CMU-CS-79-105

An Overview of the Production Quality Compiler-Compiler Project

Bruce W. Leverett
Roderic G. G. Cattell
Steven O. Hobbs
Joseph M. Newcomer
Andrew H. Reiner
Bruce R. Schatz
William A. Wulf

Abstract

The Production Quality Compiler-Compiler (PQCC) project is an investigation of the code generation process. The practical goal of the project is to build a truly automatic compiler-writing system. Compilers built with this system will be competitive in every respect with the best hand-generated compilers of today. They must generate highly optimized object code, and meet high standards of reliability and reasonable standards of performance. The system must operate from descriptions of both the source language and the target computer. Bringing up a new compiler, given a suitable language description and target architecture description, must be inexpensive and must not require the assistance of builders or maintainers of the compiler-writing system itself. This paper describes the goals and methodology of the PQCC project.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

Introduction

The Production Quality Compiler-Compiler (PQCC) project is an investigation of the code generation process. The practical goal of the project is to build a truly automatic compiler-writing system; compilers built with this system will be competitive in every respect with the best hand-generated compilers of today. They must generate highly optimized object code, and meet high standards of reliability and reasonable standards of performance. The system must operate from descriptions of both the source language and the target computer. The cost of bringing up a new compiler, given a suitable language description and target architecture description, must be small -- on the order of three man-months, without assistance from builders, maintainers, or other persons deeply involved in the original system.

This research builds on previous work in two areas: code optimization, and compiler-writing systems ("compiler-compilers"). Section 2 describes the developments in those areas which we believe are relevant to our work. The relation of the PQCC project to previous (and current) work in these areas, or to the state of the art, can be summarized briefly:

- With some notable exceptions, previous work in compiler development tools has focused on the parsing and lexical analysis phases of compilation. Thus, "compiler-compiler" has come to be almost synonymous with "parser generator". We would like to extend the function of compiler-compilers, to include the production of optimizers and code generators. We believe that compiler-compilers will become far more popular as commercial software development tools when this is done.

- A great many code optimization techniques are known and have appeared in the literature. Nevertheless, construction of optimizing compilers is still nearly a black art. As a result such compilers tend to be expensive both to build and to use. They also tend to be unreliable, in that the object code produced after optimization may be either worse than the unoptimized code, or altogether wrong. We would like to organize, even to formalize, the huge "bag of tricks" associated with code optimization. Ultimately optimization should be done as routinely, cheaply, and reliably as parsing.

In the next section we will examine what these broad statements mean in specific terms, and we will outline the limitations and specific objectives of the PQCC project. Sections 3 and 4 give an overview of the strategy and methodology of the project. The remaining sections, until the summary, are discussions of each of the phases of the code generation process, in our proposed organization of it.

1. Goals of the project

1.1. Subgoals

Estimating the "quality" of a compiler is a complex matter. In order to evaluate the success of a compiler-compiler, we must have standards for each of several dimensions for a compiler produced by it: the quality of optimization (size and speed of code produced), the size and speed of the compiler itself, its reliability, the friendliness of its user interface, and perhaps other characteristics of it which are not user-visible, such as the time required to produce it and the costs of maintenance.

We consider the quality of code produced by the Bliss-11 compiler [25] to be a standard. That is, to whatever extent the compilers produced by the PQCC fail to meet this standard, our treatment of optimization will be considered incomplete. Several tools, of varying degrees of sophistication, are available for measuring the quality of code produced by a compiler, by comparison with other compilers [54, 55, 20]. A great deal of subjectivity remains in this measurement, in spite of these advances.

It is easier, but less useful, to set hard standards for compiler size and speed, and for the time required to produce the compiler. In the former areas, the Bliss-11 compiler sets standards that are reasonable but not high. We do not wish to constrain ourselves to particular limits of size and speed without having a thorough (quantitative) understanding of the costs of various code generation and optimization techniques. It seems clear that with this information we could choose a collection of techniques with acceptable performance. We estimate that the time required to produce a compiler for a different language and/or target machine, possibly including the time required to write the necessary descriptions of the language and machine, should be fewer than three man-months. But at this early stage of the research, it is not possible to be very precise about this goal.

1.2. Limitations

We have chosen with some care the means of keeping our research task to finite size. We have retained ambitious goals, as described above, with regard to the saving in the cost of compiler construction and the degree of code optimization attainable. However, we have placed real limits on the range of languages to be compiled and the range of target machines. In addition to these, we have set some other limits to our ambitions, having to do with the process of production of compilers; these will be described in section 3.2.

We consider only the "algebraic" languages. This is a vague term; it is easier to describe

what sort of language characteristics we do not expect to handle, than to describe what we do expect to handle. We exclude languages which provide the programmer with a well-supported and non-trivial data abstraction from the start, e.g. list processing (LISP), array processing (APL), string processing (SNOBOL). While these languages present interesting and worthwhile optimization problems, optimization and code generation must be done in terms of run-time subroutines rather than instructions. With our instruction-level machine descriptions and optimization techniques geared to saving individual registers or reducing code size in words, we have little to offer to the advancement of that variety of technology. We are more at ease with ALGOL, PL/I, FORTRAN, BLISS, PASCAL, C, Ada, and other general-purpose languages.

We consider a wide but still limited range of target machine architectures. These include one-address (PDP-8, NOVA), two-address (PDP-11), general-register (PDP-10, 1108, S/370), and three-address (VAX) architectures, and mixtures of these (almost all real computers are mixtures). We exclude pure stack machines, and other designs even more specialized to high-level languages, for reasons similar to the reasons for excluding special-purpose languages. For these architectures, many of the usual techniques for code generation and optimization, such as global and local register allocation and targeting, are partly or even wholly irrelevant. Techniques that are useful, on the other hand, are outside our domain of expertise. In addition, we do not consider the issues raised by vector machines, or by vector instructions on otherwise conventional machines, or by machines implemented with pipelining or other schemes for achieving parallelism.

Note that particularly poorly designed architectures are not excluded. Thus, we are prepared to deal with very asymmetric instruction sets, or systems in which supposedly general-purpose registers or other locations must be very nearly dedicated to particular purposes. (This latter category includes instruction sets in which common arithmetic operations may only read their operands, or write their results, in highly restricted categories of registers. The 8080/Z80 machines are a well-known example.) In spite of some visible progress in the art of instruction set design, new architectures appear every so often which are no better than those of twenty years ago, and we cannot afford to ignore these.

1.3. Supergoals

The goals of the PQCC project arise from several broader goals:

- We would like to attack the high cost of software, by making high-level languages more viable as software construction tools. Our approach to this is to try to decrease the costs of compiler construction, and to improve the quality (in terms of both effectiveness and reliability) of code generation. To some extent the PQCC project is an outgrowth of the Bliss-11 optimizing compiler project

[25].

- We are interested in applications of formal, symbolic descriptions of computer architectures. Other research in this area has investigated the generation of assemblers [56], verification tools [30], emulators, and other tools directly from computer architecture descriptions. The PQCC project is partly an outgrowth of the SMCD (Symbolic Manipulation of Computer Descriptions) projects [48].

- We believe there are serious unanswered questions about the design of intermediate languages for target machine independent compiling systems, especially when the production of high-quality code is an issue. We discuss some of these in section 4.2; our proposal, called TCOL, may be thought of as another entry in a field that includes the original UNCOL [52], BCPL OCODE [38], and JANUS [28].

- We would like to formalize code generation and optimization techniques, in much the same way that parsing [39] and flow analysis optimization techniques [45] have been formalized. More generally, we wish to know how techniques are related to one another, and for any technique, to have a good understanding of its costs, its applicability, and its effectiveness.

2. Research context

2.1. Compiler-writing systems

Many compiler-writing systems provide little or no support, or even a well-defined framework, for code generation. Typical of the usual approach is YACC [58]. This is a parser generator, which works from a language description in the form of a context-free grammar, or set of *productions*. With each production, the compiler writer may associate a subroutine he has written, and an indication of what arguments are to be passed to it. (Arguments may include references to tokens in the production.) These subroutines must do all the processing necessary for semantic analysis and code generation. They are written in a general-purpose language (C [9] or a dialect of Fortran); except for the ability to refer to tokens in the productions, no support is provided to the writer.

A slightly more interesting system is GCL, a language for writing code generators, used to build the PL/I optimizing compiler [19]. This is based on a general-purpose language, but it offers automatic facilities for generating labels in the code stream, allocating (very) temporary storage (and using it in the object code), and generating any of the target machine (S/360) instructions. More importantly, it offers a significant abstraction to the user, that is, a framework for the code generation process. A parse tree is assumed, and there is a set of primitives for extracting information from it and for "walking" it: using its structure to direct the flow of control. The GCL system is typical of another class of compiler-writing systems

[14], in which some attention has been paid to the code generation phase, but the system builder must still design and implement all the strategies, including most optimizations, himself. Sometimes these systems come with a pre-code-generation phase which does some rudimentary flow analysis on the program. In general, however, such systems are not adequate for, and even preclude, techniques that involve multiple phases, or non-trivial auxiliary data structures.

In recent years there have been several attempts to formalize the code generation process, with a view to building compiler-writing systems. These efforts are closer in their aims to the PQCC project than are the more popular compiler-writing systems mentioned above. A survey of these has been written by Cattell [10]. In addition, two efforts should be mentioned which were too recent to appear in the survey [22, 12]. We will discuss this code generation research in more detail in section 9.2.

2.2. Code optimization

The literature on code optimization begins at least as early as the Fortran I compiler [23]. Since then there have been several famous (or notorious) optimizing compilers about which landmark papers (or books) have been written [21, 19, 25], and dozens, perhaps hundreds, of papers about individual optimization techniques. ([2] is a fairly recent bibliography; partial catalogues can be found in [3, 50, 32].) Unfortunately, except for a few very well-researched areas, such as global (intraprocedural) flow analysis [4], and evaluation order and targeting [7], much of the published literature is difficult to use. Algorithms are frequently very dependent upon the target architecture, or upon the nature of the run-time support, or upon peculiarities of the source language; sometimes they can be generalized in non-obvious ways, but usually the authors make no attempt to do so. Frequently authors are unaware of other attempts to solve identical problems (a phenomenon not at all limited to literature on optimization, or even computer science). Performance measurements are exceedingly rare, whether of algorithm size or speed, or of effectiveness in improving object code quality.

As a result, standards of optimization in the compiler community are low. Most compilers perform no significant optimization at all. Good optimizing compilers are usually the result of expensive projects requiring many real years as well as man-years of effort. Many optimizers do not reliably produce correct code, causing users to routinely disable them when running the associated compilers. Others use "heuristic" algorithms, which sometimes make the resulting code worse than the unoptimized code. Useful optimization techniques are rediscovered many times by different compiler writers -- or, more often, not rediscovered.

Some significant optimization techniques are almost wholly independent of the architecture

of the target machine. These include data and control flow analysis [24, 4], evaluation order rearrangement [7], and the "packing" phase of global register allocation [15, 29]. In a few other areas, such as code selection and local register allocation [49], attempts have been made to parametrize a technique, such that it can be moved from one target machine to another with changes only to a set of tables describing the machine. This is our principal method; the bulk of our work is to find formalizations for as many as possible of the known useful optimization techniques, such that they can be easily parametrized.

2.3. Current research

At least three other promising research or development efforts, with goals similar to those of PQCC, have recently been described. A comparison of their goals and methodologies with ours will help to clarify the PQCC effort.

The Experimental Compiling Systems (ECS) project [16, 17, 18] is addressing the problems of target-machine and language independence in optimizing compilers. They are primarily interested in modularization and standardization of optimizing compiler design; though the goal of this work is not a compiler-writing system, it would be a very logical by-product. Their model of the compilation process involves translation to a nearly uniform intermediate language (IL), followed by a loop in which machine independent optimizations are performed for as long as is deemed to be useful, followed by a "machine tailoring" phase in which the IL representation is brought much closer to the instruction level of the target machine.

There are major differences in methodology between the two projects. The ECS and PQCC treatments of optimization are two widely separated points on a philosophical spectrum, from "knowledge-impoverished" to "knowledge-based" systems (to use terms borrowed from Artificial Intelligence research [37]). Central in the ECS approach is the attempt to avoid special case analysis -- to use a few powerful and general optimization techniques, such as global flow analysis, constant folding and propagation, procedure integration, and dead code elimination, to subsume optimizations that are usually discovered only by more specialized, or even *ad hoc*, methods. The PQCC methodology, by contrast, is knowledge-based optimization -- we are willing to use all the (dozens of) known optimization techniques and sub-techniques, subject only to the condition that we can parametrize them to use easily understood machine and language descriptions. This difference in philosophy is reflected in, among other things, the difference between the intermediate representations for programs, discussed in section 4.2.

The other major difference in methodology is the means of adapting a compiler to different languages and machines. The procedure given in [16] for the "particularization" process

involves writing programs: a storage mapper, defining procedures written in IL, and perhaps others (a final component which does peephole optimizations is mentioned briefly). As discussed earlier, we are strongly committed to the use of formal descriptions, which tend to take the form of tables of information, rather than programs; the compiler writer's view of the PQCC system will be nearer the declarative end of the "procedural vs. declarative" spectrum of knowledge organization. A good summary of the arguments on both sides of this issue can be found in [57].

The MUG2 project [34, 35, 43], like the PQCC project, is an effort to produce (multipass) optimizing compilers from language and machine descriptions. The model of the code generator presented in [43] is closely related to our own, and is described further in section 9.2. [35] describes a language for describing optimizations, especially tree transformations of the kind discussed in section 7.5; the language, OPTRAN, is to allow the compiler-writer to impose a control structure of his own choosing on the search for such optimizations.

The portable C compiler [41, 42] is not a compiler-compiler system, but a compiler for C written with an attempt to isolate the machine-dependent portions. It has been transferred to a large number and variety of machines with a change of roughly 20% of the code; the transfer required about three man-months of effort, by the author of the original compiler. The portable C compiler does almost no global optimization, but the local code generation is of high quality [20]. In addition, compilers for another language (Fortran 77) have been written using approximately the same portable C code generator. Thus the portable C effort has made some progress toward the goals of language and target machine independence which we are interested in; it is difficult to tell from the published descriptions, however, whether the system could be extended to do more optimization, or to compile slightly higher-level languages (e.g. Pascal), or whether the system could be ported easily by a person other than the original implementor(s).

3. Overall strategy

3.1. System organization

Figure 1 is a crude box diagram of the PQCC system and a compiler generated by it. The compiler consists of a skeleton compiler (PQC) and a set of tables which it uses as input (in addition to the user program); the tables contain all the necessary language and target machine dependent information. Note, however, that these tables are not themselves written or modified by the compiler writer, that is, they are not the original descriptions of the language or target machine. They are instead output from the "compiler-compiler" system (PQCC), which takes as input the original descriptions.

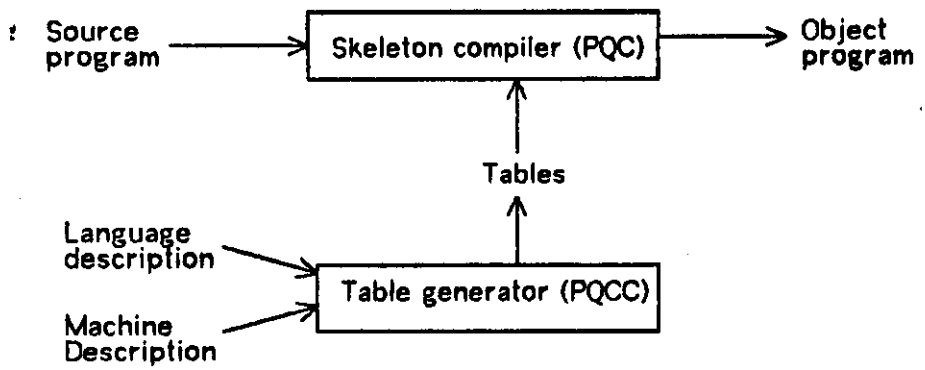


Figure 1: Crude box diagram of PQCC system

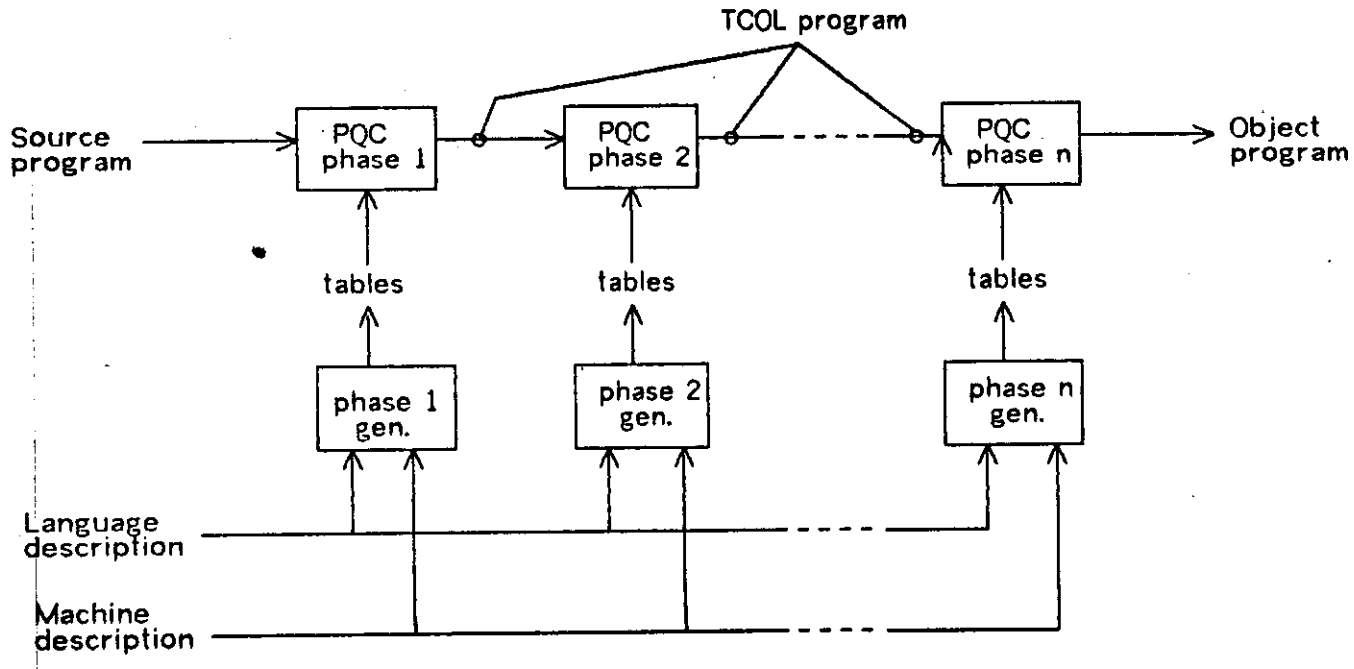


Figure 2: Detailed box diagram of PQCC system

Figure 2 shows some of the structure of the PQC and of the PQCC. The division of the compiler into phases has two important consequences to our system development methodology:

- The phases operate serially (at least in the experimental versions of the system), each taking as its input the entire output of the previous phase and performing some intellectually manageable subtask of the compilation process. This allows the PQCC itself to be decomposed into intellectually manageable portions: with each phase there is associated a "phase generator", which creates the language/machine environment information required by that phase.

- Work on each phase can proceed independently of work on (almost) any of the other phases. Thus, if some phase does not work, or does not even exist, progress can still be made on the phases following it in the compiler. This has been very difficult to achieve. It requires that any phase should be debuggable in "stand-alone mode", that is, without being plugged in to the entire compiler. This in turn requires several conditions:

    - It must be possible to prepare input for that phase, and examine output from it. Thus, the phase must be written to read from, and write to, standard input-output devices (usually disk files). It must read and write text files, not binary files, because humans must be able to read and write them as well.

    - The notation in which the files are written must be sufficiently general, or sufficiently rich, so that the entire state of the compilation can be recorded. (We have opted for generality instead of richness.)

    - The notation must be nearly uniform from phase to phase. This allows us (sometimes) to experiment with the ordering of the phases, or to run a phase even if we cannot provide a "dummy" version of the previous phase. The one exception to this is the notation used for output from the CODE phase (and input to the FINAL phase); this has the same syntax as the notation used for previous phases, but does not use the same semantics.

  The intermediate notation is discussed in later sections: LGN syntax in section 4.1, and TCOL semantics in section 4.2.

3.2. Disclaimer

There are two aspects of the compilation process which we resolutely refuse to study. Whatever systems we build will have to be plugged into other systems, not written by ourselves, to carry out these necessary tasks:

- Parsing/lexical analysis: we will presume the existence of a compiler front-end sufficient to transform a source program into a TCOL-like parse tree. The art of parser generation is sufficiently well developed that we are not interested in trying to advance it. Note that the problem of imposing uniform semantics on TCOL is a different matter; we would like to understand the problems of language semantics well enough to go beyond *ad hoc* techniques for "fixing" the output of given parsers.

- Loading and relocation: we will presume the existence of a compiler "back end" sufficient to turn the output of our last phase into executable code. We can provide instructions formatted for an assembler, or even formatted for binary

files, using information present in our target machine descriptions; but we are not interested in the problems of loader independence (which are indeed difficult).

3.3. A note on phase ordering problems

A problem common to multi-phase optimizing compilers is that there is no "ideal" ordering of phases. Phase A transforms the program in ways that obliterate some optimizations that otherwise could have been performed by phase B, which follows it; but if the order of the two phases is switched, phase B performs optimizations that deprive phase A of opportunities. The dual of this situation is one in which the two phases open up new opportunities for each other. We refer to these as *phase ordering* problems, or "chicken/egg problems", after the famous question, "Which came first, the chicken or the egg?". The first type are *negative* phase ordering problems, and the second type are *positive*. There are several ways of dealing with such situations.

- One can use iteration. The two phases are simply repeated, one after the other, until the program settles down. We are not accustomed to the use of iteration until convergence in compilers, but in fact this technique is highly successful in the FINAL phase of the Bliss-11 compiler [25], and it is used in the PQC FINAL phase as well (see section 10). Of course, it is no solution to negative phase ordering problems, but it may be the only optimal solution for positive problems. In order for it to be applicable, there must be some guarantee of convergence; in order for it to be practical, convergence must almost always occur after a very small number of iterations.

- One can ignore the problem. This approach is reasonable if the interaction between the two phases is rare, or is otherwise unimportant, or if there is no other practical alternative.

- One can use iteration without convergence; that is, the number of iterations of each phase may be fixed ahead of time.

- One can use what we call "wild guess" techniques. These may be used for both positive and negative phase ordering problems. Phase A may make some simple set of assumptions about the results of phase B, and act on them. This gives some of the effect of having two iterations of phase B. The advantage of it over having two iterations is that phase B need not be programmed to allow iteration, that is, it need not be idempotent. This is important, because writing a phase to be idempotent, especially if it computes new attribute values for program tree nodes rather than simply performing tree transformations, requires at least a strong programming and design discipline, and at worst may be hopelessly difficult.

We will encounter various phase ordering problems in our discussion of the phases of the compiler; we will try to indicate for the important ones how we have dealt with them.

4. Intermediate Representations

4.1. LGN

The following list should be appended to the list of requirements, presented in section 3.1, which must be met by our external representation of programs:

- The notation should be able to represent an arbitrary directed graph with many links, including cyclic links.

- The notation should be able to represent information independently of its implementation, e.g., representing a sequence of data which may be stored as a list, a set, a vector, etc. This is because the optimal representation of information for one compiler phase may not be optimal for any other.

- The notation should be transformable to an efficient representation, e.g., a highly packed bit representation with single bits for booleans, small fields for small values, etc. In particular, it should be possible to discard the input-output between two phases and collapse them into a single program; ultimately, we could develop source code that would integrate into a single compiler without any intermediate representation other than the binary image in memory.

- The implementation should not lose information because a particular phase does not need it; in particular, it must preserve such information idempotently from input to output.

The "Linear Graph Notation", or LGN[1] meets these goals to varying degrees. The syntax of LGN describes nodes of the graph in terms of a node type and attribute-value pairs associated with the node. Each node is labeled, and the label was chosen to be an octal number so that an implementation could emit the machine address of the node as its label (as indeed our debugger does). However, the only important requirement on labels is uniqueness within a particular graph. This is checked on input (duplicate labels indicate an error condition) and is expected to be preserved on output.

4.1.1. Example

An example here might give more of the flavor of LGN:

---

[1] The name indicates that the notation is a linear representation of a generally non-linear, even non-planar, internal graph structure.

```
17: OBJECT
        (NAME BALL)
        (COLOR YELLOW)

23: ACTOR
        (NAME JACK)
        (AGE 6)

31: RELATION
        (NAME PLAYS-WITH)
        (WHAT 23:)
        (TOWHAT 17:)
```

This example was chosen because it has nothing whatever to do with compilers. It is therefore possible to concentrate on what the notation says without worrying about what we must say to describe a compiler data structure; that is discussed later in section 4.2 (see figure 5). This shows that there exist things called OBJECTs that have names and colors, ACTORs that have names and ages, and RELATIONs for connecting actors to objects (or possibly objects to actors), which have names and directed arcs WHAT and TOWHAT. Attribute names such as "NAME", "AGE", "WHAT", and "TOWHAT" are not interpreted by the LGN support system -- any other identifiers could have been used equally well. Moreover, the NAME fields in the three types of nodes, OBJECTs, ACTORs, and RELATIONs, are not necessarily related to each other, or confused or connected with each other in any way by the LGN system. Thus LGN could be the external representation of a conventional record structure, as provided by languages like PASCAL.

4.1.2. Primitive data types

The primitive types for the attribute values are:

integer	represented externally by a string of digits, or by a symbolic name;
label	represented by an octal number followed by a colon (forward references are handled correctly);
identifier	represented by a string of letters, digits, and even some punctuation marks;
string	quoted strings of arbitrary characters;
sequence	sequences of values (separated by blanks) of any of the above types, possibly with various types intermixed.

Values of the *identifier* type are represented internally by unique integers generated by the LGN system; two of them can be tested for equality, but no other meaningful operations can

be performed.

An LGN support package provides the software necessary to work with these representations in a program. It contains:

- A definition-file generator, which takes a specification of the node types, attribute names, and allowable value types and values, and produces definition files used by the source program. These files provide the necessary access to the fields, to the node information, and to the representation. They additionally define the tables required by the input/output support.

- Input/output runtime support, which reads and writes LGN files.

- Runtime utility support, which provides procedures for set and list manipulation, storage management, creation and deletion of nodes and complex values, and error handling.

Attributes of type *integer* and *identifier* frequently appear similar in the external representation. This is because of the facility for defining symbolic names for integer attribute values. Consider, for instance, the attribute COLOR, of "object" things. The user can specify that the only legitimate colors have symbolic names BLUE, RED, YELLOW, and GREEN, and can further specify which integers these four names represent. If, alternatively, the COLOR attribute had type *identifier*, then any name would be a legitimate color; two colors could be tested for equality, but no other operations (such as typical integer operations) would be meaningful.

Attribute names and symbolic names, like identifiers, need only conform to the very permissive LGN syntax for identifiers. Since most languages (BLISS in particular) have more restricted identifier syntax, the LGN facility for defining them allows them to be associated with "internal" identifiers, which are expected to obey the rules of the host language.

4.1.3. Composite data types

The internal representation of a sequence is defined by the user; thus, the sequence

```
(SUBNODES 17: 44: 76: 122: 5:)
```

may be stored as

an *array*: the order is preserved, and the i^{th} element of the array is the i^{th} value in the sequence;

a *set*: the order is not preserved, and duplicate entries are omitted. Insertion and retrieval are efficient;

a *list*: the order is preserved, and insertions and deletions are efficient while

indexing is not (lists are doubly linked).

(All of these representations are fully supported by the LGN software.)

In addition, atomic types or arrays may contain values of type *item*. An *item* has a value which can be any of the atomic types or composite types, and has a type-tag indicating which type the value possesses. For example, the following sequence could be stored only in an item-array, set, or list:

```
(THING-SEQUENCE  "string" 17: 45 any-id)
```

Similarly, the following two nodes would require that the VALUE field be of type *item*, and the type of the item would be determined at run time by examining a tag field.

```
17: SOMENODE
        (VALUE 44:)

23: SOMENODE
        (VALUE 5)
```

In this example, the type tag associated with the VALUE field of node 17: would indicate that the type of the VALUE field is *label*, and the type tag of the VALUE field of node 23: would indicate that the type of the value field was *integer*. As with "union mode" or "variant record" features in many languages, this feature defeats some of the type checking that normally is done.

4.1.4. The definition file generator

The LGN definition file generator is a program which reads a description of the nodes, including the names of the fields, type of value, possible values, etc. and emits the source code necessary for defining these data structures to the language used for the implementation. Although currently all the source code is in BLISS-10 [6], the specifications are in LGN itself and the definition file generator could emit record declarations for SAIL, PASCAL, C, or whatever the desired target language would be.

A particularly useful feature, mentioned earlier, is the ability to construct tables for dispatching to specific procedures based upon the value of a field in a node. For example, one may wish to "understand" in some way the properties of the binary operators:

```
+       commutative, associative
*       commutative, associative
/       not commutative, associative
-       not commutative, not associative
mod     not commuative, not associative
...
```

which might involve the following code:

```
case node[operator] of
        begin
[Addop]         COMASSOC(node);
[Mulop]         COMASSOC(node);
[Divop]         NCOMASSOC(node);
[Subop]         NCOMNASSOC(node);
[ModOp]         NCOMNASSOC(node)
        end;
```

In any language, this is awkward, because as the set of operators is extended, each site in the program which involves such a dispatch table must be extended in kind. While this certainly can be done, it is tedious, and difficult to do correctly (consistently). The LGN system allows dispatch tables to be built without regard to the actual values assigned to the symbolic names allowed for a particular *integer* field; it sorts the set of dispatch table entries provided by the user, and fills in default entries for symbolic names which the user has failed to provide for. In the BLISS system, the dispatch table is simply a table of subroutine addresses, and the dispatching code looks like:

```
.DispatchTable[.node[Operator]](.node);
```

Note that the code at the dispatch site is not sensitive to the actual size of the table, i.e. to the number of operators. The table is sufficiently simple that it can be generated by the definition file generator; this assumes that the act of incorporating this table into the actual source program is accomplished by some compiler directive, such as (depending on the language) **require...source, include, copy**, etc.

The file generator has the advantage of knowing the exact values it has assigned to the bit patterns which represent the tokens "+", "/", etc. in the operator field of a node, and can therefore emit a correctly sorted table. Declarations exist for what to do with unspecified operators.

4.1.5. Observations on LGN and its support

The use of LGN has not been entirely ideal. The reader/writer routines have a significant impact on the performance; several minutes (of real time) are required to read in a large database such as a machine description. This limitation is almost entirely at the I/O and buffer-scanning level, rather than in storage allocation or manipulation. Once the data structure is read in, we pay no time penalty for the use of LGN, although we incur some space penalty due to the internal symbol tables it maintains.

The current implementation does not allow a typical space optimization used in real systems, which is to re-use the (idle) space of one phase in another phase. Thus, we were unable to create data structures in which the fields of one phase overlapped the fields of another phase, even though there was no conflict in their use. Although largely an artifact of one particular implementation of the LGN support package, this did cause us problems in that a large number of words (currently 19 words) are required to represent a tree node, when in fact fewer than half of these contain useful information at any one time.

Neither of these defects are serious enough to cause us to reconsider the use of LGN. The ease with which we were able to manage a large, complex database (the representation of the basic tree nodes) and create many small, special-purpose representations for each phase contributed significantly to the rapid implementation of the first compiler; in fact, about six months elapsed between the beginning of serious coding on the compiler and the prototype demonstration system.

The interactive debugger for Bliss (SIX12) was extended to use the internal LGN tables, and we could thus examine and display our data structures using LGN-like syntax. The ability to debug in terms of our conceptual data structures rather than their internal representation was another factor which contributed to rapid development. Although some systems, such as LISP, provide such features as a matter of course, very few algebraic languages or implementation languages provide such facilities.

4.2. TCOL

An issue not settled by the design of LGN described in the previous sections is which data structures are to be used to represent the source program. The initial choice is between various linear representations, such as triples or quadruples (two-address and three-address "instructions"), and a two-dimensional representation (such as a parse tree). We chose the latter. Some arguments for this choice are presented in [19, 14]. We summarize our principal reasons as follows:

- The linear representations are too confining in their prescriptions for the storage and retrieval of operation results. It is difficult to generate good code for three-address machines from triples, or for two-address machines from quadruples.

- The linear representations require that source-level control constructs be broken down into tests and jumps. The disadvantages of doing this too early in the compilation process are discussed below.

- The tree representation has a good deal of psychological "naturalness". It seems to be much easier for a human reader to grasp the data dependencies of a program from a (suitably prettyprinted) parse tree than from a (suitably prettyprinted) list of triples or quadruples; the latter has all the readability problems of assembly code.

It's important to remember that the two kinds of representation are essentially equivalent in power. It is equally possible to construct a control flow graph (see section 6) from either form, and in fact we maintain a linear list of all tree nodes in execution order. Conversely, the subtree pattern-matching optimizations described in section 7.5 could be described and performed with almost equal facility on a linear representation of the program -- if it were thoroughly analyzed and augmented with enough information to reconstruct the tree structure. However, much less effort, both in coding and in understanding, is required to regenerate linearity information from the tree structure, than to regenerate data dependency information from a linear structure.

We have elevated our representation of programs to the status of a notation or language, called TCOL. This acronym is historically motivated; the last three letters are from the earlier acronym UNCOL [52], and the first letter stands for "Tree". An appraisal of the aims and approaches of UNCOL-like systems, by comparison with our own, can be found in [36]. Although those systems are not especially appropriate for producing good-quality code, we have borrowed one central idea from them: the notion of a single, (almost) uniform intermediate language, to allow simultaneous compatibility between the output of many parsers and the input of many code generators. Unlike most UNCOLs, which are like instruction sets of a stylized computer, TCOL contains no implicit computer model; it can be regarded as a stylized programming language.

We do not work with a parse tree, but with a tree that reflects *abstract syntax*:

- operator precedence rules, and parenthesization by the programmer to get around them, are no longer evident in the tree;

- constructs which are indefinite sequences of other constructs are represented as nodes with indefinitely many operands, though they may have been represented in the parse tree as (highly unbalanced) trees. An example of this is given in figure 3;

- terminals in the grammar are not necessarily represented by leaves in the tree. In particular, a binary operator expression is represented by a node with two operands; the value of the **op** attribute is the identity of the operator. In a parse tree, the node would have had three operands, of which the middle one would be the operator.

In addition, some semantic rules must be followed:

- All operations, including coercions and dynamic checking of subscript ranges and types, must be explicitly represented in the tree. This includes the "dereferencing" coercion, so effectively TCOL uses the Bliss "dot" notation [6]. Figure 4 shows some typical examples of the results of this policy, for various programming languages. Figure 5 shows a TCOL tree represented in LGN; this is (slightly simplified) output from a BLISS parser which we use for testing purposes.

- Different operators must be represented by different names. Thus integer and real addition are just as distinct from each other as are addition and multiplication.

- Control structures are represented by structures, not broken down into tests and jumps. This means that the control structures in TCOL vary somewhat depending on what language is being implemented. For instance, one of the TCOL operators for Algol 68 is **for**; for BLISS there are **incr** and **decr**; for FORTRAN there is **do**. All the expressions which are parts of the construct are operands of the operator; thus the tree node for a **for** loop in an Algol 68 program has a **from** operand, a **by** operand, a **to** operand, and a **do** operand.

- Declarations are not ordinarily represented in the tree. Exceptions are declarations with which code is specifically associated, such as the BLISS **enable** declaration; an initializing declaration of a variable would be represented as an assignment.

- No tree node may appear in more than one basic block in the ultimate control flow graph. The terms *basic block* and *control flow graph* are defined in section 6; the practical meaning of this rule is that the code generated for a control construct node must be straight-line code. This sounds bizarre; what it means is that some such nodes must have dummy operands, which represent various pieces of straight-line code. Here are some examples:

    - The Algol 60 **if-then-else** statement is represented by one node. Its operator is called **if**. It has three operands: the boolean expression, and the two alternative statements. NO CODE will be generated for the **if** node. This is because the testing and conditional branching will be generated automatically for the boolean expression node, and the branch at the end of the **then** part will be generated automatically for that node. Therefore, the **if** statement node presents no problem.

    - The **if-then-else** expression, on the other hand, requires some care. Given that the value of the expression is required, it may have to be moved around. The value of the node will be held at run time in some target machine storage location, such as an accumulator or a memory location.

The values of the **then** part and the **else** part will also be held in locations, and all three of these locations may be different from each other. Code must be generated to move the contents of the latter two locations into the first location. There will thus be two straight-line pieces of code, which for simplicity should be represented by two dummy operand nodes. Each dummy operand node has an operator named **copy**. It stands between the **if** node and the corresponding alternative node; that is, the **then** part is no longer an operand of the **if** node, but rather an operand of the first **copy** node.

- Stepping loop constructs, in many languages, require considerable care. Consider the Algol 68 **for** expression:

  for *i* **from** *e*1 **by** *e*2 **to** *e*3 **do** *e*4 **od**

  The semantics of this construct require that the **by** and **to** parts must not change during execution of the loop. Therefore the initialization code, which already includes the initialization of *i* to *e*1, must also (possibly) initialize two compiler-created variables to *e*2 and *e*3. The initialization code is represented by a dummy operand node whose operands are *i*, *e*1, *e*2, and *e*3. The loop increment code is represented by a dummy operand node, and the loop test code by yet another dummy operand node. Finally, some languages, such as BLISS, allow for the loop to return a value, in which case there is some code to store the value in an appropriate location; this code is represented by the **incr** node itself.

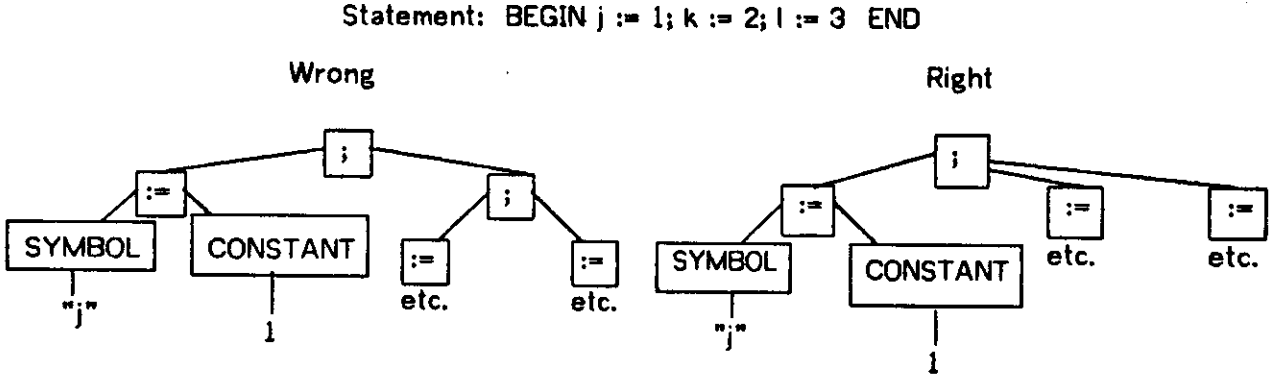


Figure 3: Right and wrong TCOL representations of a compound statement

Many artifacts of the compilation process are not reflected in TCOL. The TCOL tree representing a program should not be affected by the presence or absence of such parser features as: macros and open subroutines; generic operators and routines; compile-time type checking and verification condition checking; source-file inclusion facilities (PL/I **include**, COBOL **copy**); and conditional compilation. Moreover, block structure and encapsulation, which

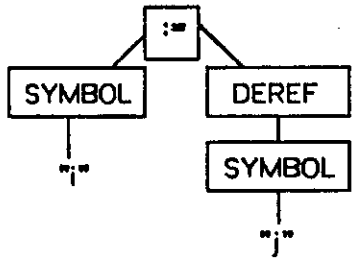
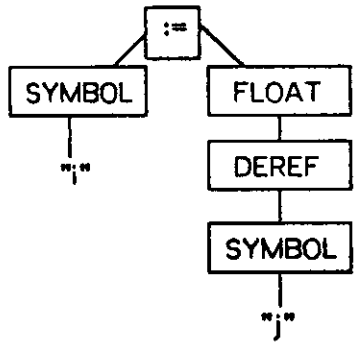
Statement	(Source Language)	TCOL tree
i = j i := j i ← j	(Fortran) (Algol 68) (Bliss)	
a = j a := j or a := REAL(j)	(Fortran) (Algol 68)	

Figure 4: Examples of TCOL trees

play important roles in language syntax definition, leave few traces in the TCOL representation of a program. Along with the parse tree there must be a symbol table, but this has no structure; it is just a set of LGN graph nodes, one for each user variable, with no necessary relation to each other. Each node holds just enough information about the variable so that later phases can allocate storage for it (e.g. whether its lifetime is static, stack-like, or heap-like), determine what its aliases may be, and generate debugging information (e.g. a print name) for it.

The semantics of each TCOL operator may vary from target machine to target machine and from language to language. At the very least, the meaning of each arithmetic operator changes when the word size of the target machine changes; and TCOL operators which are not relevant to a particular language (such as string operations in a language that does not provide strings) may simply be dropped. More subtle or more interesting changes may occur: the relationship between negation and logical complementation is different for two's complement and one's complement machines; some arithmetic operations may be checked for

overflow and may therefore have potential side-effects; the semantics of built-in data abstractions such as strings may differ slightly from one language to another. These differences do not appear in TCOL, but in a set of axioms which are incorporated into the language definition. The use and nature of these axioms are described briefly in section 9 and in more detail in [11].

There are other issues of semantic variation from one TCOL to another that we have not yet dealt with. These are variations of semantics of basic language constructs: parameter passing, aliasing, the treatment of limit and increment parameters of stepping loops (touched on above). We hope eventually to develop notations for expressing these characteristics of languages, and to determine precisely their effects on the PQC compiler. Until then we will have to deal with these issues in an *ad hoc* manner, whenever bringing up a compiler for a new language.

The treatment of array and record accesses is to represent the address computations explicitly in the TCOL tree. The motivation for this will become clear when we describe optimizations performed on these address computations in various phases (FLOWAN and several DELAY phases); it is impossible to manipulate these computations, and compare them with each other, in a rational manner without exposing the arithmetic operations of which they are composed. The penalty paid for this is that information is lost, or at any rate scattered and made difficult to use: target machine indexing operations, so obvious in the source code, must be rediscovered in a separate phase (see section 7.6) of the compiler.

The treatment of control structures is still more unusual, for the opposite reason. As mentioned earlier, these are represented in a tree form close to their original parse tree form; if we treated them as we treat data structures, we would instead break them down into tests, jumps, and labels. The tradeoff is different here. The benefit of breaking control constructs into tests and jumps is not in better optimization. In fact optimization is likely to be worse: it is extremely difficult to recognize the applicability of a special target machine instruction for loop control, if the components of the stepping loop are scattered hither and yon through the program tree. Rather, the benefit is in slightly decreased cost of transporting a compiler from one target machine to another. For each control construct operator in a TCOL, one or more descriptions of code sequences that can implement it must appear in the code generation template library; the significance of this will become clear in section 9. In the absence of quantitative knowledge or even estimates of this increased cost, we have tentatively chosen the scheme that allows better optimization.

```
VERSION 16 NODES #11618
|       0001    begin
|       0002     local X;
|       0003     own A,B;
|       0004     X ← .A + .B
|       0005    end

ROOT TREE 2343:

1: SYMBOL
     (NAME "X")
     (LIFE DYNAMIC)
     (SIZE 2)

2: SYMBOL
     (NAME "A")
     (LIFE STATIC)
     (SIZE 2)
     (GLOBAL NO)

3: SYMBOL
     (NAME "B")
     (LIFE STATIC)
     (SIZE 2)
     (GLOBAL NO)

2343: TREE
     (OP :=)
     (SUBNODES 2344:  2345:)

2344: LEAF
     (OP SYMBOL)
     (ANCESTOR 2343:)
     (VAL 1:)

2345: TREE
     (OP +)
     (ANCESTOR 2343:)
     (SUBNODES 2346:  2347:)

2346: TREE
     (OP DEREF)
     (ANCESTOR 2345:)
     (SUBNODES 2350:)

2350: LEAF
     (OP SYMBOL)
     (ANCESTOR 2346:)
     (VAL 2:)

2347: TREE
     (OP DEREF)
     (ANCESTOR 2345:)
     (SUBNODES 2351:)

2351: LEAF
     (OP SYMBOL)
     (ANCESTOR 2347:)
     (VAL 3:)
```

Figure 5: Example of a TCOL tree represented in LGN

5. Constant folding and constant propagation

Conceptually the first compiler phase to operate on the program is one of constant folding. We distinguish between *constant folding* and *constant propagation*:

- Constant folding is the evaluation of arithmetic expressions at compile time, when their operands are known. This also may include provision for operands which are addresses of variables -- not necessarily known until load time, or even run time. Thus, the address computation involved in an array access with constant subscripts is done during constant folding. Reduction of expressions involving identities is done during this phase as well; for instance, "a + 0" is reduced to "a".

- Constant propagation is a different optimization, involving data flow analysis (see section 6). The contents of a program variable may be known at compile time for certain limited sections of a program. (These usually are the sections immediately after initialization of the variable.) In these sections, references to the contents of the variable may be replaced by references to the compile-time constant.

We assume that if the language allows declarations of names for constants (e.g. the **constant** declaration of Pascal), these names have been replaced by their meanings in the program tree. Thus it might seem that constant folding is an easy step, once done and then forgotten. However, it is here that we encounter the first of the (positive) phase ordering problems discussed in section 3.3. Subsequent optimizations create new opportunities for constant folding. This is particularly true of the tree-transformation subphases (see section 7). For instance, the expression:

(a + 3) + 4

will be simplified to

a + (3 + 4)

at which point it would be useful if the constant folding phase could be run again. Our solution to this particular phase ordering problem is to modularize the constant folding phase. It consists of a module which controls the walk of the program tree, and a module which, given any expression node whose operands are leaf nodes (constants), does the arithmetic necessary to "fold" the expression, returning a single leaf node representing the result. This latter module is available to be called by subsequent phases.

Constant folding may be extended to include compile-time determination of control flow results. Thus, the expression:

if *true* **then** *thenpart* **else** *elsepart* **fi**

may be reduced to just the then-part. This is useful if the constant-folding feature of the optimizer is to be used to construct a conditional compilation facility. Ordinarily, however,

conditional compilation will have taken effect before the code generation phases are begun -- supported, perhaps, by a macro processor or preprocessor.

6. Flow Analysis

The FLOWAN, TNBIND (section 8) and FINAL (section 10) phases all do some sort of data and control flow analysis of the program. The first of these is FLOWAN; the purpose of this phase is to detect optimizations involving code motion and elimination of redundant computations.

Crucial to the action of this phase, and later phases, is the construction of a *control flow graph*. This is a graph whose nodes represent *basic blocks*; a basic block [23] is a section of the program containing no branches, except at its end, and no labels, except at its beginning. In our representation, of course, a basic block is an ordered set of nodes from the program tree. Links in the graph are directed, and represent possible transfers of control between basic blocks. This data structure is basic, in that it is common to almost all flow analysis systems [4]; other structures will be built on top of it, including the *execution list*, a linked list of all the program tree nodes threaded in (approximate) execution order.

We will not try to catalogue all the optimizations that are or will be done by this phase, but a brief summary of them will give some of the flavor of it. We will refer to the diagrams in figure 6, which cover two separate pages. In these, bits of Algol 68 program text are presented, along with arrows denoting (possible) transfers of control from one to another. Greek letters are used to indicate pieces of program text that are irrelevant to the optimization being described. "*t*" always denotes a variable created by the compiler, guaranteed to have a name distinct from those of all user variables and all other compiler-created variables (CCV's).

1. Diagram a shows an example of *redundant expression elimination*. (The misnomer "common sub-expression detection" refers to the same transformation.) Before optimization the expression $a + b * c$ is evaluated twice. After, it is evaluated once; its value is saved in a CCV, and used twice. There are always restrictions on the "irrelevant" code in order that it be truly irrelevant, e.g. it must contain no assignments to *a*, *b*, or *c*, and there may be no control transfers from outside it to within it. Diagram b shows a more complicated example.

2. Diagram c shows the *hoisting* code motion: two occurrences of the same expression or statement are combined into one, by moving them "backwards" over a fork in the program flow graph. If they are expressions (or even if only one of them is an expression), a CCV must be created, as in the examples of redundant expression elimination. Diagram d shows *reverse hoisting*; only statements can participate in this code motion.

3. Diagram e shows another kind of code motion; for lack of a better name, we will

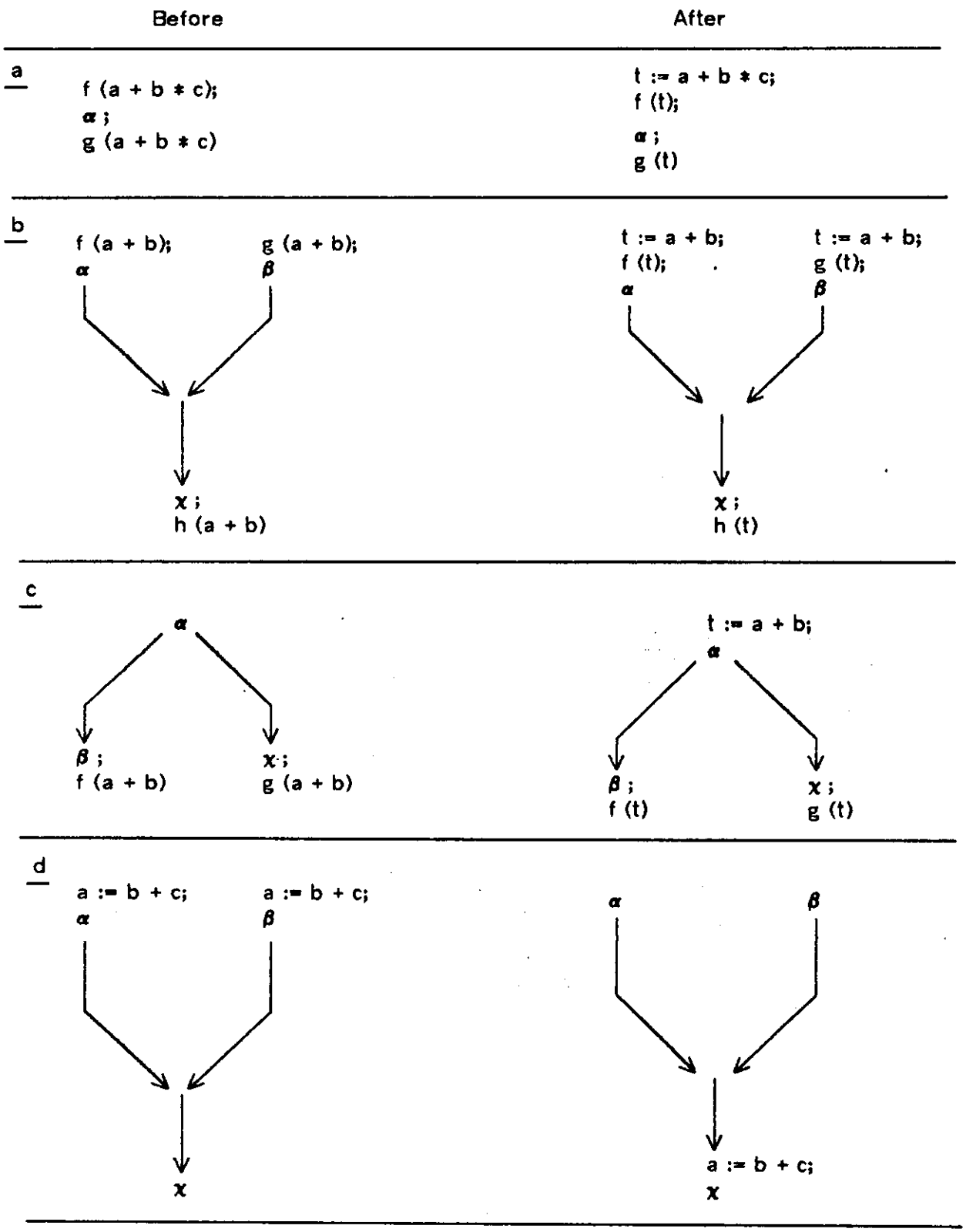
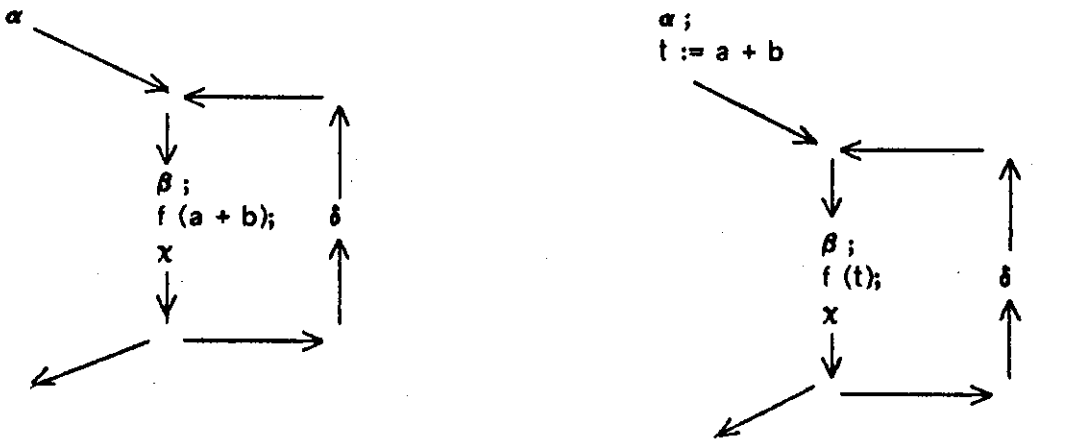


Figure 6: Diagrams for flow analysis optimizations

Before | After

e

f



g

FOR i FROM 1 TO 10 DO f (i * 5) OD

FOR i FROM 5 BY 5 TO 50 DO f (i) OD

h

FOR i FROM α TO b DO β OD

Figure 6 continued

call this *rho motion*, the name used in [25]. Here, an expression or statement is moved so that it is only evaluated once on each path leading to a join point in the flow graph. This is most useful when the join point is the entry point of a loop, and the expression is being moved out of the loop. A degenerate case of this optimization occurs when the two instances of the expression are the same instance, i.e. the same node in the program tree. This is shown in diagram f. This is the optimization classically known as "moving invariant code out of loops".

4. Diagram g shows an example of *strength reduction*. This optimization changes an expensive operation in a loop (such as multiplication) to a simpler operation (such as addition), or even gets rid of the expensive operation altogether by changing the loop control, as in the example. While the example is rather exotic-looking, this optimization finds considerable use in the treatment of array subscripts [23]. Where the source program increments an array subscript by 1 each time around a loop, the object program may increment a pointer by some number different from 1; this is useful for arrays of more than one dimension, but also for one-dimensional arrays, if (as on the IBM S/370 or DEC PDP-11) the address difference between two consecutive array elements is more than 1.

5. Finally, diagram h shows, not an optimization, but an example of source code for which the FLOWAN phase will make an important determination about the object code structure. The Algol 68 specifications of the semantics of stepping loops are that the increment and limit parameters do not change during the loop. Thus, if the loop body contains code that may change the value of variable *b*, the loop initialization must copy the contents of *b* into a CCV *t*, and use the contents of *t* instead of *b* as the loop limit. FLOWAN determines whether or not this copying is required. (Note that even if it is not required, it may be useful, e.g. if *b* is not in a fast register, but could be copied into a fast register for the duration of the loop. This is discovered in TNBIND (see section 8), not in FLOWAN.)

The optimizations described in paragraphs 1, 2, and 3 above are frequently unprofitable in situations where they happen to be feasible. There are two possible reasons for this. One is the well-known tradeoff between time and space. If a hoisting code motion requires a CCV, it may increase the time required for execution of the program, even though it decreases the size of the object code. Contrarily, if rho motion requires a CCV, it may decrease the execution time, while increasing object code size. Although most program optimizations are beneficial to both execution time and code size, the tradeoff is occasionally important, and it has been suggested that it be under control of the programmer (or at least under control of the project manager). The other reason is that when these optimizations involve expressions, and hence require CCV's, sometimes the expressions involved are so easy to compute that they are swamped by the cost of manipulating the CCV's.

This last consideration is rather important. Unfortunately it has received almost no attention in the literature (that we know of). Code motion and redundant expression elimination are most useful when they are applied to address computations, such as for indexing into arrays or for record accesses; but it is precisely in these cases that target

machine features, such as indexing and indirection, tend to make expressions very easy to compute. The analysis necessary to determine whether one of these optimizations is profitable (*desirability analysis*) is very target machine dependent; we have separated it from the FLOWAN phase, and it is discussed under DELAY (see section 7.2). Thus the FLOWAN phase does not actually carry out those optimizations, but leaves indications that they are feasible. This phase is entirely independent of the target architecture, but dependent on the semantics of the language.

7. The DELAY phases

DELAY is the name given collectively to a group of phases which do source-to-source transformations of the program tree before the register allocation and code generation. With some exceptions, these transformations are target machine dependent. The AMD and context determination phases do not, strictly speaking, transform the tree, but associate semantic information with each node. It is reasonable to expect that some of these phases could be run concurrently with each other, but for our (research) purposes, clarity is more important than efficiency, and we have kept them separate. The name "DELAY" is a misnomer; its history is explained in section 7.3.

7.1. Context determination

We define the *context* of an expression to be the information, derivable from its position in the program tree, about how its value is to be used. For instance, in languages which allow an assignment to be either an expression or a statement, the context of an assignment determines which it will be. From the earlier discussion of FLOWAN, it is clear that at least some context information must be available before that phase, and in fact we will probably move part of the context determination phase to a position before FLOWAN.

At present we distinguish four different contexts which an expression may be in:

- the *no-value* context: its value is thrown away, that is, it is a statement.
- the *flow-value* context: its value is used only to determine the subsequent path of control. This is the context of, for instance, the boolean part of a conditional expression, or the while part of a while-do loop.
- the *operand* context: this is the usual context of an expression; its value is used as, for instance, the source of an assignment, or one of the operands of an arithmetic operator.
- the *address* context: its value is used as an address; for instance, it may be the destination of an assignment, or the operand of the "dereference" operator (see section 4.2).

In addition, if an expression is in the operand or address context, we determine how many bits of it are to be used; the knowledge that part of a value is irrelevant sometimes enables better code to be generated for the computation of it.

The above context information is obviously useful for expressions whose values are simple machine words. Other context information might be valuable for more interesting data types: for complex numbers, only the real or imaginary parts might be used; for strings, only the length might be used, or only certain character positions.

If an expression involving one of the logical operators (**and**, **or**, etc.) has a flow-context, it may be that no code at all need be used to evaluate it. Instead, its operands are evaluated, and the flow of control is determined directly from them. This means that code to evaluate the second operand may not be executed, if the flow of control can be determined from the first operand. (Of course, the legality of this coding requires that the second operand have no side effects.) This is referred to in [50] as the *McCarthy conditional* transformation. It sometimes "competes" with some of the FLOWAN optimizations for the right to affect a piece of code; for example, if the second operand of a logical operator is to be made conditional, it must not contain the initializations of any CCV's. Thus, there is a negative phase ordering problem between context determination and FLOWAN.

Context determination is implemented as a simple top-to-bottom tree walk. Context information may be passed from the destination of an assignment to its source, or from an expression or statement to its operands.

7.2. Desirability analysis

Suppose that FLOWAN finds two instances, *e1* and *e2*, of the same expression in a program, and finds that *e2* is redundant with respect to *e1*. The straightforward course of action is to designate a CCV for this pair of expressions; code is generated so that *e1* is stored in the CCV before being used (it may in fact be computed directly in the CCV), and the section of the program which would have used *e2* uses the contents of the CCV instead. This course may be less profitable than leaving the code alone, however, for any of three reasons:

- subsumption: the computation of *e1* and *e2* might be carried out implicitly because of the nature of their ancestors in the program tree. For instance, consider:

  $a := b[x + 1];$
  $c := d[x + 1];$

  where both b and d are statically allocated arrays. Here, the two instances of $x + 1$ are of interest. If no CCV is designated to represent them, the addition

operation will be "folded" into the address computation, so that it costs nothing. To make this clearer, we give two code sequences for the DEC PDP-10 for the above program text, illustrating the two ways of dealing with the redundant expression situation:

```
"x + 1" computed in CCV          "x + 1" not computed in CCV

    MOVE    R1, X                    MOVE    R1, X
    ADDI    R1, 1                    MOVE    R2, B+1(R1)
    MOVE    R2, B(R1)                MOVEM   R2, A
    MOVEM   R2, A                    MOVE    R2, D+1(R1)
    MOVE    R2, D(R1)                MOVEM   R2, C
    MOVEM   R2, C
```

In the second code sequence, a CCV has been designated to represent the two occurrences of x. When one set of redundant expressions is broken up, the operands become eligible to be treated as redundant.

- addressing hardware: the computation of *e1* and *e2* might be done cheaply, by using the effective address computation available on the target machine. Generally address computations have some cost, but if there are not too many of them, they may be cheaper than the instructions necessary to manipulate a CCV. An example of this is:

$$a := b\,[x]$$
$$b\,[x] := c$$

Here, it is feasible to compute the address $b\,[x]$ in a CCV (by moving the contents of x into it and adding the address of b to it), but this is wasteful on almost any target machine, because the address can be computed more cheaply by an indexing operation.

- operand vs. flow-value context: some operators, including of course the logical operators **and** and **or**, but also including (on most target machines) the operators for comparison between values ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$), require substantially less computation when they are in a flow-value context than in an operand context. If *e1* and *e2* are both in flow-value contexts, and they are in this set of operators, it is usually not profitable to compute *e1* and save it in a CCV, because this effectively puts it in an operand context. However, if one of them is already in an operand context, the boolean value must be computed anyway, so it might as well be stored in a CCV.

Note that in the last two cases (but not the first), the number of instances of the expression involved is important: the more there are, the more attractive is the option of using a CCV.

The desirability phase examines each redundant expression set, and each code motion involving an expression, deciding whether or not to carry out the optimization as specified by FLOWAN. Sometimes a single expression can be removed from a set, if it is one of the redundant members, leaving the rest of the set intact. When a set is broken up, similar sets

are formed using the operands of the original expression. The phase takes a number of parameters describing the target machine, including a table of the relative costs of the code sequences for each operator in each of the relevant contexts (flow-value, operand, and address).

7.3. Unary complement operator propagation

The "unary complementation" operators, such as negation and logical complementation, are involved in a several interesting kinds of optimizations. Sometimes it is possible to push them upwards, so to speak, so that they are subsumed into operations occurring at higher points in the program tree. Diagrams a and b in figure 7 show examples of this. The policy of delaying code generation for unary complement operators as long as possible, used by the Bliss-11 compiler [25], was one of the functions of the DELAY phase in that system, and the origin of its name.

	Before	After	PDP-11 code
a	$(-\alpha) + (-\beta)$	$-(\alpha + \beta)$	
b	$(-\alpha) * (-\beta)$	$\alpha * \beta$	
c	a := (NOT b) AND c	a := c AND (NOT b)	MOV c,a BIC b,a
d	a := b - a	a := -(a - b)	SUB b,a NEG a
		or a := (-a) + b	NEG a ADD b,a

Figure 7: Diagrams for unary complement optimizations

More interesting are the optimizations which involve taking advantage of target machine instructions which perform peculiar combinations of binary operators and unary complement operators. Most machines have a subtraction instruction; some but not all have negation; fewer still have an instruction which negates a quantity while moving it from one location to another. On the DEC PDP-10, all 16 of the possible logical functions of two variables can be performed in one instruction; on the PDP-11, only five (six on some models). Sometimes it is possible to get better code by taking advantage of the commutativity of an operator.

Diagrams d and e show examples of these optimizations; in both cases, the PDP-11 code associated with the optimized form of the expression is also shown.

7.4. Evaluation order and targeting

Targeting is an optimization that is useful on one-address, two-address, and general-register architectures. It is the attempt to make use of the commutativity of arithmetic operators, in order to avoid loads and stores. On a two-address or general-register machine, it may be possible to compute an expression in the destination to which it is to be assigned; if one of the operands of the expression is the former value of that destination, even the initial loading of that location may be elided. (This optimization is called "renaming" in [46]). Referring to figure 8, diagrams a and b

	Before	After
a	a := b + c * d	a := c * d + b
b	a := c * d + a	a := a + c * d
c	(a + b) * ((c * d) + (e * f)) evaluation on a single-accumulator machine	
	straightforward evaluation order using two temporaries (t1 and t2)	better evaluation order using one temporary (t1)
	load a add b store to t1 load c multiply by d store to t2 load e multiply by f add t2 multiply by t1	load c multiply by d store to t1 load e multiply by f add t1 store to t1 load a add b multiply by t1

Figure 8: Diagrams for targeting and evaluation order

show examples of targeting. It was touched on briefly in the examples for section 7.3; a single algorithm, to be discussed in a future paper, performs both targeting and unary complement transformations.

Note that questions about side effects do not arise in the decisions about whether or not to commute the operands of an operator. Regardless of whether or not the operands are

commuted, they can be evaluated (if their evaluation requires any code) in the order specified in the source program. *Evaluation order determination*, which must deal with the question of side effects, is a separate optimization.

Diagram c shows an expression for which changing the order of evaluation of the operands allows better code to be generated. This is the *smallest* example of such an expression, and since expressions of this complexity are extremely rare in user programs [31], it is clear that this optimization is of limited value. Nevertheless it has received a great deal of attention in the literature [7], partly because it has been confused with the targeting problem, and partly because it seems so cleanly defined, easy to study.

For simple expression trees, without redundant expressions or side effects, there is a simple algorithm involving a bottom-to-top tree walk for finding the optimal evaluation order. The most that need be done to the original program is that, for some expressions with two operands, the order of evaluation of the operands is switched. Complications are introduced when trees contain either initializations or uses of CCV'S; in addition, it may be desired to treat more than one tree, as with a sequence of statements in a block each involving an expression tree. Either of these complications makes the evaluation order problem NP-complete. It may be necessary to interleave evaluation of operands of an operator, not just to switch them.

We are currently using a simple heuristic modification of the algorithm for single expression trees, to handle single trees containing references to CCV's. This is described in [25]. A more sophisticated algorithm, involving a constrained sorting of the tree nodes, is under consideration (also see [1]).

7.5. Other tree transformations

An interesting and useful set of optimizations meet all the following requirements:

- They are easily characterized as transformations of the program tree.
- They are target machine independent.
- The conditions necessary to apply any transformation can be verified by inspecting a fixed-size subtree of the program tree.

These optimizations are performed by the *tree transformation* phase. This phase is driven by a table of transformation patterns. Each pattern consists of two subtree skeletons; the first describes the conditions that must be met for some transformation to be applicable; the second describes the result of the transformation. (The library of patterns is organized to speed the search for applicable transformations; this is discussed in section 9.) The phase is

organized as a tree walk; at each node, all transformation patterns which might be applicable are *matched* against the local portion of the program tree. This pattern-matching operation uses *variables*: the nodes which match the leaves of the pattern subtree are remembered, and can be referred to in the result subtree.

Some examples which make this process clear are given in figure 9. Diagram a is a simple example given in [50]. The transformation being described is from "(*e*1 - *e*2) > 0" to "*e*1 > *e*2", for any expressions *e*1 and *e*2. This example already raises some interesting issues.

- The same transformation is applicable for each of the other five comparison operators; the pattern notation should be rich enough to describe all six transformations with only one pattern.

- The pattern matcher should understand that some operators are commutative, and indeed that the comparison operators have a peculiar variation of commutativity: "0 < (*e*1 - *e*2)" is equivalent to "(*e*1 - *e*2) > 0".

- There is a negative phase ordering problem between this phase and FLOWAN. If "*e*1 - *e*2" is represented by a CCV, the transformation cannot be performed; but if the transformation could have been performed, perhaps "*e*1 > *e*2" could have been represented by a CCV instead. One possible resolution of this problem is to have a "canonicalization" transformation phase before FLOWAN: the reverse transformation (see diagram b) is performed, in anticipation of creating as many CCV's as possible.

Diagram c illustrates the distribution of multiplication over addition. Here, some of the leaves of the subtree must match constants (i.e. leaves of the program tree), but the values of the constants are not specified. This optimization is not useful by itself, but occurs when the resulting "+" node can be involved in further optimizations. In particular, it occurs when the newly created constant which is the right-hand operand of the node can be "folded" into another constant by taking advantage of the associativity of addition.

This brings up another issue. The tree transformation phase being described cannot be responsible for taking advantage of associativity of operators. The TCOL representation of these operators is binary, and thus the size of the subtree that must be examined in order to find associativity optimizations cannot be bounded. One solution of this problem is to represent such operators as n-ary, that is, to allow them to have any number of operands, and to combine them during this phase, perhaps sorting the operands (assuming the operator is commutative) to expose as much constant folding as possible. We will probably use a different scheme, involving a separate phase, in order to retain the simplicity of having to deal only with binary operators in later stages of the compiler.

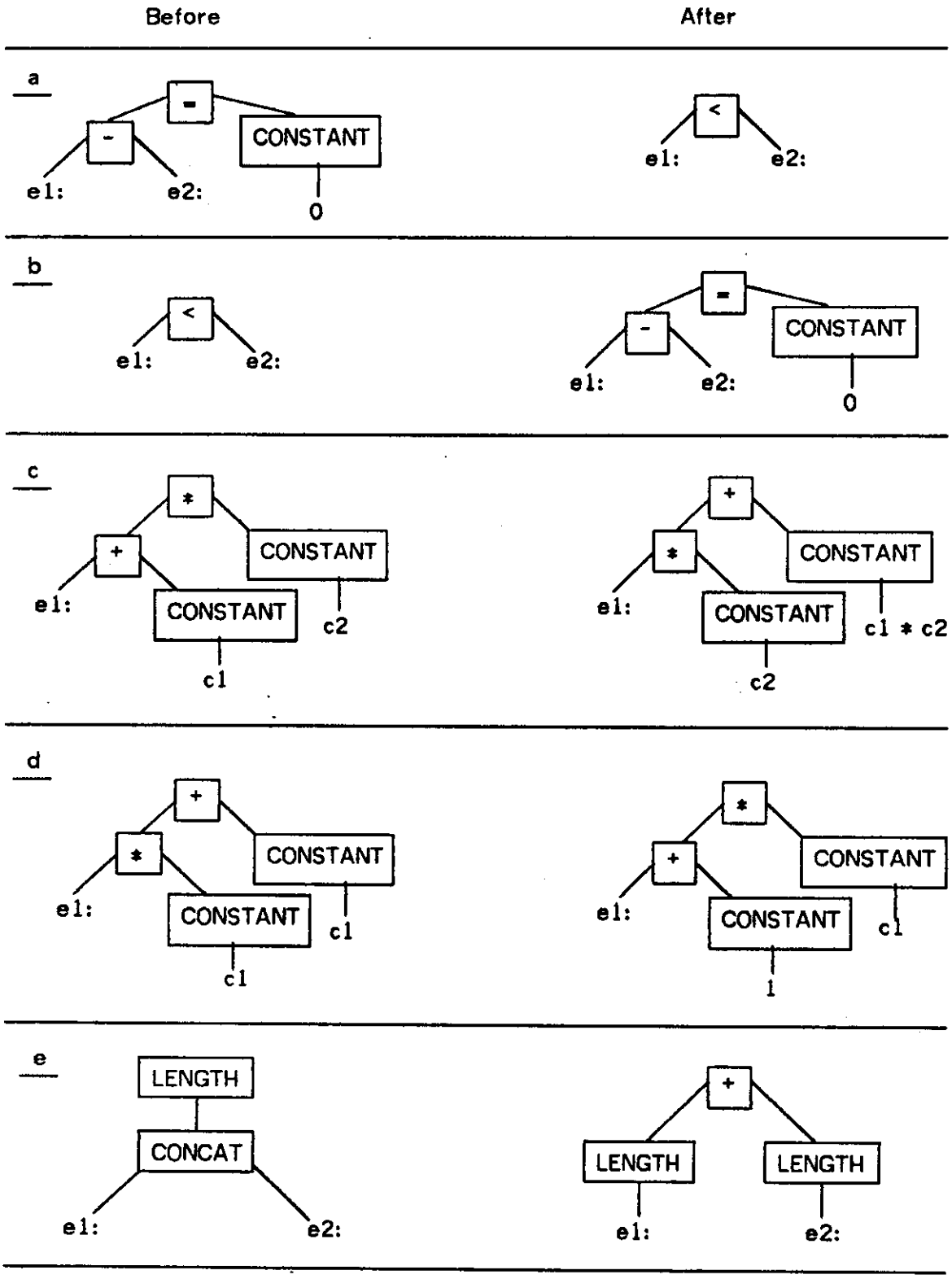


Figure 9: Tree transformation phase examples

The distribution optimization is useful, as mentioned above, only when there is some potential for using the "+" node so created in some further optimization. In order to detect this while examining only the fixed-size subtree, we must somehow encode the potential for further optimizations as "context" information in the original "*" node. We expect to do this by making a guess about this potential during the context phase (see section 7.1). This will be discussed in more detail in section 7.6.

The reverse of the distribution transformation (see diagram d) can also be useful, if it produces a new "+" node whose constant operand is 1, and if on the target machine the addition of 1 is easier than the addition of other constants. This transformation is thus not entirely target machine independent, in that it is useful on some machines but not on others; however, it may be useful to include such transformations in this phase. Notice that two of the nodes which match leaves of the pattern subtree must be identical (in this case, identical constants). This is denoted by having one leaf refer to the pattern variable defined by the other one.

Diagram e describes a transformation mentioned in both [19] and [16] as being a useful test case. It is reassuring that the pattern-result notation is concise enough to describe this optimization cleanly and elegantly, without recourse to details of the design or implementation of the tree transformation phase.

7.6. Access mode determination ("AMD")

Many target architectures allow address computations that do more than simply indexing and indirection. A standard example is the S/370, which frequently allows indexing with two index registers. More recent examples are the DEC VAX-11/780 [53] and the LLL S-1 [44]; these both allow indexing combined with (implicit) shifting of the index register, and double indexing combined with indirection. For these architectures, the question of how much of the program calculation can be performed by the addressing hardware is non-trivial; the complication is compounded by the possibility of using CCV's to assist with address computation. Even for the DEC PDP-11, with a simple addressing architecture allowing simple indexing and indirection (possibly combined), the use of CCV's makes the problem sufficiently difficult that the simple algorithms presented in [25] for solving it are sometimes inadequate.

The *Access Mode Determination (AMD)* phase assigns to each program tree node a list of the ways in which it could be represented as an instruction operand. For instance, consider the BLISS expression

..(.x + 4)

using the DEC PDP-11 as a target machine. For readers unfamiliar with BLISS, "." is the explicit deferencing operator. Thus x is being used as a variable which holds a pointer to an array of pointers. $.x$ is the value of the variable, that is, an address; $.(.x + 4)$ is the value retrieved by an array access, that is, another address (one of the pointers); and the value of the whole expression, $.(.x + 4)$, is the contents of the location pointed to (an integer, perhaps). Suppose it appeared in an operand context, such as the source of an assignment. The assignment would eventually result in the generation of a MOV instruction, of which the source operand could take any of ten forms:

@4(R3)	-- if .x were loaded into register **R3.**
@0(R3)	-- if .x+4 were a CCV, loaded into register **R3.**
@(R3)+	-- same as the above, but this is the last reference to the CCV.
@6(SP)	-- if .(.x+4) were loaded into a memory location six bytes distant from the top of the run-time stack.
@(SP)+	-- same as the above, but the memory location is at the top of the stack, and this is the last reference to the expression.
@R3	-- if .(.x+4) were a CCV, loaded into register **R3.**
6(SP)	-- if the whole expression, ..(.x+4), were a CCV, loaded into a memory location six bytes from the top of the stack.
@SP	-- same as the above, using the top of the stack.
(SP)+	-- same as the above, at the last reference to the CCV.
R3	-- if ..(.x+4) were a CCV loaded into register **R3.**

Of these, three involve the use of the auto-increment feature of the PDP-11 hardware: the third, the fifth, and the ninth. This feature will be discussed in section 10; at this stage of compilation, they are treated as identical to the forms directly above them -- the second, fourth, and eighth respectively. Skeletons for any of the other forms could be associated with the tree node for "$..(.x + 4)$", depending on which of the subnodes could be represented by CCV's (in the absence of CCV's only the first and fourth forms are relevant). With each skeleton there is a list of "actual parameters" -- indications of which subnodes must be loaded into registers or memory locations in order to use the form.

We will discuss in sections 8 and 9 how these forms are used in code generation. An interesting point of methodology is raised by this phase, for we are (tentatively) following a

different strategy with operand access modes than we follow in the other phases. Specifically, we provide an exhaustive enumeration of all the access modes relevant to a node. Our policy throughout the compiler phases, whenever several possible ways of generating code are available, has been to choose one (possibly correcting it later) and ignore the rest. This is a straightforward way of avoiding slow searching algorithms (though it does not, of course, ensure that the compilation process will be fast). We have abandoned it in this case, in the belief that in general there will be few access modes associated with a node. Most nodes cannot be computed at all by effective address hardware; for these nodes the only possible access modes are those for direct access to the various kinds of registers and memory. Skeletons for these are usually not associated with the nodes (i.e. they are implicit); this greatly decreases the cost of the AMD phase.

This phase is parametrized by a list of the possible operand access modes (AM's), each associated with the skeleton program subtree that it represents. This list is preprocessed so that, using a simple bottom-up tree walk, access modes can be associated with nodes without the use of pattern matching. The preprocessing involves creation of something like a finite state machine, in which the "state" of a node depends on its operator and the states of its two operand nodes.

There is a complete set of mutual (negative) phase ordering problems between FLOWAN, the desirability analysis phase, the phase to perform associativity transformations, and the AMD phase. We have put FLOWAN first arbitrarily, but we will attempt to resolve the conflict between the latter three phases by the "wild guess" method. The context phase attempts to guess whether a node will be involved in an address computation, or in an associativity transformation, by using a set of very approximate, local, easily computed criteria. The desirability phase reads the record left by the context phase, giving less preference than usual to using a CCV for a node if the node is likely to be active in the later (associativity and AMD) phases.

8. Register allocation

The term "register allocation" has been used in the literature to describe at least three related activities. We will call these *local*, *pseudo-local*, and *global* register allocation.

Local allocation is two tasks: selecting accumulators and other specialized storage locations for evaluation of simple arithmetic expressions, and selecting storage in which to save the results if they are not used immediately in the evaluation. Historically [47], this topic has also included the targeting and evaluation order problems discussed in section 7.4, but we have treated them separately here.

Pseudo-local allocation is an intermediate task, the allocation of storage for an entire basic block. Consider the compilation of a long sequence of statements, involving much arithmetic, subscript calculation, and indexing, for a target machine with a limited number of accumulator and index registers. In the compiled code, variables, subscripts, and intermediate computation results will constantly be moved into registers, used, perhaps modified, and returned to memory. The task of pseudo-local allocation is to choose the "right" registers to use for each action that occurs, and thus to reduce the amount of traffic between registers and memory. [26] is a classic paper on the subject; various other algorithms have been presented that are more suitable for one-pass compilers.

Global allocation is the allocation of storage for more than one basic block, or an entire subroutine, at a time. Successively more ambitious compilers have attempted to keep constants and CCV's in registers [8], to keep user variables in registers [25], to keep variables in registers during loops and other bounded program regions [21], and to keep variables in registers during arbitrary program segments [5]. Other techniques are possible. The goal of global optimization may be to reduce the traffic between memory and registers, to reduce the traffic between registers, to allow use of register operands instead of memory operands in instructions, to reduce the amount of memory used, or any combination of these.

There is a difficult phase ordering problem in the relation between local allocation, global allocation, and code generation. Local allocation is closely tied to code generation in an obvious way: the storage which is required for the result (and operands) of an expression is determined by the instructions used to compute it. Global allocation, on the other hand, not being a one-pass process, must be done separately from code generation. Moreover, local allocation can make good use of information about register availability and correlations between registers and variables, derived by global allocation, while global allocation can make good use of information about register availability and prospective loads and stores, derived by local allocation.

Our approach to this problem is related to those presented in [5] and especially [25, 29]. Register allocation and code generation are not two but three phases. The first of these can be thought of as a "guess" at code generation.

In order to discuss these three phases it is first necessary to define a basic concept, the *Temporary Name (TN)*. TN's are records, used to hold all the information about the results of register allocation that must be available to the code generator. They are used to represent entities that must be allocated storage, such as variables or the results of expressions; but a TN can only correspond to one storage location, so that it may be necessary to represent a single variable with more than one TN. Note that local, pseudo-local, and global allocation are

unified, in the sense that they all use TN's; this facilitates communication between them.

The first of the three phases can be called *TN assignment*; it determines the correspondence between TN's and program entities. The second phase is *packing*; it establishes the correspondence between TN's and target machine storage locations. The third phase is *code generation* -- a scarcely avoidable misuse of a term that we have used earlier to describe the whole compilation process after parsing. TN assignment is discussed in section 8.1, packing in section 8.2, and code generation in section 9. The first two phases together are referred to as TNBIND, the name for the module of the Bliss-11 compiler which performed their functions.

8.1. TN assignment

The association of TN's with local entities, such as expression results or operands, is done by a traversal of the tree (or of the execution list) closely related to that done for code generation. The description of this traversal is postponed until section 9. The same library of code sequences is used; however, instead of associating a code sequence with a node, this phase associates information abstracted from the code sequence. This includes not only the requirements for various types of local storage by that sequence, but also usage information; thus, one or more TN's may be associated with a node, and with each TN there will be information about when it is used (i.e. at what nodes it appears in instruction operands) and how it is used (i.e. the relative costs, in words of code or in memory references at execution time, of allocating the TN to the various different categories of storage).

The derivation of this usage information is discussed in [29]. In conjunction with the information left by the AMD phase, a two-dimensional table of target machine dependent information is used: for every category of storage (there may be several kinds of registers, plus static and stack memory), and for every formal parameter of every instruction-operand tree skeleton, there is an entry indicating the cost of trying to use a unit of that kind of storage as an actual parameter in that skeleton. (The instruction-operand tree skeletons are discussed in section 7.6.) Note that each TN will, in general, appear in several instruction operands, using different access modes (e.g. direct access, indexing, indirection).

Note that the abstraction of TN-related information from code sequences can be done by the compiler-compiler system; that is, the library of code generation templates may be separated into two sub-libraries, one of which contains only information relevant to TN assignment, the other of which contains information for code generation. This can be expected to save substantial storage during both compilation phases.

TN's must also be associated with global entities, such as variables (including CCV's). This

is done separately; it is done before the local assignment pass, in order to associate usage information with global as well as local TN's. The global assignment phase must guess when it would be useful to assign more than one TN to a variable; this guess would ordinarily be based on the loop structure of the program, as determined by FLOWAN. Note, however, that if the guess is overoptimistic and assigns too many TN's, little harm is done; if there are not enough fast registers to accommodate the most important of these TN's, they will revert to the same location as the others, just as if only one TN had been assigned to the variable.

After the assignment passes, complete lifetime information is associated with each TN. (A TN is *alive* at some node in the program tree if there is some possible path of execution, starting at that point, along which the next reference to the TN fetches from it, rather than assigning a new value to it. Thus if two TN's are both alive at some point, they may not be allocated to the same location.) Each TN has a list of (disjoint) "lifetime pairs": pairs of points in the program, between which the TN is alive. Given two TN's, it can easily be determined whether their lifetimes overlap, by inspecting their lists of lifetime pairs.

Note that this organization of lifetime information is completely different from that given in [4], in which the information is associated with basic blocks (or, at a finer grain, with program points). Thus different flow analysis algorithms must be used to gather it efficiently. The organization of lifetime information is determined by the plans for the packing phase (discussed in the next section).

The TN assignment phase is also responsible for gathering preference information. This will be defined and discussed in the next section.

8.2. Packing

It might seem that the packing phase would be very target machine dependent, because of the wide variety of register configurations in architectures currently in use:

- There are machines with separate index registers and accumulators, machines in which some registers can serve both functions, and machines in which any register that can serve the former function can also serve the latter.

- There are machines in which some registers of a class are arbitrarily restricted. Thus the DEC PDP-11 has six floating-point registers, but two of them cannot be used for a wide variety of operations.

- On some machines, consecutive pairs of registers are required for some operations. Sometimes the first member of the pair must be even, or must be odd.

- Some storage classes are finite (such as registers), while some are effectively infinite (such as various types of memory).

However, the target machine dependence of this phase is very easily parametrized. A notation has been developed to describe how many members each class of storage has, how the classes overlap with each other, and how they are related to each other (e.g. class A is the restriction of class B to its first four members); once a target machine has been described in this notation, the packing phase has enough information about it to run. (Note that this machine description is also used by the TN assignment phase, to describe restrictions on TN's and to organize the usage information about them.)

The fundamental data structure of the packing phase is an *interference graph* (see figure 10). Each graph node represents a TN; a link between two nodes indicates that the two TN's have overlapping lifetimes (i.e. may not be allocated to the same location). (It would also be possible to use the reverse of this graph, in which a link indicates that the lifetimes do not overlap.)

Superimposed on the interference graph is a *preference graph* (see figure 11). Each link in the preference graph indicates that the two TN's "should" be allocated to the same location, to avoid loads or stores. These links are labeled with counts of the loads and stores involved, possibly weighted to give more importance to those which occur in loops.

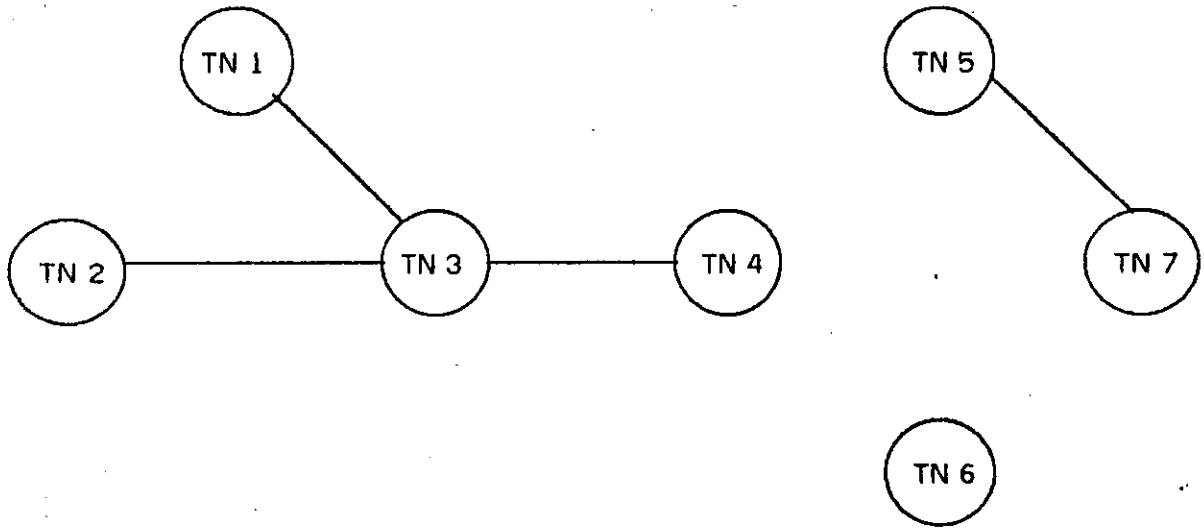


Figure 10: An interference graph

A good packing must meet four criteria:

- No two TN's which are connected by an interference arc may be packed in (allocated to) the same storage location.

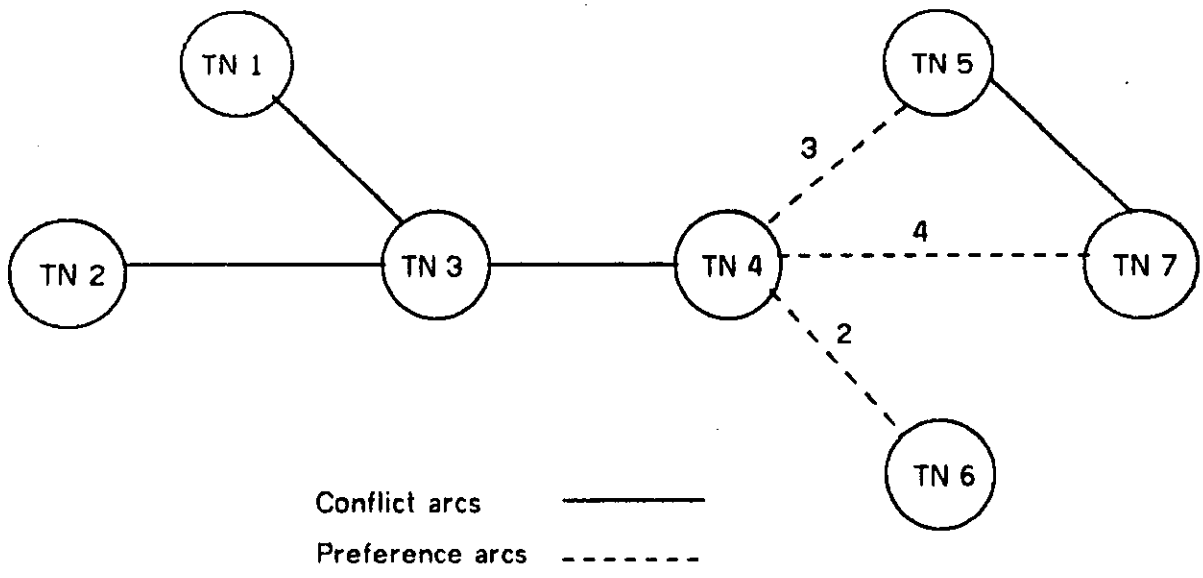


Figure 11: A preference graph and an interference graph

- The cost measure determined by summing the relative costs of all TN's, as derived from the usage information discussed in the previous section, and from the knowledge about which storage class each TN has been packed in, should be kept low (perhaps minimized).

- The profit measure determined by summing the values of all preference arcs that connect two TN's packed to identical locations should be kept high (perhaps maximized).

- For some storage classes, there may be a cost associated with using any member of the storage class, which is fixed regardless of how the member is used. For instance, a run-time convention for the preservation of register contents across routine calls may require that if a register is used by a routine, it must be saved at the beginning of the routine and restored at the end. Thus there is a cost measure determined by the number of locations (of certain classes) which are used in a given packing; this should be kept low.

It can easily be seen that packing is a particular kind of graph coloring problem. Also, the resemblance of packing to bin packing problems has been observed in [29]. A number of algorithms have suggested themselves to us, in addition to the many which have appeared in the literature, and we plan to do considerable experimentation with the packing phase. Following the policy described in section 7.6, we are primarily interested in algorithms that are one-pass, usually "greedy", rather than those which exhaustively search the space of

possible packings. However, experience with systems described in [25] and [29] has shown us that a modest, controlled amount of backtracking can be worth its cost.

9. Code Generation

The code generator described here is basically an extension of the one built by Cattell [11]. The extensions are primarily for the use of information gathered in previous phases, including register allocation, access mode determination, and evaluation order and target path determination. Both the code generator and the method of target machine description are of interest; we will summarize them in the following sections, but for details we refer the reader to [11] and [13].

9.1. Machine Descriptions and Code Generator Generation

The code generation phase is driven by a library of *code generation templates*. Each template consists of a *subtree pattern* and a *code sequence*. Examples of templates for the PDP-10 are given in figure 12; briefly, if a subtree of the program tree matches the subtree pattern in the obvious way, and various other conditions (not shown) are met, the template is *instantiated*, that is, the instructions given in the code sequence are emitted, their operands being determined from the values of the *pattern variables* (the program tree nodes which matched the leaves of the subtree). A better description of the use of the library will be given in the next section; what concerns us here is how the library is constructed, from a still more primitive machine description.

We will refer to the more primitive machine description as the *MD table*.[1] This representation is genealogically related to ISP [27, 33]: there are descriptions of the storage classes available, descriptions of the effective address calculations, and descriptions of the computation performed by each instruction. The principal difference is that each instruction is represented, not by an algorithm to simulate it, but by a group of preconditions and postconditions (in terms of the machine state, i.e. the contents of various locations) of its execution. It is easier in this formulation to understand the effect of an instruction; some characteristics become harder to describe, mostly involving the effects of different serializations of the address computation steps.

Given descriptions of the instructions, in a TCOL-like notation, it is possible to find a sequence of instructions to evaluate any TCOL tree, by a search process reminiscent of those in many Artificial Intelligence systems. This is too slow to be used in the compiler itself;

[1] MD stands for "Machine Description"; in [11] this is called the MOP table.

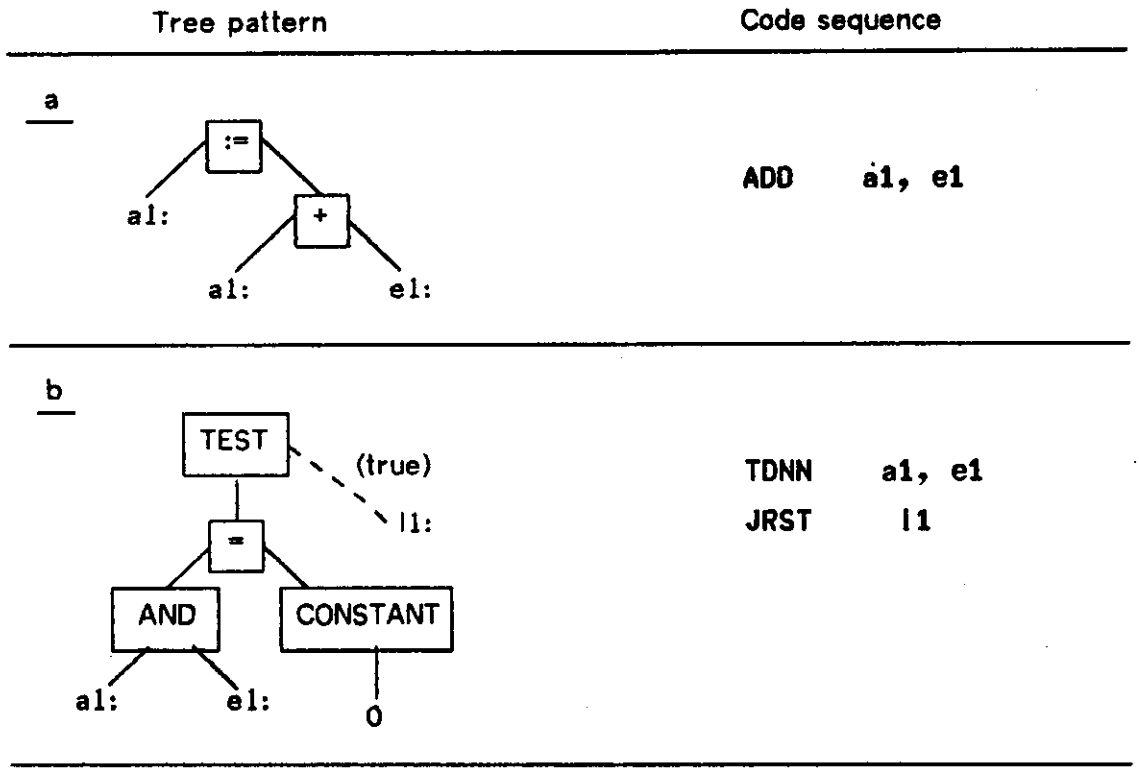


Figure 12: Some sample code generation templates for the PDP-10 (simplified)

instead, it is used to create code generation templates for a predetermined set of "interesting" subtrees. This set is generated using a number of rules. Two rules in particular serve to insure that some kind of code can be generated for any program:

- There is one template for every TCOL operator, whose pattern tree is simple enough that it is always applicable wherever that operator occurs. The first example in figure 12 is one of these.

- There is one template for every possible data move (load or store); that is, if there are n different kinds of storage on the target machine, there are n^2 templates for generating code to move a data item from one type of storage to another.

In addition there is at least one template for every instruction, in which the pattern tree describes the effects of that instruction; the second example in figure 12 is an example of these. Other rules encourage the discovery of "clever code sequences"; however, finding all of these is equivalent to solving the halting problem, and the number of such sequences that are ultimately represented by templates depends on how long the template search is left to

run.

Some arithmetic operators in a given TCOL may not appear anywhere in the MD table. For instance, the logical exclusive or (XOR) operation does not appear anywhere in the description of the DEC PDP-11/20 architecture; thus using the MD table alone it would not be possible to determine how to generate code for this operator. The search process must know about the following *language axiom*:

a **xor** *b* ≡ *a* **and not** *b* **or** *b* **and not** *a*

This must be included in the set of axioms alluded to in section 4.2. For typical machines, the only control operators which appear explicitly in the MD table are **go to** ... and **if** ... **then go to** ..., so templates for the other control operators must also be derived using axioms about control.

The most important omission from the MD table is information describing the run-time conventions and subroutines, or *CWVM (Compiler Writer's Virtual Machine)*, of the planned compiler. This includes:

- fundamental conventions, such as the representation of simple boolean variables;
- data structure conventions, such as the representation of arrays, records, and invocation frames;
- detailed operating rules, such as the names and calling sequences of precoded support routines.

We have dealt with these in an *ad hoc* manner up to now -- an approach that is reasonable for languages with minimal run-time support, such as Bliss. In the near future we expect to study the CWVM issues more closely.

9.2. The Maximal Munching Method

The hasty description of the use of code generation templates in the previous section left out the most interesting parts. In this section we will describe the template library in more detail, and the algorithm for walking the program tree, known as the "Maximal Munching" algorithm.

The template library is divided into several sub-libraries, or *schemas*. The choice of schemas to be searched to find a pattern matching a node is determined by the context of the node. The two patterns in figure 12 would be found in different schemas: the first would be in an operand-context schema, the second in a flow-value-context schema. An additional classification of patterns is done, to speed up the code generation search: they

are hashed by the principal operator. Thus the first pattern would only be tested in generating code for an (integer) addition; the second only for an equality comparison in a flow-value context. Further hashing may prove to be useful in some situations.

All templates make some requirements on the forms of their operands: the AM's associated with them by the AMD phase must include some which are suitable in the instruction operands in the template's code sequence. However, the pattern match with a template does not fail if the operands do not meet these requirements; instead, load or store code is generated so that they do meet them. This is the use of the *fetch decompostion* rule; the corollary *store decomposition* rule is used when the pattern imposes requirements on the destination of the principal operator. The top node in a pattern tree in the operand-context schema is always assignment (":="); but an actual assignment need not be present in the program tree in order to match such a pattern, thanks to fetch and store decomposition. In the TN assignment phase, the use of fetch or store decomposition causes a new local TN to be created and associated with the appropriate node; in the code generation phase, the TN is used as the destination of the load or store code that is generated.

The second example in figure 12 is a pattern for one of the special cases of testing for equality. Not only is there a requirement on its right operand (it must be a constant, whose value must be zero), but also there is a requirement on the operator of the node which is its left operand. For such patterns to be used, the library must be ordered, so that they are always tested before the "general case" patterns are tested. In addition, and most importantly, the order in which the program tree nodes are treated must be top down. In this example, for instance, the code generator must not reach the **and** node before reaching the "=" node -- otherwise code would be generated for the logical **and** operation, and the special case would no longer be applicable. This backwards traversal of the program is absolutely crucial if the large amount of special case testing characteristic of good code generators is to be kept intellectually manageable. The use of templates sorted so as to "bite off" as large a subtree as possible of the program tree, with the backwards traversal of the tree, is described in [11] as the "Maximal Munching Method".

The mechanism of the backwards tree traversal is meshed with the mechanism of fetch decomposition. If an operand does not require fetch decomposition, that is, it is a tree which exactly matches the skeleton tree for some appropriate access mode, no code need be generated, in order to represent it as an instruction operand. Usually, however, one of its subtrees does not match the corresponding part of the access mode tree; or, the whole operand tree does not match any access mode tree. In this case, code must be generated to evaluate the non-matching tree, leaving the result in the designated destination; this causes the code generator to be called, recursively.

For some target machines, especially those with large instruction sets, there may be several code generation templates whose pattern trees differ from each other only in the requirements on the access modes of the nodes. For instance, two of the PDP-10 instructions to do integer addition are represented by the same pattern tree; the only difference is that one leaves its result in an accumulator, while the other leaves its result in the other operand (generally a memory location). Neither of these templates is a special case of the other. The choice between them, therefore, cannot be determined beforehand; that is, in the ordering of templates in the template library, neither of these may precede the other. Instead, the code generator must choose between them, by estimating the amount of fetch and store decomposition that must occur for each one, and using the one that will cost the least. Because register allocation has been done in a previous phase, the estimate may be constrained to use local information only: information about the nodes which match the root and leaves of the pattern tree.

In the presence of the more advanced evaluation order algorithms hinted at in section 7.4, and the code motion optimizations described in section 6, it may prove to be necessary to walk the execution list rather than the program tree to generate code. However, this will not affect the code generation templates, or even the Maximal Munching Method. It means only that a coupling is broken: the coupling described above, between the discovery that there is not a perfect match between some operand node and the corresponding AM skeleton tree, and the subsequent call of the code generator to evaluate that node. This raises some annoying implementation problems, but these are outside the scope of this paper.

The idea of generating code using libraries of templates is not new, and it is useful to compare the system described above with others that have appeared in the literature. A well-known example was used in the Fortran-H compiler [21]. In that system, there is one template per operator. The template contains all the instructions which could ever be used in coding that operator, including loads and stores. Each instruction is associated with a vector of bits which describes something about when it is to be used. When the template is applied, another vector of bits describing the "status" of the operands (i.e. whether they are in registers or in memory, and whether or not they must be in registers after the operation) is put together; then a curious algorithm is used, which will not be described here, to use this vector and the bit vectors in the template to determine which of the instructions will be emitted.

The Fortran-H scheme is not addressed to the problem of target machine independence, and thus is less general than our system in obvious and perhaps easily remediable ways: the meanings of the operand status bits are closely tied to the S/360 architecture; there is little

or no provision for the use of interesting AM's. More important are differences in the structure of the templates. In our system, load and store instructions do not appear in the templates for the basic operators; this makes the latter simpler, while somewhat complicating the code generation process. In addition, if there are several instructions each of which could be used to evaluate an operator, depending on where its operands are, we will have several templates instead of just one; this also contributes to the simplification of individual templates. The simplification of the templates contributes to the *maintainability* of the template libraries; at present we are concerned that the libraries be understandable and maintainable, because we anticipate that this will contribute to usability of the compiler-writing system as a whole by persons other than ourselves.

There are also differences in descriptive power between the Fortran-H templates and ours. There does not appear to be any mechanism in the former system to take advantage of knowledge about special operand cases, such as operands which are constants, or constants whose values are known. (However, some particular optimizations, such as converting multiplications by powers of two into shifts, are described; these are the kind of transformations that we discuss in section 7.5.) Even more specialized cases, such as the example shown above in which one operand must itself have a particular operator, likewise cannot be accommodated into the Fortran H templates.

Snyder [49] describes a template system in which there may be more than one template for each operator; in addition, some provision is made for special cases of constant operands (although the further special cases involving multiple operators are not detected). Thus the code generation algorithm itself is somewhat similar to our own. There is no provision for a separate register allocation phase, or for operations which require completely temporary storage (i.e. storage which is distinct from the operands and the result).

The code generation templates described by Ripken [43] are very close, in their design and in the overall approach to the problem of target machine independence, to the templates we have described here. The principal difference is in the control structure of the code generation algorithm itself. Ripken's algorithm first walks the tree from the bottom up. (As we have noted, this precludes the use of special-case templates, such as the second one in figure 12). This phase addresses the problem described earlier, of multiple templates for the same tree pattern: all the templates applicable to a given node are recorded (that is, their costs are recorded, though no code is generated). The next phase uses the cost information to allocate registers locally, walking back down the tree and choosing code sequences. This has the flavor of a dynamic programming algorithm; though there is no provision for any interaction between local and global register allocation, the local allocation may be optimal for a given global allocation because of the exhaustive enumeration of possible code sequences

in the first (bottom-up) phase. (Optimality may be lost in some circumstances because of some simplifications in the cost recording mechanism, but in the usual cases it is achieved.)

The LIS [12] system for constructing code generators is of interest, because it involves a primitive description of the target machine, with separate descriptions of each instruction; the instructions are automatically grouped by the system, so that different instructions can be chosen for a given operator, depending on what kinds of storage locations are to contain the operands and result. However, despite claims to the contrary, this system is only minimally addressed to the issues of target machine independence. It would require substantial recoding (and redesign of basic data structures) to describe machines other than the UNIVAC 1108; and describing machines that are not either two-address or general-register architectures would seem to be impossible.

10. The FINAL phase

The output of the code generation phase is a doubly linked list of intermixed instructions and labels. These are in a format that retains only minimal target machine independence. The instruction operands are lists of actual parameters for the subtree skeletons of the AM's; these are constants or references to storage locations. In principle the effect of any instruction could be reconstructed, forming a TCOL tree, from the MD table entry for it and from its operands. The reason for this is that further optimizations are done on the program while it is in this format.

Notwithstanding the arguments in section 4.2 for the use of tree-structured intermediate representations, there are some optimizations which are more easily done on the final linear form of the object code. These may arise from any of several circumstances:

- Instructions may have effects other than those for which they were emitted. For instance, on a machine with condition codes, an instruction which was emitted by the code generator because it performs some arithmetic operation may, incidentally, set the condition codes. Instructions whose sole purpose is to set the condition codes can sometimes be subsumed in instructions which accidentally set them. It is not desirable for the code generation phase to "know" all the effects, or indeed any of the effects, of the instructions it emits; this knowledge is more appropriately delegated to a separate phase.

- Reverse-hoisting code motion (referred to as *cross-jumping* in [25]) that was not evident at the TCOL level may become evident at the instruction level. This is because operations which are single nodes in the TCOL tree (or single instructions in triple or quadruple program representations) may produce multiple instructions; and frequently the cleanup instruction sequences for non-identical TCOL nodes may be identical. This is especially true for operations such as subroutine calls, which are likely to involve highly stereotyped instructions to maintain the CWVM, such as instructions to cut back the run-time

stack.

- The code generator is more intellectually manageable if it does not have to be too careful about the senses and destinations of conditional and unconditional branch instructions. That is, it may emit a branch instruction whose destination is another (unconditional) branch instruction, or a branch whose destination is the next instruction, or a conditional branch which only skips over an unconditional branch -- code that is correct but wasteful in obvious ways. It is easiest to leave glitches like this to be cleaned up by a conceptually separate phase.

- Some optimizations can only be performed with a detailed knowledge, not only of the effects of instructions, but of their sizes as well. A classic example of this is the resolution of short and long branch instructions on the PDP-11 [25, 51]. A branch instruction can take two forms: the short form if its destination is known at assembly time and is within 256 bytes of the origin, or the long (costlier) form otherwise. The algorithm to determine which form is used for each branch must take account of the number of bytes required by each instruction, including, of course, the branch instructions themselves.

FINAL performs some optimizations that are usually called "peephole" optimizations [40]. However, since the whole list of instructions is kept in core, that term is inappropriate; in fact, for any optimizing transformation which involves two or more related instructions, FINAL allows for indefinitely many intervening instructions, provided that they are irrelevant to the optimization.

The FINAL phase relies on control flow graph data structure attached to the program, as described in section 6. The structure of the graph may change from FLOWAN to FINAL, and a basic block now represents a list of instructions rather than a list of program tree nodes.

The control structure of the FINAL phase is based on an infinite loop, as discussed in section 3.3. Four classes of optimizations are performed, each of which may open up new opportunities for most of the others:

- removal of useless instructions: an instruction whose effects are ignored by subsequent instructions may be removed. A complete data flow analysis is performed on the control flow graph, so that at any instruction, it is possible to determine whether the locations it modifies, including "status" locations such as condition codes, are *alive* after it is executed (see section 8 for the definition of aliveness).

- branch and label optimizations: the consequences of deliberately careless code generation, described above, are corrected.

- cross jumping: the reverse hoisting optimization mentioned above is performed. For each basic block, the instructions which could (if it were useful) be moved to the end of the block are collected into its *postlogue set*; the postlogue sets of basic blocks which share a common join point are compared with one another.

- peephole transformations: a set of optimizations on instructions, pairs of instructions, or larger groups, are performed; the optimizations are taken from a library of patterns which will be described below. Typical optimizations may reduce two instructions to one, or replace two instructions by two cheaper instructions, or replace one or more operands by cheaper operands.

Passes are made over the whole program, or over individual basic blocks, until convergence is achieved, i.e. until no further optimizations can be found. Some optimizations, such as the resolution of short and long jumps described above, may be done as separate subphases. In fact, we know that there are negative phase ordering problems in the interactions of some of the different peephole optimizations, and the simple structuring of FINAL as an infinite loop is not sufficient to take account of these. Not only must there be some splitting of the phase into subphases, but also the criteria for doing this splitting must be specified in terms of easily described characteristics of the target machine; the splitting can then be done by the FINAL phase generator, or if necessary by the compiler writer.

There are non-trivial problems in the parametrization of the target machine dependence of parts of FINAL. These are primarily the problems of dealing with the peephole optimizations: the library of them must be generated; their applicability conditions must be determined; and the conditions under which they improve, rather than worsen, code must be determined. We have chosen to allow the compiler builder to "suggest" optimizing transformations; this is reasonable if there are not too many of them. (The current Bliss-11 compiler [25] uses about 30.) A program, the *legality checker*, determines the conditions under which a transformation is correct, i.e. it lists a set of restrictions on the instruction operands and other parameters to the transformation pattern, such that it does not change the meaning of the program. Experience has shown that the human compiler writer is very likely to do this task incorrectly. Sometimes there are errors due to neglect of special cases, or even outright misunderstandings of the effect of the transformation; sometimes there are clerical errors. The last component of the parametrization is a *cost function*: a function that, given an instruction in FINAL format, returns some suitable measure of its cost (for example, its size in bytes, or its execution time in cycles). This function is easily derivable from information in the MD table. Whenever a peephole transformation is found to be applicable at some point in a program, the costs of the instruction sequences it creates and destroys are compared, to determine whether it should in fact be applied.

Figure 13 shows an typical pattern for a peephole optimization, for the DEC PDP-11. The purpose of the transformation is to reduce two instructions to one. The second instruction increments a register by two; since the PDP-11 is byte-addressable and each word consists of two bytes, this is a common operation, used in stepping a pointer through an array of words. The transformation subsumes the second instruction into an operand of the first, by

Before		After	
OPC1=<any>	,@R_1,	OPC1	,(R_1)+,
ADD	#2,R_1		

Figure 13: A typical peephole optimization

making use of the "autoincrement" feature of the PDP-11: the use of the autoincrement access mode automatically steps the register by two, while acting otherwise as if simple indirection through the register had been done.

This example, simple as it is, raises some interesting points about the pattern notation, the legality checker, and the FINAL phase as a whole:

- The opcode of the first instruction is irrelevant, and the commas surrounding its operand indicate that it may have other operands as well. The notation must be rich enough to allow for a great deal of unspecified information. (Implicitly it is assumed that any instructions whose effects are known not to be relevant to this optimization may come between the two relevant instructions.)

- Correspondingly, the legality checker must be able to detect when the unspecified information must obey certain limits. This transformation is invalid if the opcode is one which operates on bytes instead of words (in these cases the register is stepped by one instead of two). It is also invalid if there is an operand after the one given, and this other operand makes reference to the same register. The legality checker must make sure that both these conditions are imposed.

- The reader may wonder why the given optimization wasn't included in earlier compiler phases, perhaps by means of cooperation between the AMD phase and the phases that use the code generation template library. The reason is really that we don't understand it well enough. (This is true, generally, of all optimizations involving the autoincrement and autodecrement features of the PDP-11 and similar features on other target machines.) There is a dangerous tendency for optimizations to slip into the FINAL phase simply because we don't know how, at the current stage of the project, to include them in earlier phases of the compiler. This tendency has to be monitored carefully, because it is almost always true that when an optimization can be performed both at the TCOL (source-program) level and the FINAL (object-program) level, it can be done more powerfully, that is, on a larger scale, at the TCOL level.

The target machine dependence of peephole optimizations is obvious, and probably inherently difficult to parametrize. This accounts for the scarcity of previous literature on peephole optimization techniques. We have reason to hope, however, that using the system sketched here, we can take most of the work of construction of this phase out of the hands of the

compiler writer.

11. Summary

11.1. Optimizations not mentioned above

The reader who is familiar with the optimization literature will undoubtedly think of a number of optimization techniques which were not mentioned in the previous pages. In most cases, we are interested in these techniques, but simply have not integrated them into our framework at this writing. Here are some of the better-known ones:

- Removal of "tail recursion". If it happens that, along some execution path leading to the end of a subroutine, the last thing executed is a recursive call to that subroutine, that call can be relieved of a great deal of the overhead usually associated with subroutine calls. It can be coded as a resetting of the values of the parameters (if any) of the current call, followed by a jump to the beginning of the subroutine. This is an attractive optimization, not uncommon in recursive programs, and it is trivially discoverable in the TCOL representation of the program, supplemented by the control flow graph.

- Topological sorting of basic blocks. It is frequently useful to change the order in which the basic blocks of the program are coded. This may be done to minimize the number of branches from one block to another, and to move them to minimize the number which occur in loops. The problem of finding the best ordering seems to suggest some kind of topological sorting.

- Inter-procedural analysis and optimizations. A wide variety of benefits can be gained by extending the compiler to hold data about more than one procedure in core at a time. These include expansion of procedures in line (i.e. replacement of procedure calls by procedure bodies), acquisition of better information about variable aliasing for use in flow analysis, and automatic allocation of global variables and procedure parameters to registers and static memory locations. Few of us have any familiarity with these optimization techniques, because we are accustomed to compilers that deal with one subroutine or even one statement at a time. But we suspect that the day is coming when small subroutines are the rule rather than the exception, and interprocedural optimizations will be everyday tools rather than exotic subjects of research.

11.2. Status

At this writing, some phases of the skeleton compiler have been implemented; a few (some of the smaller DELAY phases) are close to final form. Other phases exist only in preliminary form or not at all. The phase generator for code generation has been implemented; no other phase generators have even been designed. The phases and their generators, however, are only part of the complete PQCC system.

- TCOL must be exactly specified. It must be possible for an uninitiated programmer, given a parser of any kind which produces as output a parse tree as any kind, to figure out from our description just what semantic requirements his parse tree should meet and how he should meet them. We have started work, therefore, on a detailed TCOL manual.

- A somewhat smaller task, though just as necessary, is the exact specification of LGN, and the standard representation of TCOL in it. Again, we have the beginnings of a manual. Some projects at CMU are already using LGN-TCOL for their own work in language implementation. The short term benefit of this is the use of the (reasonably well-developed) LGN support system; in the long term, of course, there is the benefit for us of additional testing of the code generation system.

- The rules for constructing MD tables must be drawn up. In addition, there must be documentation for the format of the code generation template tables and the table of tree transformation patterns; we expect that users of the system will find it desirable or even necessary to extend these tables "by hand", for reasons outlined in earlier sections. Though this documentation is far from complete, we have had some success with teaching uninitiated programmers how to construct and understand MD tables, and (untested) descriptions of several popular architectures have been constructed by them.

11.3. Acknowledgements

In addition to the authors, a number of other people have made contributions to the PQCC project, some of them substantial. Special mention must be made of Gary Feldman and Paul Hilfinger, who are primarily responsible for the success of the LGN design and the support package. Other participants have included (in alphabetical order) Mario Barbacci, Ron Brender, Steve Byrne, Frank DeRemer, Peter Hibbard, Andy Hisgen, Regis Hoffmann, Lee Szewerenko, and Gideon Yuval.

References

[1] A.V. Aho and J.D. Ullman.
Optimization of straight line code.

SIAM Journal of Computing 1(1):1-19, 1972.

[2] F.E. Allen.
Annotated Bibliography of Selected Papers on Program Optimization.
Technical Report RC 5889, IBM Thomas J. Watson Research Center, March 1976.

[3] F.E. Allen and J. Cocke.
A Catalogue of Optimizing Transformations.
In R. Rustin, editor, *Design and Optimization of Compilers (Courant Computer Symposium 5)*, pages 1-30. Prentice-Hall, 1972.

[4] F.E. Allen and J. Cocke.

A Program Data Flow Analysis Procedure.

Communications of the ACM 19(3):137-147, March 1976.

[5] J.C. Beatty.
Register Assignment Algorithm for Generation of Highly Optimized Object Code.
IBM Journal of Research and Development 18(1):20-39, January 1974.

[6] W.A. Wulf, D.B. Russell, and A.N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December 1971.

[7] J. Bruno and R. Sethi.
Code Generation for a One-Register Machine.
Journal of the ACM 23(3):502-510, July 1976.

[8] V.A. Busam and D.E. Englund.
Optimization of Expressions in Fortran.
Communications of the ACM 12(12):666-674, December 1969.

[9] D.M. Ritchie, S.C. Johnson, M.E. Lesk, and B.W. Kernighan.
The C Programming Language.
Bell System Technical Journal 57(6):1991-2019, July-August 1978.

[10] R.G.G. Cattell.
A Survey and Critique of Some Models of Code Generation.
Technical Report, Carnegie-Mellon University, 1977.

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

[12] Chi Corporation.
Language Implementation System.
manual (undated, Cleveland, Ohio).

[13] R.G.G. Cattell, J.M. Newcomer, and B.W. Leverett.
Code Generation in a Machine-Independent Compiler.
(to appear).

[14] S. Warshall and R.M. Shapiro.
A General-Purpose Table-Driven Compiler.
In *Proceedings of the Spring Joint Computer Conference*, pages 59-65. AFIPS, 1964.
Also in S. Rosen, editor, Programming Systems and Languages, McGraw-Hill (1967), pages 332-341.

[15] W.H.E. Day.
Compiler Assignment of Data Items to Registers.
IBM Systems Journal 9(4):281-317, 1970.

[16] F.E. Allen et al.
The Experimental Compiling Systems Project.
Technical Report RC 6718, IBM Thomas J. Watson Research Center, September 1977.

[17] W. Harrison.
A New Strategy for Code Generation -- the General Purpose Optimizing Compiler.

In *Fourth ACM Symposium on Principles of Programming Languages*, pages 29-37. SIGPLAN-SIGACT, January, 1977.

[18] J.L. Carter.
A Case Study of a New Code Generation Technique for Compilers.
Communications of the ACM 20(12):914-920, December 1977.

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

[20] W. Wulf, P. Feiler, J. Zinnikas, and R. Brender.
A Quantitative Technique for Comparing the Quality of Compiler Code Generation.
Technical Report, Carnegie-Mellon University, (to be published).

[21] E.S. Lowry and C.W. Medlock.
Object Code Optimization.
Communications of the ACM 12(1):13-22, January 1969.

[22] C.W. Fraser.
Automatic Generation of Code Generators.
PhD thesis, Yale University, July, 1977.

[23] J.W. Backus et al.
The FORTRAN Automatic Coding System.
In *Proceedings of the Western Joint Computer Conference*, pages 188-198. AFIPS, February, 1957.
Also in S. Rosen, editor, Programming Systems and Languages, McGraw-Hill (1967), pages 29-47.

[24] C.M. Geschke.
Global Program Optimizations.
PhD thesis, Carnegie-Mellon University, October, 1972.

[25] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
The Design of an Optimizing Compiler.
American-Elsevier, 1975.

[26] L.P. Horwitz, R.M. Karp, R.E. Miller, and S. Winograd.
Index Register Allocation.
Journal of the ACM 13(1):43-61, January 1966.

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

[28] S.S. Coleman, P.C. Poole, and W.M. Waite.
The Mobile Programming System, Janus.
Software: Practice and Experience 4:5-23, 1974.

[29] R.K. Johnsson.
An Approach to Global Register Allocation.
PhD thesis, Carnegie-Mellon University, December, 1975.

[30] W.H. Joyner Jr., W.C. Carter, and D. Brand.

Using Machine Descriptions in Program Verification.
Technical Report RC 6922, IBM Thomas J. Watson Research Center, 1978.

[31] D.E. Knuth.
An Empirical Study of FORTRAN Programs.
Software: Practice and Experience 1(2):105-134, 1971.

[32] D.B. Loveman.
Program Improvement by Source-to-Source Transformation.
Journal of the ACM 24(1):121-145, January 1977.

[33] R.G.G. Cattell.
Using Machine Descriptions for Automatic Generation of Code Generators.
In *Proceedings of the 3rd Jerusalem Conference on Information Technology*, pages 503-507. (publisher or organization?), 1978.

[34] K. Ripken.
Generating an Intermediate-code Generator in a Compiler-Writing System.
In E. Gelenbe and D. Potier, editor, *International Computing Symposium*, pages 121-127. North-Holland, 1975.

[35] H. Ganzinger, K. Ripken, and R. Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In B. Gilchrist, editor, *Information Processing 77*, pages 535-540. North-Holland, 1977.

[36] Joseph M. Newcomer.
Machine-independent Generation of Optimal Local Code.
PhD thesis, Carnegie-Mellon University, May, 1975.

[37] Nils J. Nilsson.
Artificial Intelligence.
Technical Report 89, Stanford Research Institute, Artificial Intelligence Center, April 1974.
Also in Proceedings of the IFIP Congress (invited paper), Stockholm, Sweden, August, 1974.

[38] M. Richards.
The Portability of the BCPL Compiler.
Software: Practice and Experience 1:135-146, 1971.

[39] A.V. Aho and J.D. Ullman.
The Theory of Parsing, Translation, and Compiling.
Prentice-Hall, 1972.

[40] W. McKeeman.
Peephole Optimization.
Communications of the ACM 8(7):443-444, July 1965.

[41] S.C. Johnson and D.M. Ritchie.
Portability of C Programs and the UNIX System.
Bell System Technical Journal 57(6):2021, July-August 1978.

[42] S.C. Johnson.
A Portable Compiler: Theory and Practice.
In *Fifth ACM Symposium on Principles of Programming Languages*, pages 97-104.

SIGPLAN-SIGACT, January, 1978.

[43] Knut Ripken.
Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinecode-erzeugung aus attributierten Programmgraphen.
PhD thesis, Technische Universitat Munchen, July, 1977.
In German; reviewed in English by Bert Speelpenning, *A Review of Ripken's Thesis*, unpublished.

[44] Lawrence Livermore Laboratories.
S-1 Processor Manual.
1978.

[45] M. Schaefer.
A Mathematical Theory of Global Program Optimization.
Prentice-Hall, 1973.

[46] B.R. Schatz.
Algorithms for Optimizing Transformations in a General Purpose Compiler: Propagation and Renaming.
Technical Report RC 6232, IBM Thomas J. Watson Research Center, October 1976.

[47] R. Sethi.
Complete Register Allocation Problems.
SIAM Journal of Computing 4(3):226-248, September 1975.

[48] M. Barbacci and D. Siewiorek.
Some Aspects of the Symbolic Manipulation of Computer Descriptions.
Technical Report, Carnegie-Mellon University, 1974.

[49] A. Snyder.
A Portable Compiler for the Language C.
Master's thesis, MIT, May, 1975.

[50] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors.
The Irvine Program Transformation Catalogue.
Technical Report, UC Irvine, Department of Information and Computer Science, 1976.

[51] T.G. Szymanski.
Assembling Code for Machines with Span-Dependent Instructions.
Communications of the ACM 21(4):300-308, April 1978.

[52] T.B. Steel.
A First Version of UNCOL.
In *Proceedings of the Western Joint Computer Conference*, pages 371-377. AFIPS, 1961.

[53] Digital Equipment Corporation.
VAX-11/780 Architecture Handbook.
1977.

[54] B.A. Wichmann.
Basic Statement Times for Algol 60.
Technical Report NAC 42, National Physical Laboratory, 1973.

[55] B.A. Wichmann.
How to Call Procedures, or Second Thoughts on Ackermann's Function.

Software: Practice and Experience 7:317-329, June-July 1977.

[56] J.D. Wick.
Automatic Generation of Assemblers.
PhD thesis, Yale University, December, 1975.

[57] T. Winograd.
Frame Representations and the Declarative/Procedural Controversy.
In D.G. Bobrow and A. Collins, editor, *Representation and Understanding*, pages 185-210. Academic Press, 1975.

[58] S.C. Johnson and M.E. Lesk.
Language Development Tools.
Bell System Technical Journal 57(6):2155-2175, July-August 1978.