A Survey and Critique of Some Models of Code Generation

R. G. Cattell
Computer Science Department
Carnegie-Mellon University

November 1977

Abstract

Various work on code generation is discussed, particularly from the point of view of simplifying and/or automating the derivation of this phase of compilers. Code generators, which typically translate an intermediate notation into target machine code in one or more steps, have been relatively ad hoc as compared to the first phase of compilers, which translates a source language into the intermediate notation. Progress in formalizing the code generation process is summarized, with the conclusion that considerably more work remains. Future directions of research are suggested.

Acknowledgements

I'd like to thank Bill Wulf, Mario Barbacci, Joe Newcomer, and Raj Reddy for their comments on earlier versions of this paper.

This work was supported in part by the Defense Advanced Research Projects Agency under contract number F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

1. Introduction

The classical compiler has two main phases: *recognition*, and *generation*. The *recognizer*, which translates a source program into an internal notation, typically consists of lexical, syntactic, and static semantic analysis. A code *generator* then translates the internal notation into object code, in one or more subphases, with various degrees and kinds of optimization.

Although there is more work to be done, considerable progress has been made toward formalization and mechanization of the recognition phase of compilation. This paper is a survey and analysis of some recent work toward doing the same for the code generation phase, which heretofore has remained unpleasantly ad hoc and has received less theoretical attention. The recurrent theme of these reviews is the goal of simplifying and automating the construction of this phase of compilation. Particular attention is paid to the contributions and shortcomings of the various approaches with respect to this goal.

Unfortunately, the assumptions and compilation models of authors in this area differ widely, making a generalized summary of their work virtually impossible. Instead, the approaches are discussed individually, drawing comparisons where appropriate. Then, in the figures given at the end of this paper, a few salient features will be compared along some common dimensions. The reader may want to refer to these tables as the approaches are presented.

2. General Background

Wilcox, in his thesis at Cornell [1971], provides one of the first comprehensive discussions of code generation for a high-level language compiler. In particular, he abstracts methodology from his work on PL/C, a PL/1 compiler developed at Cornell. As well as describing the general structure of the compiler and code generator, he discusses in some detail addressing and data reference, register management, and the translation process between his internal notations, the APT and SLM. The APT is an Abstract Program Tree which is essentially a parse tree of the language, but oriented towards sequencing of operations rather than the phrase structure of the input. The APT is linearized, then translated into a sequential SLM (Source Language Machine) notation which is essentially an assembler-like notation especially oriented towards executing PL/1. After some optimization on the SLM, it is translated into 360 machine code.

Some of Wilcox's work, such as the idea of an APT, and his scheme for data description, have significance for general code generation. Unfortunately for our purposes, a large part of the work is specific to the machine and language, with few hints as to how it might be generalized. Also, the structure of the compiler precludes certain kinds of optimization; more on this later.

A recent thesis by Simoneaux [1975] provides a more readable and general discussion of compiler organization, intermediate representations, and optimization, in the same light as this paper. However this thesis constitutes little or no new results, and therefore will just be pointed to as a reference here.

Weingart [1973] also presents a summary of code generation techniques, and a promising formalization of the code generation process; his work will be discussed in section 4.

In general, there have been two kinds of approaches to more automatic production of code generators. The first is the development of a specialized language for code generators, with built-in machinery for dealing with common details of the process. The second extreme is the development of a program to build a code generator for a language from a purely structural and behavioral machine description. Rather than being mutually exclusive, these *procedural* and *descriptive* language approaches, respectively, represent points in a continuum of degrees of automatic programming. It is sometimes difficult to classify an approach along this dimension; for example, authors sometime refer to "descriptions" which are really just tabular representations of procedures. The *descriptive* approach is probably ultimately more desirable; not surprisingly, there has been more success with the former approach.

3. The Specialized Procedural Language Approach

Elson and Rake's [1970] GCL, Generate Coding Language, was used as a procedural specification of code generation in a large PL/1 compiler. In the implementation described in their article, GCL is translated into an internal code and interpreted. Code generation is performed in one pass, a tree walk, in which node-specific routines (called OPGEN Macro Definitions, written in GCL), are invoked at each node. Previous passes have expanded certain operations (such as indexing and type coercions) and performed global optimizations on the tree. Elson and Rake seem to have been reasonably successful with this approach, except perhaps for compilation speed, which would be improved by compiling GCL, or by simply improving the implementation, which they suggest.

A recent thesis taking the specialized language approach was written by R. P. Young at the University of Illinois [1974]. The organization and internal notation he uses is similar to that of Wilcox (who is now at U of I and served as his advisor). The code generation process is described in ICL (Interpretive Coding Language), which is stored and interpreted by the Coder. The input to the Coder from the compiler front end is in the form of a sequential SLM notation. ICL is based on templates for each SLM instruction, having capabilities for decision-making, automatic handling of various data accesses (an elaboration of Wilcox's *data descriptors*), and register allocation. ICL turns out to be hard to read and write, so Young proposes a higher-level TEL (Template Language) which is compiled into ICL.

Young's approach is a considerable improvement over simple machine code macro substitution for source language operations. He recognizes that "Simple substitution

into a code skeleton ... leads to inadequate code", which is what led him to ICL.[1] This is not to say that he has solved all the problems of a specialized language for code generation, however. There are, in particular, some serious questions with respect to the organization of the system. For example, the decision to do all the translation in one pass makes forward references difficult (reserving space for address calculation for an instruction and "then either no-op instructions or a branch around the unused space is inserted..." patching it later!) In discussing optimization, he suggests that the SLM instructions should somehow be "rearranged to perform the required computation in a more efficient manner", but this makes difficult the optimizations which depend on the original APT, or the peculiarities of the machine architecture. It could also be argued that the intermediate SLM notation is not necessary at all, that generation could proceed directly from the APT.

Implementation of Young's coder was unfortunately not done, although an ICL description for the 360 was proposed.

There is a comparatively long history of compiler-writing systems, dealing with code generation to lesser or greater extents. These efforts have all taken the specialized language approach. An early example is Feldman [1966], who uses a language FSL for description of programming language semantics (code generation). In combination with a syntax description, it was used in a compiler-compiler. It was somewhat primitive, but did deal with errors, forward references, and simple storage allocation. Feldman and Gries [1968] and McKeeman et al [1970] survey more advanced translator writing systems. More recently, White [1973] and Ganzinger et al [1977] describe compiler-generation systems along this line. Traditionally, compiler-generation systems have been weak on automating the later stages of compilation, specifically code generation. But as the formal methods and grammars applied have become better understood and more powerful, their scope has gradually been evolving towards the later stages of compilation.

Ripken [1975] describes the intermediate code generator currently used in the latter compiler-generator, MUG2 (Ganzinger et al [1977]).

There are three distinct phases of code generation in his scheme, as shown in Figure 2. The first phase takes as input an APT-like tree with attributes attached to the nodes, constructed and optimized in earlier phases of MUG2. This APT is translated into a zero-address virtual machine code. A Tree-Walking Push-Down Transducer (TPDT) performs this translation, using a set of code templates (indexed by APT operator) in a specialized notation allowing testing of attribute values attached to nodes and output of code. In the second phase, the zero-address machine operations are translated into an SLM-like n-address form (which Ripken calls the Intermediate Language, IL). This is a fairly simple procedure, in which the zero-address virtual machine, with its several stacks, is simulated, and SLM-instructions emitted for each

---

[1] Inadequate code is generated because simple substitution doesn't allow analysis of the context in which a construct appears. For example, we might want to generate different code for an addition if the result is used as an address (indexing), or if one of the arguments is 1.

zero-address instruction (except PUSHes and POPs). Finally, the SLM instructions are translated into target machine code, using macros for the SLM instruction provided by the user (as in Miller [1971]).

The MUG2 group has gone much farther than previous compiler-generator projects in several ways. They have been able to integrate formalizations of all phases of the compiler, including the later stages, and optimizations in particular. This has largely be made possible by the richness and numerousness of their intermediate notations. These allow each phase to operate on the notation most appropriate and efficient for its use. The APT representation in general, and attributes in particular, have been used to make the description of various tree transformations and optimizations possible in a concise way. Also, the zero-address machine phase make it possible to neatly separate temporary allocation, because temporaries and their lifetimes are made explicit on a stack.

Like earlier compiler-generation systems, MUG2 avoids machine dependence until quite late in the compilation process. This is a mixed blessing. It means that changing machine could be as easy as changing the the SLM macros in the last stage. However, if one wants to make machine-dependent optimizations, particularly those which involve recognizing early in the compilation the tree segments that should be performed by certain target instructions,[2] then we are not in as good of shape. This is particularly troublesome because simplification rather than automation of machine dependence is the approach taken. However, work on MUG2 is still under way, and the authors should have more advances forthcoming.

4. More Automation, More Descriptive Languages

P. L. Miller's thesis [1971] was probably the first attempt toward automating code generator production. His goal is statedly the descriptive language approach. Although he only attacked a portion of the complete problem, the limitations and applicability of his work are fairly clearly specified.

In Miller's model, construction of a code generator occurs in two phases. First, the language is "described" by a set of macros in MIML, a procedural Machine Independent Macro Language. These provide a machine-independent skeleton for a code generator. Then, the machine is described in OMML, a declarative Object Machine Macro Language, which is used to "fill out" the code generator skeleton. Specifically, the OMML specifies the registers and memory on the machine, instructions to move data between them, word size and alignment information, and most importantly, the instructions to emit for each "macro" (actually, an SLM-like instruction) produced by the MIML procedures. The SLM-like intermediate code is a sequence of two-operand virtual instructions, but is essentially a binary tree because each instruction is numbered and one of the operands may be [the number of] a previous instruction.

---

[2] for example, using a subtract-one-and-skip-if-zero for a loop, indexing for an addition, or a shift for multiply by two.

In order to eventually achieve both language and machine independence, it is important that the descriptions of the two be separated. In the past, it has not been necessary to attack this difficult task. Miller's model of code generation looks interesting in this respect, in that it might be possible to do this in his system, if the details can be ironed out.

His scheme falls short of our goals in two major ways. First, only two subproblems are attacked: arithmetic expressions, and data access; and the solutions are not completely general. For example, in data access, he allows only simple address calculation (index, displacement, base), assumes all of memory is addressable, and that registers have certain properties. Second, his approach cannot properly be called purely descriptive. Obviously, this hasn't been achieved with respect to the language, as the MIML macros are essentially programs. With respect to the machine, it is necessary to specify instruction sequences for each macro, which is debatably pure description, although it is a step in the right direction.

M. K. Donegan [1973] has attempted to generalize Miller's scheme in some ways. The heart of his system is a finite-state machine model of code generation: in the process of generating code for a node in a parse tree, the code generator enters various states, dependent on the properties of the operands and the machine registers available. Code is emitted and operations performed on the basis of this state; then another state is entered, or (in a *terminal* state) code generation for the node is complete.

Donegan points out that the state transition table with associated actions is easier to understand and debug than routines in a language tailored to code generation such as those described earlier in this paper. He suggests a language CGPL (Code Generator Preprocessor Language) for conveniently describing the states and actions, and a preprocessor which translates CGPL into a program in a high-level language such as PL/1. The program, when compiled, would constitute the code generator. The preprocessor must analyze the state transitions to generate a program utilizing the shortest paths to each terminal state, checking for input errors such as circular paths. It must also make some assumptions about register allocation and other tasks performed in the code generation process.

Donegan's biggest contribution is his characterization of the code generation process in such a simple way, a finite state machine. The simplicity of the model aids human understanding, as mentioned above, as well as making mechanization easier. His forte may also be his weakness, however: has the process been oversimplified? For example, the model as presented seems to have trouble with register allocation, when there are more than one or less than an effectively infinite number of registers. Donegan points out that "Any attempt to assign states to each possible register conditions would be rather hopeless" in such a case, and discusses various alternatives, none of which look very attractive. The finite state model as described seems to be more of a convenient mechanism for handling data access characteristics of instructions, than a panacea for code generation. However, more elaborate state tables look promising as an efficient notation for constructing or driving a code generator from the output of an analysis program.

Unfortunately, Donegan didn't implement the system, so the only assurance we have that problems are surmountable is the usual expression of confidence that the basic ideas are sound, that implementation would fill in details. This isn't particularly a fault of Donegan's work, but rather is true to more or less of a degree of all the work discussed in this paper: the whole task is too large to undertake these studies. Consequently, it is especially necessary to try to foresee how major details might be handled in these models.

Another contribution of Donegan is that he has neatly separated code generator generation time from code generation time, a concern with some other models. Donegan makes little mention of *language* independence, incidentally, presumably he had in mind dealing with multiple machines and a single language.

Concurrently with Donegan, Weingart [1973] developed a model of code generation that is more powerful, and demonstrably practical. His code generator uses a pattern tree, or discrimination net, to select code sequences for an APT-like input parse tree.

The code generator works as a coroutine to the parsing process. APT tokens (operator, symbol, and constant nodes of the tree) are passed to the code generator, which stores them on a stack, and traverses the pattern tree in an attempt to match the stack tokens (the stack is a preorder representation of the APT tree). The tree walking commences with the top node of the pattern tree; the nodes encountered are of two sorts:

(1) output actions, which occur at the leaves of the pattern tree, and specify instructions to generate. After processing such an output action, the code generator returns to the top of the pattern tree, using the next piece of APT input.

(2) match nodes, which specify an operator, operands (register, memory, constant), or one of the predefined classes of operators or operands; these are matched against the current input token. Upon a successful match, the tree walk continues at the right son of this pattern tree node, with the next input token; otherwise, it continues at the left son.

This pattern tree, used as a discrimination net, is a compact and efficient way to represent most of the machine-dependent information in a code generator. Weingart demonstrated the method by modifying an IMP-10 (Bilofsky[1973]) compiler (which already used this internal representation), to generate code for the PDP-11. There is still some machine-dependency not built-in to the tree, particularly with respect to instruction and data format, but this does not look infeasible for future work.

Weingart found that creating the pattern tree for the PDP-11, despite its simple and compact form, was quite difficult. This is not surprising, since all of the potential code sequences and patterns must be interwoven into the one pattern tree with the proper ordering to generate good code. This prompted Weingart to engage in the second part of his thesis, attempting to automatically generate the tree from a machine description. Unfortunately, his ideas here are not nearly as universal as his formalization of code generation; the problem has been vastly oversimplified.

Basically, it is assumed that each machine instruction corresponds to one language operator, with a small tree representing the action of the instruction. When that tree is found in the input, it is assumed that we should generate that instruction. It is immediately apparent that this is insufficient, as Weingart observes, in the cases where no instruction implements some portion of an input tree. To fix this, he adds "conversions". These correspond to instructions that "convert" a data item to a form in which one of the other insructions can operate on it, for example moving it into a register. The thesis is quite weak on conversions; the PDP-11 is only partially dealt with, and the scheme does not appear to be general enough to handle other machines. However, Weingart did succeed in writing a program which (for the PDP-11) automatically constructs the pattern tree from a special representation of the instruction actions, and automatically adds the conversions, given a human-generated input file which sets these up (e.g., specifies the necessary conversions).

Weingart does not show the code generator or examples of its output. The automatically generated pattern tree was not compared to the manually generated one, nor was the quality of the code discussed. The thesis is lacking in evaluation of results, with respect to both performance and generality.

More recently, Newcomer [1975] presents a more promising approach to the selection of code sequences. In his scheme, a set of *attributes*, *T-operators*, *language axioms*, and some other specifications are prepared for a machine and language, and these are analyzed to produce code templates (specifying code to generate for language constructs) for a compiler. The analysis uses APT-like language parse trees, but with attributes attached to the nodes. The attributes are selected by the user to specify useful properties and other information about the nodes in the tree. For example, they might specify the location to be used for a temporary required in the calculation of the node's result, "fudge factors" such as whether the negative of the originally specified result is to be computed for efficiency reasons, or common sub-expression information.

The *T-operators* are of two kinds: *terminal* T-ops, and *transformation* T-ops [my terms, not Newcomer's], which probably should be thought of as completely different animals, although they have been given the same name in the thesis.

Terminal T-ops specify trees for which code can be generated "immediately". When a tree is not of this form, transformation T-ops specify operations which can be performed to change its form. This might include generation of some code, for example, to load an operand into a register, or it might not, for example in transformations of the tree based on arithmetic properties. Like terminal T-ops, transformation T-ops specify the form and attributes of the trees to which they are applicable; in addition, they specify the form and attributes of the trees after the operation is performed. For efficiency, T-ops are indexed by the (top) language operator (L-op) of the trees to which they apply. It is also necessary to know the cost of using the T-op, and its requirements and effects on the global program state (for example, with respect to allocation of resources). Conveniently, cost can be measured as whatever the user desires to optimize; the only assumption made is that a cost can be given to each code sequence, and that these are additive.

Because Newcomer's thesis does not include a complete description of the code template generation scheme in one place, a summary of the process may be useful here. Although his mathematical notation would make this description much shorter, it has not been used, to improve readability. In the following, "template" means a language tree with attributes (a pattern) plus a code sequence which implements that tree.

Given an APT-like language tree L for which we wish to generate a code template (comments on this selection below), we search for code sequences as follows:

S1. Look up the top operation of L, to get all terminal T-ops which might be applicable, call this set P.

S2. If any are directly applicable (shape & attributes match L) then go to S7.

S3. Form a Preferred Attribute Set (PAS) by putting together the sets of attributes required (of the operands) by the terminal T-ops in P.

S4. Recursively perform this algorithm for each operand of the top node of L, passing the PAS as a parameter. Each son will return a set of templates with potential code sequences for their evaluation.

S5. Form the cross-product of possibilities for these templates, collecting all possible concatenations of the code sequences.

S6. Form the cross-product of this set with P (P gives the possible code sequences to implement the top L-op), but only include those operand evaluation sequences whose attributes satisfy the attribute (domain) requirements of the corresponding element of P.

S7. If a non-empty subset of these satisfy the PAS *we* were passed, return this subset (the first time the algorithm is performed, the PAS is empty and trivially satisfied. Otherwise:

S8. Attempt to satisfy the PAS by means-ends analysis similar to that used in GPS [Ernst & Newell, 1969], but exhaustive: For each template, find the difference between its requirements (attributes) and the PAS. Use this difference to look up transformation T-ops, and try applying them to get templates which satisfy the PAS.

S9. If successful with a non-empty subset of templates, return this set (with the attributes and code sequences, as modified by the transformation T-ops to satisfy the PAS). Otherwise, give up.

The language axioms are used to determine all other trees equivalent to the given one, and this algorithm is performed for each.

Although this algorithm could be used directly in a compiler, to generate code for a

tree L, it would be too expensive to do the analysis at compile time. Instead, it is envisioned that a driving program would somehow select a complete set (but no larger than necessary) of small language trees for which to find optimal code. This gives us templates (language trees with corresponding code sequences) which could then be used in the compiler. Both the selection of the trees and the use of the templates in the compiler are open for future research.

Note that it is possible for S9 to "give up". This should not normally happen if there exists any way to implement L on the machine and the T-ops provided are adequate. However, in order to avoid an infinite loop in which the algorithm repeatedly applies a sequence of T-ops which unknowingly return it to its original state, it was necessary to limit the depth of search. To avoid this, it would be necessary to leave the means-ends analysis paradigm and incorporate memory of previous search.

One of the best strengths of Newcomer's template idea is that it appears to be suitable for use in an optimizing compiler such as Bliss-11 (Wulf et al [1975]), in which code generation occurs in several phases. The templates can be used in an earlier phase to enumerate potential code sequences to guide, say, register allocation, and later, in the actual output of code.

The most conspicuous drawbacks of Newcomer's scheme are that it is *too* general, and that it is not general *enough*. It is probably *too* general in the sense that Newcomer has applied general but weak AI methods to the problem, and experienced difficulties with the amount of time required to analyze even simple trees. It might be possible to achieve tolerable performance through heuristics; he suggests what effects these heuristics would have to have, but gives no actual mechanisms. Another alternative is to use a stronger method with correspondingly stronger assumptions and built-in knowledge about code generation.

Like Donegan, Newcomer deals only with arithmetic expression trees. An open area for research is to determine whether control constructs and other operations can be incorporated into his scheme. Also open to further work is a way to automatically discover semantic equivalence of trees; the general problem is undecidable, an approximate solution is desired to reduce proliferation of identical cases.

Inventing the attributes, transformation & terminal T-ops, and other specifications for this system is still a non-trivial task, even though the case analysis is automated. A mechanism for deducing them from a machine description would be desirable, for this reason as well as others.

Concurrently with Newcomer, Snyder [1975], at MIT, wrote a thesis with somewhat less ambitious goals with respect to formalization and automation of code generation, but interesting in that it provides ideas for different generalizations. His paper describes the implementation of a compiler for the programming language C, in which a large part of the machine dependence of the code generation process has been abstracted into tables.

The first phase of the compiler code generation produces a 3-address code for an abstract C-machine. The second phase then translates this abstract machine code into

assembly code for the target machine (a macro expansion scheme is used, which takes advantage of properties of typical assembly languages). These two phases are analogous to Miller's MIML and OMML, but more refined, as we will see shortly.

The instructions for the abstract machine, which Snyder refers to as AMOPs (Abstract Machine Operations), are essentially L-ops which include the types (real, integer, pointer) of their operands. Pseudo-instructions are also permitted. These are basically keyword macros for storage allocation, procedure linkage, and other information. The addresses for the abstract machine are called REFs; a REF may specify an abstract register, static or stack variable, label, indirect reference, or constant.

The user provides a machine description, in the form of a set of specifications which map the abstract machine onto a real machine. These specifications are translated by a stand-alone program GT which generates tables for the code generator. The machine description maps the abstract machine onto a real machine in two ways. Part of the mapping is occurring in GT before the compiler is produced, the other part in the compiler itself; keep this in mind to avoid confusion.

The user's machine description consists of three kinds of specifications. First, the user defines the data storage and access structure to be used in the abstract machine (i.e., the target machine structure). To do this, he defines the register names, classes of registers, conflicts (either real or due to the abstract representation), memory alignment, and addressable unit sizes. Snyder uses a quite readable declaration-like notation to specify this information:

```
regnames(X0,X1,X2,X3,X4,A,Q,F);
class X(X0,X1,X2,X3,X4), R(A,Q);
size 1(char), 4(int,float), 8(double);
```

Then, the user specifies the data access properties of the machine instructions, in a "operand1, operand2, result" notation. For example,

```
+d: F,M,F
+i: R,M,1
*i: Q,M,Q [A]
```

specifies that: double precision add (+d) takes its first operand form the F-register, its second operand from memory, and leaves its result in the F-register; integer add (+i) takes its first operand in a register, the second from memory, and leaves the result in the first operand location; integer multiplication (*i) multiplies a memory location into the Q-register and destroys register A in the process.

Then, the user defines a mapping from abstract machine operations to assembly language, using a macro-expansion scheme. This is somewhat complex to describe here, but a simple example is:

```
+i: " AD#R  #S"
```

where #R and #S are macros which expand to the names of the result and second operands, respectively. An occurrence of integer addition might then expand to

```
ADA X
```

to add X to accumulator A, for example. When the features supplied for macro expansion are insufficient, it is possible to specify this in the form of a C routine.

Altogether, Snyder has made some hopeful advances toward our goals, particularly in the convenient specification of data storage and access. Unfortunately, it is necessary to perform the case analysis of code sequences, and to construct macros and C routines to perform the translation. Further work combining the successes of Snyder and Newcomer will eventually be necessary.

Snyder brings up an important point with respect to abstract machines, or in fact any intermediate notation between source and object languages:

> "If the abstract machine is of a high level (i.e., very problem-oriented), then the program [compiler] will be very portable, but the implementation of the abstract machine will be difficult. On the other hand, if the abstract machine is of a low level (i.e., more machine-oriented), then, unless it corresponds closely to the target machine, either the code will be inefficient or the implementation will be complicated by optimization code."

In the case of an UNCOL for multiple languages, there is yet another constraint, that it correspond to the *high* level language, for both implementation and code efficiency. Nevertheless, the motivation for an UNCOL is great, and this author believes that these will become more prevalent, probably with some language or machine restrictions (e.g., see Coleman [1974]). Snyder seems fairly definite, however, about sticking with one language.

The fact that Snyder fully implemented his system was a great asset in evaluating his ideas, because he was forced to fill in details, even if only for a couple machines. He was surprisingly successful in converting his compiler to generate code for another machine in a few days time. Of course, to achieve an implementation of an entire compiler in reasonable time it was necessary to simplify by restricting machine architectures and ignoring optimization to a large extent. For example, the register allocation is performed on the fly by a simple local algorithm.

The directions Snyder points out for further work are "bigger and better": more general machine model, more complicated languages, and optimization. Our goal of more fully automating code generator generation could be added to this list.

5. Related Formal Treatments

At the opposite pole from implementation, Aho & Johnson [1976] deal formally with the problem of generating optimal code from a parse tree, using a model similar to Newcomer's. They propose a model of machine and language, and show several interesting results. In particular, they show that a simple brute-force optimal code generation scheme is linearly proportional to the tree size and instruction set size. Unfortunately, they also make some unacceptable simplifying assumptions. For example, with respect to the language, they are only dealing with expression trees, and only ones without common subexpressions. With respect to the machine, an instruction must compute and store a single result into a register, registers must be of one symmetric kind, and of course they do not deal with any other processor state or intricacies of control, only the arithmetic instructions in one-to-one correspondence with arithmetic L-ops.

Samet [1975], in a voluminous thesis, presents a method for proving that a code generator has correctly translated a program; he has implemented the prover for translations of CMPLISP, a subset of LISP, into LAP, a PDP-10 assembly code. Samet's system proves that a particular program was correctly translated independent of the code generation process, rather than proving that a code generator is correct independent of the input program. This approach has the advantage that no knowledge of the the code generator, be it human or machine, is required, but, of course, the disadvantage that a new proof is required for each translation.

Basically, to prove that a set of machine instructions correctly implements a CMPLISP function, his system symbolically executes the machine instructions in such a way as to produce a tree representation of their effect, and this is then proved equivalent to the CMPLISP function. The cornerstone of the system is a *canonical* tree representation, obtained using a set of semantic equivalence axioms which Samet derived from McCarthy[1963]. Both the original program and the output of the symbolic execution are expressed in this form, as shown in Figure 2, reducing the equivalence/non-equivalence proof to a comparatively simple matching process.

The "machine description" in Samet's system consists of a set of LISP procedures, one for each LAP instruction. When the procedure for an instruction is executed, it updates a computational model as appropriate to the effect of the instruction. It also performs certain control operations; for example, when a condition is tested, either the condition value is known from previous results and that path is taken, or both paths are processed (there are mechanisms to stack alternatives and test for loops).

Samet's system has been oversimplified here. However, it should be clear that his ideas, in particular the symbolic execution, the axioms of semantic equivalence, and the canonical tree form, have potential applicability to our goals of *generating* a translator.

6. Summary

Figure 1 is a condensation into tabular form of many of the observations made in this paper. The abbreviated interpretations should be fairly self-explanatory at this point. The entries are classified according to the primary goal of their approach, as they were in this paper: simplifying or automating the generation of code generators (sections 3, 4, resp.). Note that all of the authors model code generation as a multiple-step process from source language to intermediate language-machine notation(s) to the target machine code. Some used an internal notation like Wilcox's SLM; others used an APT-like notation. Wilcox used both. Regardless of its form, the utility of the internal notation is that it provides a low-level but machine-independent UNCOL (Conway [1958]) representation, and/or it allows information and transformations to be more easily and concisely represented. One of the most important aspects of the various techniques, but the hardest to compare in any simple way, is the algorithm itself; these are simply summarized in a short phrase in the figure. The last two columns give a short evaluation of the approach.

A frequently confusing aspect of systems involving more than one level of interpretation or translation, such as many of those described, is what is being interpreted/translated by/into what at what point. For example, the approaches differ as to whether the translator is table-driven, or generated from tables, and the degree to which the case analysis of instructions is performed at translation time as opposed to translator construction time. A simple notation, developed by the author in conjunction with S. Saunders [1977], is used in Figure 2 to try to clarify these relationships for selected systems. The notation is quite simple; two primitives are used. An arrow from language L1 to language L2, with T connected to the side of the arrow, indicates that T *translates* the text (program) in language L1 to language L2. If L2 is missing, i.e., the head of the arrow is replaced by an electrical-engineering grounding symbol, then T *interprets* L1 (one can think of this as translating L1 into *action*, perhaps).

It should be noted that the assumptions the various authors make about the definition of code generation and its relationship to the rest of the compiler differ somewhat. For example, the stage at which register allocation is performed differs, and this affects the flexibility and information available to other stages (see comments on MUG2). The post-processing assumptions also differ, for example whether machine or assembly code is generated. Snyder even takes advantage of the syntax of the assembler language in building code generation macros. Although there are these differences, all of the authors have in common the "core" function of code generation: the selection of machine instructions on the basis of the intermediate language constructs.

In summary, this paper has attempted to point out the potential drawbacks and advantages of several models of code generation, particularly with respect to possibilities for simplifying and automating the creation of this phase of the compiler. Progress has been made, yet all of these works have non-trivial deficiencies with

respect to this goal, pointing directions for future research. It is likely that there will be more interest in this field in the near future.

Bibliography

Aho, A. V., and Johnson, S.C.: "Optimal Code Generation for Expression Trees", *JACM* 23, 3 (July 1976), pp 458-501

Coleman, Samuel S.: *JANUS: A Universal Intermediate Language*, PhD thesis, Electrical Engineering, University of Colorado, 1974

Conway, M. E.: "Proposal for an UNCOL", *CACM* 1,10 (October 1958) pp 5-8

Donegan, Michael K.: *An Approach to the Automatic Generation of Code Generators*, PhD thesis, Computer Science & Engineering, Rice University, 1973

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

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

Feldman, J.: *A Formal Semantics for Computer-Oriented Languages*, PhD thesis, Computer Science, Carnegie-Mellon University, 1964

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

Ganzinger, H., Ripken, K., and Wilhem, R.: "Automatic Generation of Optimizing Multipass Compilers", IFIPS Proceedings 1977, pp 535-540

McCarthy, J.: "A basis for a Mathematical Theory of Computation", in *Computer Programming and Formal Systems* (Eds: Baffort and Hirshberg), North Holland, 1963

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

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

Newcomer, Joseph M.: *Machine Independent Generation of Optimal Local Code*, PhD thesis, Computer Science, Carnegie-Mellon University, 1975

Ripken, Knut: "Generating an Intermediate-Code Generator in a Compiler-Writing System", Proceedings of the International Computing Symposium, 1975, pp 121-127

Samet, Hanan: *Automatically Proving the Correctness of Translations involving Optimized Code*, PhD thesis, Computer Science, Stanford University, 1975

Saunders, S., and Cattell, R.: "A Notation for Translation and Interpretation Systems", Blackboard in Science Hall 4114, March 1977.

Simoneaux, Donald C.: *High-Level Language Compiling for User-Defineable Architectures*, Electrical Engineering, Naval Postgraduate School, 1975

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

Weingart, Steven W.: *An Efficient and Systematic Method of Compiler Code Generation*, PhD thesis, Computer Science, Yale University, 1973

White, John R.: *JOSSLE: A Language for Specifying and Structuring the Semantic Phase of Translators*, PhD thesis, University of California at Santa Barbara, 1973

Wilcox, Thomas R.: *Generating Machine Code for High-Level Programming Languages*, PhD thesis, Computer Science, Cornell University, 1971

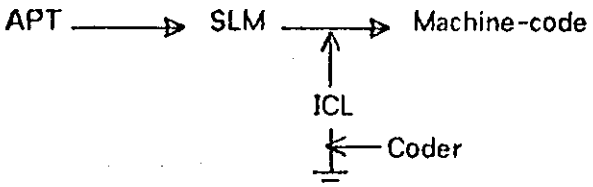
Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C.: *The Design of an Optimizing Compiler*, American Elsevier, 1975

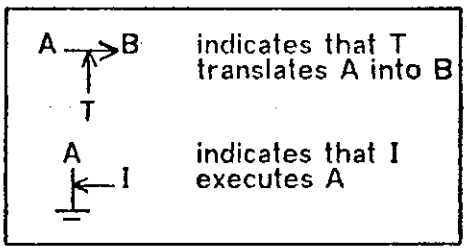
Young, Raymond: *The Coder: A Program Module for Code Generation in High-level Language Compilers*, MS thesis, Computer Science, University of Illinois, 1974

Work	Goal	Intermediate Notation(s)	Algorithm	Implementation	Advantages / Contributions	Limitations
Wilcox 1971	Formalize	APT&SLM	still relatively ad hoc	PL/C: 360	internal notations, describing locations	machine/language specific, mainly coded-in
Feldman 1964	Simplify	parse tree	user-written	FSL	semi-automated system for whole compiler	only some of compiler functions automated
Elson & Rake 1970	Simplify	APT	interpreted coding language	GCL	workable system for large language	slow. some generality, but still lacking
Young 1974	Simplify	APT&SLM	interpreted coding language	none	practical? advanced with respect to simplification	optimization; generality
Ripken 1977	Simplify	APT w/ attrs, &SLM	tree transfms & code macros	still under development	optimization. attributes & intermed. not. advances. integrated with compiler	machine dependence, automation
Miller 1971	Automate	linear APT	specialized macro expansion	limited	separated lang/mach dependence. macros	only addrsing, exprs handled, and in limited way
Donegan 1973	Simplify/ Automate?	APT	state-transition table	none	simplicity	practicality? limited w.r.t. constructs handled
Weingart 1973	Simplify/ Automate	APT	discrimination-net driven matcher/ code emitter	IMP10: PDP10	simple formalization; practical, efficient	limited w.r.t. automation
Newcomer 1975	Automate	low-level APT w/attributes	attr-diff driven exhaustive search	limited	search approach: automated case analysis	too slow. requires setup, not general enough w.r.t. machine
Snyder 1975	Simplify/ Automate	3-addr SLM	combined macros/ table driven	C-compiler	many machine properties tabularized	too C-/machine- specific; still much manual constr.
Aho & Johnson 1976	Formalize	APT using machine-ops	exhaustive search	none	some complexity limits, formal alg'm statement	too strong assms about machine arch. & ops
Samet 1975	Verify	Canonical (LISP) APT	transfm rules, symb simulation, matching	CMPLISP: PDP10	formal tree equivalence methods; verification	still limited w.r.t. control constructs

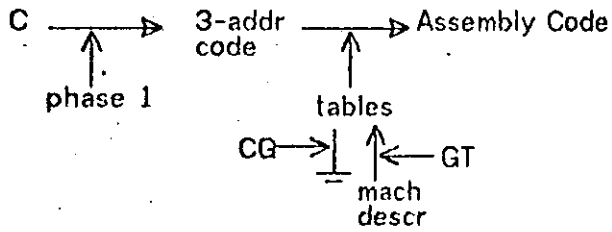
Figure 1. See text for explanations. Comments on contributions/limitations are relative to predecessors/successors, not w.r.t. all.

Young

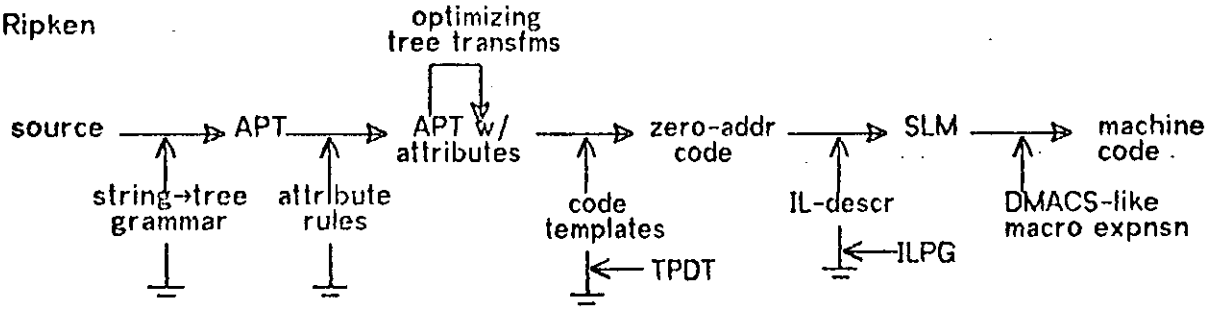




Snyder



Ripken



Samet

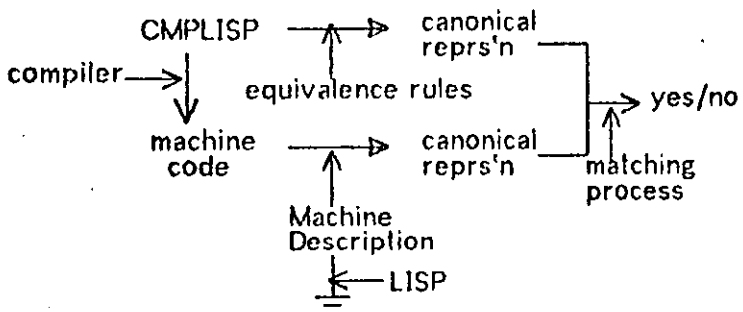


Figure 2. Interpretation & Translation diagrams for selected systems.