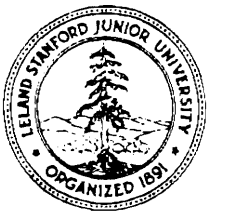
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055



Automatic Compiler Code Gene ration

Mahadevan Ganapathi,
Charles N. Fischer,
John L. Hennessy

Technical **Report No. 225**

November 1981

This research was supported by National Science Foundation Grant MCS78-02570, and the Office of Naval Research under a contract to the University of California Lawrence Livermore Laboratory, LLL Contract 9628303.

Automatic Compiler Code Generation

Mahadevan Ganapathi,
Charles N. Fischer,
John L. Hennessy

Technical **Report No. 225**

November 1981

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

A classification of. automatic code generation techniques and a survey of the work on these techniques is presented. Automatic code-generation research is classified into three categories: formal treatments, interpretive approaches and descriptive approaches. An analysis of these approaches and a critique of automatic code-generation algorithms are presented.

Key Words and Phrases: Code Generation, Intermediate Representation, Machine Description, Machine-dependent Optimization, Code Generator Generator, Compiler Compiler.

Charles Fischer and Mahadevan Ganapathi are associated with the Computer Science Department, University of Wisconsin - Madison. Mahadevan Ganapathi's current affliation is Intel Corporation, Santa Clara, California.

1. INTRODUCTION

Code generation is the process of mapping some intermediate representation of the source program into assembly or binary machine-code. This complex task involves selecting machine instructions to correctly implement programming language constructs in line with the following considerations:

(1) Evaluation order of programming language statements,

(2) Frequency analysis of variables and register allocation,

(3) binding source-language variables to storage locations (i.e. operand binding), and packing variables into machine locations,

(4) accessing variables (i.e. setting up run-time display linkage during procedure calls and operand addressing),

(5) instruction selection (i.e. evaluating arithmetic and Boolean expressions, executing control constructs and evaluating predicates without storing an explicit Boolean result), and

(6) machine-dependent optimization

Previous research in code generation can be broadly classified into three categories: formal treatments, interpretive approaches and descriptive approaches. Formal treatments thus far have considered arithmetic expressions only. Interpretive approaches are improvements over ad-hoc code generation (because only P+M translators are needed to implement P languages on M architectures). But in such schemes machine descriptions are intermixed with the code generation algorithm. Retargeting thus requires changing the code generator for every new machine. Descriptive approaches separate the machine description from the code generation algorithm, providing a higher degree of

portability. In such schemes, pattern matching is used to replace interpretation.

2. INTERPRETIVE APPROACHES

Two-level translation schemes were suggested to help design portable compilers [Ers58, Str58, Ste61]. Code is produced for a virtual machine and is then expanded into real machine instructions. Such schemes use code generation languages specifically designed to describe the code generation process along with the target machine instructions (GCL [Els70], BCPL Ocode [Ric71], ICL [Wil71, You74], CGPL, CGGL [Don73, Don79], Pascal P-code [Amm77]). There are two classes of interpretive approaches. The first category comprises of straight-forward hand-written interpreters that implement a **1-to-1 or 1-to-** many mapping between the virtual machine and the real machine. instructions. Hand-written P-code translators [Amm75] and the UCSD Pascal code **interpreter** [Shi78] fall under this category. They require a lot of code generation decisions to be made by the compiler front-end. The other class of interpretive approach is more pragmatic: interpretation is done by a controlled set of schemas. In this schema-driven interpreter, Ucode [Per79] dictates transformations on an abstract machine. **Interpretation is carried out** by observing the **state** of this machine and then obtaining the correct code from the schema. **In effect, an m-to-1** mapping is obtained when interpreting the abstract machine onto the real machine. This scheme is explained in detail towards the end of this section. While hand-written interpreters are a distinct improvement over ad-hoc methods, they suffer from serious limitations:

(1) Due 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 (e.g. whether a hardware stack exists) in one virtual machine. Interpreters tend to be very large and complex (In Elson and Rake's implementation [Els70], macros had to be paged in from disk).

(2) Code generation languages are closely tied to a specific language or machine. Thus they cannot be considered truly portable.

(3) The description of the target machine is mixed with the code generation algorithm; the description cannot be changed without changing the algorithm.

(4) The implementor must perform a tedious case analysis of code sequences and make all low-level decisions as to 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.

(5) 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 (i.e., expressions) and Boolean values that need only be tested (i.e., predicates),
   (c) branch chaining and other flow-dependent optimizations [Wul75].

We now consider further the schema-driven interpretation scheme. The code generation scheme is based on two concepts:

(1) The translation from UCode to the target machine code is controlled by a set of schemas.

(2) The translation process is tracked by using a map between the virtual Ucode 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 these come from the fact that little or no context is used in expanding a particular macro. Alternatively, extra context can be incorporated with a great deal of special casing usually done by the programmer in a conventional code generator.

The approach used by UGEN differs because the state of the virtual machine records the necessary context at any time. A particular UCode operation may cause the code generator to alter the state of the virtual machine. Thus, th'e machine dependent portion of the code generator is isolated in the schemas and the code generation algorithms deal with a parameterized version of the target machine (parameters allow handling of issues such as register allocation and differing word lengths) and the virtual machine state.

The schemas essentially specify a mapping between the virtual machine and the target machine; the complexity of the mapping is directly related to the complexity of the target machine. The use of transformations on the virtual machine results in the collapsing of several UCode operators into a smaller sequence of target machine instructions, without requiring the programmer to be aware of, or to construct, all possible instruction sequences leading to the same virtual state.

The virtual machine consists of several components:

(1) The instruction stream of UCode instructions.

(2) The static global data area.

(3) The stack allocated activation record frames (which are symbolically addressed).

(4) One or more portions of memory that can be addressed using pointers.

(5) A computation stack.

The UCode machine instructions include

(1) Loading and storing from the memory areas and stacks to the computation stack.

(2) Performing computations on the computation stack.

(3) Creating or destroying activation frames.

(4) Changing the choice of instructions (calls, jumps, returns, etc.)

The machine instructions appear in postfix format. **Instruction** operands have data type, memory type, length, and address fields associated with them.

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 which may use 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, e.g. register contents.

The code generation algorithm is also responsible for:

(a) Register allocation.

(b) Loading and storing between target machine memory types (registers and memory): this is always directly tied to register allocation.

(c) Tracking the state of the target machine to eliminate subsumed instructions.

The schemas are chosen on the basis of the UCode instruction to be executed and the virtual computation stack of the machine. The schemas may be parameterized by the fields of the instruction or components of the virtual stack.

An Example

The Source: `A[i] := A[i] + 1;`

Ucode:

```
LDA     A       ;load address
LOD     I       ;load i
INDEX   4       ;index an array of component length 4
ADD
LDA     A
LOD     I
INDEX   4       ;index an array of component length 4
ADD
ILOD            ;indirect load
LDC     1
ADD             ;A[i] + 1
ISTR            ;indirect store
```

Instruction	Virtual Machine State	Code
LDA A	Addr(A)	
LOD I	Addr(A) | I	
INDEX 4	Addr(A) | R1	L R1,I SLA R1,4 ;load i & shift
ADD	$Offset_A$(R1,FP)	;Offset@-(A)=offset from FP
ADD	$Offset_A$(R1,FP) | $Offset_A$(R1,FP)	
ILOD	$Offset_A$(R1,FP)|R~2	L R2,$Offset_A$(R1,FP) ;indexed load
LDC 1	$Offset_A$(R1,FP)|R 2 |1	
ADD	$Offset_A$(R1,FP)|R 2	INCR R2 ;increment R2
ISTR		Str R2,$Offset_A$(R1,FP)

A version of UGEN has been done for two machines (Motorola 68000 and the DEC-20) with a third underway (VAX). The results have been quite encouraging. The code on the 68000 is comparable to that for the C compiler including the post optimizer for the C compiler. On the DEC-20, the code is about 20% faster than the one pass Stanford/Hamburg Pascal compiler. The time to port the 68000 was less than 1 man month (ignoring about 1 week in learning the instruction set and 1 week in making low level C interfaces and learning the C linkage convention).

3. FORMAL TREATMENTS

Newcomer [New75] uses means-end-analysis [New69] to generate code templates (not machine instructions) from a parse tree and a set of operators, His scheme is very restrictive and of very little practical importance because: .

(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 due to
   (a) a possibly inadequate set of operators, or
   (b) a limitation of the depth of search performed by means-end-analysis (a 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. Sometimes, weeks of computer time are required to analyze even 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. 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 (i.e. wherever '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. Aho & Johnson show that this algorithm requires time linear in the number of tree nodes and exponential in the number of instruction and addressing mode choices at each point.. *The shortcomings of this model are:

(1) It only deals with arithmetic expression trees excluding common sub-expressions.

(2) Only mathematically clean instructions are dealt with, avoiding asymmetric registers and special instructions found in real computers.

4. HAND-WRITTEN CODE GENERATION ALGORITHMS

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.

Miller made the first attempt to isolate machine-dependent issues from the code generation algorithm [Mil71]. Source language is mapped to two-address code sequences, which are macros written in MIML (Machine Independent Macro Language). These macros specify the actual code generation algorithm, e.g.

```
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, e.g., macros Iadd and Fadd in OMML (Object Machine Macro Language) form the description of addition on the target machine. Thus, for the IBM-360:

```
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, r) 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 with a set of permitted states only. The designer is therefore required to specify transitions between memory and registers so that the code generator automatically moves to a permitted state if needed (e.g.

movement of an operand in storage to a register 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, however, is too restrictive because it deals with expression evaluation and very simple addressing schemes only: it does not allow indexing, auto-increment, or indirect addressing.

Weingart introduced pattern matching to avoid interpretation [Wei73]. Target machine characteristics are encoded into a single pattern tree that 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 because:

(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.** He tried to generate the tree from a machine description automatically, but did not succeed very well.

(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 (e.g. using 'increment' instead of '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).

Snyder [Sny75] attempted to write a portable compiler for the language C [Rit78] (but he did not succeed very well). 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.' 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 to a large extent ignored object code optimization.

A number of Snyder's ideas are used by Johnson in his successful implementation of the portable C compiler [Joh77, Joh78]. Templates and a template-matching algorithm form the central idea around which code generation is designed. Templates specify:

(1) the operator of the subtree (e.g. an assign-op),

(2) the desired result location on the target machine (e.g. 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: the number of temporaries and scratch registers needed for implementing the subtree,

(5) a rewrite rule specifying how to replace a subtree by another, and

(6) the machine instruction(s) to be emitted on a successful match; the opcodes and operands are, in general, macros that are expanded into assembler mnemonics of the target machine (e.g. emit Integer-Opcode, 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) through (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, for example,

```
a += b becomes a = a + b
x++    becomes ((x += 1) -1)
```

Feldman uses C's code generator in his implementation of the portable Fortran 77 compiler [Fel79]. Most register and temporary allocation is taken care of by the code generator. However, mapping the different flavors of Fortran integer variables to C's types and generating the necessary type conversions are not easy tasks. Furthermore, Fortran's power 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:

(1) 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.

(2) The intermediate representation (IR) 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 very easy to map references that are neither local nor global (i.e. at an intermediate nesting level) to the C compiler's back-end.

(3) 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.

(4) On a mismatch, machine-dependent rewrite rules call the code generator for possible tree alterations. Such rules potentially can produce infinite loops.

(5) Not much thought is given to machine specific optimizations (but there is a specific plan for a post pass by a peephole optimizer). Condition codes are not saved between expressions.

(6) About one-third (-1000 lines) of the code generator needs to be rewritten to do a transport.

5. TABLE-DRIVEN CODE GENERATION

To suit a variety of target architectures, a lot of flexibility and tuning of the code generation algorithm is usually necessary. Lately, research has concentrated on providing this flexibility by 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, instruction selection. These problems interact strongly. In particular, the choice of an optimal set of instructions to implement a Direct Acyclic Graph (DAG [Aho77]) depends on register availability etc.; 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 68000.

Fraser [Fra77] uses ad-hoc rules (coded as MLISP [Smi70] subroutines) to minimize machine dependency in code generation. He uses XL, a machine-independent IR that may need to be adapted to accommodate new source languages or target machines. Rules are used to perform storage allocation (more on this topic in the next section) and in this process XL is rewritten into. ISP' [Wic75] (a modified version of ISP [Bel71]). Code generation then consists of matching this ISP' form with machine instruction patterns that are also 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, load non-accumulator operands into accumulators. The rules, of course, do not guarantee that a code sequence will eventually be found.

Fraser performs syntactic analysis of ISP [Bel71] descriptions at code generation time to recognize stack operations, macros that set condition codes, index registers and accumulators. Rules (subroutines in **MLISP)** are used to allocate storage for variables and 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 where small integers can be stored in a half-word or a byte, the allocation rule for integers is inefficient. On architectures with no index registers (e.g. **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 (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:

(1) Rules compromise generality for efficiency. They are based on the observation that many computer architectures are similar in design (as Wick postulated in assemblers). The same rules are not usable for diverse architectures: often completely new rules are necessary. Some rules such as "load non-accumulator operands into accumulators** could potentially contradict other parts of a compiler (such as the-register allocator). **In** Fraser's scheme, redundant loads and stores are unavoidable.

(2) **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 the best alternative.

(3) The code generator is very slow: the implementation in Lisp on a PDP-10 KA10 generates one line of assembler code each second.

Cattell [Cat78, Cat79, Cat80, Wu179, Wu180] 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, further matches to be recursively performed (e.g. a statement within a control construct). Templates are grouped into schemata representing the context (e.g. flow result, 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 therefore be composed recursively to match an entire program tree. Operand mismatches are forcefully resolved by a subtargeting operation that 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 (details in the next section). Multiple matches are handled by sorting the alternatives with decreasing preference and choosing the first (e.g. x <- x+1 occurs before x <- 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 to derive code sequences for **IR** operators that do not have equivalent machine opcodes (e.g. subtraction on the PDP-8) by using tree equivalence axioms (e.g. **DeMorgan's** laws, relations between addition and subtraction) and heuristic search. The Mop assertion templates are then augmented with such derived sequences and pseudo-operations (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').**

			code emitted
IR:	**c <-** a & b		
goal:	c **<-** c **& ~d** **'bic d,c'**		
mismatch:	a with c, decomposition	**'c <-** a'	mov **a,c**
	b with **~d,** transformation	**'b <- ~~b'**	
IR:	c **<-** c & **~~b**		
goal:	c **<-** c & **~d** **'bic d,c'**		
mismatch:	**~b** with d, decomposition	**'d <- ~b'**	
heuristic search obtains 'corn'			
IR:	d **<- ~b**		
goal:	d **<- ~d** 'corn **d'**		
mismatch:	b with d, decomposition	'd **<-** b'	mov **b,d**
match:			**com** d
match:			**bic d,c**

Attempts are also made to match the IR with other potentially useful instructions ('c <- a & b' with 'c <- ~c' ('corn')) but the search is too deep and 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, a conditional jump requires several transformations), so Cattell suggests doing such searches before code generation and tabulating the results for the code generator. In practice, it is very hard and time consuming (if not impossible) for such an axiomatic approach to automatically derive code sequences for floating point operations or doing a '2n'-bit arithmetic on an 'n'-bit machine (e.g. 16-bit arithmetic on the Intel 8080 or 32-bit arithmetic on the PDP-11). The Intel 8080 has no explicit 'branch on greater' or 'branch on equal' instructions. It has '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,

e.g. '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:
```

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:

(1) 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' [Wul75] 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 [Joh75]. Interfacing the code generator within Bliss' framework might therefore be hard.

(2) Templates are part of the code generation algorithm because some result sequences specify further matches to be performed. It is therefore hard to alter the templates without changing the algorithm. .

(3) 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. Even if it does not' there would be unnecessary moves.

(4) 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-111780 that have both two-address and three-address operations for a single IR operator, the optimal choice

depends on whether the operator is commutative and the operands are destroyable (e.g. a <- b + a).

(5) 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.

(6) Code sequences that are produced do not make effective use of all machine resources. Special case subsumption operations such as auto-increment 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 [Mck70, Fra79, Fra80] may solve some of these problems but there may be conflicts with other parts of the compiler (such as the register allocator [Rud79]).

Ripken [Rip77] uses an extended version of Aho & Johnson's algorithm to generate optimal code from expression trees. His IR 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 attri-

buted tree patterns with a set of attribute transformation (AT) rules (more details in the next section). A pre-pass to code generation maps simple (i.e. non-aggregate) 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 & Johnson's) determines the order of Al-rule applications and which AT rule is to be applied at each node. The difference between Aho & Johnson's algorithm and Ripken's is that Ripken considers real instruction sets with several register classes and addressing modes. Transfer operations (not only '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 & 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 (e.g. (bytes, 15), (words.' 16), (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.**

E.g. addition **on** the **Intel** 8080: template: **+ 01 02 -> 03**

```
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.
```

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.

```
e.g.    :=  Registergair  HL_pair
                mov  2*i, H      (i=address of register pair)
                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 (which represent individual statements rather than the basic blocks of [Aho77]) of different expression trees. These inefficiencies include redundant loads and stores, and failure to take advantage of auto-increment/decrement possibilities.

Glanville [Gla77, Gla78, Gra80] made a significant breakthrough, not entirely perfect but a big step. The generated code is reasonable and the code generator is fairly retargetable. He 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 and they form 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 = l "inc rl"** specifies that an addition of **1 to register1** (with the sum going to the same register) can be obtained by an 'inc r1" instruction. A one-to-one mapping is as-

sumed between productions (serving 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 usually ambiguous, a modified LR(l) parsing algorithm is used. The table driven code generator is automatically derived from instruction patterns (more on this technique in the next section). In practice, reasonably compact tables are obtained and also, because standard context-free parsing techniques (which forbid backup) are used, a linear time algorithm is obtained. Multiple matches produce shift-reduce or reduce-reduce conflicts and are resolved heuristicly. 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.

Glanville's machine description (Polish prefix expressions) is not very formal. Different data types of the target machine (e.g. bytes, words, floating point) and special addressing modes (e.g. auto-increment' auto-decrement) are not used. The code generating IR parser is automatically constructed from the instruction-set description using an LR(1)-like table constructor [Aho76]. Correctness of the code generator is emphasized. 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. Instruc-

tion grammars are analyzed for uniformity (all operands being 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 (i.e. no error actions are encountered). Although some of the semantics (e.g. 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 an earlier and more primitive form of attributes that are used in [Gan80]. Sometimes semantic restrictions (e.g. 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. All uses of a pattern are restricted and the input to the code generator does not satisfy these restrictions. 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.

While Glanville's scheme is very efficient (easily the fastest among comparable code generation schemes) and provably correct, it is not completely portable because:

(1) 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 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 address computations as opposed to 24 bits). Since storage allocation and binding issues are avoided, any change in the implementation of (e.g.) 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.

(2) Very good code cannot be generated by purely context-free expansion (e.g. '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 (e.g. in an addressing context, explicit addition is performed in the intermediate representation and in the generated code instead of using indexing).

(3) 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).

(4) Glanville ignored machine-dependent optimization issues. The code genera-

tor does not worry about information retention (e.g. values left in registers from previous computations). Thus, redundant load and store elimination, recognition of equivalent locations, subsumption of addition or subtraction via auto-increment and decrement are not done.

In [Gan80] attribute grammars are used to specify translations from an intermediate representation (a linear representation of parse-trees) to a target code representation of programs. A code generator may be obtained automatically for any compiler using attributed parsing techniques. The code generator is also easily retargetable to different machine architectures. Implementations of a code generator based on this model exist for the VAX-111780 and the PDP-11/70.

Code Generation Goals

The goals are:

(1) structure the code generation process so that target machine dependency does not taint other phases of a compiler. Interfacing the code-generator package with other phases of a compiler should therefore become considerably easier.

(2) devise a simple and clean model of code generation and machine-dependent optimization using attribute grammars [Rai80]; ideally, one that is simpler and cleaner than Newcomer's and Cattell's'means-end-analysis model [New69].

(3) retain the speed and efficiency of Glanville's [Gla77] approach by using a fundamentally single-pass code generation scheme.

(4) include machine-dependent optimizations that have not been included in other portable code-generators. These optimizations include choosing between three-address and two-address instructions, subsuming (via auto-increment) additions widely separated from the current instruction (in effect, 'floating' an addition across many instructions), subsuming subtractions via auto-decrement in a similar fashion, removing redundant loads and stores' replacing memory references by register references, delaying operand movement into costlier storage locations and span-dependent optimizations [Szy78, Szy80]. Such optimizations are hard to incorporate as a separate pass of peephole optimization [Fra79, Fra80] since the instruction could have effects outside the window (e.g. condition-code setting and register contents).

Attributed-Prefix Intermediate Representation

The design of an intermediate representation (IR) plays an important role in compiler portability and code generator efficiency. An attributed Polish Prefix representation is used as the IR. The rationale behind this choice, its relative standing with respect to other IR proposals in the literature and the detailed design of attributed prefix IR is explained in [Gan81b]. The main goal in this design is to minimize the effort required to retarget compiler back-ends. The level of the IR serves to demarcate language-dependent issues from machine-dependent issues. The use of a Polish Prefix representation facilitates efficient pattern matching between the IR and the attribute grammar of the target machine (explained below). Attributes provide the flexibility needed to accommodate the diversity found in programming languages. Further-

more, they provide ease of interface between machine-independent aspects of compilers (such as live/dead variable analysis) and machine-dependent issues. Thus, they are influential in guiding the code generator to produce efficient target code.

Attribute-Grammar Machine Description

For purposes of pattern matching and instruction selection' the instruction set of the target architecture is represented as a set of attribute-grammar productions. These productions form the input to a program that generates a code generator for the target machine. All productions are of the form 'LHS -> RHS', where LHS stands for lefthand side, RHS for righthand side. The LHS is a single non-terminal usually appearing with synthetic attributes. The RHS contains:

(1) terminals with synthetic attributes (prefixed by a '↑'),
(2) non-terminals with synthetic attributes,
(3) disambiguating predicates [Mi177] (underlined) with inherited attributes (prefixed with a '↓') and
(4) action symbols (capitalized) with synthetic and inherited attributes.

Attribute occurrences may be constants or variables. Constant attributes (with the exception of self-defining constants) are enclosed within quotes. The kinds of productions needed for an entire code generator can be broadly classified into addressing-mode productions and instruction-selection productions. Although examples in this section pertain to the PDP-11/70 and the VAX-11/780, the technique is generally applicable and feasible, as demonstrated by our specific implementations.

Addressing-mode productions:

Each production has an RHS specifying the pattern of an IR addressing mode.

The production creates the proper machine address (in an action symbol). For example, the following production is used to specify an index addressing mode on the PDP-11/70 or a displacement addressing mode on the VAX-111780 (',' denotes indexing in the **IR):**

Address↑a -> , Disp↑b Base↑c ADDR (↓b↓c↑a) .

"Disp" represents a local variable with attributes specifying the machine data type and offset from a frame pointer. These attributes are determined when **IR** variables are bound to locations in the target machine. The attribute variable "c" specifies the base (or display) register of the **IR** variable. The action symbol ADDR synthesizes an address for a datum on the target machine. The attribute "a" represents this address: **In** this implementation' it has the following components:

(1) a base register,

(2) an offset from the base register,

(3) an optional level of indirection,

(4) an index register (if any) and

(5) the name of a variable (in case it is global).

These components may vary when the code generator is retargeted to new machines. However, for a variety of machines, including the VAX-111780, IBM-370 and the PDP-11/70, this structure seems to suffice. The addressing mode productions determine the components used.

Instruction-selection productions:

Each production has an RHS specifying a pattern in the **IR** and the corresponding code sequence to be emitted on a match. The LHS may be an explicit result location (a register or a memory location), in which case it specifies the

data type of the result, or a condition code location, or simply a non-terminal place-holder. Consider addition on the VAX-11/780. There are two-address and three-address add op-codes. Furthermore, the increment instruction can be used for adding one. For a byte datum' these three forms of addition are expressed as follows:

```
Bytefr  -> + Byteta Byte↑r IsOne (↓a) IsTemp (↓r) EMIT (↓'incb' ↓r)
        -> + Byte↑r Byte↑a IsOne (↓a) IsTemp (↓r) EMIT (↓'incb' ↓r)
        -> + Byteta Bytefr TwoOp (↓+↓a↓r) EMIT (↓'addb2' ↓a ↓r) .
        -> + Bytetr Byteta TwoOp (↓+↓a↓r) EMIT (↓'addb2' ↓a ↓r)
        -> + Byteta Bytefb GETTEMP (↓'byte' ↑r) EMIT(↓'addb3'↓a↓b↓r)
```

The first and second productions specify the addition of 1 to "r". Both productions are needed to represent the commutativity of addition. **In case either** production is selected, the op-code incb (increment byte) is emitted. The non-terminal on the LHS (Byte) and its attribute (r) specify the.data type and address of the result respectively. The third and fourth productions specify two-address addition of "a" and "r" using op-code addb2. Similarly, the last production specifies three-address addition of "a" and "b" using op-code addb3. **In** this case, the sum is stored in "r" that is obtained from action symbol GETTEMP. The location "r" may be a free register or the LHS of an assignment statement whose previous contents need not be preserved.

An addition of two **IR** data in byte format will match the RHS of one of these productions. The choice of the RHS is determined by attribute values and the disambiguating predicates. If an operand is 1 then an incb instruction is selected. Productions three through five handle addition of a constant other than **1 as** well as variables. **In** an assignment context, a global attribute

keeps track of the target address of the assignment statement. The disambiguating predicate TwoOp evaluates to true if either operand is the target of assignment or its value need not be preserved after addition. Consequently, a two-address addb2 is selected. If TwoOp evaluates to false, then a three-address addb3 is selected.

Code-Generator Generator and Code Generation Algorithm

The attribute grammar for the target machine is input to a code-generator generator (CGG) whose output is a specific code generator for the machine. The code generator consists of a set of transition tables and a driver for these tables. This driver serves as a push-down automaton that parses the IR form. Instructions (machine operations) are selected during parsing. To transport compilers to a new machine, the attribute-grammar description of that machine is given to the CGG. Transition tables for the machine are then automatically obtained and the same driver is used. The CGG constructs a context-sensitive parser [Wat74, Wat77]. Bottom-up parsing is preferred to top-down parsing (for a rationale, the interested reader is referred to [Gan80]). The attributed bottom-up parser with disambiguating predicates employs the standard LR(k) parsing loop with added code to manipulate attributes. Since the set of attributes is relatively small (the prototype code generator uses ten attributes in all, covering many architectures), the parser does not need to be able to handle fully general attribute sets.

An Example

In this section we illustrate examples of using attributed parsing to generate

VAX-111780 code. These examples emphasize the PDP-11/70 and the VAX-11/780. However' attributed parsing is a generally applicable technique for compiler code generation and optimization usable on almost any architecture.

Consider the translation of the statement "A := B - 1"

on typical architectures with several lengths of integers (e.g. byte, word, long). The IR after storage-binding is:

*- Address↑a - Addresstb Addresstc

where the attribute variable a includes 'long' and address information for A' b has 'word' and other information for B, and c has 'byte' and 1 as the actual value. We now trace the parsing process.

(1) The following production is recognized:

Long↑x -> Addresstx IsLong (↓x)

This production matches any Address with attributes that declare that its type is long. Because x appears twice in this production, it is implicitly copied . from Address to Long. Thus, the attributes of A are carried forward. We now have

. = Long↑a - Addresstb Addresstc

(2) Next, production

Wordfx -> Addresstx IsWord (↓x)

matches Addresstb, because it is a word. The local attribute variable "x" is instantiated as "b". We reduce the IR further to:

:= Long↑a - Word↑b Addresstc

(3) Now, the following production is matched:

Long↑x -> Word↑y ConvToLong (↓y) GETTEMP (↓'long'↑x) EMIT (↓'cvtwl'↓y↓x)

We convert from word to long format by first allocating a temporary (say re-

gister r_1) through the action symbol GETTEMP, then issuing a 'convert word to long' instruction through the action symbol **EMIT.** We now have reduced the **IR to :**

. = Longta - Long↑r_1 Address↑c

(4) Next, the constant **1** is reduced to a Long by the production

Longfx -> Address↑x IsLong (↓x)

We have reduced the **IR to:**

. = Longta - Long↑r_1 Long↑c

(5) The following production is now matched:

Longtx -> - Longtx Longty IsOne (↓y) IsTemp (↓x) **EMIT** (↓'decl' ↓x)

This production describes a special-purpose decrement instruction, applicable only if the second operand is the constant **1.** We have reduced the **IR to:**

. = **Longta** Long↑r_1

(6) Finally, the following production is matched:

Instruction -> := Longtx Longty NotEquate (↓x↓y) DELAY (↓'movl' ↓y ↓x)

-> := Longtx Longty (no code emitted)

NotEquate evaluates to false if **"x"** and **"y"** are equivalent locations and consequently, reducing by this production does not produce any code. DELAY is a variant of **EMIT** that can delay generation of an instruction pending future instructions. **In** this case, the move of r_1 to A is delayed so that future references to A can be replaced by r_1. Also' the move may be completely suppressed if, for example, another assignment to A is encountered before it is referenced.

The use of attribute values to control parsing the **IR** allows us to significantly improve the quality of generated code with little effort. For example, in step (5), above, if the left operand had not been in a temporary, we would have generated two instructions (a 'move', then the decrement). A better code

sequence would be to use the VAX's three address format to generate, for example, "sub13 I' B, r_1".

If the x's attributes show it is not a temporary, IsTemp evaluates to false' and recognition of this production is blocked. Instead, an equivalent (but longer) instruction is generated by this alternate production:

Long↑x -> - Long↑y Long↑z GETTEMP (↓'long'↑x) EMIT (↓'sub13' ↓z ↓y ↓x)

Optimization

This implementation attempts to express optimization in a non-procedural form, replacing the hand-coding of machine-dependent optimizations by the use of attribute grammars. Our intention is not to expand on the vast store of optimization techniques, but to cleanly organize 'tricky' machine-dependent optimizations (especially those optimizations that are both popular and effective). Some of these optimizations, such as removing redundant loads/stores and using arithmetic shifts instead of multiplications, are commonly used in compilers with the help of specially hand-coded routines. Others, such as the use of 'sob' (subtract-one-and-branch) on the PDP-11/70 and auto-increment, are not commonly used. Our implementation formalizes machine-dependent optimization within the attributed parsing framework under the following categories:

(1) addition of attribute grammar productions to incorporate special instructions,
(2) delaying generation of code till the end of a basic block,
(3) code subsumption within addressing modes,
(4) deletion of redundant code and
(5) code alteration (back-patching) using information gathered after instruction selection.

Experience with Cg

Implementations of Cg exist for the PDP-11/70, VAX-111780 and the **Intel** 8086.

The generator occupies 86K bytes on the PDP-11/70 and 115K bytes on the VAX-111780. The 11/70 implementation generates about 30 lines of assembler code per second (real time), while the VAX version generates about 50 lines per second. In contrast, the C compiler produces about 4.0 lines per second on the PDP-11/70 and about 60 lines per second on the VAX-111780.

We have retained the speed of Glanville's code generator by using a one-pass linear parsing technique. It is faster than those implemented by Fraser and Cattell and is expected to be much faster than that proposed by Ripken [Gan81a].

The code-optimization results are very encouraging. Cg produces code comparable to hand-written assembler code for user programs. It produces code far superior to the unoptimizing C compiler on both the PDP-11/70 and the VAX-111780. In most cases, Cg produces code that uses 35-50% less space than the code produced by the C compilers prior to "-0" optimization. Even with the peephole optimization performed by the C compiler, the code produced by Cg is usually 5-10% smaller.

An amazingly wide variety of code-generation optimizations can be realized in a highly modular manner. Almost all optimizations can be realized by addition of new attribute grammar productions. Furthermore, when the code generator is retargeted to a new machine, most of the basic (non-specialized) productions can be retained. In particular, a simple (but un-optimized) code generator can be implemented for a machine easily and rapidly. As time permits, and the need arises, improvements can be included by adding new rules to the machine description and automatically re-generating the code generator. The chief

difference between an optimized and an unoptimized code generator is how carefully and thoroughly the production rules reflect the details and complexities of the target machine.

The shortcomings of the implementation are:

(1) Register allocation is not driven by machine description. (This. shortcoming is questionable in view of recent research in architectures such as the **Intel iAPX-432** that do not contain general purpose registers. **If** these architectures become more prevalent in the future, register allocation is an irrelevant issue.)

(2) Most optimizations are not extended beyond basic blocks.

(3) The code generator is not intelligent enough to automatically derive code sequences in cases of non-orthogonal instruction sets (Cattell's scheme could possibly derive code sequences).

(4) Certain compiler generated temporaries need not be allocated memory space since they can reside in registers for the entire duration of a procedure activation. **In** the implementation, although memory space is allocated for such temporaries, this storage space is never used because redundant stores are never emitted (i.e. if there is no need to do a 'store' then no moves are emitted). Thus, code quality is not affected.

6. SUMMARY AND CONCLUSIONS

There has been much theoretical research in code generation. Very 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. But for such schemes machine descriptions are intermixed with the code generation algorithm. Retargeting thus requires changing the code generator for every new machine. UGEN is a schema driven code generator that uses techniques based on Johnson's approach. It simplifies pattern matching by modifying the state of its abstract machine enabling it to compile a more complex pattern match that would normally.be hard to handle under Johnson's scheme. It works, ports are easy and fast and it produces fairly good code. Its philosophy of reversibility at all stages is good but with its use of postfix, some preliminary information is needed to obtain optimal code.

Descriptive 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). Fraser, Glanville, Ripken and Cattell have tried to automatically derive code generators from a machine description though their

methods are very much different. Fraser's rule based system is inefficient and its portability is questionable, 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. 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 point of view 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.

As an extension and natural successor to Glanville's work, attribute grammars are used to specify translations from a linear representation of parse trees [Gana81] to a target code representation of programs. The intermediate representation is at a higher level than that proposed by Glanville. A more complete machine description is used by adding attributes to instruction-set productions (including machine data types and addressing mode productions). Storage allocation is viewed as part of the issue of portable code generation. The following contributions of this technique are noteworthy:

(1) Conventionally, it has been an extremely difficult task to organize the different phases of a compiler. The design of a flexible (attributed) Polish-prefix intermediate representation and attributed parsing framework seems to have solved this problem. Almost all machine-dependent aspects of compiler code generation (with the notable exception of packing) have been isolated to a single software package.

(2) Attributes in the intermediate representation have provided a convenient

interface between the machine-independent and the machine-dependent parts of a compiler. They have helped in solving the difficult problem of operand binding (an issue not addressed by other researchers in the area of automatic code generation).

(3) Machine-dependent and peephole optimizations have been incorporated in a routine, cheap and reliable manner within the attributed parsing framework of code generation. This attempt seems to be the first to organize optimization within any framework.

(4) This technique has demonstrated code generation retargetability. Three code generators have already been completed (for the PDP-11/70, VAX-111780 and Intel 8086).

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 in a realistic way. Table-driven code generation should be viewed as one more step in the advancement towards automated compiler generation where such compilers can be as competent as today's hand-written compilers!

Acknowledgements

The help provided by Johannes Heigert, Raphael Finkel and William Leland in improving the readability of this paper is appreciated.

Bibliography

[Aho73] A.V. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling", Vols. 1 and 2, Prentice-Hall, Inc., 1973.

[Aho76] A.V. Aho and S.C. Johnson, "Optimal Code Generation for Expression Trees", JACM Vol. 23 No. 3 pp. 488-501, 1976.

[Aho77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Amm75] U. Ammann, K. Nori, K. Jensen and H. Nageli, "The Pascal (P) Compiler Implementation Notes", Institut for Informatik, Eidgenossische Technische Hochschule, Zurich 1975.

[Amm77] U. Ammann, "On Code Generation in a Pascal Compiler", Software-Practice and Experience, Vol. 7 No. 3 pp.' 391-423, June/July 1977.

[Bel71] C.G. Bell and A. Newell, "Computer Structures: Readings and Examples", McGraw Hill, 1971.

[Cat78] R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD thesis, Carnegie Mellon University, 1978.

[Cat79] R.G.G. Cattell, J.M. Newcomer and B.W. Leverett, "Code Generation in a Machine-Independent Compiler", ACM SIGPLAN Symposium Compiler Construction, Denver, Colorado, August 1979.

[Cat80] R.G.G. Cattell, "Automatic Derivation of Code Generators. from Machine Descriptions**, ACM Trans. Programming Languages and Systems, Vol. 2, No. 2 pp. 173-190, April 1980.

[Don73] M.K. Donegan, "An Approach to the Automatic Generation of Code Generators", PhD thesis, Rice University, Houston, Texas, 1973.

[Don79] M.K. Donegan et.al., "A Code Generator Language", ACM SIGPLAN Symposium Compiler Construction, Denver, Colorado, August 1979.

[Els70] M. Elson and S.T. Rake, "Code Generation Technique for Large Language Compilers", I.B.M. Systems Journal Vol. 9 No. 3 pp. 166-188, 1970.

[Ers58] A.P. Ershov, "On Programming of Arithmetic Operations", CACM Vol.1 No. 8 pp. 3-6, 1958.

[Fel79] S.I. Feldman' "Implementation of a Portable Fortran 77 Compiler using Modern Tools", ACM SIGPLAN Symposium Compiler Construction' Denver, Colorado, August 1979.

[Fra77] C.W. Fraser, "Automatic Generation of Code Generators', PhD thesis, Computer Science Department' Yale University'. New Haven, Conn., 1977.

[Fra79] C.W. Fraser' "A Compact Machine Independent Peephole Optimizer"' Principles Of Programming Languages, 1979.

[Fra80] C.W. Fraser and J.W. Davidson, "The Design and Application of a Retargetable Peephole Optimizer", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 2, 1980.

[Gan80] M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars"' PhD dissertation, University of Wisconsin - Madison, 1980.

[Gan81a] M. Ganapathi and C.N. Fischer, "A Review of Automatic Code Generation Techniques', Tech. Report #406, University' of Wisconsin - Madison, 1981.

[Gan81b] M. Ganapathi et al., "Linear Intermediate Representation for Portable Code Generation", Technical Report #435, University of Wisconsin - Madison, 1981.

[Gla77] R.S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers', PhD thesis, University of California' Berkeley, Dec. 1977.

[Gla78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation', Conference Record Fifth ACM Symposium Principles of Programming Languages, Jan. 1978.

[Gra80] S.L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol. 13 No. 8 pp. 25-34, August 1980.

[Joh75] R.K. Johnsson, "An Approach to Global Register Allocation", PhD dissertation, Carnegie-Mellon University, 1975.

[Joh77] S.C. Johnson, "A Tour through the Portable C Compiler", Bell Telephone Laboratories, 1977.

[Joh78] S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symposium Principles of Programming Languages, pp. 97-104, Jan 1978.

[Mck70] W.M. Mckeeman, "Peephole Optimization", CACM Vol. 8 No. 7 pp. 443-444, 1970.

[Mil71] P.L. Miller, "Automatic Creation of a Code Generator from a Machine Description", **M.I.T.** Tech report MAC TR-85, **1971.**

[Mil77] D.R. Milton' "Syntactic Specification and Analysis with Attribute Grammars", PhD thesis' University of Wisconsin-Madison, **1977.**

[New69] A. Newell and G.W. Ernst, "GPS: A Case Study in Generality and Problem Solving", Academic Press' **1969.**

[New75] **J.M.** Newcomer' "Machine **Independent** Generation of Optimized Local Code", PhD thesis' Computer Science Department, Carnegie Mellon University, **1975.**

[Per79] D.R. Perkins and R.L. Sites, "Machine **Independent** Pascal Code Optimization", Proceedings of the ACM **SIGPLAN Symposium on** Compiler Construction, Denver, Colorado, August 6-10, **1979.**

[Rai80] **K.J.** Raiha, **Bibliography on Attribute Grammars", ACM **SIGPLAN** Notices, Vol. 15 No. 3 pp. 35-44, Mar **1980.**

[Ric71] M. Richards, "The Portability of the BCPL Compiler", Software Practice and Experience, **1,** pp. 135-146, **1971.**

[Rip77] K. Ripken, "Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe", Technische Univer. Munchen, Munich, Germany, **July 1977.**

[Rit78] D.M. Ritchie and B.W. Kernighan, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New **Jersey, 1978.**

[Rud79] A. Rudmik and E.S. Lee, "Compiler Design for Efficient Code Generation and Program Optimization", ACM **SIGPLAN Symposium** Compiler Construction, Denver, Colorado, August **1979.**

[Shi78] K.A. Shillington and G.M. Ackland (editors), "UCSD Pascal Version **1.5", Institute** for Information Systems, University of California, San Diego **1978.**

[Smi70] D.C. Smith, **MLISP,** Stanford Artificial **Intel1** igence Project Memo **AIM-135,** Stanford University, **1970.**

[Sny75] A. Snyder, "A Portable Compiler for the Language C", master's thesis, MIT, Cambridge, Mass., May **1975.**

[Ste61] T.B. Steel, **Jr.,** "A First Version of UNCOL", **Proceedings** WJCC, **19,** pp. 371-378, **1961.**

[Str58] J. Strong et. al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM Vol.1 No. 8 pp. 12-18, 1958.

[Szy78] T.G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions", CACM, Vol. 21 No. 4 pp. 300-308, April 1978.

[Szy80] T.G. Szymanski and B. Leverett, "Chaining Span-Dependent Jump Instructions", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 3, 1980.

[Wat74] D.A. Watt, "L.R. Parsing of Affix Grammars", PhD thesis, University of Glasgow, Report #7, 1974.

[Wat77] D.A. Watt, "The Parsing Problem for Affix Grammars", Acta Informatica, Springer Verlag, 1977.

[Wei73] S.W. Weingart, "An Efficient and Systematic Method of Compiler Code Generation", PhD thesis, Computer Sciences Department, Yale University, 1973.

[Wic75] J.D. Wick, "Automatic Generation of Assemblers", PhD Dissertation, Yale University, 1975.

[Wil71] T.R. Wilcox, "Generating Machine Code for High Level Programming Languages",. Tech. report 71-103, PhD thesis, Department of Computer Sciences, Cornell University, 1971.

[Wul75] W. Wulf et. al. "The Design of an Optimizing Compiler", American Elsevier Publishing Co., 1975.

[Wul79] W. Wulf et. al., "An Overview of the Production Quality Compiler-Compiler Project", Tech. Report CMU-CS-79-105, Carnegie Mellon University, Feb. 1979.

[Wul80] W. Wulf et. al., "An Overview of the Production-Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8 pp. 38-49, August 1980.

[You74] R. Young, "The Coder: A Program Module for Code Generation in High Level Language Compilers", M.S. thesis, Computer Sciences Department, University of Illinois, 1974.