Retargetable Compiler Code Generation

MAHADEVAN GANAPATHI

Computer Systems Laboratory, Stanford University, Stanford, California 94305

CHARLES N. FISCHER

Computer Sciences Department, University of Wisconsin—Madison, Madison, Wisconsin 53706

AND

JOHN L. HENNESSY

Computer Systems Laboratory, Stanford University, Stanford, California 94305

A classification of automated retargetable code generation techniques and a survey of the work on these techniques is presented. Retargetable code generation research is classified into three categories: interpretive code generation, pattern-matched code generation, and table-driven code generation. Interpretive code generation approaches generate code for a virtual machine and then expand into real target code. Pattern-matched code generation approaches separate the machine description from the code generation algorithm. Table-driven code generation approaches employ a formal machine description and use a code-generator generator to produce code generators automatically. An analysis of these techniques and a critique of automatic code generation algorithms are presented.

Categories and Subject Descriptors: D.3.4 **[Programming Languages]**: Processors—*code generation*; *compilers*; *optimization*

General Terms: None

Additional Key Words and Phrases. Code generator generator, compiler compiler, intermediate representation, machine-dependent optimization, machine description

INTRODUCTION

Code generation is a collective term for the different synthesis phases of compilation. First, language-dependent aspects such as name resolution, operator identification, and type rules are analyzed, and an intermediate form of a program is produced. This intermediate form is then input to a code generator, which, in turn, produces code for the target machine. Code may be assembly code, object code in linker format, or absolute binary machine code.

The problems associated with synthesizing code are

(1) evaluation order of operands and expressions;
(2) register allocation, assignment and management;
(3) storage allocation, binding and run-time access;
(4) context switching;
(5) instruction selection;
(6) machine-dependent optimizations.

Let us consider each of these areas in more detail.

On single sequential processor architectures, the evaluation order of expressions

CONTENTS

and operands can minimize the total number of simultaneous intermediate results. Sometimes this order is fixed by the semantics of the source language. When it is not, the code generator may evaluate expressions in an order that reduces the cost of evaluation. The issue of evaluation order is complicated by differing sizes of operands and their possible locations, and the presence of multiple uses of operands, called common subexpressions.

Register architectures provide the opportunity to improve target code quality through optimization of register allocation, assignment, and management strategies. An estimation of variables that should be placed in registers is based on frequency analysis of variables. Intermediate values are ranked by a heuristic weighting formula [WULF75]. The most "expensive" values, that is, those that will cost the most to access if resident in main memory, are assigned to registers. Simultaneous register use is estimated by subexpression analysis [JOHN78]. Whenever the simultaneous use is very large, values are assigned to memory. Register spills may be needed if the instruction-selection phase fails to find as good an instruction sequence as the labeling algorithm assumes. In the PL.8 compiler [AUSL82], a graph of value lifetimes is maintained [CHAI82]. Register assignment is performed by a coloring algorithm. All these preestimation methods decide in advance which intermediate values are to be kept in registers. All except the graph coloring algorithm separate register allocation from register assignment.

On-the-fly register allocation and assignment schemes track register contents during code generation to utilize them efficiently [AMMA75]. They attempt to reuse register contents whenever reuse is cheaper than recomputation. Typical tracking information includes last use of register, copy in another register or memory, distance to next use, currently unused registers, active memory value that must be saved, active instruction result, and content, which is a constant value.

The code generator must map source-language types to machine data types. Storage locations must be allocated for maximum scope and lifetime of source program variables. Allocation may be at fixed memory locations, stack or heap. Frame allocation can be done based on the stack or the heap. Compiler-generated temporaries may also need to be allocated storage locations, although not unless they are register spills or their size necessitates deferred allocation. Space must also be allocated for passing procedure parameters and holding results. Alignment and packing are spatially related machine-dependent issues. Variables are accessed through one of the following run-time display linkage and basic access mechanisms: display in registers [DIJK60], display in memory [GRIE71], static-link chains of enclosing scopes [AHO77], or indirect access through a descriptor containing a reference parameter or a pointer, the size of objects, and the layout information. These mechanisms are translated to machine-addressing modes when an operand value is to be fetched.

During procedure calls and returns, code must be emitted for saving information about the caller's environment, such as registers and status information, for passing parameters and returning values, and for setting up and removing information related to called subroutines and processes. This section of code is typically encapsulated as rigid sequences in the procedure body's prologue and epilogue routines.

Instruction selection involves the complex task of selecting target machine instructions to implement programming language constructs. This phase includes evaluating expressions, executing control constructs, and evaluating Boolean expressions to control conditional execution. When carrying out these computations, the full

repertoire of rich instruction sets usually makes instruction selection a craft requiring significant creativity on the part of the compiler writer.

Machine-dependent optimizations attempt to improve the quality of target code size and speed. Examples of such optimizations are

- using special addressing modes such as double-register indexing on the VAX, Intel-8086, Zilog-Z8001/2, and Motorola MC-68000;
- using special machine instructions such as AOBLEQ (add one and branch less than or equal) on the VAX;
- avoiding extra register loads by using values directly from memory and reusing values in registers;
- avoiding extra stores by reusing computed values in registers, by storing registers only if the value is to be used subsequently, and by computing directly into memory;
- optimizing branches by branch chaining, by merging common tails of code sequences preceding the junction (called cross-jumping [WULF75]) and by wisely choosing long versus short branching (called span-dependent optimization [SZYM78, LEVE80]);
- peephole optimizations [MCKE70]: examining a window of several instructions to make local substitutions and to eliminate redundant instructions.

There has been much formal investigation in code generation. Formal treatments investigate optimal (or near-optimal) code generation algorithms for various machine models [SETH70, AHO76, SZYM78]. Real machines often do not match these formal models. Furthermore, optimal algorithms are usually too costly to be practical. Instead, good heuristics are used. However, formal models do serve to establish principles, discover limits, and provide insights.

1. RETARGETABLE CODE GENERATION

The proliferation of programming languages and computer architectures has created the need for automating the code-synthesis phase in compilers. Recent progress in automatic code generation has made retargetable code generation less of a chore [WULF80, GRAH80, ALLE80, GANA81a]. Some of these code generation techniques have become economically feasible and are gaining wide acceptance in practice. Furthermore, these schemes seem to be promising vehicles for supporting architectural experimentation at the level of determining what impact a particular architecture has on a particular system's code size and execution time.

Let us return now to the main problems of retargetability. In the preceding paragraphs we discussed different aspects of the code generation problem and the basic ideas that are used in developing a code generator. The problem that remains, and which the following paragraphs address, is how these issues must be tackled in retargetable code generation: in changing an existing compiler to produce target code for a new machine. A solution's success is measured by the following key criteria:

(1) minimization of machine-dependent modules;
(2) reasonable speed and compactness of target code;
(3) reasonable compilation speed.

We classify research in retargetable code generation into three categories: *interpretative code generation*, *pattern-matched code generation*, and *table-driven code generation*.

Interpretative code generation approaches generate code for a virtual machine and then expand that code into "real" target code. They create compilers for programming languages L, on machine architectures M, using L "front ends" and M code generators. Examples of such approaches include UNCOL [STRO58], P-code [AMMA77], and U-code [PERK79]. Code generation is "macro"-like, although, as in an interpreter, "state" is also needed to produce good code quality. Case analysis and selection is organized either by mutually recursive routines used with templates written in a coding language [WILC71] or by a table of templates used in conjunction with a template-matching routine that uses shapes as subgoals [JOHN78]. These approaches mix the machine description with the code generation

algorithm. Such code generators are usually hand coded and informal.

Pattern-matched code generation approaches separate the machine description from the code generation algorithm. Pattern matching is used to replace interpretation with case analysis. Instruction patterns may be tree structured, as used by Cattell [CATT78] and by Wulf [WULF80], or linear, as used by Glanville and Graham [GLAN78] and by Ganapathi and Fischer [GANA82]. Depending on the choice of the instruction patterns, pattern matching is performed either by heuristic search [FRAS77, CATT78] or parsing [GLAN77, GANA80]. Heuristic search techniques create a list of subgoals as the search continues, and use heuristics to select subgoals and to determine the order in which to try patterns. Parsing techniques use simple LR-like parsing [GLAN77] or attributed parsing [GANA80] algorithms. In these techniques, the results of transformations become nonterminals in the grammar. Each transformation of the input code to the intermediate representation of a program causes code-generator action. Transformations are syntactic but may require certain conditions of semantic attributes of symbols [GANA82] (e.g., that registers occur or that a constant is in a given range).

Table-driven code generation approaches are automated enhancements of approaches that employ pattern matching. They employ a formal machine description and use a code-generator generator to produce code generators automatically [CATT80, GLAN78, GANA82].

The following sections describe each of these individual approaches in detail. Related reviews have been done by Cattell [CATT77] and by Ganapathi [GANA81c, GANA81d].

2. INTERPRETATIVE CODE GENERATION

Two-level translation schemes were suggested in the late 1950s in order to help design portable compilers [ERSH58, STRO58, STEE61]. Their basic function is to produce code for a virtual machine, which code is then expanded into real machine instructions. Such schemes use code generation languages that describe the code generation process. Current examples are GCL [ELSO70], BCPL O-code [RICH71], ICL [WILC71, YOUN74], CGPL, CGGL [DONE73, DONE79], and PASCAL P-code [AMMA77].

There are two classes of interpretative code generation approaches. The first class is made up of handwritten interpreters that implement a one-to-one or one-to-many mapping between the virtual machine and the real machine instructions. Handwritten P-code translators [AMMA75] and the UCSD PASCAL code interpreter [SHIL78] fall in this class. The entire class requires a lot of code generation decisions to be made by the compiler front end.

Miller made the first attempt to separate machine-dependent issues from the code generation algorithm [MILL71]. He mapped source language to two-address code sequences, which are macros written in MIML (Machine-Independent Macro Language). These macros specify the actual code generation algorithm, for example,

```
macro Add x, y
  If type of x = integer
    and type of y = integer
      then Iadd x, y
  else if type of x = float
    and type of y = float
      then Fadd x, y
  else error
```

The specifications of macros Iadd and Fadd in OMML (Object Machine Macro Language) form the description of addition on the target machine. Thus, for example, for the IBM-360, the macro Iadd would read

```
macro Iadd a, b
  from a in R1, b in R2
    emit (AR a, b) result in R1
  from a in R, b in M
    emit (A a, b) result in R
  from a in M, b in R
    emit (A b, a) result in R
```

States are defined as configurations of operand locations. A state is "permitted" if, from that state, code can be emitted with operands unmoved. Every macro is associated only with a set of permitted states. The designer is required to specify transitions between memory and registers so that the code generator can automatically move to a permitted state if needed. For example,

suppose that an operand in storage must be moved to a register in order to implement storage-to-storage addition. To retarget a compiler to a new machine, Iadd and Fadd must be changed. The algorithm represented in Add remains unchanged. Miller's model, we now see, is too restrictive because it deals only with expression evaluation and very simple addressing schemes; it does not allow indexing, autoincrement, or indirect addressing.

Snyder [SNYD75] attempted to write a portable compiler for the language C [RITC78]. His compiler uses a two-phase translation scheme very similar to Miller's. In a first phase, the code generator walks an expression tree and generates three-address instructions. Both the classification of registers and the register requirements of these instructions are defined by the programmer. A second phase then translates three-address instructions into assembler code for the target machine. Macros and C routines are used to perform tedious case analysis of code sequences. Snyder mostly ignored object code optimization.

While these handwritten interpreters are an improvement over ad hoc methods that have been employed in monolithic compilers, they suffer from serious limitations.

- Owing to the diversity in addressing modes, target machine data types, and instructions, it is very hard, if not impossible, to anticipate a variety of machine organizations in one virtual machine—such as one having a hardware stack and frame pointer, for instance. There, interpreters tend to be very large and complex. For example, in Elson and Rake's implementation of ELSO70, macros have to be paged in from disk.
- Code generation languages are closely tied to a specific language or machine, and so they cannot be considered fully portable.
- The description of the target machine is mixed with the code generation algorithm; the description cannot be changed without changing the algorithm.
- The implementor must perform a tedious case analysis of code sequences and make all low-level decisions about what kind of code is to be generated. The quality of the code produced depends on the implementor's ability to design and debug code generation routines.
- The implementor has a very local view of the code to be generated. It is hard to incorporate context-dependent optimizations such as

  (a) use of indexing instead of explicit addition in an addressing context;
  (b) differentiation between Boolean values to be stored (expressions) and Boolean values that need only be tested (predicates);
  (c) branch chaining and other flow-dependent optimizations [WULF75].

The other class of interpretive approaches is made up of interpreters that are based on a controlled set of schemata. The IR (intermediate representation) consists of instructions to an abstract machine. Translation of the abstract instruction stream to instructions for an actual machine is done by interpreting the changes to the abstract machine state and generating actual machine instructions that cause equivalent actions on the actual machine. This interpretation of the state of the abstract machine allows the code generator to replace multiple abstract instructions with a single machine instruction.

The code generator UGEN is a schema-driven, retargetable code generator for the intermediate form U-code [PERK79]. U-code, an extension of P-code, consists of a set of instructions for an abstract stack machine. Retargetability is obtained by basing the translation of U-code to target machine instructions on a set of schemata. The translation process incorporates the context of the U-code instruction being translated and tracks the actual machine state by using a map between the virtual U-code machine and the actual target machine.

The notion of schema- or macro-based translation between an intermediate form program and a target machine language has been used numerous times to design conventional and retargetable code generators. In most cases, a simple macro expansion based on each intermediate-form operator generates the code. However, this method has many inefficiencies, most of which stem from the fact that little or no context is

used in expanding a particular abstract instruction. Extra context can be incorporated with a great deal of special casing of code as is usually done in a conventional, nonretargetable code generator.

The approach used by UGEN differs from other interpretive approaches in that the state of the virtual machine records the necessary context at any time. A particular U-code operation may cause the code generator to alter the state of the virtual machine and then to generate instructions. Instruction generation also updates the virtual machine state to correspond to the state of the actual machine. Thus, the machine-dependent portion of the code generator is isolated in the schemata. The code generation algorithms deal with a parameterized version of the target machine and of the virtual machine state. Parameters allow handling of issues such as register allocation and differing word lengths. The code generation algorithm proceeds as follows:

(1) Using the next instruction in the instruction stream and the state of the virtual machine (the computation stack), choose a translation schema.
(2) Use the schema to
    (a) emit code, using the information tracked in the virtual machine to record the location of operands on the target machine and the operand types (some operators may not cause any code to be emitted);
    (b) transform the virtual machine state, and perform related transformations on the target machine;
    (c) finally update the target machine state, for example, register contents and condition codes.

The code generation algorithm is also responsible for

(1) register allocation;
(2) loading and storing between target machine memory types (registers and memory); this action is always directly tied to register allocation;
(3) tracking the state of the target machine to eliminate subsumed instructions.

The schemata are chosen on the basis of the U-code instruction to be executed and the virtual computation stack of the machine.

A version of UGEN has been done for three machines: the DEC-20, the Motorola 68000, and the S-1, with a fourth underway (VAX). The results have been quite encouraging. The code on the 68000 is comparable to that for the C compiler including the postoptimizer for the C compiler. On the DEC-20, the code is about 20 percent faster than the code on the one-pass Stanford/Hamburg PASCAL compiler. The time to port the 68000 was about one-and-a-half man-months, including approximately two weeks designing the linkage convention and working on run times. The port to the S-1 required approximately one month and produces code that is approximately 10 percent better than a hand-coded code generator.

The major advantages of UGEN as opposed to other interpretative approaches comes from the use of schemata to specify the translation, and the incorporation of the virtual machine state into the translation scheme. The virtual machine state provides both the ability to incorporate context in choosing instructions and a machine-independent method of dealing with the actual machine resources. The disadvantages of this technique are several. First, certain types of intermediate form instructions are not easily handled by schemata; examples are translating PASCAL set operators, and choosing the best code sequences for moving between memory types. Second, while the schemata serve primarily to isolate machine-dependent concepts, some aspects of the code generation algorithm tend to appear in the schemata. This appearance increases the work needed to retarget the code generator.

3. PATTERN-MATCHED CODE GENERATION

For reasons of portability, code generation research has concentrated on separating machine descriptions from the code generation algorithm itself. The advantage of this approach is the potential ability to use one code generation algorithm for all machines.

Weingart introduced pattern matching to avoid interpretation [WEIN73]. Target ma-

chine characteristics are encoded into a single pattern tree; this tree is expected to be a compact and efficient means of representing most machine-dependent information. The code generator is a tree traverser that accepts information from a parse tree of the source language and stores tokens until a suitable match can be found in the pattern tree. To transport this code generation scheme to a different target machine, the user creates a new pattern tree for the new machine. In practice, Weingart's ideas are not easy to use for the following reasons:

(1) Creating a single tree structure to encode all potential instruction patterns and code sequences is often hard. For example, Weingart had difficulties creating the pattern tree for the PDP-11 by generating the tree from a machine description automatically.
(2) There exists a possibility that on some machines no instructions at all will match parse trees. Pattern mismatches are handled by a set of conversion patterns. However, there is no way to determine if a sufficient set of conversion patterns has been supplied. The code generator might therefore fail to produce any code for some legal subtree of the source language.
(3) Some machines provide a choice of instructions to implement a source language construct. Code quality depends critically on the selection of the most appropriate instructions. For example, an "increment" can be used instead of an "add one." Special care must therefore be taken by the tree traverser to make the best possible instruction choice. Weingart's technique cannot make such a choice.

Newcomer [NEWC75] uses a goal-directed recursive heuristic search called "means-end-analysis" [NEWE69] to generate code patterns—which are not machine instructions—from a parse tree and a set of operators. His scheme is restrictive and of little practical importance for the following reasons:

(1) It only deals with arithmetic expression trees.
(2) Real machine architectures are not always readily representable in his specification scheme.
(3) His code generation algorithm can fail to produce a code template owing to
    (a) a possibly inadequate set of operators, or
    (b) a limitation of the depth of search performed by means-end-analysis. This limit is needed to prevent the code generator from possibly looping.
(4) The algorithm is exhaustive and therefore far too expensive to be used in a production compiler. Weeks of computer time could be required to analyze even some simple trees.

Aho and Johnson [AHO76] consider a similar exhaustive "brute force" optimal code generation scheme. They use a three-phase dynamic programming algorithm to derive optimal code sequences for expression trees. With dynamic programming, the results of each evaluation are saved and reused for subsequent identical evaluations, rather than being regenerated.

In the first phase, trees are traversed bottom up. For each vertex v, all possible machine instruction translations of the subtree rooted by v are used to compute an array $Cr[v]$ ($1 \leq r \leq$ total number of registers) of costs. Cr is the minimum number of instructions required to compute the subtree using r registers. All permutations of evaluation order are considered. At the end of the first phase the following are determined:

(1) the optimal instruction sequence required to compute the subtree rooted by vertex v, and
(2) the optimal evaluation order for the subtree of v.

The second phase uses the cost arrays and traverses top down to mark tree nodes that must be computed in memory locations. It identifies subtrees where "stores" are necessary because of too few registers. The last phase walks each marked subtree and generates code to evaluate the subtree, followed by appropriate "stores" into temporary memory locations. The complexity of code generation is linear in the number of tree nodes, linear in the number of cov-

ering instructions, quadratic in the number of available registers, and exponential in the number of operands used by an instruction. The shortcomings of this model are as follows:

(1) It only deals with arithmetic expression trees excluding common subexpressions.
(2) Only mathematically clean instructions are dealt with, avoiding asymmetric registers and special instructions found in real computers.

Ripken [RIPK77] uses an extended version of Aho and Johnson's algorithm to generate optimal code from expression trees. His IR (intermediate representation) consists of attributed expression trees linked together as a graph according to the flow of control of the source program. The instruction set of the machine is described as attributed tree patterns with a set of AT (attribute transformation) rules. A prepass to code generation maps simple (nonaggregate) source language types to characteristic value ranges of machine storage locations.

Code generation then consists of a two-phase transformation of the IR. In the first phase, AT rules (derived from an analysis of the machine's AT rules) are used to generate code for expression trees. A machine operation is assumed to exist for every IR operator and its attribute values. A three-pass tree traversal scheme (very similar to Aho and Johnson's [AHO76]) determines the order of AT-rule applications and which AT rule is to be applied at each node. The difference between Aho and Johnson's algorithm and Ripken's is that Ripken considers real instruction sets with several register classes and addressing modes. Transfer operations (in addition to "stores") between machine storage locations are also considered (sufficient transfer operations are assumed to allow operand transfer between all storage classes), together with register, temporary allocation, and assignment. Like Aho and Johnson's, Ripken's first phase emphasizes locally optimal code. The second phase linearly arranges such locally optimal code blocks and generates the necessary branch instructions among them.

In Ripken's scheme, storage locations are described as pairs containing an operand class and address, such as (bytes, 15), (words, 16), and (register, 2). An operand is described by its address descriptor and value range (e.g., n bit, $-2**(n-1)..2**(n-1)$, 2's complement). Operations are described by tree patterns (at least one pattern per IR operator) with predicates on attribute values and evaluation rules to describe the semantics.

Consider addition on the Intel 8080; the addition template for the Intel 8080 is shown in Figure 1. From these AT rules, IR operand-specific application rules are selected for code generation. Ripken also requires templates for operand transfer between two storage classes even if the machine architecture does not have a "move" instruction between them. This specification is needed so that transfer paths exist from any storage class to any other. The following template is an example of such a specification:

```
:= Register_pair HL_pair
   (i = address of register pair)
     mov 2*i, H
     mov 2*i + 1, L.
```

Addressing modes are also represented as tree templates. They are inserted in applicable places for operands in the IR tree before code generation. An attempt is made to subsume address computations by machine addressing modes.

Ripken did not implement his proposal. A straightforward implementation would require a great deal of computation of different permutations with combinatorially explosive choices. A code generator based on this model would be very slow. Also, in spite of emphasis on optimal code generation, certain inefficiences are likely to occur at the border between code blocks of different expression trees that represent individual statements rather than the basic blocks of Aho and Ullman [AHO77]. These inefficiencies include redundant loads and stores, and failure to take advantage of autoincrement/decrement possibilities.

Fraser [FRAS77] uses ad hoc rules, which are coded as MLISP [SMIT70] subroutines, to minimize machine dependency in code generation. He uses XL, a machine-inde-

```
AT-rules·
  choice (1)
    predicates:
      cell_class(01)        = accumulator
      cell_class(02)        = immediate mode
      value(02)             = 1
    evaluations:
      cell_class(03)        := accumulator
      address(03)           := address(02)
      code                  :=                    inr A
      Z, S, P, AC affected

  choice (2)
    predicates:
      cell_class(01)        = H and L register pair
      cell_class(02)        = H and L register pair
      value_range(01)       = F ± 15
      value_range(02)       = F ± 15
    evaluations:
      cell_class(03)        := H and L register pair
      value_range(03)       := F ± 16
      code                  :=                    dad HL
      CY affected

choices (3), (4)  similar to above.
```

Figure 1. Template: +01 02 → 03.

pendent intermediate representation (IR) that may need to be adapted to accommodate new source languages or target machines. Rules are used to perform storage allocation, and in this process XL is rewritten into ISP′ [WICK75], a modified version of ISP [BELL71]. Code generation then consists of matching this ISP′ form with machine instruction patterns that are also written in ISP′. Pattern mismatches invoke rules (subroutines written in MLISP) that try to rewrite the ISP′ form of the IR. Examples of such rules include: invert relational tests and alter control flow, replace indirect references with indexed ones, and load nonaccumulator operands into accumulators. The rules do not guarantee that a code sequence will eventually be found.

Fraser performs syntactic analysis of ISP [BELL71] descriptions at code generation time to recognize stack operations, macros that set condition codes, and index registers and accumulators. Rules, specified as subroutines in MLISP, are used to allocate storage for variables and to classify registers as index registers and accumulators. Examples of such rules are "store integers in the widest possible memory that can participate in an add instruction" and "if a single instruction can add a register to some offset and use the result to index some memory, then the register is an index register." On machines such as the IBM-360 or the PDP-11 in which small integers can be stored in a half-word or a byte, Fraser's allocation rule for integers is inefficient. On architectures with no index registers, such as the Intel 8080, the index register rule is useless. In general, Fraser's rules are ad hoc and machine specific. Machine descriptions could be used as a substitute for some of these rules because it is not hard for the user to specify index registers and accumulators as part of the machine description.

Fraser's knowledge-based approach has several shortcomings.

- Rules compromise generality for efficiency. They are based on the observation, similar to Wick, that many computer architectures are similar in design.
- The same rules are not usable for diverse architectures; often completely new rules are necessary. Some rules such as "load nonaccumulator operands into accumu-

lators" could potentially contradict other parts of a compiler such as the "register allocator." In Fraser's scheme, redundant loads and stores are unavoidable.
- It is hard, if not impossible, to utilize special instructions and addressing modes of a target machine. An XL primitive such as "a = a + 1" may match multiple machine instructions ("add #1, a" and "inc a" on the PDP-11). It is not clear when, if ever, his code generator would resolve such multiple matches and choose a good alternative.
- The code generator is very slow: the implementation in LISP on a PDP-10 KA10 generates one line of assembler code each second.

A number of Snyder's ideas [SNYD75] are used by Johnson in his successful implementation of the portable C compiler [JOHN77, JOHN78]. Templates and a template-matching algorithm form the central idea around which code generation is designed. Templates specify

(1) the operator of the subtree, such as an assign-op;
(2) the desired result location on the target machine, such as a register location or a condition-code setting;
(3) the machine addressing mode and the language data type of the operands of the expression, if any (e.g., register mode, pointer type);
(4) the resource requirements, that is, the number of temporaries and scratch registers needed for implementing the subtree;
(5) a rewrite rule specifying how to replace one subtree with another; and
(6) the machine instruction(s) to be emitted on a successful match. The op-codes and operands of these instructions are, in general, macros that are expanded into assembler mnemonics of the target machine (e.g., emit Integer-Op-code, Address-form-of-Left-Operand, Address-form-of-Right-Operand).

The template-matching algorithm tries to match a subtree against suitable templates in an attempt to transform the subtree. Such transformations must consider the result location specified in the template. For an efficient implementation of the algorithm, it is essential to restrict the search for an acceptable template. A template matches a subtree when all the template specifications (1)–(5) match. Condition (4) includes a call to a resource allocator; the match fails if it is unable to allocate the required resources. On a failure, an attempt is made to transform the subtree using default or machine-dependent rewrite rules, such as, for example,

$$a \mathrel{+}= b \quad \text{becomes} \quad a = a + b$$
$$x{+}{+} \quad \text{becomes} \quad ((x \mathrel{+}= 1) - 1).$$

Feldman uses C's code generator in his implementation of the portable FORTRAN 77 complier [FELD79]. Register and temporary allocation is mostly handled by the code generator. However, mapping the different kinds of FORTRAN integer variables to C's types and generating the necessary type conversions are not easy tasks. Furthermore, FORTRAN's exponentiation operator (**) must be treated as a special case, and MIN and MAX functions are implemented as nested conditional expressions.

The shortcomings of Johnson's approach are as follows.

- Templates are not the only places where code selection is specified (templates deal with expression evaluation only). Other phases of the compiler must emit code for control constructs. The machine-dependent portions of the first pass (the machine-independent pass) of the C compiler includes code for subroutine prologues and epilogues, the Switch statement, branches, label definitions, alignment operations, and changes of location counters. Assignment of register numbers and stack-offset numbers are also done in this pass. Thus separating the machine-independent and machine-dependent portions of the C compiler is not simple.
- The IR (intermediate representation) is specifically designed for the C language. Language-dependent data types are embedded in the templates. The basic data types of C limit other languages to compile into its IR. It is not easy to map references that are neither local nor global (i.e., at an intermediate nesting level) to the C compiler's back end.

- Macro interpretation is used for selecting the assembler instruction from a set of possible instructions matching the subtree. The macros are really assembly language instructions for a machine; they map between the abstract instruction and the syntax for a particular assembler. Multiple matches between templates and subtrees are thus avoided, but these macros must be changed when the compiler is transported to a new machine.
- On a mismatch, machine-dependent rewrite rules call the code generator for possible tree alterations. Such rules potentially can produce infinite loops.
- Not much thought is given to machine-specific optimizations but there is a specific plan for a postpass by a peephole optimizer. Condition codes are not saved between expressions.
- About one-third (~1000 lines) of the code generator needs to be rewritten to do a transport.

4. TABLE-DRIVEN CODE GENERATION

To suit a variety of target architectures, flexibility and tuning of the code generation algorithm is necessary. Lately, research has concentrated on providing this flexibility by doing an automatic analysis of a formal description of the target machine. Table-driven approaches attempt to break the code generation problem into parts: register allocation, storage allocation, and instruction selection. These problems interact strongly. In particular, the choice of an optimal set of instructions to implement a direct acyclic graph [AHO77] depends on register availability. Likewise, register allocation depends on instruction choice. This problem could be somewhat glossed over on the PDP-11 because of restrictive address modes, but it explodes on the VAX and is a major issue on the MC68000.

Cattell [CATT78, CATT79, CATT80, WULF79, WULF80] uses TCOL, a tree-based intermediate representation, as the IR and a recursive tree-traversing algorithm to generate code. Templates of the form "tree pattern → result sequence" are used to specify the translation from a TCOL program tree to machine code. The result sequence specifies code to be generated, calls to a register allocator or label generator, or further matches to be recursively performed (e.g., a statement within a control construct). Templates are grouped into schemata representing the context, such as flow result and value result, in which code is to be generated. The code generator starts from the root of the IR tree and attempts to match templates with the largest possible subtree at the current tree node. On a match, the corresponding result sequence is processed. Templates must be composed recursively to match an entire program tree. Operand mismatches are forcefully resolved by a subtargeting operation, which consists of allocating a location of the desired data type and emitting a "move" into that location. If an IR operator does not match any template operator, an attempt is made to transform the operator using tree equivalence axioms and heuristic search. Multiple matches are handled by sorting the alternatives with decreasing preference and choosing the first. For example, $x \leftarrow x + 1$ occurs before $x \leftarrow x +$ constant.

Cattell proposes a formal model of instruction set processors, Mop (genealogically related to ISP), containing descriptions of storage locations, addressing modes, and instructions. A set of assertions, in a parenthesized LISP-like notation, are written for addressing modes and instructions. Such descriptions are significantly more useful for automating software than ISPS procedural descriptions. An attempt is made by heuristic search to derive code sequences for IR operators that do not have equivalent machine op-codes (e.g., subtraction on the PDP-8) by using tree equivalence axioms, such as DeMorgan's laws and relations between addition and subtraction. The Mop assertion templates are then augmented with such derived sequences and with pseudooperations utilizing side effects of instructions to implement IR operators.

For example, consider "c ← a & b" on the PDP-11, which does not have a Boolean "and." Heuristic search obtains the closest machine instruction "bic." Means-end-analysis is then used to try matching "c ← a & b" with "c ← c & ~d" (assertion for "bic d, c"). The entire search process is

			code emitted
IR:	$c \leftarrow a \,\&\, b$		
goal:	$c \leftarrow c \,\&\, \sim d$	"bic d, c"	
mismatch:	a with c, decomposition	"c ← a"	mov a, c
	b with $\sim d$, transformation	"b ← ~~b"	
IR:	$c \leftarrow c \,\&\, \sim\sim b$		
goal:	$c \leftarrow c \,\&\, \sim d$	"bic d, c"	
mismatch:	$\sim b$ with d, decomposition	"d ← ~b"	
heuristic search obtains "com"			
IR:	$d \leftarrow \sim b$		
goal:	$d \leftarrow \sim d$	"com d"	
mismatch:	b with d, decomposition	"d ← b"	mov b, d
match:			com d
match:			bic d, c

Figure 2. Code emitted for "c ← a & b".

shown in Figure 2. Attempts are also made to match the IR with other potentially useful instructions ("c ← a & b" with "c ← ~c" ("com")), but the search is too deep and is subsequently cut off before a code sequence can be found.

Such a heuristic search is too time consuming to be applied during code generation. On machines with condition codes, for example, a conditional jump requires several transformations. Cattell suggests doing such searches before code generation and tabulating the results for the code generator. In practice, it is both hard and time consuming, if not impossible, for such an axiomatic approach to automatically derive code sequences for floating point operations or to do "2n"-bit arithmetic on an "n"-bit machine (e.g., 16-bit arithmetic on the 8-bit Intel 8080 or 32-bit arithmetic on the 16-bit PDP-11). The Intel 8080 has no explicit "branch on greater" or "branch on equal" instructions; it has only "jz" (branch if zero flag is set) and "jp" (branch if sign flag is clear). Code sequences for such IR control statements are very long, as demonstrated in Figure 3. And the code for "Bge x y La" (if $x \geq y$ jump to La) is twice as long.

While Cattell's model is more general than Newcomer's, which only deals with arithmetic expressions, it has the following drawbacks.

- The code generator avoids machine-dependent issues such as binding variables to storage formats, space allocation, and addressing of variables. The model fits only the "CODE" part of Bliss's [WULF75] DELAY-TNBIND-CODE-FINAL model. TNBIND allocates registers before the code generation pass. The code generation pass also emits allocation commands. The allocation commands emitted by the code generator may conflict with the requirements of TNBIND [JOHN75]. Interfacing the code generator within Bliss's framework might be hard. Leverett [LEVE81, LEVE82] attempts to integrate code generation with register allocation in PQCC [WULF80].
- Templates are part of the code generation algorithm because some result sequences specify further matches to be performed. Therefore it is difficult to alter the templates without changing the algorithm.
- For subtargeting to be successful, there must be "move" instructions in the target machine between all possible location types. Otherwise, there is a chance that the code generator may block generation of code for a valid program tree. Sometimes, to prevent blocking, there will be unnecessary moves.
- Multiple matches are "statically" resolved by ordering alternatives. This strategy does not result in optimal code sequences in certain cases. For example, on machines such as the VAX-11/780 that have both two-address and three-address operations for a single IR opera-

```
"Beq x y La"   if x = y jump to La
  assumptions:   x and y are 16-bit integers
                 x is in register pair BC
                 y is in memory addressed by the HL pair

  mov A, M       accumulator ← y's low-order byte
  cmp C          compare x's low-order byte with accumulator
  jnz Lb         jump to Lb if the zero flag is not set
  inx H          increment address register
  mov A, M       accumulator ← y's higher order byte
  cmp B          compare x's higher order byte with accumulator
  jz La          jump to La if zero flag is set
  Lb
```

Figure 3. Code sequence for an IR control statement.

tor, the optimal choice depends on whether the operator is commutative and the operands are destroyable (e.g., $a \leftarrow b + a$).

- Operator mismatches invoke a heuristic search that is recursive and combinatorially explosive. The search must be cut off at some point so that the code generator will not loop or use excessive amounts of time. Consequently, no machine code may be generated in cases where the search is cut off.
- Code sequences that are produced do not make effective use of all machine resources. Special-case subsumption operations such as autoincrement are hard to describe as templates. Also, equivalent locations are not recognized. Thus, if a register contains an operand that is also in a memory location, the code generator fails to identify this equivalence and use the register. Even if a value is already in a register, it is invariably reloaded. This reload happens because the code preceding the reload could possibly be generated from an arbitrarily distant section of the program tree, and this optimization is therefore hard to recognize in any local tree context analysis. A separate peephole optimizer package [McKe70, Fras79, Davi80] may solve some of these problems, but there may be conflicts with other parts of the compiler, such as the register allocator [Rudm79].

Glanville and Graham [Glan77, Glan78, Grah80] made a significant breakthrough. While their approach is not a complete solution, it is a big step toward automating the code generation phase in compilers. The generated code is reasonable and the code generator is fairly retargetable. They chose a very low-level IR in the form of Polish prefix expressions. Storage allocation and binding are assumed to be already done by other phases of a compiler. The code generation algorithm is derived from context-free parsing theory [Aho73]. Instructions in the target machine are also expressed in prefix form. The instructions are described as grammar productions with the left-hand side (LHS) specifying the result of an operation and the right-hand side (RHS) the operation. An assembler instruction computing the RHS is supplied with each production. Thus r.1 → +r.1 k = 1 "inc r1" specifies that an addition of 1 to register 1 (with the sum going to the same register) can be obtained by an "inc r1" instruction. A one-to-one mapping is assumed between productions that serve as machine templates to the code generator and target machine instructions. Since the addressing modes of operands are explicitly described as grammar terminals, this one-to-one restriction is essential. The IR string is parsed according to the context-free grammar and the appropriate assembler instructions are emitted. Since the grammar is almost always ambiguous, a modified LR(1) parsing algorithm is used. Multiple matches produce shift–reduce or reduce–reduce conflicts and are resolved heuristically. Shift–reduce conflicts are resolved in favor of a shift so that more powerful single machine instructions are preferred to equivalent sequences of instructions. Similarly, reduce–reduce conflicts are resolved

in favor of the production with the longer RHS. In case of conflicts between identical length productions, a "best instruction first" ordering is used to select the first production.

The table-driven code generator is automatically derived from instruction patterns. In practice, reasonably compact tables are obtained and also, because standard context-free parsing techniques that forbid backup are used, a linear time code generation algorithm is obtained.

In Glanville's machine description (Polish prefix expressions), different data types of the target machine (bytes, words, floating point) and special addressing modes (autoincrement, autodecrement) are not used. For example, in the production r.1 → +r.1 r.2 add r.2, r.1, the data types of the operands and that of the result are not evident. The code generating IR parser is automatically constructed from the instruction-set description using an LR(1)-like table constructor [AHO73]. This approach emphasizes correctness of the code generator. Possible looping configurations (where $V \Rightarrow^* V$) are detected by analysis of grammar tables, using a transitive closure algorithm on a relation characterizing parser moves. Instruction grammars are analyzed for uniformity, ensuring that all operands are uniformly valid to operators independent of the context in which they appear. States are inspected to check that, for all first symbols of left or right operands, either a shift or a reduce is signaled and that no error actions are encountered. Although some of the semantics (register number, source-destination relationship) that are necessary to emitting instructions are used in productions, they are not used to control parsing. These semantics can be viewed as earlier and more primitive forms of attributes that are used by Ganapathi [GANA80]. Sometimes semantic restrictions, such as a constant required to have value 1, or required register usage, may not be satisfied for any production in the set of possible reductions in a particular state. In such cases, default instructions are needed to prevent the code generator from blocking for a valid input. If all patterns in a reduce state have semantic actions, the pattern matcher constructs an instruction sequence to compute the unrestricted pattern by removing its instructions from the description and providing the pattern as input to the code generator. Action tables are changed to consider default reductions instead of signaling an error. This consideration ensures that a necessary set of conversion patterns has been supplied and thus that the code generator cannot block for a valid IR input.

Implementations of code generators based on the Graham-Glanville model can be found in papers by Graham et al. [GRAH82], Bird [BIRD82], and Landwehr et al. [LAND82]. Graham et al. discuss the implementation of a local code generator for the VAX-11. Bird discusses the implementation of a code generator for a PASCAL compiler on an Amdahl-470, and Landwehr et al. discuss their implementation for the Siemens-7.000 and the Motorola MC-68000.

While Glanville's scheme is very efficient, easily the fastest among comparable code generation schemes, and provably correct, it is not completely portable because of the following reasons.

- The IR is very low level; it contains assumptions about the addressing structure of the target machine. The IR requires operand binding and operand addressing to be dealt with by the front end of a compiler. The mapping between operators in the IR and target machine opcodes is required to be one to one. Thus, in transporting a compiler from one machine to another, changes have to be made to the IR. Such changes are reflected in Glanville's IRs for the PDP-11 and the IBM-360 (16-bit, as opposed to 24-bit, address computations).
- Since storage allocation and binding issues are avoided, any change in the implementation of the run-time display will result in changes to the IR code to access variables. Glanville requires careful design of the IR and tailoring to the machine. Many potential IR constructs cannot be used. Some interfacing problems, such as the allocation of registers that are used for display purposes, might arise between the register allocator and the display mechanism.

- Very good code cannot be generated by purely local context-free expansion. For example, "a & b" in "if (a & b)" and "c := a & b" may need to yield different code because of the context in which it is used. Because this method uses limited context, the quality of generated code *is* strongly dependent on the exact IR form generated by the front end. For example, in an addressing context, explicit addition is performed in the intermediate representation and in the generated code instead of using indexing. The quality of target code also depends on the completeness of the machine description.
- Heuristic resolution of multiple matches fails in certain cases (e.g., in the choice of two-address or three-address instructions on the VAX-11/780).
- Glanville ignored machine-dependent optimization issues. The code generator does not worry about information retention, such as values left in registers from previous computations. Thus, redundant load and store elimination, recognition of equivalent locations, subsumption of addition, or subtraction via autoincrement and decrement are not done.
- Because each op-code/data type/addressing mode must be written as a separate production, the grammar can easily get very large (over 1000 productions for the VAX). Graham et al. [GRAH82] discuss a solution by factoring productions.
- Treatment of semantic restrictions can cause problems. For example, consider a restricted instruction I. Because the code generator may often block because of I's restrictions, the implementor may be forced to leave I out of the grammar description. Semantic restrictions are outside the simple code generation algorithm based on syntax. The grammar may be extended in some limited cases to solve this problem.

In GANA80, the code generation approach of Glanville is extended and enhanced with semantic attributes, disambiguating predicates and a semantic evaluation framework. The basic Graham–Glanville code generation algorithm is modified to provide formalized attribute processing and automated semantic handling [GANA82]. The target machine is described by attribute grammars [KNUT68, RAIH80] instead of by context-free grammars [AHO73]. Code generation is performed by attributed parsing. Semantics and context are used to do all attribute evaluation in a single bottom-up pass. Implementations of code generators (CG) based on this model exist for the VAX-11, PDP-11, and the Intel-8086/8087. CG produces 50 lines of assembler code per second (real time) on the VAX. The target code quality is comparable to handwritten assembler code for user programs, and to the code produced by monolithic (single-language/single-machine) compilers, such as the UNIX C compiler with peephole optimization for the PDP-11 and the VAX-11 and Intel's PASCAL compiler for the iAPX-86.

The abstraction used by Ganapathi and Fischer envisions a more complete code generation model and at the same time considers the problems of a real machine architecture. It is intended to better fit into the framework of an optimizing compiler. Attributes are used to blend all machine-dependent aspects of code generation into one module; they are an integral part of the code generation process including machine-specific expansions of variable addressing, peephole optimizations, dynamic common-subexpression detection, and register allocation.

The intermediate representation, called IR [GANA81b], is Polish prefix, linear, and attributed. Binding high-level-language variables to addressable locations in the target machine is done via attributes in IR. The first phase of CG reads in target machine locations, their properties, data types, and alignment restrictions. Examples of these characteristics of a target machine are stack, frame, memory, registers, direction of frame growth, positive or negative offsets for variables, byte, word, and floating-point data types and their alignment restrictions. By use of this information, variables that appear in IR are mapped to locations in the target machine.

The use of attributes in IR allows considerable flexibility for targeting various high-level language front ends (e.g., an Ada front end [BAKE82]). Machine-independent optimizations that have been found possible

by compiler phases in the front end can be expressed as attributes in IR. Since these optimizations interfere with instruction selection, the attributes in IR are used to guide and control (a) pattern matching during instruction selection and (b) register allocation and assignment.

The target machine is described as an attribute grammar comprised of three classes of productions: *addressing-mode productions, instruction-selection productions,* and *transfer productions.* Addressing-mode productions describe addressing modes in the target machine. Instruction-selection productions describe op-codes, with operand restrictions for every machine data type. Transfer productions describe conversions for a machine data type to every other data type. Conversions to correct storage types are ensured by predicates and semantic attributes. They are essential to prevent the code generator from blocking.

Consider the Base-Index addressing mode (BX) on the Z8001 microprocessor, a segmented version of the Z8000 CPU. This addressing mode allows two registers to be indexed to form an address. However, the register R0 cannot be used as the index register and the register RR0 cannot be used as the base register. This restriction on the programming model is described by an attribute predicate *Notsame* in the following production that describes the BX addressing mode:

```
Address↑a → INDEX   Register↑i Register↑b
                    Notsame (↓i ↓"R0")
                    Notsame (↓b ↓"RR0")
                    ADDR (↓"0" ↓i ↓b ↑a)
```

The symbol "Register" is a grammar nonterminal; it appears with attribute "i" to specify the index register, and attribute "b" to specify the base register. The symbol ADDR synthesizes a machine address for a datum on the target machine. The attribute "a" represents this address.

The use of semantic attributes, attribute predicates, and action symbols not only allows more information to be used to drive the parse, but also brings many of the decisions out of semantic routines into the grammar for an overall increase in clarity. The attribute-predicate adjunct to productions serves three essential purposes:

(1) to describe target-machine restrictions on the programming model;
(2) to reduce the total number of productions in a target-machine description;
(3) to handle pattern-matching conflicts in a careful fashion by controlling parsing. This control also aids in incremental development of a code generator.

The use of attribute predicates and semantic treatment of attributes effectively replaces the cross-product of op-codes and addressing modes by their additive sum. Addressing modes are independently described as addressing-mode productions and op-codes are described as instruction-selection productions. This separation significantly reduces the total number of productions and consequently yields a much smaller size of code generator tables.

Condition-code setting and side effects of instructions such as autoincrement/autodecrement effects on registers are incorporated in instruction-selection productions. In the Graham-Glanville approach, only one result can be tracked as a result of production selection. On the contrary, in the Ganapathi-Fischer approach, multiple instruction results, such as condition codes, can be tracked as semantic attributes. Transfer productions automatically handle data-type conversions for instruction operands and storage-class conversions, such as memory-to-register transfers.

Upon instruction selection, the instruction code is buffered for a basic block [AHO77]. Machine-dependent optimizations [WULF75, SZYM78] and peephole optimizations [DAVI80, DAVI82, GEIG82] are performed by limited "on-the-fly" data-flow analysis of basic blocks.

Register assignment is guided by attributes in IR and the context of operand use. Analyses carried out by machine-independent optimizers are expressed as attributes in IR. Context is determined by the prefix operator and semantic attributes in productions. For example, suppose that a value is loaded into the right kind of register in a multiplication context. This operand location may be an even-odd register pair on the PDP-11 and the AX-DX accumulator pair on the Intel-8086.

Register pairing and sharing are also considered during register management. Examples of these register cells are AL, AX on the Intel-8086, and RL0, R0, RR0, RQ0 on the Z8001/Z8002. Register tracking is done to keep a history of free, volatile, and locked registers. This tracking prevents redundant loads, stores, and register-to-register moves. Thus, register spills are avoided by a combination of attribute guidance and register tracking.

The Ganapathi–Fischer code generation model extends that of Graham and Glanville. The main extensions are in the areas of code generator structure and modularization, storage binding, machine-dependent optimizations, and retargetability. Semantic attributes and attribute predicates are used to accomplish this goal.

The shortcomings of this technique are as follows.

- The introduction of the more complicated IR, using attribute grammars, pushes more complexity on the compiler writer in describing code generation and optimization.
- The desire to generate code in a single bottom-up pass precludes iterative machine-dependent optimizations [WULF75]. The restricted attribute evaluation and flow may affect target-code quality. Multipass attribute evaluation paradigms may solve this problem at the expense of compilation speed.
- Register assignment is guided by attributes in IR and extended register history and tracking. While this scheme avoids register spills in most cases, it does not guarantee any condition on them. A separate register allocation pass is needed to guide the code generator.
- Evaluation ordering and packing [JOHN75] are not included within CG's domain. The main concerns in incorporating them are the size and speed of CG.
- Most optimizations are not extended beyond basic blocks.

5. SUMMARY AND CONCLUSIONS

There has been much theoretical research in code generation. Formal research has usually not considered the wide range of machine architectures that are available in the market. Interpretive approaches are improvements over ad hoc code generation because only "p + m" translators are needed to implement "p" languages on "m" architectures rather than "p * m" translators. However, for such schemes, machine descriptions are intermixed with the code generation algorithm. Retargeting thus requires changing the code generator for every new machine.

Pattern-matching approaches separate the machine description from the code generation algorithm, thus providing a higher degree of portability and a better theoretical foundation. In such schemes, pattern matching is used to replace interpretation. Johnson uses a template-driven, heuristic, linear programming algorithm in his implementation of the portable C compiler. While the code generator is slow when compared to ad hoc code generators, the expected explosion due to dynamic programming does not occur, since in real life the exponents are small (2–3). His approach may be the most practical one for real-world compilers and reasonably talented compiler writers. He makes an attempt to separate most machine dependence but leaves some ad hoc hand coding to be done.

Ripken has considered in detail the interaction between different phases in a compiler. However, an implementation of his dynamic programming algorithm can be expected to be prohibitively slow. Fraser's rule-based system is inefficient and its portability is questionable. Cattell uses a heuristic search algorithm to derive code sequences in cases of operator mismatch between the IR and target machine templates. Such automatic derivation using axioms is not practical for a variety of machine instructions. Glanville's scheme is best from the viewpoint of practicality. However, using his scheme in a production compiler requires a lot of machine-dependent work to be done by other phases of the compiler. His scheme calls for a complex interaction between several compiler phases. Many of the difficult code generation questions are left informal by handling them with semantic restrictions. The Ganapathi–Fischer code generation model is an extension of the Graham–Glanville approach. The use of attributes simplify the task of interfacing

machine-dependent and machine-independent parts of a compiler. They prove useful in storage binding, guiding register allocation and assignment, and machine-dependent optimizations.

A number of techniques that are prototypes of vehicles for future research in automatic code generation have been illustrated in this paper. These techniques demonstrate the feasibility of automatic code generation becoming a routine issue in compiler technology. Table-driven code generation should be viewed as one more step in the advancement toward automated compiler generation where such compilers can be as competent as today's handwritten compilers.

ACKNOWLEDGMENTS

The help provided by Johannes Heigert, Raphael Finkel, William Leland, Uma Mahadevan, Kausalya Ganapathi, Adele Goldberg, Rachel Rutherford, and the referees in improving the readability of this paper is appreciated.

This research was supported in part by NSF grant MCS78-02570 and by the Office of Naval Research, under a contract to the University of California Lawrence Livermore Laboratory, contract LLL PO no. 9628303.

REFERENCES

AHO73 AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation and Compiling*, vols. 1 and 2. Prentice-Hall, Englewood Cliffs, N. J., 1973.

AHO76 AHO, A. V., AND JOHNSON, S. C. "Optimal code generation for expression trees." *J. ACM* **23**, 3 1976, 488–501.

AHO77 AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.

ALLE80 ALLEN, F. E., CARTER, J. L., FABRI, J, FERRANTE, J., HARRISON, W. H., LOEWNER, P. G., AND TREVILLYAN, L. H. "The experimental compiling system." *IBM J. Res Dev.* **24**, 6 (Nov. 1980), 695–715.

AMMA75 AMMANN, U., NORI, K., JENSEN, K., AND NAGELI, H. *The Pascal (P) Compiler Implementation Notes*. Institut für Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

AMMA77 AMMANN, U. "On code generation in a Pascal compiler." *Softw. Pract Exper.*, **7**, 3 (June/July 1977), 391–423.

AUSL82 AUSLANDER, M., AND HOPKINS, M. "An overview of the PL.8 compiler." In *Proc. SIGPLAN82 Symp. Compiler Construction* (Boston, Mass., June 23–25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 22–31.

BAKE82 BAKER, T. P. "A single-pass syntax-directed front end for Ada." In *Proc. SIGPLAN82 Symp. Compiler Construction* (Boston, Mass., June 23–25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 318–326.

BELL71 BELL, C. G., AND NEWELL, A. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, 1971.

BIRD82 BIRD, P. L. "An implementation of a code generator specification language for table driven code generators." In *Proc. SIGPLAN82 Symp Compiler Construction* (Boston, Mass., June 23-25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 44–50.

CATT77 CATTELL, R. G. G. "A survey and critique of some models of code generation." Tech. Rep., Computer Sciences Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.

CATT78 CATTELL, R. G. G. "Formalization and automatic derivation of code generators." Ph.D. dissertation, Computer Sciences Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.

CATT79 CATTELL, R. G. G., NEWCOMER, J. M., AND LEVERETT, B. W. "Code generation in a machine-independent compiler." In *Proc. ACM SIGPLAN Symp. Compiler Construction* (Denver, Colo., Aug.), *ACM SIGPLAN Not.* **14**, 8, (Aug. 1979), 65–75.

CATT80 CATTELL, R. G. G. "Automatic derivation of code generators from machine descriptions." *ACM Trans. Program Lang. Syst* **2**, 2 (Apr. 1980), 173–190.

CHAI82 CHAITIN, G. J "Register allocation and spilling via graph coloring." In *Proc. SIGPLAN82 Symp. Compiler Construction* (Boston, Mass., June 23-25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 98–105.

DAVI80 DAVIDSON, J. W., AND FRASER, C W., "The design and application of a retargetable peephole optimizer." *ACM Trans. Program. Lang. Syst.* **2**, 2 (Apr. 1980), 191–202.

DAVI82 DAVIDSON, J. W., AND FRASER, C. W. "Eliminating redundant object code." *Proc. 9th Ann. ACM Symp. Principles of Programming Languages* (Albuquerque, New Mex., Jan. 25–27), ACM, New York, 1982

DIJK60 DIJKSTRA, E. W. "Algol 60 translation." *Suppl. ALGOL Bull.* 10 (1960).

DONE73 DONEGAN, M. K. "An approach to the automatic generation of code generators." Ph.D. dissertation, Computer Science Dep., Rice Univ., Houston, Tex., 1973.

DONE79 DONEGAN, M. K., NOONAN, R., AND FEYOCK, S. "A code generator generation language." In *Proc. ACM SIGPLAN Symp. Compiler Construction*

(Denver, Colo., Aug.), *ACM SIGPLAN Not.* **14**, 8 (Aug. 1979), 58–64.

ELSO70 ELSON, M., AND RAKE, S. T. "Code generation technique for large language compilers." *IBM Syst. J.* **9**, 3 (Dec 1970), 166–188.

ERSH58 ERSHOV, A. P. "On programming of arithmetic operations." *Commun. ACM* **1**, 8 (Aug. 1958), 3–6.

FELD79 FELDMAN, S. I "Implementation of a portable Fortran 77 compiler using modern tools." In *Proc. ACM SIGPLAN Symp. Compiler Construction* (Denver, Col., Aug.), *ACM SIGPLAN Not.* **14**, 8 (Aug. 1979) 98–106.

FRAS77 FRASER, C. W. "Automatic generation of code generators." Ph.D. dissertation, Computer Science Dep., Yale Univ., New Haven, Conn., 1977.

FRAS79 FRASER, C. W. "A compact machine independent peephole optimizer." In *Proc. 6th Ann. ACM Symp Principles of Programming Language* (San Antonio, Tex., Jan. 29–31), ACM, New York, 1979, pp. 1–6.

GANA80 GANAPATHI, M. "Retargetable code generation and optimization using attribute grammars." Ph.D. dissertation, Computer Science Dep., Univ. of Wisconsin—Madison, 1980.

GANA81a GANAPATHI, M., AND FISCHER, C. N. "Bibliography on automated retargetable code generation." *ACM SIGPLAN Not.* **16**, 10 (Oct. 1981), 9–12.

GANA81b GANAPATHI, M., FISCHER, C. N., SCALPONE, S. J., AND THOMPSON, K. C. "Linear intermediate representation for portable code generation." Tech. Rep. 435, Computer Science Dep., Univ. of Wisconsin—Madison, 1981.

GANA81c GANAPATHI, M., AND FISCHER, C. N. "A review of automatic code generation techniques." Tech. Rep. 406, Computer Science Dep., Univ. of Wisconsin—Madison, 1981.

GANA81d GANAPATHI, M., FISCHER, C. N., AND HENNESSY, J. L. "Automatic compiler code generation." Tech. Rep. 225, Computer Systems Lab., Stanford Electronics Lab., Deps. of Electrical Engineering and Computer Science, Stanford Univ., Nov. 1981.

GANA82 GANAPATHI, M., AND FISCHER, C. N. "Description-driven code generation using attribute grammars." In *Proc. 9th Ann. ACM Symp. Principles of Programming Languages* (Albuquerque, New Mex., Jan. 25–27), ACM, New York, 1982.

GIEG82 GIEGERICH, R. "Automatic generation of machine specific code optimizers." In *Proc. 9th Ann ACM Symp. Principles of Programming Languages* (Albuquerque, New Mex., Jan. 25–27), ACM, New York, 1982.

GLAN77 GLANVILLE, R. S. "A machine independent algorithm for code generation and its use in retargetable compilers." Ph.D. dissertation, Deps. of Electrical Engineering and Computer Science, Univ. California, Berkeley, Dec. 1977.

GLAN78 GLANVILLE, R. S., AND GRAHAM, S. L. "A new method for compiler code generation." In *Proc. 5th ACM Symp. Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25), ACM, New York, Jan. 1978, pp. 231–240.

GRAH80 GRAHAM, S. L. "Table-driven code generation." *IEEE Comput.* **13**, 8 (Aug. 1980), 25–34.

GRAH82 GRAHAM, S. L., HENRY, R. R., AND SCHULMAN, R. A. "An experiment in table driven code generation." In *Proc. SIGPLAN82 Symp. Compiler Construction* (Boston, Mass., June 23–25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 32–42.

GRIE71 GRIES, D. *Compiler Construction for Digital Computers.* Wiley, New York, 1971.

JOHN75 JOHNSSON, R. K. "An approach to global register allocation." Ph.D. dissertation, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1975.

JOHN77 JOHNSON, S. C. "A tour through the portable C compiler." UNIX documentation, Bell Telephone Laboratories, Murray Hill, N. J., 1977.

JOHN78 JOHNSON, S. C. "A portable compiler: Theory and practice." In *Proc. 5th ACM Symp. Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25), ACM, New York, Jan. 1978, pp. 97–104.

KNUT68 KNUTH, D. E. "Semantics of context-free languages." *Math. Syst. Theor.* **2**, 2 (June 1968), 127–145.

LAND82 LANDWEHR, R., JANSOHN, H.-ST., AND GOOS, G. "Experience with an automatic code generator generator." In *Proc. SIGPLAN82 Symp. Compiler Construction* (Boston, Mass., June 23–25), *ACM SIGPLAN Not.* **17**, 6 (June 1982), 56–66.

LEVE80 LEVERETT, B., AND SZYMANSKI, T. G. "Chaining span-dependent jump instructions." *ACM Trans. Program. Lang. Syst.* **2**, 3 (July 1980), 275–289.

LEVE81 LEVERETT, B. W. "Register allocation in an optimizing compiler." Ph.D. dissertation and Tech. Rep. CMU CS-81-103, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., Feb. 1981.

LEVE82 LEVERETT, B. W. "Topics in code generation and register allocation." Tech. Rep. CMU-82-130, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., July 1982.

MCKE70 MCKEEMAN, W. M. "Peephole optimization." *Commun. ACM* **8**, 7 (July 1965), 443–444.

MILL71 MILLER, P. L. "Automatic creation of a code generator from a machine description." Tech. Rep. MAC TR-85, Project

MAC, Massachusetts Institute of Technology, Cambridge, Mass., 1971.

NEWC75 NEWCOMER, J. M. "Machine independent generation of optimized local code." Ph.D. dissertation, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1975.

NEWE69 NEWELL, A., AND ERNST, G. W. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, 1969.

PERK79 PERKINS, D. R., AND SITES, R. L. "Machine independent Pascal code optimization" In *Proc. ACM SIGPLAN Symp. Compiler Construction* (Denver, Colo, Aug. 6–10), *ACM SIGPLAN Not.* **14,** 8 (Aug. 1979), 201–207.

RAIH80 RAIHA, K. J. "Biblioigraphy on attribute grammars." *ACM SIGPLAN Not.* **15,** 3 (March 1980), 35–44.

RICH71 RICHARDS, M. "The portability of the BCPL compiler." *Softw. Pract. Exper.* **1,** (1971), 135–146.

RIPK77 RIPKEN, K "Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe." Tech. Rep. TUM-INFO-7731, Computer Science Dep., Technische Univ., Munchen, Munich, Germany, July 1977.

RITC78 RITCHIE, D. M., AND KERNIGHAN, B. W. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N. J., 1978.

RUDM79 RUDMIK, A., AND LEE, E. S. "Compiler design for efficient code generation and program optimization." In *Proc. ACM SIGPLAN Symp. Compiler Construction* (Denver, Colo, Aug. 6–10), *ACM SIGPLAN Not.* **14,** 8 (Aug. 1979), 127–138.

SETH70 SETHI, R., AND ULLMAN, J. D. "The generation of optimal code for arithmetic expressions." *J. ACM* **17,** 4 (Oct. 1970), 715–728.

SHIL78 SHILLINGTON, K. A., AND ACKLAND, G. M. (Eds.) *UCSD Pascal Version 1.5* Institute for Information Systems, Univ. of California, San Diego, 1978.

SMIT70 SMITH, D. C. MLISP, Stanford Artificial Intelligence Project Memo. AIM-135, Stanford Univ, Stanford, Calif., 1970.

SNYD75 SNYDER, A. "A portable compiler for the language C." M.Sc. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., May 1975.

STEE61 STEEL, T. B., JR. "A first version of UNCOL." In *Proc. Winter Jt. Computer Conf.,* **19** (1961), pp. 371–378.

STRO58 STRONG, J., WEGSTEIN, J., TRITTER, A., OLSZTYN, J., MOCK, O., AND STEEL, T. "The problem of programming communication with changing machines: A proposed solution." *Commun. ACM* **1,** 8 (Aug. 1958), 12–18.

SZYM78 SZYMANSKI, T. B. "Assembling code for machines with span-dependent instructions." *Commun. ACM* **21,** 4 (Apr. 1978), 300–308.

WEIN73 WEINGART, S. W. "An efficient and systematic method of compiler code generation." Ph.D. dissertation, Computer Sciences Dep., Yale University, New Haven, Conn., 1973.

WICK75 WICK, J. D. "Automatic generation of assemblers." Ph.D. dissertation, Computer Science Dep., Yale Univ., New Haven, Conn., 1975.

WILC71 WILCOX, T. R. "Generating machine code for high level programming languages." Ph.D. dissertation, Tech. Rep. 71–103, Dep. Computer Sciences, Cornell Univ., Ithaca, N. Y., 1971.

WULF75 WULF, W., JOHNSSON, R. K., WEINSTOCK, C. B., HOBBS, S. O., AND GESCHKE, C. M. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

WULF79 WULF, W. A., LEVERETT, B. W., CATTELL, R. G. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., AND SCHATZ, B. R. "An overview of the production quality compiler-compiler project." Tech. Rep. CMU-CS-79-105, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., Feb. 1979.

WULF80 WULF, W., LEVERETT, B. W., CATTELL, R. G. G., HOBBS, S. O., NECOMER, J. M., REINER, A. H., AND SCHATZ, B. R. "An overview of the production-quality compiler-compiler project." *IEEE Comput.* **13,** 8 (Aug. 1980), 38–49.

YOUN74 YOUNG, R. "The coder: A program module for code generation in high level language compilers." M.Sc. thesis, Computer Sciences Dep., Univ. of Illinois, Urbana-Champaign, Ill., 1974.

Received September 1981; final revision accepted September 1982.