomposition can be known to PACK, and thus taken into account in the global cost analysis which was used to select the actual assignments of storage locations. If, in addition, it is necessary to save an intermediate result (the case known as "register spill" in most compilers), this cost would also have been accounted for. Because of this accountability, the assignment that has been made of TNs to locations is still as good as the heuristic solution will permit.

A particularly valuable aspect of this strategy is that it now is nearly always possible to generate code for a specific target tree, independent of the actual locations assigned to the operands. Backup to a higher level of search (with the associated overhead this entails) is no longer necessary.

The specific design of the cost/profit determination for alternate code sequences is still under design. The actual cost of a pattern is the base cost of the instruction(s) associated with it, plus the additional cost of using particular access modes (e.g., needing a full 16-bit address operand on a PDP-11), plus the cost of fetch decomposition if the operands do not satisfy the pattern, plus the cost of register spill and restore if required. However, the profit is a function of how many nodes of the target tree are instantiated by the pattern.

The code generator we've described does a reverse-execution treewalk. However, because of the presence of the flow graph structure, the tree itself must to be distorted to make the treewalk equivalent to the execution order. This can be avoided simply by following the flow graph itself (a linked list of program tree nodes) in reverse order rather than using the tree structure.

In this scheme, we use *both* the parse tree and the flow graph representations. The flow graph is used to select, in turn, each candidate node from the target tree. If code must be generated for it, we use the tree shape in the pattern match. However, unlike the current implementation which then descends to the nodes in the tree which have matched leaves of the pattern and proceeds to generate code for these, the graph-directed code generator returns control to the graph-walk algorithm after a match, and a new node is selected.

We also plan to experiment with a modification to the code generation algorithm we refer to as "look-ahead" code generation. Many machines include instructions which perform multiple actions; for example, to subtract one from a register and branch if the result is zero. Such instructions are indexed by CGG in the code generation tables for each of the potential sub-actions (e.g., subtraction, and branching). When the code generator discovers that a multiple-action instruction is applicable to the current program tree node (e.g., a subtraction), it can "look ahead" in the program flow graph to see if the next operation can also be subsumed by one of the instruction's actions (e.g., is it followed by a branch on the result?). This involves negligible increase in the compilation time, but only works if the subactions are adjacent in the program.

6. Summary

This paper has presented machine-independent code generation algorithms for code generation in a production-quality compiler. A formal model of machines has been designed to allow definition of the target machine separate from the machine-independent code generation algorithms themselves. Several phases precede the actual code generation case analysis, "decorating" the program tree with information about context, program flow, use of addressing modes, and allocation of temporaries. The program tree is then matched against code generation templates taking this information into account to produce the target machine code. The algorithms are being tested in the current prototype compiler. At the time of this writing, we unfortunately do not have statistics comparing target code size and speed to good hand-coded compilers on non-trivial programs. The PQCC group plans to do this for several conventional machines (PDP-10, IBM-360, PDP-11, Intel 8080, ...) and languages (Bliss, Pascal, ...), experimenting with the algorithms presented here as well as other optimization techniques.

Bibliography

Barbacci, M., Barnes, G., Cattell, R., and Siewiorek, D. ISPS Reference Manual, CMU Computer Science technical report, 1978.

Barbacci, M. and Siewiorek, D. Some Aspects of the Symbolic Manipulation of Computer Descriptions, CMU Computer Science technical report, 1974.

Barbacci, M. and Siewiorek, D. The CMU RT-CAD System: An Innovative Approach to Computer Aided Design, CMU Computer Science Review, 1974-1975.

Bell, C. G. and Newell, A. Computer Structures: Readings and Examples, McGraw-Hill, 1971.

Carter, A. F., Harrison, J., Loewner, W., Tapscott, R., Trevillyan, L., and Wegman, M. The Experimental Compiling Systems Project, *IBM Research Report*, IBM Yorktown, 1977.

Cattell, R. G. A Survey and Critique of Some Models of Code Generation, CMU Computer Science technical report, 1977.

Cattell, R. G. Formalization and Automatic Derivation of Code Generators, Ph.D. dissertation, TR 78-115, Computer Science, Carnegie-Mellon University, 1978.

Cattell, R. G. Using Machine Descriptions for Automatic Generation of Code Generators. In *Proceedings of the 3rd Jerusalem Conference on Information Technology*, pp. 503-507. North-Holland, 1978a. The second paper in this collection is a revised version of this paper.

Cattell, R. G., Newcomer, J. M., Leverett, B. W. Code Generation in a Machine-independent Compiler, *Proceedings SIGPLAN Symposium on Compiler Construction,* Boulder, Colorado, (August 1979). The third paper in this collection is a revised version of this paper.

Cattell, R. G. Automatic Derivation of Code Generators from Machine Descriptions. To appear, ACM Transactions on Programming Languages and Systems, 1980. The first paper in this collection is a revised version of this paper.

Digital Equipment Corporation
VAX-11/780 Architecture Handbook, 1970.

Donegan, M. K. An Approach to the Automatic Generation of Code Generators, Ph.D. dissertation, Computer Science & Engineering, Rice University, 1973.

Elson, M. and Rake, S. T. Code-generation Technique for Large-Language Compilers, *IBM Systems Journal* **9**, 3 (1970), pp. 166-188.

Feldman, J. A Formal Semantics for Computer-oriented Languages, Ph.D. dissertation, Computer Science, Carnegie-Mellon University, 1964.

Feldman, J. and Gries, D. Translator Writing Systems, *CACM,* **11**, 2 (February 1968), pp. 77-113.

Fraser, C. W. Automatic Generation of Code Generators, Ph.D. dissertation, Computer Science, Yale University, 1977.

Glanville, R. S. and Graham, S. L. A New Method for Compiler Code Generation, *Proceedings of the Fifth Conference on Principles of Programming Languages*, pp. 231-240, SIGPLAN-SIGACT, (January 1978).

Glanville, R. S. A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers, Ph.D. Dissertation, TR UCB-CS-78-01, Computer Science, University of California at Berkeley, (December 1977).

Hailpern B. T. and Hitson, B. L. S-1 Architecture Manual. Technical Report No. 161, STAN-CS-79-715, Stanford University, Department of Electrical Engineering, (January 1979).

Hobbs, S. Object Code Optimization, thesis proposal, Computer Science, Carnegie-Mellon University, 1976.

Johnson, S. C. A Portable Compiler: Theory and Practice, *Proceedings of the Fifth Conference on Principles of Programming Languages*, pp. 97-104, (January 1978).

Johnsson, R. K. An Approach to Global Register Allocation, Ph.D. thesis, Carnegie-Mellon University, (December 1975).

Leive, G. The Binding of Modules to Abstract Digital Hardware Descriptions, thesis proposal, Electrical Engineering, Carnegie-Mellon University, 1977.

Leverett, B., Cattell, R., Hobbs, S., Newcomer, J., Reiner, A., Schatz, B., and Wulf, W. An Overview of the Production Quality Compiler-Compiler Project, TR-79-105, Computer Science, Carnegie-Mellon University, February 1979 (condensation to appear in IEEE Computer).

Luckham, D. C., Park, D. M. R., and Paterson, M. S. On Formalized Computer Programs, *Journal of Computer and System Sciences* **4**, 3 (1970), pp. 220-249.

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

McKeeman, W. M., Horning, J. J., and Wortman, D. B. A Compiler Generator, Prentice Hall, 1970.

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

Newcomer, J. M. Machine Independent Generation of Optimal Local Code, Ph.D. dissertation, Computer Science, Carnegie-Mellon University, 1975.

Oakley, J. D. Automatic Generation of Diagnostics from ISP, thesis proposal, Computer Science, Carnegie-Mellon University, (December 1976).

Oakley, J. D. Symbolic Execution of Formal Machine Descriptions, Ph.D. dissertation, Computer Science, Carnegie-Mellon University, (April 1979).

Parker, A. C., Thomas, E. E., Crocker, S., Cattell, R. G. G. ISPS: A Retrospective View, in

Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications, Palo Alto, Ca, 1979.

Reiser, J., *et al.* SAIL, Stanford Artificial Intelligence Lab Memo AIM-289, Computer Science, Stanford University, 1976.

Ripken, K. Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen. Ph.D. thesis, Technische Universitat Munchen, (July 1977). Reviewed in English by Bert Speelpenning, *A Review of Ripken's Thesis* (Unpublished).

Samet, H. Automatically Proving the Correctness of Translations involving Optimized Code, Ph.D. dissertation, Computer Science, Stanford University, 1975.

Simoneaux, D. C. High-Level Language Compiling for User-Defineable Architectures, Ph.D. dissertation, Electrical Engineering, Naval Postgraduate School, 1975.

Snyder, A. A Portable Compiler for the Language C, TR-149, Project MAC, Massachusetts Institute of Technology, 1975.

Weingart, S. W. An Efficient and Systematic Method of Compiler Code Generation, Ph.D. dissertation, Computer Science, Yale University, 1973.

White, J. R. JOSSLE: A Language for Specifying and Structuring the Semantic Phase of Translators, Ph.D. dissertation, University of California at Santa Barbara, 1973.

Wick, J. D. Automatic Generation of Assemblers, Ph.D. dissertation, Computer Science, Yale University, 1975.

Wilcox, T. R. Generating Machine Code for High-Level Programming Languages, Ph.D. dissertation, Computer Science, Cornell University, 1971.

Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C. The Design of an Optimizing Compiler, American Elsevier, 1975.

Young, R. The Coder: A Program Module for Code Generation in High-Level Language Compilers, M.S. dissertation, Computer Science, University of Illinois, 1974.

XEROX

XEROX

Code Generation and Machine Descriptions

By R. G. G. Cattell

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

 Printed in U.S.A.

CSL-79-8