THE DESIGN OF AN OPTIMIZING COMPILER

William A. Wulf
Richard K. Johnsson
Charles B. Weinstock
Steven O. Hobbs

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa.
December 1973

ABSTRACT

There are important classes of programs which must be highly efficient on a particular computer, independent of how fast that computer may be; systems programs are one such class. In order to be able to write these programs in a higher-level language and accrue the benefits associated with the use of such languages, we must have compilers which will produce highly efficient representations of these programs. This paper describes the design and implementation of a highly optimizing compiler for the Bliss language [Wul7 1].

A notational scheme is described in terms of which an overview of the compiler is presented. The logical phases of the compiler are then described in some detail.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

TABLE OF CONTENTS

I	**Introduction**	**1**
II	**A Descriptive Notation**	**3**
III	**An Overview of the Bliss/11 Compiler**	**5**
IV	**Compiler Specifics**	**8**
IV.1	LEXSYNFLO	8
IV.1.1	LEX	8
	What's in a Lexeme?	8
	Symbol Table Structure	9
IV.1.2	A Meta Comment on Switching	11
IV.1.3	SYN	12
	The Recognizer	12
	The Translator	13
IV.1.4	Syntax Error Reporting and Recovery	14
IV.1.5	FLO	15
	Theoretical Background	16
	FLO Implementation	26
IV.2	DELAY	33
IV.2.1	Determination of Desirable Feasible Optimizations	34
IV.2.2	Determination of Evaluation Order	36
IV.2.3	Target Paths and Unary Complement Operators	40
IV.2.4	Other Delaying of the Plus Operator	44
IV.3	Temporary Name Binding	50
IV.3.1	TLA	51
	Targeting	51
	Cost Determination	56
	Lifetime Determination	56
	The Run-Time Stack	58
	Label Assignment	60
IV.3.2	RANK	61
IV.3.3	PACK	61
IV.4	Code	63
IV.4.1	Generating Data Moves	65
IV.4.2	Generating the incr Loop Code	70
IV.5	FINAL	73
IV.5.1	The Final Data Structure	74

IV.5.2 The First Subphase 74
IV.5.3 The Second Subphase 79
IV.5.4 The Third Subphase 79

V Conclusion 80

A Primer on the PDP-11 83

B A Short Primer on Bliss 89

C A Complete Example 92

Bibliography 103

I. INTRODUCTION

There are important classes of programs which must be highly efficient on a particular computer, independent of how fast that computer may be -- systems programs are one such class. We must assume that such programs will continue to exist in the future. In order to be able to write these programs in a higher-level language and accrue the benefits associated with the use of such languages, we must have compilers which will produce highly efficient representations of these programs on a particular computer. This paper attempts to describe one such compiler.

Not all optimization techniques known are employed in the compiler to be discussed, nor is every aspect of optimizing compilers which is discussed original. Our purpose in writing this paper is neither to survey known optimization techniques nor to present only original research. Our purpose is, rather, to present various new and known techniques in the context of a specific, real compiler and to show how these techniques interact to produce high quality code. Another effect of this presentation, however, is to illuminate aspects of compiler optimization which have received little or no attention from researchers.

There is an unfortunate tendency for textbooks and journal articles to focus on topics which can be formalized. Thus much paper has been devoted to syntax analysis and to those relatively few optimization topics which lend themselves to this type of treatment, e.g., that evaluation order of expressions which minimizes register use. We do not oppose formalization -- it is essential, and, indeed, we will use a great deal in this paper. However, the appearance of many papers on a few topics together with a paucity of papers covering complete compilers, tends to breed more papers on the same subjects and leave other important problems untouched. We sincerely hope that one effect of this paper will be stimulate research on some of these problems.

Three specific assumptions are made in the remainder of this paper:

1. The compiler being described is for Bliss/11 [Wul72a] which is similar to Bliss/10 [Wul71]; a brief introduction to the language is included as Appendix B. The impact of the Bliss/11 language on the compiler strategies arises primarily from three properties of the language:

    a) Bliss does not contain an explicit goto statement [Wul72b]. The absence of a goto obviates the need for certain global flow analysis and permits a more highly recursive and regular structure in the compiler.

    b) Data is typeless. Certain aspects of more conventional compilers for dealing with type conversion and doing correlated case analysis during code generation will not be discussed since they do not apply.

    c) Any expression may be used as an accessor for a data structure. Therefore all expressions are treated uniformly, i.e., there are no special case optimizations

applied to data accessors (e.g., array indices). These optimizations are subsumed into the general strategies of the optimizer.

2. The problem domain for which Bliss/11 is intended, the development of production quality systems programs, warrants the expenditure of a relatively large amount of compile-time in order to achieve run-time efficiency. An explicit goal of the design is to produce better object code than that written by good systems programmers on moderately large programs (say longer than 1000 instructions).

3. Some optimizations involve a speed/size tradeoff. Since the object code is to run on a mini-computer these tradeoffs have generally been made in favor of the smaller (slower) alternative.

The structure of the paper is first to present a notational scheme in terms of which an overview of the Bliss/11 compiler will be presented, and then to discuss the various pieces of the compiler. The latter section describes the components in some detail, not all of which each reader will want or need to read in depth. Hence the latter portion of the paper is divided into subsections from which the reader is encouraged to pick and choose.

II. A DESCRIPTIVE NOTATION

Conceptually a compiler is a transducer which accepts a program

source → COMPILER → object

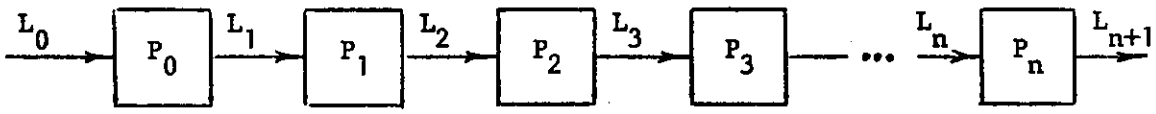
in its "source" language form and produce an equivalent "object" language form. For logical clarity, and sometimes from necessity, a compiler is broken into several "phases." A common diagram which illustrates these phases is

source → LEXICAL ANALYSIS → SYNTAX ANALYSIS → CODE PRODUCTION → object

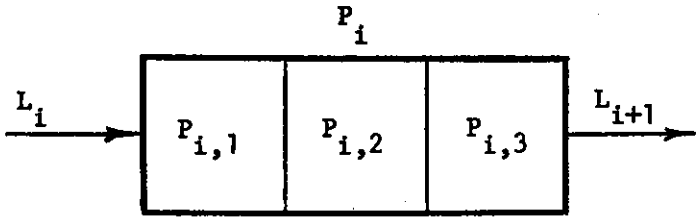
These phases may execute serially (as in overlaid "multiple pass" compilers) or act as coroutines. The diagram above need not distinguish between these modes of operation. In either case some "unit" of the program (anything from a single atom to the entire program) is successively operated upon by each phase. In this view the diagram above is intended only to show data dependencies between the phases--not control flow between them. (Control flow follows, but need not coincide with, the data dependencies.)

Diagrams of the simplicity of that shown above will not be adequate, however. While certain phases must be executed in a prescribed order, others may be performed in arbitrary orders; it may be possible to organize several logical phases into a single physical one. In order to describe these dependencies we have adopted the following notation:

1) A program shall be described as consisting of a linearly ordered set of "phases;" each phase is represented by a rectangular box with a single arrow entering it and another arrow leaving it. Each phase is to be thought of as a transducer accepting input language L_i and producing output language L_{i+1}.



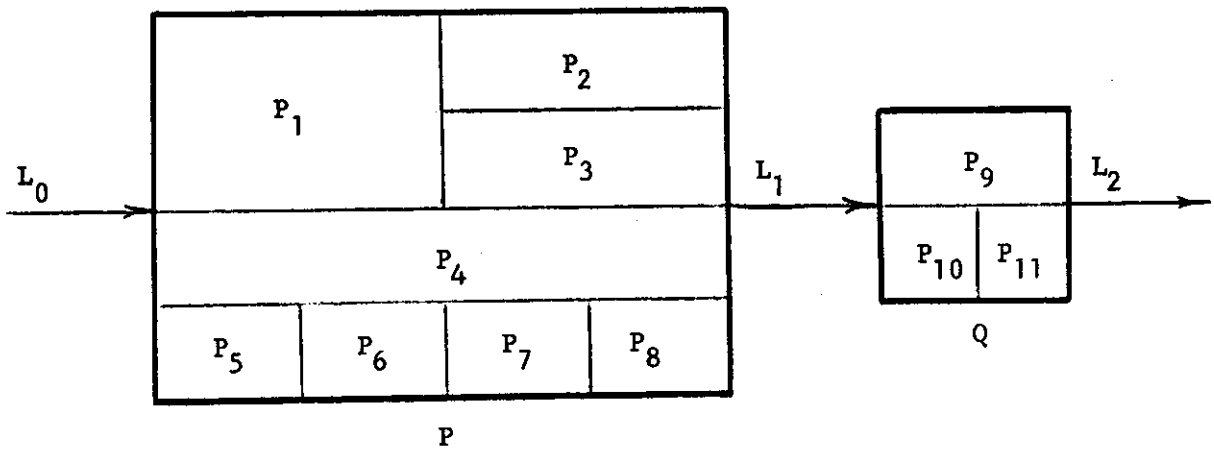
2) The box representing each phase may be divided into a number of vertical "columns." The box, say P_i, then represents a physical phase (operating over an entire unit in L_i) which is divided into logical phases, $P_{i1},\ldots,P_{in}$, which operate serially on the unit of L_i.



3) Each column may be divided horizontally into "rows" which are sub-phases of the column which may be executed in an arbitrary order.

4) Rows may be subdivided again into subcolumns, those into subrows, etc. as desired. In each case horizontal separation denotes left-to-right serial dependency, vertical separation (within a column) denotes the absence of such a dependency.

Consider the following example:



This diagram describes a system composed of two physical phases, P and Q. P accepts units of L_0 to produce units of L_1. Similarly Q accepts units of L_1 to produce units of L_2. Within the physical phase P are logical phases $P_1,\ldots,P_8$. From the diagram we see that the sets $\{P_1,P_2,P_3\}$, $\{P_4\}$, and $\{P_5,P_6,P_7,P_8\}$ are independent (may be executed in arbitrary order), but within these sets P_1 must precede both P_2 and P_3, and P_5, P_6,P_7, and P_8 are serially dependent.

III. AN OVERVIEW OF THE BLISS/11 COMPILER

Using the diagramatic notation developed in the last section, the structure of the Bliss/11 compiler is shown in Figure 1. In the case of Bliss/11, the program unit to which each physical phase is applied is the subroutine. Thus the source text for an entire subroutine is read and the phase LEXSYNFLO applied to it producing intermediate form L_1. In turn DELAY, TLA,..., and FINAL are applied to the intermediate representations $L_1,L_2,\ldots,L_6$ for the same subroutine producing, respectively, $L_2,L_3,\ldots,L_7$. The next subroutine is processed only after all phases have been applied to its predecessor. A consequence of choosing the subroutine as the unit to which successive phases are applied is that optimizations are applied to this unit, i.e., no optimizations are applied which involve detailed structural knowledge of more than one subroutine simultaneously.

The general attributes of the major phases are summarized below and then explained in more detail in the following sections.

LEXSYNFLO: This phase performs lexical analysis, declaration processing, syntax analysis, and flow analysis. The input is the source program unit in character string form. The output consists of: (1) a set of symbol table entries, (2) a tree representation of the parsed program unit, and (3) a set of lists (generally threads running through the tree) which define feasible global optimizations (constant expressions which may be moved out of loops and the like).

DELAY: Delay has three primary functions: (1) to determine the "general shape" of the object code to be generated, (2) to estimate the "cost" of each (linear) program segment, and (3) to determine the evaluation order for expressions. By the "general shape" of the object code, we mean those properties of the operators (e.g., commutativity) or properties of the target machine (e.g., indexing) which may be used to simplify the computation of a value. Decisions are also made at this point whether any (or all) of the "feasible" global optimizations are, in fact, desirable. Actual machine code is not generated; rather various flags and fields are set to guide optimal local code generation in a later phase. The cost metric is used to guide selection of evaluation order and in register allocation. The output of this phase is identical to that of LEXSYNFLO (i.e., symbol table, tree, etc.) except that certain information has been added to the tree to signal the subsequent phases of the compiler concerning the shape, cost, and execution order of the code to be generated.

TLA, RANK, PACK: The function of these phases is what is frequently called "register allocation" in other compilers; the difference being that not only registers, but memory locations, are allocated as well. The entities which are assigned to locations (registers or memory) include both

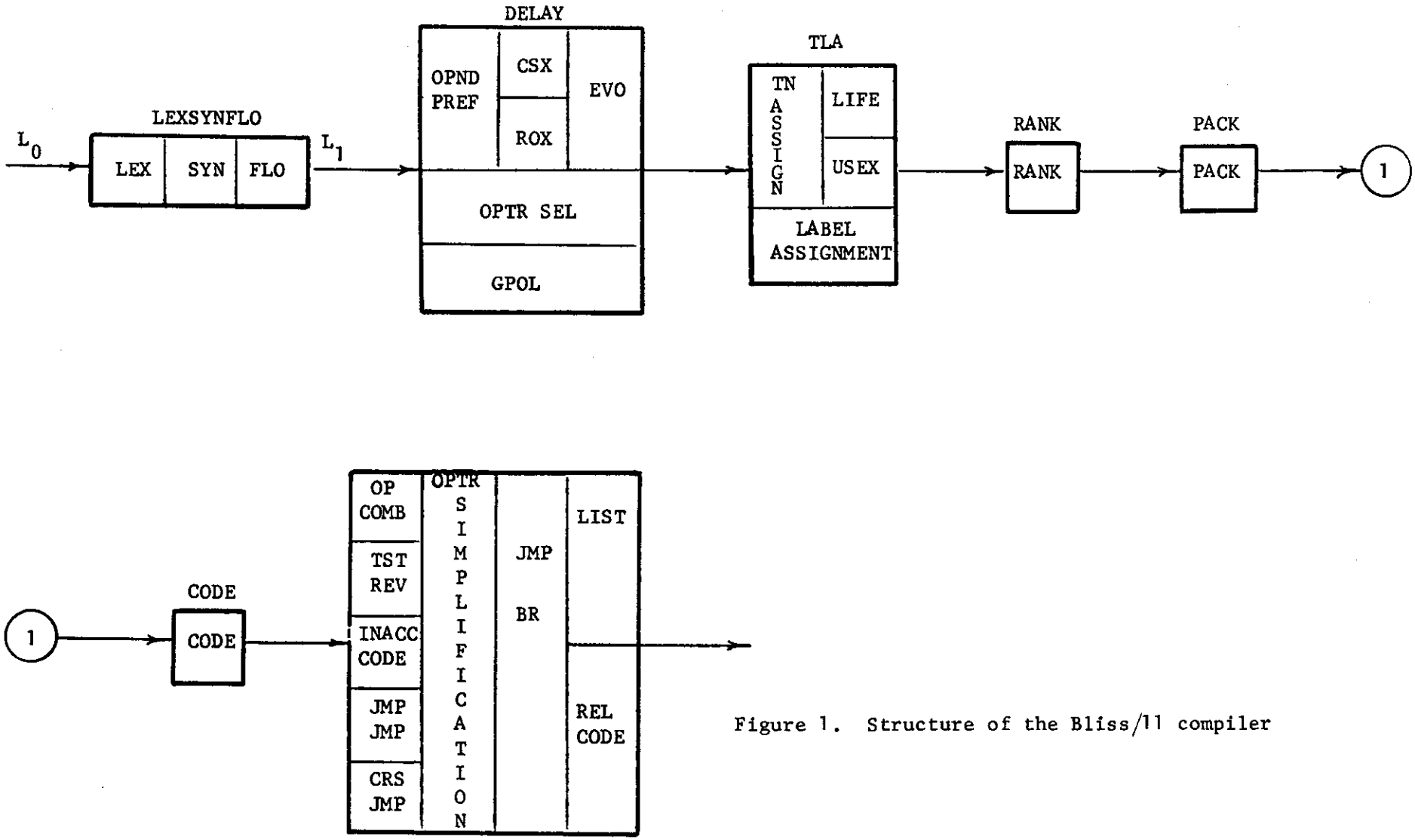


Figure 1. Structure of the Bliss/11 compiler

compiler-generated temporaries and user-defined "local" variables. The output of this phase includes that of DELAY plus the bindings.

CODE: The function of the CODE phase is to produce locally optimal code for each tree node, hence its output is a representation of the target machine language (the tree is discarded at this point). In some cases the locally optimal code is completely determined in DELAY; in these cases the action of CODE is trivial. In many cases, however, further analysis is required. For example, it is CODE's responsibility to determine the optimal sequence of shift and mask instructions to move an arbitrary subfield of one word to an arbitrary position of another.

FINAL: FINAL has two responsibilities. First, it contains a collection of relatively ad hoc "peephole" optimizations - for example, to change jump instructions which jump to another jump instruction into ones which jump directly to their ultimate destination. Second, it prepares the final listing and object code files.

It would be ideal if there were complete algorithms to guarantee the production of optimal programs. A few such algorithms are known [Set70, Ges72] and are used in the Bliss/11 compiler; more often, however, such algorithms do not exist and heuristics must be used. The sequence of phases in the compiler has been designed to provide as much global information as possible on which to base the heuristic decisions. Thus, for example, generation of code is postponed until after both the general shape of the code and binding of temporaries are known. Consequently special properties of the hardware may be exploited within this known context. Similarly, binding of temporaries is done only after the execution order has been firmly established and cost estimates obtained on the basis of the general shape of code to be generated; the first of these permits an accurate characterization of the "lifetime" of the temporary and the second permits realistic trade-offs to be made between allocating competing entities to registers. Again, evaluation order, general shape, and cost decisions are made in the context of "feasible" global optimizations.

IV. COMPILER SPECIFICS

The intent of this section is to examine in some detail the algorithms (and heuristics) used in each (sub) phase of the compiler. Since the topics of lexical and syntactic analysis have been extensively treated elsewhere, and since we make no special claims for our approach, these topics are treated rather cursorily; the emphasis is placed on the later phases of the compiler.

IV.1. LEXSYNFLO

As discussed earlier, the primary function of LEXSYNFLO is to generate a tree representation of the executable portion of a program unit together with auxiliary information such as symbol table entries and feasible optimizations. It consists of several subphases: (1) LEX which performs lexical analysis, (2) SYN which performs syntactic analysis and declaration processing, (3) KFOLD which performs compile-time arithmetic, logical, and relational operations (sometimes called constant folding), and (4) GFEAS which forms the actual trees and detects feasible global optimizations. Although implemented as a set of potentially recursive subroutines, these subphases actually bear a coroutine relation with each other with the central control residing in SYN. The subphases are discussed separately below.

IV.1.1 LEX

The function of the LEX module is to convert the input source text into atoms (or lexemes) for use by later phases of the compiler. LEX is essentially a finite state machine which reads source programs and produces lexemes. Lexemes are the smallest units of program used by the rest of the compiler, and consist of the internal representations of identifiers, reserved words, special characters, and literal values. LEX supports multiple input streams which may arise from text files named by the source program or from macro and structure invocations. We shall not be concerned with these aspects, however.

IV.1.1a What's in a Lexeme?

All lexemes contain a type field which specifies the kind of atom represented by the lexeme; in addition, each lexeme carries type dependent information:

Literal lexemes contain the actual 16 bit literal value. String literals are represented by lexemes containing a pointer to a linked list structure in which the string is stored.

Delimeter lexemes contain a class (declarator, open bracket, operator, etc.) and an index into a table of routines for performing syntax analysis. Operator lexemes also contain a

priority which controls association of operators and operands (e.g., this assures that A+B*C is treated as A+(B*C)).

Identifier lexemes contain a pointer to the symbol table entry for the identifier.

IV.1.1b Symbol Table Structure

The symbol table consists of three logically separate tables: the Name Table (NT), the Hash Table (HT), and the Symbol Table (ST). These three tables are linked together along several lists which allow easy recognition, insertion, and deletion. The relations between these tables, and the linkages between them, are illustrated in Figure 2 and explained further in the following paragraphs.

A hashing function converts the text representation of an identifier to an index into the hash table. Since more than one identifier may hash to the same index, the HT entry serves as the head of a singly linked list of name table entries. (All of the lists in the symbol table structure are effectively list implemented stacks, i.e., they obey a LIFO discipline.)

Each NT entry contains the actual text representation of the identifier and serves as the head of a singly linked list of all ST entries for that name (block structure makes it possible to have more than one ST entry for any given identifier). The HT entry also serves as the head of a singly linked list of ST entries for its hash value. Each ST entry contains a pointer to its NT entry.

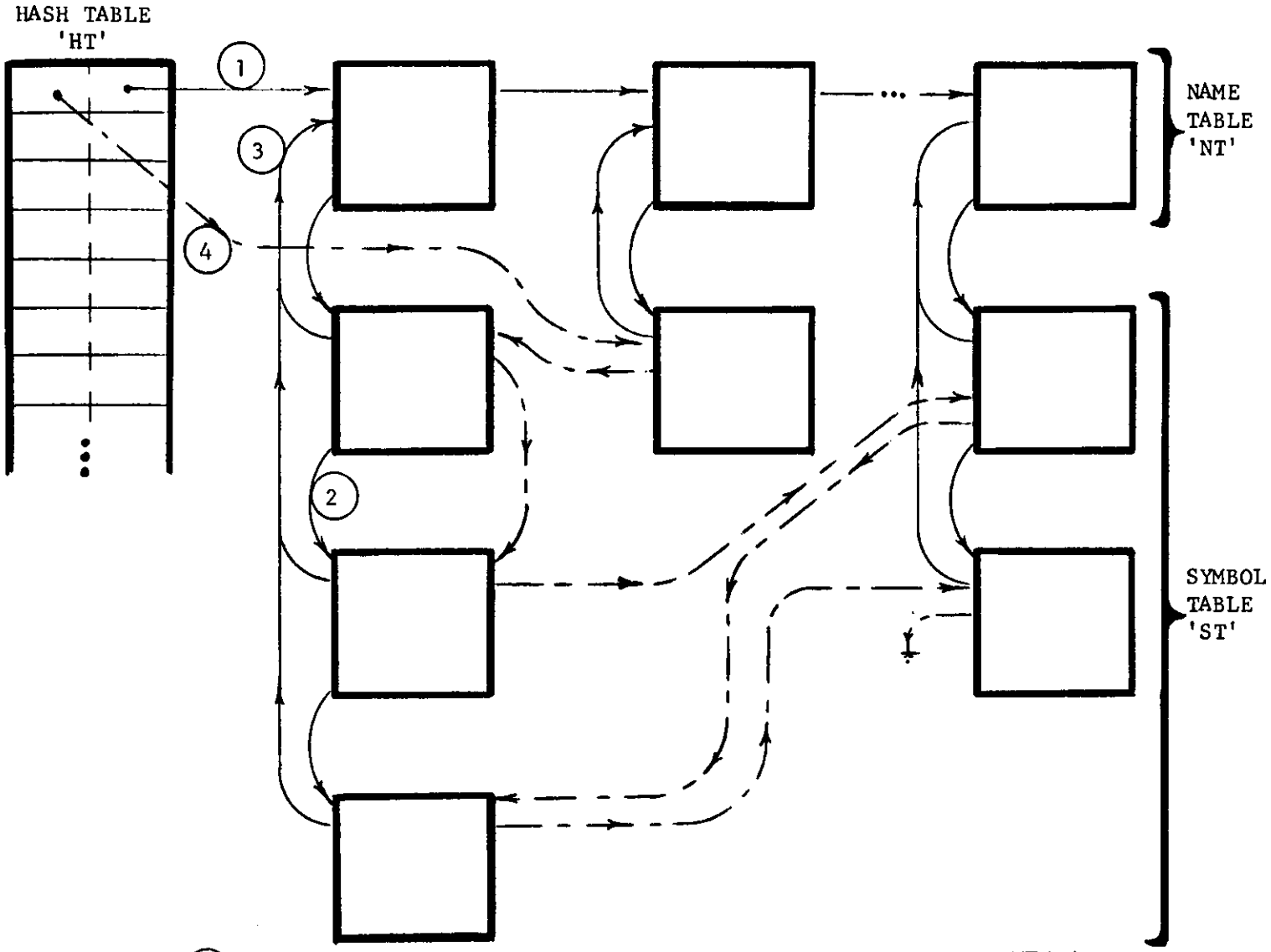
The algorithms for inserting an entry into the symbol table and for recognizing existing entries are quite simple, and should be fairly obvious from the structure.

To find the existing symbol table entry associated with a name, first form the hash function value associated withthe name, then scan the name table entries which are threaded together from this hash table entry until a character string match is found between the name and the string stored in the name table. The symbol table entry for the currently defined symbol with that name is pointed to by the NT entry.

To create a new symbol table entry for a name, e.g., when that name is (re)declared, search for an NT entry associated with the name as above; if no NT entry is found, create one and place it on the proper HT list. Create a symbol table (ST) entry, and "push" it onto the (possibly empty) stack of entries associated with the NT entry.

Deletion of entries from the symbol table is less intuitive than the insertion and recognition algorithm above, although easily implemented. At the end of a block, all ST entries created in that block must be removed from the symbol table. Since each ST entry contains the block level at which it was created, purging the symbol table is simply a matter of deleting all ST entries whose declaration level is equal to the current block level. These entries are easy to find because the most recent ST entries are at the tops of the ST lists pointed to by the hash table. The algorithm is:

Scan down the hash table and at each entry, delete the ST entries at the top of the



(1) Name table thread: runs from HT through all NT entries with the same HT index.

(2) Symbol stack thread: runs from NT through ST entries with same name; since this thread is in reverse declaration order, the entry pointed at by an NT entry is always the most recently declared instance of the named entity.

(3) Name table backpointer: every symbol table entry contains a backpointer to its name table entry.

(4) Symbol table list: this thread runs through all ST entries associated with the same HT index and is in reverse declaration order; used to remove ST entries at the end of a block.

Figure 2. Symbol Table structure.

ST list until the declaration level of the top item is less than the current block level or the list is empty.

In reality, symbols must only be removed from the ST at the end of a block in the sense that we want to prevent re-recognition. The entry itself continues to exist as long as there are references to it. To this end, symbols removed from the ST are linked onto a list of purged symbols. The symbols on this list are actually deleted after code has been generated for the current subroutine.

While it would be possible to process an entire program from source to lexemes in a separate pass independent of the rest of the compiler, this is neither necessary nor desirable. The syntax analyzer operates having knowledge only of a current "window" into the program. This window is maintained by LEX and consists of two lexemes -- a current symbol and a current delimeter. These are stored in the global variables SYM and DEL. (SYM may at times contain the "empty lexeme" since two delimeters may occur successively in a valid program.)

LEX interfaces with syntax analysis through the routine RUND (Read Until Next Delimeter). RUND is called to update the window by placing the next symbol lexeme (if any) in SYM and the next delimeter lexeme in DEL.

IV.1.2 A Meta Comment on Switching

In several of the phases to be described below (syntax, delay, tnbind, and code), a left-to-right depth-first tree traversal is done. In each of these traversals the actions to be performed depend upon the type of node. A uniform mechanism is used in each phase to accomplish the necessary switching between these actions.

Each type of tree node is uniquely associated with some delimiter in the source language (e.g., the node associated with a conditional expression is associated with the if delimiter). Each delimiter lexeme contains a unique, small integer; this integer is stored in the node to identify the node type. Each phase of the compiler contains a vector of the addresses of node-specific action routines. Switching to the appropriate routine is accomplished by performing a subroutine call indirectly through this vector.*

Since traversals and node-specific actions are handled similarly in each phase, an understanding of this mechanism will be assumed in each of the discussions below.

---

*For those familiar with Bliss, the following is an example of how this switching is actually done in the syntax phase:

```
bind  synplit = plit (soperator, scompound, sif, %...etc.  for all syntax routines%);

macro xctsyntax = (.synplit[.DEL])( ) $;
```

Each occurrence of the macro "xctsyntax" corresponds to a call on whichever syntax routine handles the construct typified by the delimiter lexeme in DEL.

IV.1.3 SYN

The compiler uses the top down method of syntax analysis known as recursive descent [Gri7 1]. The structure of the language lends itself nicely to this method. Recursive descent involves a separate, potentially recursive routine to parse each different construct in the language; each routine knows the format of its associated construct. Several properties of the Bliss/11 language that make it possible to use the technique effectively:

1. Every construct is uniquely identifiable by a key meta-variable (e.g., whenever an if lexeme is encountered, an if expression follows).*

2. It is possible to parse the language without backup.

3. The definition of the language does not contain the left recursion problem.

4. Bliss/11 is an expression language. This means that all switching between the syntax routines can be done in a central routine. When a particular construct expects to find a sub-expression , it merely calls this central routine to parse it. This routine makes use of the lexeme for the key meta-variable mentioned above to do the switching between routines. The switching technique itself is the one that was described in Section IV.1.2.

5. Bliss/11 is an operator language (between any two terminals (literals, symbols) at least one delimiter will appear). As a consequence, the syntax analyzer need only "see" at most two lexemes of the input stream at once. These are: the "current symbol" (SYM) and the "current delimiter" (DEL). These lexemes make up the window described in Section IV.1.1 above.

The task of the syntax analyzer is to transform the infix notation of the input stream into a n-ary tree. We treat only the key meta-variables as operators. Thus there is an if operator whose operands are a boolean part, a then part, and an else part.

IV.1.3a The Recognizer

Building a recognizer for Bliss/11 is a fairly easy task. A control routine, EXPRESSION, performs the switching; all that the construct associated routines need do is make repeated calls on EXPRESSION, checking for the occurrence of necessary delimiters in between. For example, the syntax of an if statement is: if e_1 then e_2 else e_3. The recognizer for if first calls EXPRESSION to parse e_1. It then checks for the presence of then, and if it is not there, it indicates an error. If it is present, it then calls EXPRESSION to parse e_2. Finally, it checks for an else and if it is found, it calls EXPRESSION to parse e_3.

Figure 3 illustrates several language constructs and recognizer routines for them. Note that they all have similar formats; they initialize error pointers, do the actions specific to the construct, and finally set more error pointers before exiting.

---

*Note that some meta-variables are used in two different ways (do, while, until). LEX can always tell which use is appropriate in context, and return the proper version of the lexeme.

```
routine sif=
    begin
    ! recognize if e0 then e1 else e2
    init;          ! initialize error pointers
    ruex;          ! parse e0*
    if .del neq "then"
        then return error( );
    ruex;          ! parse e1
    if .del eql "else"
        then ruex; ! parse e2 if else present
    end;
```

```
routine swhile=
    begin
    ! recognize while e0 do e1
    init;          ! initialize error pointers
    ruex;          ! parse e0
    if .del neq "do"
        then return error( );
    ruex;          ! parse e1
    end;
```

*ruex is an abbreviation for the sequence of instructions rund( ) (read until next delimiter) and expression( ).

Figure 3. Recognizer routines for if and while.

IV.1.3b The Translator

We are not interested in simply building a recognizer, but rather a translator. The translator, of course, differs from the recognizer in that it must do something with what it has parsed. In our case it builds a tree representation of the program. To do this, the recognizer routines are augmented with an auxilary stack (referred to simply as "the stack" for the rest of this section). The purpose of the stack is to hold the root(s) of subtrees while their ancestor(s) is (are) being parsed. Upon entry to a construct-associated routine, the current top of stack is marked so that this construct's intermediate results do not get confused with those results already developed at an outer level of the parse. At the end of a construct-associated routine, the contents of this stack (from the mark to the top) used to create a tree node.

Specifically, the translator for an if statement calls EXPRESSION to parse e_1. When EXPRESSION returns, a lexeme pointing to the sub-tree for e_1 is in SYM (this lexeme is

called a graph table lexeme). It is then pushed onto the stack. The translator then checks for the presence of the then (reporting an error if it is not found), calls EXPRESSION to parse e_2 and pushes the result onto the stack. If the else is present, EXPRESSION is asked to parse e_3 and again, the result is put on the stack. If else is absent, a literal zero is put on the stack instead. Finally, a routine called MAKGT is called to take the contents of the stack and make a tree node out of them. When it returns, the lexeme pointing to the node is in SYM.

Figure 4 illustrates the translator routines for the constructs that were described in Figure 3. Again note the similarity of their formats.

Because we are interested in doing global optimizations, the translator becomes a bit more complicated than the description above indicates. A module called FLOWAN which will be described below recognizes possible global optimizations. The flow analysis routines operate "inside out" and are invoked by the syntax routines. In particular, one flow analysis routine may be optionally called before a call on EXPRESSION and another is optionally called after EXPRESSION returns. Flow analysis routines are also potentially called before and after calls on MAKGT. The actions of these routines will be described in Section IV.1.5.

Figure 5 illustrates the translator routines with links to the flow analysis module. Note that these, too, are similar in format:

1. Initialize the error pointers and mark the current top of stack.

2. Do the construct specific actions.

3. Make a tree node from the connents of the stack between the mark and the new top of stack, and reset the stack to where it was on entry. Place a pointer to the new node in SYM.

IV.1.4 Syntax Error Reporting and Recovery

In order to help the programmer pinpoint errors, the lexical analysis routines note and record the line number and beginning character position of each lexeme. Individual syntax routines save these values for the opening bracket of the syntactic construct (e.g. begin, if, etc.), and the most recent "intermediate" bracket (e.g., the most recent ";" in a block). When an error is detected an error message is printed describing the error together with pointers to the most recently scanned symbol and the opening and intermediate brackets.

Once an error has been detected, a forward scan is made in an attempt to find a meaningful context in which syntax analysis can resume. Generally such a place is following a close bracket - notably ")", "end", or ";". However, if in the process of making this forward scan an open bracket is encountered, the syntax analyzer recurs to attempt the analysis of the interior of the bracketed construct. After the interior of the construct has been analyzed, the forward scan resumes.

```
routine sif=
    begin
    ! translate if e0 then e1 else e2
    init;                ! initialize error pointers and mark the syntax stack floor
    ruexpush;            ! parse e0 and put a lexeme for the sub-tree on the stack*
    if .del neq "then"
        then return error( );
    ruexpush;            ! parse e1 and put a lexeme for the subtree on the stack
    if .del eql "else"
        then ruexpush    ! parse e2 and put a lexeme for the subtree on the stack
        else push(zero); ! if no else, default to literal zero
    symmakgt("if");      ! make a tree of the if expression
    end;


routine swhile=
    begin
    ! translate while e0 do e1
    init;                    ! initialize error pointers and mark the stack floor
    ruexpush;                ! parse e0 and put it on the stack
    if .del neq "do"
        then return error( );
    ruexpush;                ! parse e1 and put it on the stack
    symmakgt("while-do");    ! make a tree of the while expression
    end;
```

*ruexpush is an abbreviation for the sequence of instructions rund( ); expression( ); push(.sym) where sym contains a lexeme for the expression scanned after the call on expression.

Figure 4. Translator routines for if and while.

IV.1.5 FLO

Before describing the actual strategies for global flow analysis in the Bliss/11 compiler a semi-formal treatment of the approach will be given. The formulation is due to Geschke [Ges72] and the actual implementation, except for representation issues, models the formulation quite closely.

```
outine sif=
   begin
   ! translate if e0 then e1 else e2
   init;                        ! initialize error pointers and mark the syntax stack floor
   ruexpush(fif0);              ! parse e0 and put a lexeme for the sub-tree on the stack*
   if .del neq "then"
       then return error( );
   ruexpush(fif1);              ! parse e1 and put a lexeme for the subtree on the stack
   if .del eql "else"
       then ruexpush(fif2)      ! parse e2 and put a lexeme for the subtree on the stack
       else push(zero);         ! if no else, default to literal zero
   fin("if",fif);               ! make a tree of the if expression**
   end;


routine swhile=
    begin
    ! translate while e0 do e1
    init;                       ! initialize error pointers and mark the stack floor
    ruexpush(fwh0);             ! parse e0 and put it on the stack
    if .del neq "do"
        then return error( );
    ruexpush(fwh1);             ! parse e1 and put it on the stack
    fin("while-do",fwh);        ! make a tree of the while expression
    end;
```

*ruexpush(fif0)is an abbreviation for the sequence of instructions <call flowan for preprocessing e_0>; rund( ); expression( ); push(.sym); <call flowan for postprocessing e_0>; where sym contains a lexeme for the expression scanned after the call on expression.

**fin("if",fif) is an abbreviation for the sequence <call flowan for preprocessing>; sym←makgt("if"); <call flowan for post processing>.

Figure 5. Translator routines for if and while with flow analysis linkages.

IV.1.5a Theoretical Background

We begin by considering the ordering relations inherent in a representation of a program P. There are several: the lexical order of the input text, the precedence-induced order of evaluation, both data-sensitive and data-insensitive order induced by control flow, and so forth. Two such orderings are important to the development.

The first is the order that results from considering a program as a mapping from its set

of input variables to its set of output variables. This ordering, called the essential ordering and symbolized by "$\prec$", is the ordering on evaluation of expressions that results from the application of the data flow and control flow semantics of a language L to the set of expressions E in a program P. The optimizations to be considered will regard the essential order as immutable.

The second ordering allows the selection of subsets of the total set of expressions in a program which at a given point are of interest to an optimization strategy. A representation of a program defines (at least partially) an evaluation order on its set of expressions. The ordering inherent in this particular representation may or may not correspond to the $\prec$-ordering.

The initial ordering on a program is symbolized by "$\lhd$". Intuitively, the relation $e \lhd e'$ expresses the notion that in a straightforward evaluation (i.e., that performed by a classical one-pass, non-optimizing compiler) of this representation of the program, the evaluation of e would necessarily have preceded the evaluation of e'. Alternatively, the $\prec$-ordering captures the notion of a linear flow of control passing through the related expressions. This ordering reflects the precedence relationships of the program; it also reflects the sequential nature of execution as in the case of a compound expression. It does not, on the other hand, necessarily reflect the subnode relationship between nodes. Below we give a precise definition of the initial order for Bliss; other languages would require a different set of definitions.

Definition

The initial ordering on the set of expressions E of a Bliss program is defined as follows:

Let e be a well-formed Bliss expression. Define $S(e) = \{e' \in E: e' \lhd e$ and e' is a subexpression of $e\} \cup \{e\}$. One of the following cases applies for e:

(1) e_1 <binop> e_2: $e_1 \lhd e, e_2 \lhd e$

(2) <unop> e_1: $e_1 \lhd e$

(3) begin e_1;...;e_n end: $e_i \lhd S(e_{i+1})$ $(1 \le i < n)$, $e_n \lhd e$

(4) case e_0 of set e_1; ... ; e_n tes: $e_0 \lhd e$, $e_0 \lhd S(e_i)$ $(1 \le i \le n)$

(5) if e_0 then e_1 else e_2: $e_0 \lhd S(e_1), e_0 \lhd S(e_2), e_0 \lhd e$

(6) select e_0 of nset e_1:e_2; ... ; e_{2n-1}:e_{2n} tesn:
$e_0 \lhd e$, $e_{2i-1} \lhd e_{2i}$ $(1 \le i \le n)$, $e_0 \lhd S(e_{2i-1})$ $(1 \le i \le n)$,
$e_{2i-1} \lhd S(e_{2i})$ $(1 \le i \le n)$

(7) while e_1 do e_2: $e_1 \lhd S(e_2)$

(8) do e_1 while e_2: $e_1 \lhd S(e_2)$

(9) incr 1 from e_1 to e_2 by e_3 do e_4:
$e_1 \lhd e_2 \lhd e_3 \lhd e$, $e_1 \lhd S(e_2)$, $e_2 \lhd S(e_3)$, $e_3 \lhd S(e_4)$

(10) $e_0(e_1, \ldots, e_n)$: $e_i \lhd S(e_{i+1})$ $(0 \le i < n)$, $e_n \lhd e$

(11) leave <label> with e_1: $e_1 \lhd e$.

Then e initially precedes e' (notation: $e \lhd e'$) if and only if in the $\lhd$-transitive closure of E there is a subset $\{e_1, \ldots, e_k\}$ such that $e \lhd e_1 \lhd \ldots \lhd e_k \lhd e'$.

In the case of simple non-control expressions the $\triangleleft$-ordering reflects the precedence-induced ordering of the binary operation. Most languages, however, contain control environments whose components are potentially $\triangleleft$-order independent. Consider the following compound expression:

$$(A \leftarrow .B;\ C \leftarrow .D;\ E \leftarrow .F),$$

where A, ..., F are distinct variables. Certainly the $\triangleleft$-order of execution of these three assignments can be altered. For example

$$(E \leftarrow .F;\ A \leftarrow .B;\ C \leftarrow .D)$$

produces the same effect. Even within the context of a simple expression such as

$$.A * .B + .C * .D$$

the commutivity of the "+" operator is reflected in the fact that the $\triangleleft$-order of the two multiplications is not defined. Nevertheless, the $\triangleleft$-order still reflects the requirement that both products be evaluated before the addition. Our observations to this point on the $\triangleleft$-order and $\prec$-order can be summarized by noting that in general the $\triangleleft$-order is weaker than the $\prec$-order, i.e., $e \prec e'$ implies that $e \triangleleft e'$ whereas the converse does not necessarily hold. In some instances the fact that e has been placed "before" e' (in the $\triangleleft$ sense) by the programmer is essential and sometimes it is not. The optimization strategies discussed below will alter the $\triangleleft$-ordering in a program. Since the validity of such an alteration is constrained by the $\prec$-ordering, a means must be provided for expressing the validity of transformations of a program. Given a pair of expressions e, e' where $e \triangleleft e'$, two aspects of the essential ordering can be identified that decide the validity of an optimizing transformation re-ordering e and e'.

Definition

Let $e_1, e_2 \in E$. e_1 is a necessary constituent of e_2 (notation: $e_1 < e_2$) if and only if (iff)
(1) e_1 is a subexpression of e_2, and
(2) evaluation of e_2 requires prior evaluation of e_1.

At first sight conditions (1) and (2) above may appear redundant. Indeed, if the language is Algol, they are redundant. However, in an expression language such as Bliss they are not (e.g. compound expressions).

Definition

Let $e_1, e_2 \in E$. The expression e_1 is an essential predecessor of e_2 (notation: $e_1 \ll e_2$) iff
(1) $e_1 \triangleleft e_2$
(2) there exists a context such that the evaluation of the sequences $\{e_1,e_2\}$ and $\{e_2\}$ ($\{e_2,e_1\}$ and $\{e_1\}$) may produce distinct values for e_2 (e_1).

It is important to understand the relationship between the orderings $<$ and $\ll$ and the

$\prec$-ordering. If these orderings are considered in their standard mathematical representations as subsets of ExE, then their relationship can be stated as; $\{\prec\} \subset \{<\} \cup \{\ll\}$. Hence it follows that if $e < e'$ or $e \ll e'$ then $e \prec e'$; or equivalently if e does not precede e' in the $\prec$-ordering then $e \not< e'$ and $e \not\ll e'$.

Definition

Let $e_1, e_2 \in E$. e_1 is independent of e_2 (notation: $e_1 \diamond e_2$) iff $e_1 \not< e_2$, $e_2 \not< e_1$, $e_1 \not\ll e_2$, $e_2 \not\ll e_1$.

Independent expressions are those whose $\prec$-ordering is not determined by the semantics of the language. The usefulness of these primitive relations will become apparent during the discussion of the optimization strategies involving code motions.

Next we introduce an equivalence relation called congruence on ExE which is an extension of the equality relation on E. Intuitively, two expressions are congruent if there exists a one-to-one correspondence between them that preserves the tree structure and in which the corresponding nodes are identical operators or terminals. More precisely, the elements of E, considered as nodes in the tree representation, can be decomposed into non-terminal (N) and terminal nodes (T). Moreover T itself can be decomposed into names and literals. Hence congruence is defined as follows:

Definition

Let $e, e' \in E$. e is congruent to e' (notation: $e \cong e'$) iff either

(1) $e, e' \in N$,
e[operator] = e'[operator],
e[# of operands] = e'[# of operands] = n, and
$e[operand_i] \cong e'[operand_i]$ $(1 \le i \le n)$;

(2) $e, e' \in T$,
e and e' are equal literals or identical names*

The notion of common subexpression (cse), which specifies a subset of the collection of all redundant computations, is defined in terms of the primitives developed so far.**

Definition

e and e' are common subexpressions (cses) (notation: e = e') iff

*The statement that e and e' are identical names is stronger than character string equality. Here we mean that they in fact refer to the unique variables accessible by that identifier within the present environment.

**It should be noted that the concept of common subexpressions as defined here is a subset of the concept as it has been used in some earlier papers. We shall refer to the more global concept as "redundant expressions," which is consistent with yet other papers. The redundant computations not captured by our definition are covered by other notions introduced later.

(1) $e \cong e'$,
(2) $e \lhd e'$ or $e' \lhd e$, and
(3) assuming $e \lhd e'$, $\forall\, e''$ such that $e \lhd e'' \lhd e'$, $e \not\lhd e''$.

The intuition to be conveyed by this definition of a cse is that if e = e', then (1) the values returned from the evaluation of e and e' are always identical and (2) the control flow of P is such that whenever e' is evaluated then e has been evaluated prior to it (or vice versa). The components of the definition mirror this intuition by saying that (1) e and e' are congruent, (2) the evaluation of e initially precedes e' (or vice versa) by definition of the $\lhd$-ordering, and (3) all the expressions that intervene between e and e' have the property that they do not produce side-effects that affect the value of e (equivalently: e') nor does e produce side effects on them. The latter condition says intuitively that the evaluation of e' is unnecessary since its value is available from the evaluation of e.

The literature on object code optimization in the presence of control flow identifies a collection of optimization strategies called code motions. The collection of code motion optimizations about to be described are all predicated on a recursive inside-out approach for their detection. For example, in detecting code motions relative to an if-then-else control environment, the detection proceeds by first invoking the optimization on the "then" and "else" expressions. The optimization on each of these expressions will (1) detect the feasible optimizations within its own local environment, and (2) return information to be used in detecting optimizations relative to the if-then-else environment. This overall approach requires that a means be provided for stating precisely what information about the sub-components of a control expression is required in order to detect optimizations for the control expression itself. The notion of a linear block, β is introduced for this purpose. Roughly speaking, β corresponds to those subexpressions of e through which a linear (i.e., $\lhd$-order) flow of control passes.

Definition

Let $e \in E$ and $E' = \{e' \in E: e' \text{ is a subexpression of } e\}$. The linear block β relative to e (notation: $\beta|e$) is the set $\beta|e = \{e' \in E': e' \lhd e\}$.

Since in the context of the use of $\beta|e$ the expression e is quite often obvious, "|e" is simply omitted in most cases; otherwise, by convention, the linear block relative to e_i will be denoted by β_i. In flow diagrams, linear blocks are depicted as unbroken vertical lines (flow passing from top to bottom):

The following definition introduces three sets which make the succeeding definition less cumbersome.

Definition

Let $e \in \beta$, β a linear block.
pro-dominator(β,e) = $\{e' \in \beta: e' \lhd e, e' \ll e\}$,
epi-dominator(β,e) = $\{e' \in \beta: e \lhd e', e \ll e'\}$,
post-dominator(β,e) = $\{e' \in \beta: e \lhd e', e' \not\diamond e\}$.

The pro-dominator set contains those elements of β which initially precede e such that they produce a side effect on e or e produces a side-effect on them. The epi-dominator set differs from the pro-dominator only in that its elements initially follow e. Intuitively the pro-dominator (epi-dominator) contains those elements of β which prevent the movement of e backward (forward) to the head (tail) of β because they produce a side-effect on e or vice versa. The post-dominator set consists of those elements of β which initially follow e and are not independent of e. Hence the post-dominator consists of those elements which prevent the movement of e forward either because of a side-effects relationship or because their evaluation requires the evaluation of e. It follows from the definitions of "$\ll$" and "$\diamond$" that: epi-dominator(β,e) $\subseteq$ post-dominator(β,e).

Definition

Let β be a linear block.
prolog(β) = $\{e \in \beta:$ pro-dominator(β,e) = $\emptyset\}$,
epilog(β) = $\{e \in \beta:$ epi-dominator(β,e) = $\emptyset\}$,
postlog(β) = $\{e \in \beta:$ post-dominator(β,e) = $\emptyset\}$.

Note that it follows immediately that postlog(β) $\subseteq$ epilog(β).

Code Motion Optimizations

We can now define various code motion optimizations. Consider, for example, a branching control construct of the form shown in Figure 6 where $\in$ functions as a selector among the n branches. This form of control represents both if-then-else and case types of control environments.

The first feasible optimization exploits code motions out of the linear blocks $\beta_1, \ldots, \beta_n$ to produce a flow diagram of the form shown in Figure 7. The linear blocks α and ω contain those expressions factored forward and backward from all of the branches, β_i.

A primary goal of the development is to provide a means of concisely describing the set of feasible members of sets such as α and ω. To that end an operator on the power set of E is introduced.

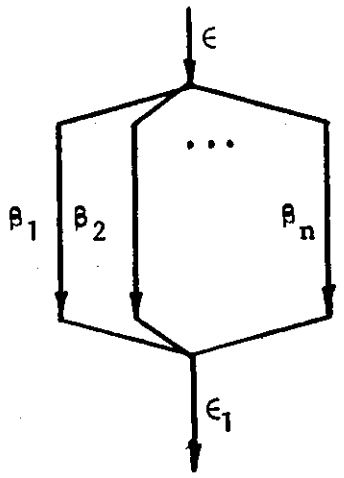


Figure 6. An n-way branching structure.

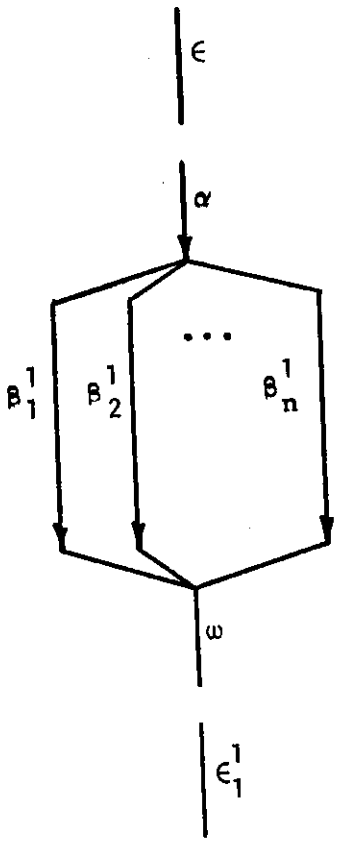


Figure 7. α and ω code motion.

Definition

Let $E_1, \ldots, E_n$ be subsets of E. The formal intersection of the sets E_i is defined as

$$\Lambda E_i = \{e \in E: \forall\, i,\ 1 \le i \le n,\ \exists\, e_i \in E_i \text{ such that } e \cong e_i\}.$$

While formal intersection is different from ordinary set intersection the analogy should be obvious; it differs from set intersection in that the equivalence relation of equality of elements is replaced by that of congruence.

The notion of formal intersection provides us with a powerful tool to concisely define the sets α and ω.

Given a forked control environment with branches $e_1, \ldots, e_n$, the domain of elements (α) available for pre-evaluation is described by: $\alpha \subseteq \Lambda\ \text{prolog}(\beta_i)$. The domain of elements (ω) available for post-evaluation is described by: $\omega \subseteq \Lambda\ \text{postlog}(\beta_i)$.

Moreover,

Given a forked control environment with selector expression ϵ and branches $e_i = (\epsilon; e_i)$ and $\beta_i' = \beta_i' | e_i'$, $1 \le i \le n$. Then the set of expressions whose evaluation at the merge point would be redundant is the set: $\Lambda\ \text{epilog}(\beta_i')$.

The looping constructs we shall consider consist of a body β_1 and a predicate β_2 to be evaluated on each iteration. We shall consider two types, a "while-do" form which has its test at the top of the loop and a "do-while" with its test at the bottom of the loop. These constructs are illustrated in Figures 8a and 8b respectively. Other forms of loops such as counting types can be modeled by these forms.

The first optimization is the pre-evaluation of the "loop invariant expressions," i.e., those whose values do not change on any iteration of the loop.

Given a loop control environment, the set of loop invariant expressions is described by: $\chi = \text{prolog}(\beta) \cap \text{epilog}(\beta)$, where β is the linear block relative to the compound expression $(\beta_1; \beta_2)$ in the "do-while" case and $(\beta_2; \beta_1)$ in the "while-do" case.

The description has intuitive appeal since it simply states that any expression whose evaluation is not affected by occurring either before or after the loop is not changed by execution of the loop.

Cyclic Re-evaluations

The cyclic nature of loop control gives rise to a particular class of "redundant" computations. Consider the example shown in Figure 9. Clearly the expression .A*.B is not invariant throughout the loop. However if the expression .A*.B were pre-evaluated at entry to the loop and stored in a temporary T and if after each computation of A or B the expression .A*.B were again evaluated in T, there would be no need to re-evaluate .A*.B at the top of the loop on each iteration. The restructured computation is shown in Figure 10.

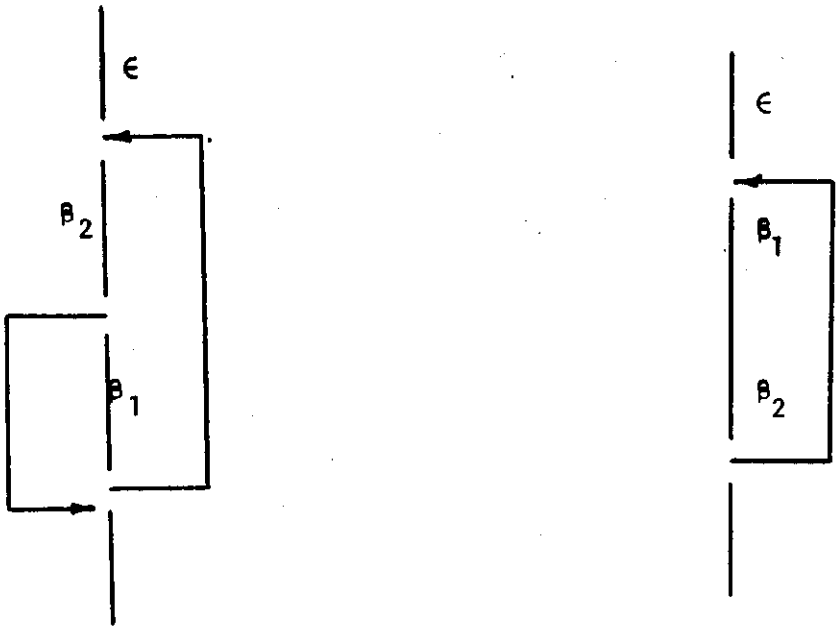


Figure 8. "while-do" and "do-while" constructs.

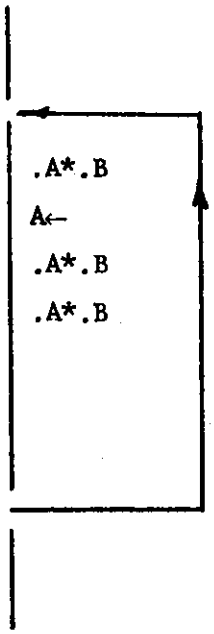


Figure 9. Sample looping construct.

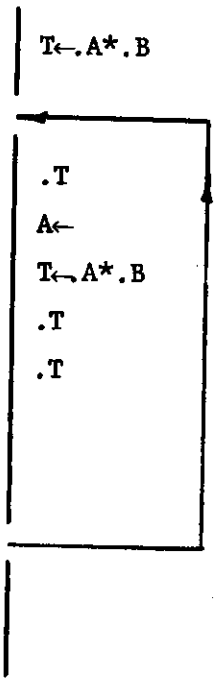


Figure 10. ρ code motion in a loop.

Given a loop control environment where β is the linear block relative to the expression $(\beta_1; \beta_2)$ ("do-while") or $(\beta_2; \beta_1)$ ("while-do"), the set of expressions whose evaluations at the head of β are redundant to evaluations at the tail of β are described by the set: ρ = prolog(β) $\wedge$ epilog(β).

Finally, let us point out how loops participate in the exposure of the set of redundant expressions to their surrounding environment.

In the case of a "while-do" construct (Figure 8a) the set of expressions whose values are available on exit from the loop is the set epilog(ϵ; β_2).

For the case of a "do-while" construct (Figure 8b) the set of available expressions on exit is epilog (ϵ; β_1; β_2).

IV.1.5b FLO Implementation

In the preceding theoretical background the importance of congruent expressions should be obvious. Therefore the structure of FLO has been organized to make the recognition of congruent expressions extremely efficient. Because it is efficient to do so, cses (in the sense defined above) are also detected at the same time. Figures 11 and 12 illustrate the data structures used for this recognition - principally a set of threads running through the tree representation of the program.

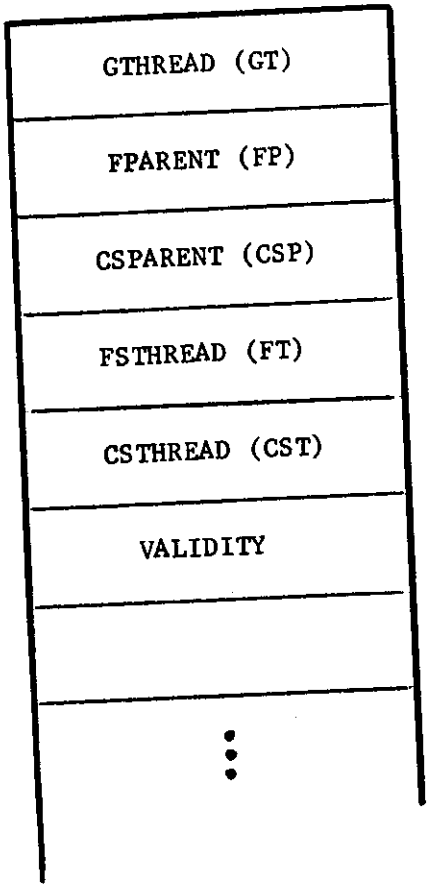


Figure 11. Data fields for FLO.

Suppose the set of expressions in a particular program is denoted by E and that a subset of these, C, are congruent. Each element of E is a node in the tree representation of the program. One element of the set C is designated to act as the representative of the class and is called the "formal parent" of the class. Every element of the class contains a reference to the formal parent (FPARENT) field in Figure 11). Furthermore, the elements of C may be partitioned into one or more sets of cses, $C_0,\ldots,C_n$, each of which is a unique collection of cses. An element of each of the sets C_i is designated as the representative of the set and called its "cse parent." Elements of C_i all contain a reference to this representative (CSPARENT field in Figure 11). Elements of C_i are threaded together from the representative (through the CSTHREAD field) and CS-parents are threaded together from the formal parent (through the FSTHREAD field). Formal parents for expressions involving the same root operator are threaded together (through GTHREAD,Figure 11); this thread is rooted in an auxiliary vector called GTHASH.

The following discussion is divided into two pieces which are actually merged in the implemenation. The first piece deals with the recognition of cses "on-the-fly;" that is, as the program is parsed and the tree representation program is built. The second piece of the discussion deals with recognition of additional redundant expression evaluations which cannot be safely recognized "on-the-fly" and with detection of feasible code motion optimizations.

On-The-Fly Recognition

Congruent expressions and (most) cses are recognized as the program is parsed and the tree representation is built. Two expressions are congruent iff: (1) they involve the same operator, (2) they involve the same number of operands, and (3) all of their corresponding operands are congruent. Therefore, to determine whether an expression which is just being built is congruent to an existing one the compiler searches the GTHREAD corresponding to the operator of the new expression. Each node on this list is the representative of a distinct equivalence class under congruence. Hence, for each node on the list the compiler tests to see whether the number of operands is identical to that of the new expression and checks whether the corresponding operands are congruent. Note that $e \cong e'$ if and only if e[FPARENT] = e'[FPARENT] so the latter check is simple. If the new expression is not congruent to existing ones, the node is added to the GTHREAD thread as the representative (formal parent) of a new equivalence class; otherwise, a check is made to determine whether the new expression belongs to one of the cse classes.

Expressions e and e' are cses, e = e', iff: (1) $e \cong e'$, (2) $e \lhd e'$, and (3) $\forall e'' \ni e \lhd e'' \lhd e', e \not< e''$. The first condition, congruence, is detected as described above. The second and third condition are encoded in the VALIDITY field (Figure 11) in a manner which will be described below. The general strategy is, having detected that a new expression is congruent to one or more extant ones, to scan from the formal parent through the cse parents (via the FSTHREAD) to determine whether there exists one of these whose value is valid at that point. If so, the new expression is added to the appropriate set; otherwise, the expression becomes the cse parent of a new set.

The validity field consists of six subfields as shown in Figure 13:

PD "purged," one bit, set when node is permanently invalid for cse recognition.

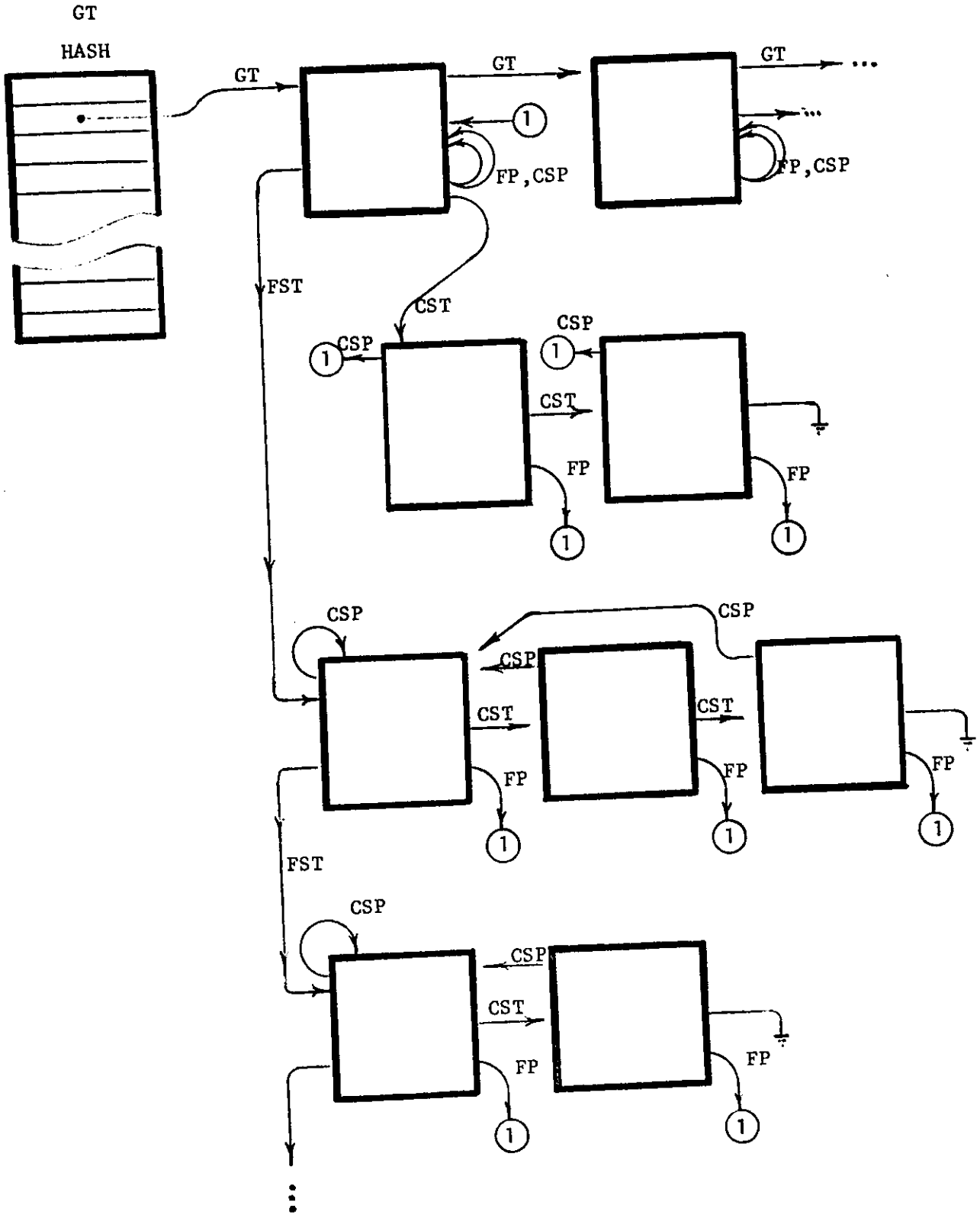


Figure 12. FLO links in the Graph Table.

RM	"real mark," one bit, set when node is invalid in the current linear block.
JM	"join mark," one bit, set when real-mark must be set at join of a forked construct.
MM	"must mark," one bit, set when effect of a side-effect must be noted later.
CRLEVEL	"creation level," encoding of the linear block, β, in which the expression was created.
MKLEVEL	"mark level," encoding of the outermost linear block, β, in which the expression was marked, i.e., invalidated.

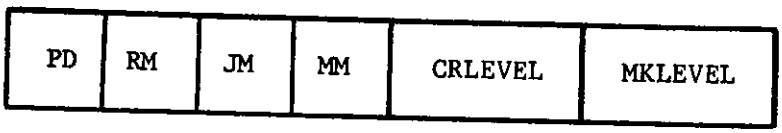


Figure 13. Subfields of VALIDITY.

These fields are maintained, together with global variables CEILING and FLOOR, such that the value represented by a node is valid if:

$$\sim PD \wedge \sim RM \wedge (CRLEVEL \leq CEILING) \wedge (CRLEVEL \geq FLOOR)$$

The encoding of the validity field is possible because of the fully nested (i.e., no goto) nature of Bliss and the "inside-out" invocation of the recognition mechanism. This mechanism will be explained in the context of the processing of two specific syntactic constructs; the if-then-else and do-while. Before describing the handling of these constructs, however, several aspects of the implementation must be described.

(1) A global variable, LEVEL, is maintained such that its value is incremented whenever a new linear block is encountered and reset when the end of that linear block is detected. Therefore a necessary, but not sufficient,* condition for $e \lhd e'$ is $e[CRLEVEL] \leq e'[CRLEVEL]$. (The variables CEILING and FLOOR assume values of LEVEL and define a "window," $FLOOR \leq L \leq CEILING$, in which "on-the-fly" recognition of cse's is possible. The interpretation of these variables will be discussed in context below.)

(2) Whenever a side-effects producing operation, e.g., assignment or subroutine invocation, is encountered the "must mark" bit of nodes whose value depend upon the affected variables is set. As may be seen from the definition of validity given above, setting this bit does not invalidate the value of the node. The semantics of Bliss specify that the effect of side-effects need to be accounted for at prescribed places, notably at the semicolon. Hence, at the appropriate points a primitive,

*The condition is not sufficient since linear blocks at the same nesting level need not satisfy the initial ordering; for example, the then and else portions of the conditional are at the same nesting level, but neither initially precedes the other.

"markmmnodes," is invoked to convert MM-bits to RM-bits (at this time the mklevel field is also set).

(3) The construction of the recursive-descent parser is such that any invocation of the driver, EXPRESSION, can be (optionally) preceded and/or followed by invocation of a context-specific flow primitive. The particular case of the conditional expression, for example, involves invocation of six flow primitives, f_0-f_5, as shown below.

```
if   e0   then   e1   else   e2
   ↑  ↑        ↑  ↑        ↑  ↑
   f0 f1       f2 f3       f4 f5
```

Ignoring the recognition of the α and ω sets described earlier, the flow primitives for the conditional expression are:

```
f0:
f1: markmmnodes; increment (ceiling);
f2:
f3: markmmnodes; refresh;
f4:
f5: markmmnodes; refresh; decrement (ceiling); join;
```

where

```
markmmnodes: ∀e(e[rm] ← e[rm] ∨ e[mm]; e[mm] ← ∅);
increment(x): (level ← level + 1; x ← level);
refresh:     ∀e(if e[crlevel]≥ ceiling
                 then e[pd] ← 1
                 else if e[mklevel]≥ ceiling
                      then(e[jm] ← e[jm] ∨ e[mm] ∨ e[rm]; e[rm] ← e[mm] ← ∅));
decrement(x): (level ← level - 1; x ← level);
join:        ∀e › e[mklevel] ≥ level (e[rm] ← e[rm] ∨ e[jm]);
```

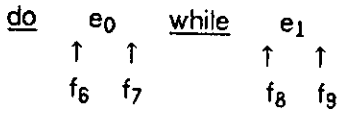
The operation of this collection of routines is probably best explained by tracing its execution for the expression "if e_0 then e_1 else e_2." The boolean expression, e_0, is an element of the linear block determined by the context in which the entire expression occurs. If e_0 contains uses of cses created earlier they will be recognized; new cses may be created in e_0; if e_0 produces side-effects, the MM bits of affected nodes will be set. After e_0 has been parsed, however, a flow primitive, f_1, is invoked. One of the prescribed places in Bliss at which side-effects are to be accounted for is following a boolean expression, hence "markmmnodes" is invoked to set RM bits. At this point LEVEL and CEILING are incremented to reflect that a new linear block will be entered to evaluate e_1.

During the parse of e_1 cses will be recognized, potentially new cse parents generated, MM and RM bits set, etc. After e_1 has been completely parsed the flow primitive f_3 is invoked and performs "markmmnodes" and "refresh." The purpose of refresh is: (1) to "purge" all nodes created "in the attic," i.e., with a creation level ≥ CEILING, and (2) to set the "join mark," JM, bit of nodes invalidated in the attic (and simultaneously reset their RM and MM bits). During the left-to-right parse of the program we must assume that values

created on one branch of the conditional will not be available after the forks rejoin; hence the purge of nodes created in the attic (we'll come back and pick up values created on both branches later). On the other hand, nodes created prior to the branch and invalidated on one branch will be available for the other; hence nodes marked along the branch are reset to "valid" (RM reset to zero). The JM bit of these nodes is set, however, to serve as a reminder that the value of the expression will not be available after the forks rejoin.

The else-branch, e_2, is treated similarly. Notice, however, that neither CEILING nor LEVEL is altered since both branches, even though they are distinct linear blocks, are at the same nesting level. Also, after e_2 is parsed, LEVEL and CEILING are reset and "join" is invoked to set the RM bits correctly for nodes which were invalidated along either (or both) branches.

The do-while expression invokes flow primitives f_6-f_9 as shown below:



where

f_6: increment (floor);
f_7: markmmnodes;
f_8:
f_9: markmmnodes; decrement (floor);

Since, in the case of a do-while, both e_0 and e_1 are guaranteed to be executed at least once,there is major problem with "on-the-fly" recognition of cses. We wish to prevent recognition of expressions whose value may be valid on the first execution of the loop, but, because of side-effects later in the loop, will be invalid on the second and subsequent iterations. The manipulation of FLOOR, together with the definition of "valid" expressions given earlier, prevent recognition of such expressions. Values available prior to the loop and not altered by the loop are detected at a later stage in a manner which will be described below.

Other Redundant Expressions and Code Motion Optimizations

The detection of the remaining redundant expressions and feasible code motion optimizations hinges primarily upon formation of the prolog, epilog, and postlog sets described earlier. To facilitate the formation of these sets a new global variable, the "atomic-block-count" or ABCOUNT, is introduced. This variable monotonically increases whenever side effects are invoked; an auxiliary stack is maintained such that the top element of this stack contains the value of ABCOUNT when the current linear block was entered. Also, each symbol table entry for a variable in a program is augmented with two sets -- CHANGELIST and USELIST. Whenever a value is changed as well, an entry consisting of the atomic block count and a pointer to the node at which the action occurred is made in the appropriate set. The sets themselves are represented by linked lists; the references are ordered in a manner which minimizes the search time for the intersection operators.

The prolog set consists of those expressions which have no essential predecessors in the block. This set is formed "on-the-fly" as the program is parsed. A newly formed node, e, belongs to the prolog of its linear block iff no previous operation in the linear block has had a side effect on the operands of e. There are three interesting cases:

(1) $e = e_1$<binop>e_2: $e \in prolog(\beta)$ iff $e_1 \in prolog(\beta) \wedge e_2 \in prolog(\beta)$

(2) $e = .e_1$: $e \in prolog(\beta)$ iff

(a) e_1 is a variable and e[changed] < ABCOUNT $\forall x \in$ CHANGELIST(e)(X[ABCOUNT] < ABCOUNT(β)), where ABCOUNT(β) is the top of the auxiliary stack of ABCOUNT values described above,

(b) or e_1 is an expression and $e_1 \in prolog(\beta)$

(3) e = <constant>: by definition, all constants are in prolog(β)

Although these definitions are recursive, the information necessary to make the first test, (1), can be (and is) encoded in a single bit in each node.

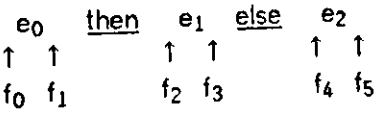
The epilog set consists of those expressions in a linear block such that no expression following them in the linear block has a side-effect on them; that is, they are not the essential predecessor of any expression in the block. It follows from the earlier discussion of "on-the-fly" recognition of cses that

$$e \in epilog(\beta) \text{ iff } e \in \beta \wedge PD \wedge RM$$

Thus, at the end of a linear block, a simple linear scan of the expressions is adequate to determine membership in epilog(β).

The postlog set contains those expressions in a linear block which are neither essential predecessors nor necessary constituents of the expressions which initially follow them in the linear block. Since postlog(β) $\subseteq$ epilog(β), it is only necessary to determine those nodes which are not necessary constituents of following nodes; in the case of Bliss the only candidates are non-value-producing components of a compound expression (e.g., e_1, e_2, ..., e_{n-1} in (e_1; ...; e_{n-1}; e_n)) and hence a simple tree walk from the root of the linear block will detect the elements of postlog(β).

Returning to the example of the conditional expression



and concentrating only on those operations necessary to detect feasible optimizations not exposed in the previous section, we have

f_0:

f_1:

f_2: pushflow;
f_3: $pr_1 \leftarrow$ prolog; $ep_1 \leftarrow$ epilog; $po_1 \leftarrow$ postlog; popflow;
f_4: pushflow;
f_5: $pr_2 \leftarrow$ prolog; $ep_2 \leftarrow$ epilog; $po_2 \leftarrow$ postlog; popflow;
$\alpha \leftarrow pr_1 \wedge pr_2$; $\omega \leftarrow po_1 \wedge po_2$; $\pi \leftarrow ep_1 \wedge ep_2$;

where: (1) "pushflow" increments ABCOUNT, pushes the stack of these values, and sets up a new prolog set: (2) "popflow" pops the stack of ABCOUNT values, creates the "prolog," "epilog," and "postlog" for the linear block and return a reference to them; and "$\wedge$" is the formal intersect operator. Note, that the set "π" is the set of expressions created on both branches whose values are therefore available after the merge point; expressions whose values are available after the merge point, because they were created prior to the fork point, are handled by the mechanism described previously.

Although we have avoided describing all the details of the implementation, we hope that the description is adequate to suggest the general approach. Before leaving the subject of FLO it should be re-emphasized that the only action taken is to detect feasible optimizations. The full tree representation of the program remains. Threads have been added to link together cses and separate lists have been built to identify feasible code motions, but no transformations have been performed on the tree itself. Hence later phases, especially DELAY, may choose to exploit these feasible optimizations or not depending on context.

IV.2. DELAY

As mentioned previously, DELAY has three primary functions: (1) to determine the "general shape" of the ultimate object code to be produced, (2) to form an estimate of the cost of each program segment, and (3) to determine evaluation order for the expressions in a program segment. Although it is impractical to specify how each of these functions is implemented for each syntactic construct, detailed discussions of various important situations will be given.

Before proceeding to details, however, an overview of the DELAY strategy is desirable. All of the various functions (subphases) of DELAY are performed on a single tree walk. Node specific routines are invoked on the way down the tree by the mechanism described in Section IV.1.2; the general form of one of these node-specific routines is:

(1) Determine whether the feasible optimizations discovered by FLO are, in fact, desirable in the present context.

(2) Make some decisions concerning the nature of the "preferred shape" of the operands of this node in terms of the node type and the context in which it occurs.

(3) Invoke the appropriate node specific routines on the operands of this node passing "context" to them based on the decisions made in (2).

(4) Determine the "general shape" of the code to be generated for this node given both the context in which it occurs and the "general shape" of its operands (determined in (3)).

(5) Estimate the "cost" of the code associated with the node (and its descendants). The estimated cost is expressed both in terms of code size and number of registers involved.

(6) Determine the preferred evaluation order for the operands (if appropriate).

At each level of this tree walk it is necessary for the node-specific routine to know something (not everything) about the context in which the specific node occurs. This is done by having each routine pass relevant context information to its descendants. Specific types of context information will be dealt within subsequent sections; some examples of the type of context information passed down from level to level are given below:

(1) The kind of value needed: In certain contexts the value of given expression may not be needed at all, in other cases the result of an expression may be used to generate control flow, and finally, in many contexts the value of an expression must be a genuine bit pattern.

(2) The kind of addressability required of the expression: The expression appearing on the lefthand side of an assignment operator must specify the address of a location into which a value will be stored. The expression on the righthand side of an assignment operator, on the other hand, must evaluate to the value to be stored. Because of asymmetries in most hardware the two contexts are not identical; hence the expression delayers must know which of the two contexts applies.

(3) Sign preference: In many cases it is less expensive, locally, to generate the negative of a result rather than the value with its correct sign. In some cases this local optimality leads to global optimality and in others it does not, depending on global context.

All of these will be explained in greater detail below. For the reader who may not be familiar with the instruction set of the PDP-11, Appendix A contains a brief description of the machine.

IV.2.1 Determination of Desirable Feasible Optimizations

The feasible optimizations detected by FLO are, for the most part, also desirable, but not all are. We shall illustrate using various cses. Consider, for example, the simple assignment expression

X ← .A

where ".A" is a cse. Generally the value of a common subexpression is loaded into a register; doing so, however, may not produce optimal code. If, for example, there are precisely two uses of the common subexpression.

```
X ← .A
...
Y ← .A
```

If the value of ".A" is treated as a cse and loaded into a register, the following code would be produced

```
MOV A,RØ
MOV RØ,X
...
MOV RØ,Y
```

If, on the other hand we ignore the fact that ".A" is a cse, we produce

```
MOV A,X
...
MOV A,Y
```

Both code sequences occupy six words; however, the latter has two advantages: it involves one fewer instruction executions (hence is faster), and it avoids tying up a register over some span of the program (registers are a scarce resource on the PDP-11). If there were more than two uses of ".A", however, as a cse the situation would be quite different, as is illustrated by the following, more interesting, example:

```
X ← ..(.A+4)
```

where "..(.A+4)" is a cse. Following the usual strategy of loading the value of a cse into a register would produce the following code:

```
MOV  A,RØ
MOV  @4(RØ),RØ
. . .
MOV  RØ,X
. . .
```

We might, however, have produced the following code:

```
MOV  A,RØ
MOV  4(RØ),RØ
. . .
MOV  @RØ,X
```

Or the following:

```
MOV  A, RØ
. . .
MOV  @4(RØ),X
```

Which of these alternatives is best depends upon more global context. The first alternative requires four words of code to form "..(.A+4)" in a register and no additional code for each use. The second alternative involves the same amount of code but involves an additional memory reference for each access. The third alternative involves only two words initially but requires an extra word of code for each reference. If there are only two references to "..(.A+4)" all three alternatives require the same amount of code, but the third is best since it requires one fewer instruction executions. If ".(.A+4)" is also a cse, the second alternative is best in terms of code size and requires one fewer registers.

IV.2.2 Determination of Evaluation Order

Consider the following arithmetic expression: a*b+(c+d)/(e+f). A simple left-to-right code generation scheme would produce code similar to the following:

```
a*b → R1
c+d → R2
e+f → R3
R2/R3 → R2
R1+R2 → R1
```

However, the following sequence will produce the same result and requires one fewer registers.

```
c+d → R1
e+f → R2
R1/R2 → R1
a+b → R2
R2+R1 → R2
```

Obviously since (c+d)/(e+f) requires two registers but produces its result in a single register the second of these may be used for the evaluation of a*b. Several authors [Nak67, Red69, Set70] have noted the advantages of rearranging the evaluation order in such cases and have given essentially equivalent algorithms for determining an evaluation order which minimizes the maximum number of registers used. These algorithms label each node with the number of registers necessary to evaluate that node (including its subnodes) and then specify that the preferred evaluation order (for a binary operator) is to evaluate that subnode which requires the maximum number of registers first, thus leaving enough registers free to evaluate the other subnode.

The labeling algorithm is simple. Consider a binary expression ϵ of the form "ϵ_1<binop>ϵ_2" and let $r(\epsilon_i)$ be the number of registers required to evaluate ϵ_i. Then $r(\epsilon)$, the number of registers required to evaluate the entire expression is:

(1) $r(\epsilon_1) \neq r(\epsilon_2)$: $r(\epsilon) = \max(r(\epsilon_1), r(\epsilon_2))$. In this case the operand requiring the larger number of registers should be evaluated first, leaving $r(\epsilon)-1$ registers free. Since the other operand requires at most $r(\epsilon)-1$ registers no additional registers are required. The entire expression can be evaluated using only $r(\epsilon)$ registers.

(2) $r(\epsilon_1) = r(\epsilon_2)$: $r(\epsilon) = r(\epsilon_1)+1$. In principle either expression may be evaluated first. Since the expression evaluated first leaves its result in a register only $r(\epsilon_1)-1$ will be left free. Since the other subnode also requires $r(\epsilon_1)$ registers the total number of registers required will be $r(\epsilon_1)+1$.

Unfortunately the algorithm described above does not account for the possibility of cses. In fact, no computationally reasonable algorithm (i.e., other than complete enumeration) is currently known to determine optimal evaluation order when cses exist. Therefore the Bliss/11 compiler uses an approximate technique which includes the scheme above as a degenerate case.

Consider a node ϵ_0 with subnodes ϵ_1 and ϵ_2. Assume that the evaluation of ϵ_1 will require three registers but, because it involves the last use of two cses, the evaluation also results in freeing two registers. Further, assume the evaluation of ϵ_2 requires four registers and does not involve any cses. Finally assume no new cses are generated. The situation is characterized by the diagram in Figure 14. Prior to the evaluation of ϵ_0 some number of registers, r, will be occupied and following the evaluation, independent of the order of evaluation of ϵ_1 and ϵ_2, r-1 (=r-2+1) registers will be occupied. (Two registers are freed by the evaluation of ϵ_1, but one extra is needed to hold the value of ϵ_0.) Our objective is to minimize the maximum number of registers, k, needed at any point during the evaluation process.

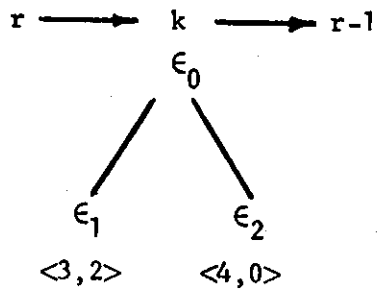


Figure 14. Determination of number of registers required to evaluate ϵ_0.

The algorithm described above, which does not account for common-sub-expressions, would specify that ϵ_2 should be evaluated first; this results in k = r+4. However, if ϵ_1 is evaluated first, only r-1 registers will be in use after its evaluation; hence we obtain k = r+3.

The example suggests that for each node, ϵ, if we knew how many cse values were "created" (first use) and "deleted" (last use) below that node a precise determination of the optimal evaluation order would be possible. Suppose each node, ϵ_i, were labeled with a triple $<u_i,c_i,d_i>$, where:

u_i = registers "required" for the evaluation of ϵ_i in the same intuitive sense as above; the derivation of form u_i is the purpose of the following analysis.

c_i = total number of cses "created" during the evaluation of ϵ_i

d_i = total number of cses "deleted" during the evaluation of ϵ_i.

Now consider a binary expression with nodes labeled as shown in Figure 15. Prior to the evaluation of ϵ_0 some number of registers, r, will be "in use," i.e., holding the values of various partial results and/or cses. After the evaluation of ϵ_0, independent of the order of evaluation of ϵ_1 and ϵ_2, $r+1+(c_1+c_2-d_1-d_2)$ registers will be in use. At some point during the evaluation of ϵ_0 and its descendants a maximum number of registers, r+k, will be in use; in general the value of k does depend upon the order of evaluation and this is the value we wish to minimize.

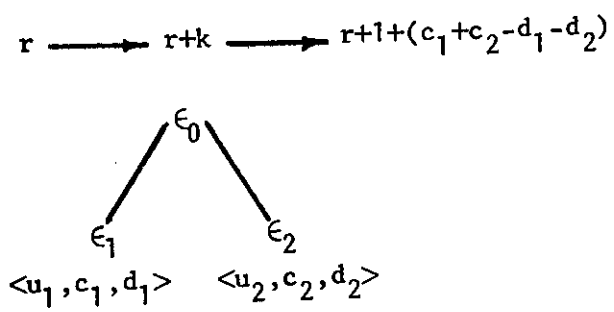


Figure 15. Determination of number of registers required to evaluate ϵ_0.

Under the assumption that c_i and d_i are known, which is equivalent to saying that the first and last uses of all common-sub-expressions are known, the number of registers that will be in use after each of ϵ_1 and ϵ_2 are evaluated is:

$$I = \delta(\epsilon_1) + \delta(\epsilon_2) + (c_1-d_1) + (c_2-d_2)$$

where

$$\delta(\epsilon) = \begin{cases} 0 & \text{if } \epsilon \text{ is a terminal node} \\ 1 & \text{if } \epsilon \text{ is a non-terminal node.} \end{cases}$$

The number of registers required to evaluate ϵ_0 if ϵ_1 is evaluated first, u_{12}, and the number required if ϵ_2 is evaluated first, u_{21}, are then given by

$$u_{12} = \max(u_1, u_2+1, I+\eta(\epsilon_1,\epsilon_2))$$
$$u_{21} = \max(u_1+1, u_2, I+\eta(\epsilon_1,\epsilon_2))$$

where

$\eta(\epsilon_1,\epsilon_2)$ = 1 if both ϵ_1 and ϵ_2 are terminal nodes and/or cses which are not last uses (i.e., neither register may be re-used)

Ø otherwise

Clearly we will choose the evaluation order depending upon whether u_{12} is smaller than u_{21} or vice versa. The labeling scheme is then

$$u_0 = \min[\max(u_1,u_2+1,\ 1+\eta(\epsilon_1,\epsilon_2)),\ \max(u_1+1,u_2,1+\eta(\epsilon_1,\epsilon_2))]$$
$$c_0 = c_1+c_2+\gamma(\epsilon_1)+\gamma(\epsilon_2)$$
$$d_0 = d_1+d_2+\lambda(\epsilon_1)+\lambda(\epsilon_2)$$

where

$\gamma(\epsilon)$ = 1 if ϵ is a cse creation
Ø otherwise

$\lambda(\epsilon)$ = 1 if ϵ is the "last use" of a cse
Ø otherwise

Unfortunately this scheme cannot be used. It depends on knowing which are the first and last uses of cses. This information cannot be known until the evaluation order is fixed, and hence cannot be used to determine the evaluation order. The analysis does, however, suggest the approximate technique used in the compiler.

For various reasons the compiler requires that the creation of a cse value be done by that instance of the expression which occurs earliest in the initial order; hence c_i is known. It does not, on the other hand, specify which is the last use; hence, d_i cannot be determined exactly. However, if n_i is the number of occurrences of a cse, then $1/(n_i-1)$ is the probability that any particular use is the last. The compiler uses exactly the algorithm implied by the analysis above except that it substitutes

$\lambda(\epsilon_i) = 1/(n_i-1)$ if ϵ_i is a cse use (not creation)
Ø otherwise.

It should be noted that the algorithm is exact if no cses are present ($n_i = 2$).

In addition to the evaluation order determinations based on register usage, another set of evaluation order decisions are based on a measure of the code complexity of the subnodes. We shall consider two cases: the order of evaluation of the operands in a boolean expression and the order of placement of the then and else portions of a conditional expression.

The semantics of Bliss specify that: (1) the operands of a boolean (notably and and or) may be evaluated in any order, and (2) a boolean which produces a flow result (e.g., in the boolean expression of a conditonal) need only be evaluated as far as necessary to determine truth or falsity of the expression. Therefore, in a construct of the form "if(ϵ_1 and ϵ_2) then..." it is advantageous to first evaluate whichever of ϵ_1 and ϵ_2 is most likely to

determine the falsity of "ϵ_1 and ϵ_2" with the least effort. Lacking knowledge of the probability of the truth or falsity of the expressions ϵ_1 and ϵ_2 independently, the compiler simply chooses to evaluate the least complex one first. Note that the issue of register-use-complexity is irrelevant here since the and itself need never be formed. The complexity measure used is the (approximate) amount of code required to evaluate the operands.

The pecularities of the target machine give rise to another use of the code-complexity. Consider the conditional expression

$$\text{if } \epsilon_0 \text{ then } \epsilon_1 \text{ else } \epsilon_2$$

which is obviously equivalent to

$$\text{if not } \epsilon_0 \text{ then } \epsilon_2 \text{ else } \epsilon_1$$

Normally one would not expect one of these forms to be preferable to the other (unless, of course, a symmetric set of relational operators were not available). However, the fact that the conditional branch instruction on the PDP-11 can branch only a short distance makes whichever form places the smaller (less complex) code segment in the then portion more advantageous.

IV.2.3 Target Paths and Unary Complement Operators

These two topics will be discussed together since they are frequently, but not always, related. In particular in this section we shall choose our examples from the plus, "+", operator where the two are inextricably intertwined.

Consider a binary expression, ϵ_0, whose operands are ϵ_1 and ϵ_2. Suppose that the results of ϵ_1 and ϵ_2 are available in temporary locations (registers) t_1 and t_2 respectively and that the results of the entire expression is to be produced in t_0. If the operator of ϵ_0, call it op_0, is commutative, two instruction sequences are possible:*

$$(1)\ \ t_1 \rightarrow t_0;\ t_0\ op_0\ t_2 \rightarrow t_0$$

or

$$(2)\ \ t_2 \rightarrow t_0;\ t_0\ op_0\ t_1 \rightarrow t_0$$

We refer to a subnode as lying on the "target path" of its ancestor if the value of that node is first moved to the ancestor's temporary and then operated upon by its sibling; thus in (1) ϵ_1 is on the target path.**

---

*In this section we are purposely avoiding the question of whether ϵ_1 or ϵ_2 is evaluated first; see Section IV.2.2.

**It is clearly advantageous for a node on the target path and its ancestor to use the same temporary location (in which case the move can be eliminated); see Section IV.3.1.

The term "unary complement operators" seems to have been introduced by [Fra70]. More formal treatments than will be given here may be found in [Set70] and [Bea72]. The term refers to the fact that the properties of real numbers and boolean values may be exploited to reduce the complexity of evaluating certain expressions. Identities such as the following are used:

(1) $x-y \equiv x+(-y)$
(2) $x-y \equiv -(y-x)$ x,y are real (or integer) values
(3) $x*y \equiv (-x)*(-y)$
(4) $a \wedge b \equiv \sim(\sim a \vee \sim b)$ a,b are boolean values

The set of identities exploited may also be extended to include those related to the machine repesentation of values such as, for two's complement machines, $-a = \sim a+1$.

The term "unary complement" is derived from the properties of negate and complement that they are their own inverses, e.g., $--a = a$. However, the term is also applied to those cases in which one of these operators may be subsumed into a binary operator at a higher level in the tree. To see the effect of these transformations consider the expression "-y*(a-b)". A simple left-to-right evaluation of this expression would produce code of the form:

```
y → R1
-R1 → R1
a → R2
R2-b → R2
R1*R2 → R1
```

By applying identities (2) and (3) given above it is possible to rewrite this expression as y*(b-a) which produces

```
y → R1
b → R2
R2-a → R2
R1*R2 → R1
```

By further changing the evaluation order to "(b-a)*y" as specified in above we obtain

```
b → R1
R1-a → R1
R1*y → R1
```

In order to apply these transformations it is not necessary to perform actual transformations of the tree; rather a set of fields may be added to the tree to specify which, if any, transformations have been performed. In the remainder of this section we shall concentrate on the binary plus operator and its interaction with the unary negate and with common-sub-expressions. For these transformations the following set of fields are adequate:

NEG (1 bit) =	Ø	the result is "correct," i.e., has the same sign as specified in the source program
	1	the result is the negative of that specified in the source program
NDT (1 bit) =	1	the result is <u>not</u> contained in a "destroyable temporary;" this will be true of all programmer-defined variables and common-sub-expressions
	Ø	the result is contained in a "destroyable temporary"
TPATH (1 bit) =	Ø	the target path is the left (natural) descendant operand
	1	the target path is the right descendant operand
NLOD (1 bit) =	Ø	load the value of the target path operand with sign unmodified
	1	load the negative of the value of the target path operand
OP		encoding of the operator to be performed

The transformations to be described will utilize the NEG and NDT fields of the operands of a node to determine the NEG, TPATH, NLOD, and OP fields of the node. To make an intelligent determination, however, it is necessary to know something about the global context in which the expression represented by the node occurs; in particular for a "+" node it is necessary to know whether, in the context of its immediate ancestor, it is preferable to develop the positive or negative of the value represented by the "+" node in the source program. Therefore whenever a delay routine is called the caller may specify its preference for various attributes of the operand. In the particular case of the sign of an operand the caller may specify: (1) the correct sign is preferred (but the opposite sign is acceptable), (2) the opposite sign is preferred (but the correct sign is acceptable), (3) the correct sign <u>must</u> be generated, or (4) either is equally acceptable.

For the purposes of this section the delay routine for the plus operator has the following structure:

(1) determine sign preference for one operand (e.g., the "left" one)
(2) delay that operand, passing sign preferences from (1)
(3) determine the sign preference for the other operand
(4) delay that operand, passing sign preference from (3)
(5) determine NEG, TPATH, NLOD, and OP based on caller's sign preference and the attributes of the operands.

A full explanation of the determination of sign preference in (1) is rather tedious but is based on considerations such as, if the caller prefers the negated result, it is preferable for at least one operand, and possibly both, to be negated. Decision tables for the sign-preference decisions are given in Figures 16 and 17.

After the operands of a "+" node have been delayed it is possible to use a decision table such as that shown in Figure 18 to determine the NEG, TPATH, NLOD, and OP fields of the "+" node itself. While most of this table should be obvious, a few things should be noted:

GIVEN			PREF
CONTEXT	NDT_L	NDT_R	
DON'T CARE	0	0	+
or	0	1	+
POSITIVE	1	0	+
PREFERRED	1	1	+
	0	0	-
NEGATIVE	0	1	-
PREFERRED	1	0	-
	1	1	-
	0	0	+
MUST BE	0	1	+
POSITIVE	1	0	-
	1	1	+

Figure 16. Sign preference for left operand of "+".

(1) As noted earlier, it is advantageous for an ancestor node and its target path descendant to use the same temporary location. Although the advantage is not enforced at this point in the compiler the prototype code shown in the code field assumes it will be exploited; this was done to accent the situations in which an explicit move is required.

(2) In some cases the caller's sign request cannot be satisfied without additional code. It should be clear that it never requires more code to modify the sign at a higher level in the tree; hence, since these requests are not binding, the caller's preference is ignored.

(3) In two cases the caller's preference could be satisfied without additional code by using a "load negative." The PDP-11 does not have such an instruction, however, and hence the request is not honored. These two cases would be changed for a machine for which "load" and "load negative" incurred comparable costs.

The actual implementation of the decisions represented by Figure 18 consists of computing an index (based on the caller's sign preference, NEG_1, DT_1, NEG_2, and DT_2), using this index to extract the information represented in the table and storing the extracted information into the node.

CONTEXT	GIVEN NEG_L	GIVEN DT_L	GIVEN DT_R	PREF
DON'T CARE or POSITIVE PREFERRED	0	0	0	DC
	0	0	1	DC
	0	1	0	+
	0	1	1	DC
	1	0	0	+
	1	0	1	DC
	1	1	0	+
	1	1	1	-
NEGATIVE PREFERRED	0	0	0	-
	0	0	1	DC
	0	1	0	-
	0	1	1	+
	1	0	0	DC
	1	0	1	DC
	1	1	0	-
	1	1	1	DC
MUST BE POSITIVE	0	0	0	DC
	0	0	1	DC
	0	1	0	+
	0	1	1	DC
	1	0	0	+
	1	0	1	+
	1	1	0	+
	1	1	1	+

Figure 17. Sign preference for right operand of "+".

IV.2.4 Other Delaying of the Plus Operator

Other delaying operations are possible on addition besides those described in the previous section; in particular, the indexing portion of address calculation provides a potential implicit addition. Exploitation of such operations is highly machine specific. Thus, while optimizations analogous to those described in this section are possible on most machines, the details will vary widely (on machines such as the IBM 360, for example, where double indexing is possible). Because of the machine specific nature of these optimizations we shall assume that the reader is familiar with the PDP-11 addressing structure.

To see the effect of these optimizations, consider the following trivial program:

(EXTERNAL A,B,C; A←.(A+.B+3+.C)+4)

CONTEXT (REQUEST)	OPERANDS (LEFT) NEG	(LEFT) NDT	(RIGHT) NEG	(RIGHT) NDT	BITS NEG	BITS TPATH	BITS NLOD	BITS OP	RESULT	CODE	
DON'T CARE and POSITIVE PREFERRED	0	0	0	0	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	0	0	0	1	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	0	0	1	0	0	0	0	-		$t_2 - t_1 \rightarrow t_2$	
	0	0	1	1	0	0	0	-		$t_2 - t_1 \rightarrow t_2$	
	0	1	0	0	0	1	0	+		$t_2 + t_1 \rightarrow t_2$	
	0	1	0	1	0	0	0	+	$t_1 \rightarrow t_0$	$t_0 + t_2 \rightarrow t_0$	
	0	1	1	0	1	1	0	-		$t_2 - t_1 \rightarrow t_2$	(1)
	0	1	1	1	1	0	0	-	$t_1 \rightarrow t_0$	$t_0 - t_2 \rightarrow t_0$	
	1	0	0	0	0	0	0	-		$t_2 - t_1 \rightarrow t_2$	
	1	0	0	1	1	0	0	-		$t_1 - t_2 \rightarrow t_1$	(1)
	1	0	1	0	1	0	0	+		$t_1 + t_2 \rightarrow t_1$	(1)
	1	0	1	1	1	0	0	+		$t_1 + t_2 \rightarrow t_1$	(1)
	1	1	0	0	0	1	0	-		$t_2 - t_1 \rightarrow t_2$	
	1	1	0	1	0	1	0	-	$t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	1	1	0	1	1	0	+		$t_2 + t_1 \rightarrow t_2$	(1)
	1	1	1	1	1	0	0	+	$t_1 \rightarrow t_0$	$t_0 + t_2 \rightarrow t_0$	(2)
NEGATIVE PREFERRED	0	0	0	0	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	(1)
	0	0	0	1	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	(1)
	0	0	1	0	1	1	0	-		$t_2 - t_1 \rightarrow t_2$	
	0	0	1	1	0	0	0	-		$t_1 - t_2 \rightarrow t_1$	(1)
	0	1	0	0	0	1	0	+		$t_1 + t_2 \rightarrow t_1$	(1)
	0	1	0	1	0	0	0	+	$t_1 \rightarrow t_0$	$t_0 + t_2 \rightarrow t_0$	(2)
	0	1	1	0	1	1	0	-		$t_2 - t_1 \rightarrow t_2$	
	0	1	1	1	1	0	0	-	$t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	0	0	0	1	0	0	-		$t_1 - t_2 \rightarrow t_1$	
	1	0	0	1	1	0	0	-		$t_1 - t_2 \rightarrow t_1$	
	1	0	1	0	1	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	1	0	1	1	1	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	1	1	0	0	0	1	0	-		$t_2 - t_1 \rightarrow t_2$	(1)
	1	1	0	1	1	0	0	-	$t_1 \rightarrow t_0$	$t_0 - t_2 \rightarrow t_0$	
	1	1	1	0	1	1	0	+		$t_0 + t_1 \rightarrow t_2$	
	1	1	1	1	1	0	0	+	$t_1 \rightarrow t_0$	$t_0 + t_2 \rightarrow t_0$	
MUST BE POSITIVE	0	0	0	0	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	0	0	0	1	0	0	0	+		$t_1 + t_2 \rightarrow t_1$	
	0	0	1	0	0	0	0	-		$t_1 - t_2 \rightarrow t_1$	
	0	0	1	1	0	0	0	-		$t_2 - t_1 \rightarrow t_2$	
	0	1	0	0	0	1	0	+		$t_2 + t_1 \rightarrow t_2$	
	0	1	0	1	0	0	0	+	$t_1 \rightarrow t_0$	$t_0 + t_2 \rightarrow t_0$	
	0	1	1	0	0	1	1	+	$-t_2 \rightarrow t_0$	$t_0 + t_1 \rightarrow t_0$	
	0	1	1	1	0	0	0	-	$t_1 \rightarrow t_0$	$t_1 - t_2 \rightarrow t_2$	
	1	0	0	0	0	0	0	-		$t_2 - t_1 \rightarrow t_2$	
	1	0	0	1	0	0	1	+	$t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	0	1	0	0	1	0	+	$-t_1 \rightarrow t_0$	$t_0 - t_2 \rightarrow t_0$	
	1	0	1	1	0	0	1	+	$-t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	1	0	0	0	1	0	-		$t_2 - t_1 \rightarrow t_2$	
	1	1	0	1	0	0	1	-	$t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	1	1	0	0	1	1	-	$-t_2 \rightarrow t_0$	$t_0 - t_1 \rightarrow t_0$	
	1	1	1	1	0	0	1	-	$-t_1 \rightarrow t_0$	$t_0 - t_2 \rightarrow t_0$	

Figure 18. Target path and unary complement decision for "+".

The code produced by the compiler for this example is:

```
MOV @#C,R5
ADD @#B,R5
MOV A+3(R5),R5
ADD #4,R5
MOV R5,A
```

Examination of this example will show that both "A+" and "+3" were performed implicitly by indexing in the MOV instruction.

In order to perform this type of optimization two kinds of information are necessary - the context in which the expression occurs and the "state" of similar optimizations performed on its operands. Once again, although some of the optimizations are applicable to other operators, we shall focus on the plus operator for a concrete example.

Because of asymmetries in the PDP-11 addressing structure, and, indeed, in that of most machines, it is desirable to distinguish two contexts in which an expression may occur; that of an operand and that of an address. Consider the expression ".A+4" in the two examples below and the difference in the code produced for each:

(a) X ← .A+4

```
MOV  A,X
ADD  #4, X
```

(b) X ← .(.A+4)

```
MOV  A,RØ
MOV  4(RØ), X
```

In (a) the expression ".A+4" is an operand and its value must actually be generated somewhere. In (b) the value of ".A+4" is not needed explicitly - only implicitly in forming an address - and thus may be formed when needed by the indexing operation.

Since the ancestor of a node is aware of the context in which its descendant occurs, context information is passed down from ancestor to descendant by the mechanism which should be familiar by now. The descendant, in turn, passes context information to its descendants and generates code (if any) as appropriate on the basis of the state of its descendants, sets its own state and returns to the ancestor. In the particular case of the address vs. operand context, four context specifications are permitted:

(1) operand	Code must be generated, as in example (a) above, to form the value of the expression.
(2) temporary	This is essentially equivalent to the "operand" case except that the value must be generated in a compiler-generated temporary, i.e., a register. This case is used almost exclusively for frequently used cses.
(3) address	Exploitation of indexing, etc., is possible and desirable.
(4) arbitrary	Either form is allowed and the called routine is free to generate the "least expensive" form; the caller will generate appropriate code if he doesn't like what he gets back.

The delaying actions in the case that an "operand" (or "temporary") is requested are controlled exclusively by the target path, unary complement operators, and evaluation order discussed previously; here we will focus on "address" (or "arbitrary") requests. For these there are six "interesting" states which correspond, roughly, to addressing modes which may be optimized further; they are:

L	literal	These are simple literal values.
RN	relocatable name	These are names which will have an absolute address when the program is loaded. They would be equivalent to literal values except they involve an unknown, and unknowable, relocation constant.
T	temporary	The value of the node is contained in a temporary, i.e., a register.
TL	temp + literal	The value of the node is the result of adding a literal to a value contained in a temporary; i.e. indexing is indicated.
TRN	temp + relocatable name	Similar to TL except, again, for the unknown relocation constant.
Z	other	All other cases.

The state for an expression is determined from the states of its operands. Rules for deriving these states are given in Figure 19. Each entry in this table contains three pieces of information (the first and/or third of which may be empty): the code produced (if any), the new state, and the compile time operations necessary to derive the parameters of the new state. In order to describe these attributes of the state transition T_0, T_1, and T_2 are used to name the temporary of the node and its left and right operands respectively, L_0, L_1, and L_2 denote the corresponding literal parameters associated with the T and TL states; and N_0, N_1, and N_2 denote the corresponding relocatable names associated with the RN and TRN states. For example, consider the expression ϵ_0 which is actually "$\epsilon_1+\epsilon_2$," and assume that the content for ϵ_0 requires an "address." Further suppose that the states of ϵ_1 and ϵ_2 are TL and TRN respectively. Then the table specifies that the contents of the two temporaries should be added together and the resulting state should be TRN with the resulting name differing by a (known) constant amount (L_1).

The effect of these transformations in a larger context may be seen by returning to the example introduced at the beginning of this section:

(EXTERNAL A,B,C; A ← .(A + .B + 3 + .C) + 4)

The tree of the assignment expression is shown in Figure 20 with its arcs labeled by the context information which would be passed down (a ≡ address, o ≡ operand, b ≡ arbitrary). The states of the nodes and code generated for each are shown in Figure 21. It should be noted that:

LEFT OPERAND STATES			RIGHT OPERAND STATES					
			L	RN	T	TL	TRN	Z
			L_2	N_2	$.T_2$	$.T_2+L_2$	$.T_2+N_2$	Z_2
L	L_1		L $L_0=L_1+L_2$	RN $R_0=N_2+L_1$	TL $L_0=L_1$	TL $L_0=L_1+L_2$	TRN $N_0=N_2+L_1$	$T_0 \leftarrow Z_2$ TL $L_0=L_1$
RN	N_1		RN $N_0=N_1-L_2$	$T_0 \leftarrow N_1$ TRN $N_0=N_2$	TRN $N_0=N_1$	TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_2+N_2$ TRN $N_0=N_1$	$T_0 \leftarrow Z_2$ TRN $N_0=N_1$
T	$.T_1$		TL $L_0=L_2$	TRN	$T_0 \leftarrow .T_1+.T_2$ T	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+Z_2$ T
TL	$.T_1+L_1$		TL $L_0=L_1+L_2$	TRN $N_0=N_2+L_1$	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_1$	$T_0 \leftarrow .T_1+.T_2$ TL $L_0=L_1+L_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=L_1+N_2$	$T_0 \leftarrow .T_1+z_2$ TL $L_0=L_1$
TRN	$.T_1+N_1$		$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_1+N_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1$	$T_0 \leftarrow .T_1+.T_2$ TRN $N_0=N_1+L_2$	$T_0 \leftarrow .T_1+.T_2+N_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_1+Z_2$ TRN $N_0=N_1$
Z	Z_1		$T_0 \leftarrow Z_1$ TL $L_0=L_2$	$T_0 \leftarrow Z_1$ TRN $N_0=N_2$	$T_0 \leftarrow .T_2+Z_1$ T	$T_0 \leftarrow .T_2+Z_1$ TL $L_0=L_2$	$T_0 \leftarrow .T_2+Z_1$ TRN $N_0=N_2$	$T_0 \leftarrow Z_1+Z_2$ T

Figure 19. Exploitation of addressing modes in a "+" node.

(1) New temporaries (T_1,T_2,T_3) are introduced only where explicit code is generated. Otherwise the temporaries are assumed to propogate up the tree. (In fact, comparison of the tree with the code shown earlier shows that all the temporaries have been bound to the same register, R5.)

(2) Note the transition of the highest "+" node is different - the transition is not given in our table because the "+" node receives an "operand" request from the "←" node.

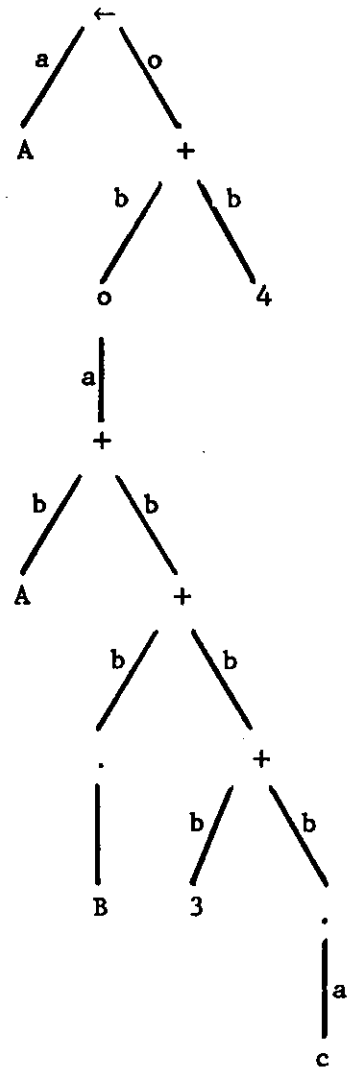


Figure 2Ø. Tree of example program with context information.

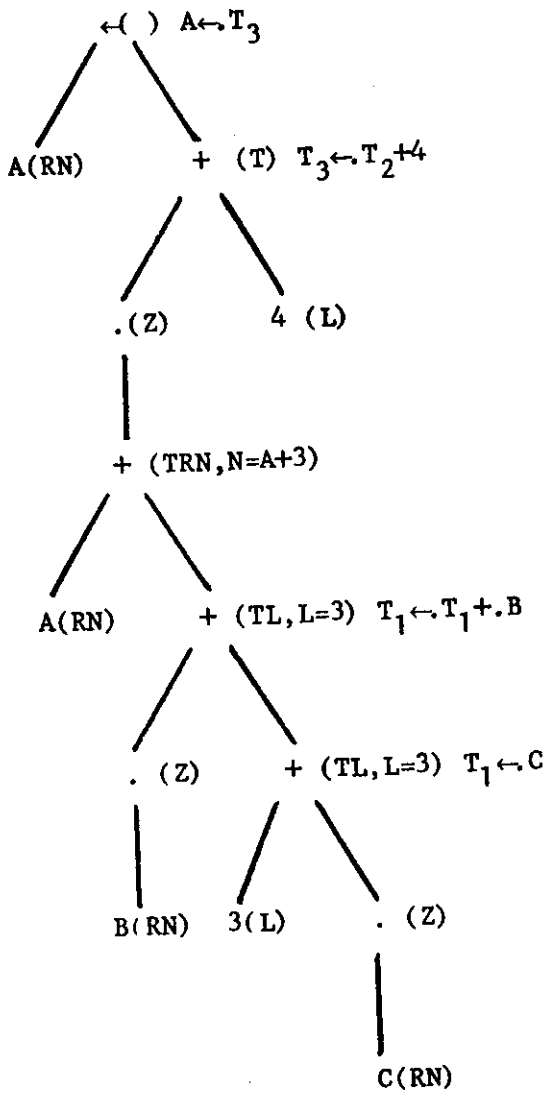


Figure 21. Tree of example program with state information.

IV.3. Temporary Name Binding

The phases TLA, RANK, and PACK are collectively referred to as "temporary name binding," or TNBIND. The function of these phases is to allocate temporary storage locations, both registers and memory. The majority of these temporary locations are used for generating and holding intermediate results and common-sub-expressions; however, explicitly declared "local" and "register" variables are assigned by the same mechanism. The compiler makes virtually no distinction between compiler and user defined temporaries, and none will be made in this discussion.

TNBIND consists of three phases. The first, TLA, creates "temporary names," TN's, to represent the location in which results will be held; at this stage the TN's are not bound to actual locations - they are purely symbolic. During TLA a "targeting" strategy attempts to insure that results will be placed in optimal locations (see below) and information is gathered on how the temporary is used and on the "lifetime" of the temporary. An optimization unrelated to TN binding, that of label targeting is also performed during TLA. The second phase, RANK, uses the information gathered by TLA to create a list of all TN's in order of their "importance." The final phase, PACK, actually assigns (binds) the TN's to actual locations - it does so by binding the most important TN's first to the most desirable locations. Each of these phases is described in more detail below.

IV.3.1 TLA

The discussion of TLA is divided into five pieces: targeting, cost determination, lifetime characterization, the run-time stack, and label assignment.

IV.3.1a Targeting

Consider the following simple program:

(own x,y; routine f(z) = (local l; l←.x+.z; .l);...)

The tree for the routine is shown in Figure 22. For convenience the nodes in this tree have been labeled in the order that they would be encountered while backing out of an execution-order tree walk. In principle, of course, a unique temporary name could be associated with each node plus one additional TN for the local variable, l. To do so, however, would lose much of the structural information available in the tree. One of the prime goals of TLA is to avoid such loses.

TLA is organized as a set of mutually-recursive, node-specific action routines. An execution-order tree-walk is performed invoking these routines at each node in a manner analogous to that in DELAY. Each routine is passed a (possibly null) TN as its "target." If possible and reasonable, the node-specific routine will use this target TN for the result of its node. It does so by passing the target TN to the routine which will handle its "target-path" sub-node -- recall that the "target-path" was determined in DELAY. Upon return from the action-specific routines for its subnodes, which will assign TN's for their results, it is necessary to resolve the target request from above with what was actually done below. In the simple case the target path will have been assigned to the target TN, and that TN will be used for the current node as well. Other cases are more complex and will be discussed below; however, first let's return to our example.

The root node "8:routine" has no ancestor and hence is not passed a target. The node-specific action, however, knows this and also knows that the value of a routine is returned in a standard place. It therefore creates a TN, say t_0, and passes this as a target to its subnode. The node specific action for compounds passes no targets to all but the last subnode; the last subnode is passed the target from above. The node specific action for store, "←", is quite complex, but in cases such as that shown will pass the location of the

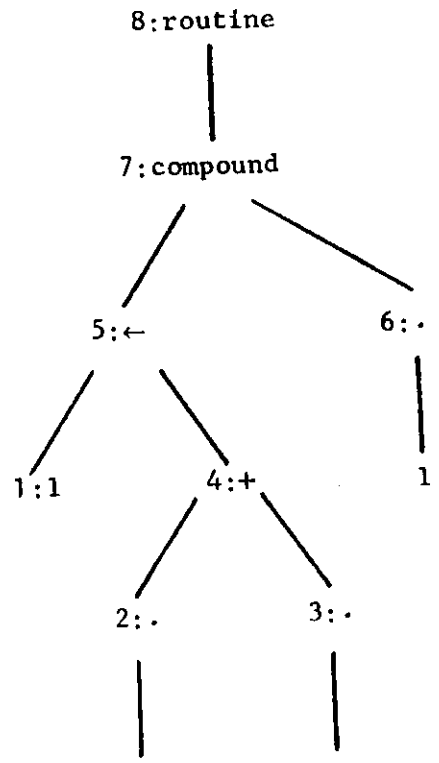


Figure 22. Tree of example routine.

left-hand-side as a target to the right-hand-side. Thus after the compound the state is characterized by the diagram in Figure 23 in which targets are shown by directed arrows adjacent to the tree and assignments are indicated in parentheses next to the node. Notice that the "compound" had to resolve the request that its result be placed in t_0 with the fact that its last subnode placed its result in t_1. Such resolutions are invariably made in favor of leaving the value where it is since at most one "MOV" is implied and it might as well be generated higher in the tree. However, at this stage we can note that it would be "nice" if, even though t_0 and t_1 are distinct temporary names, they were bound to the same location. Therefore another mechanism, "preferencing," is introduced. If two or more TN's belong to the same preference class they will (if possible) be bound to the same location by PACK. In particular, for this example it is possible to use the same register (RØ) for both the value of the routine and the local "I", and the code produced for the entire routine is:

```
F:  MOV Z,RØ
    ADD X,RØ
    RTS PC
```

With this introduction to the function of TLA we can return to the complexities of TLA alluded to earlier; these complexities arise because of node-specific idiosyncracies and common-sub-expressions. The generic form of node specific actions is (roughly):

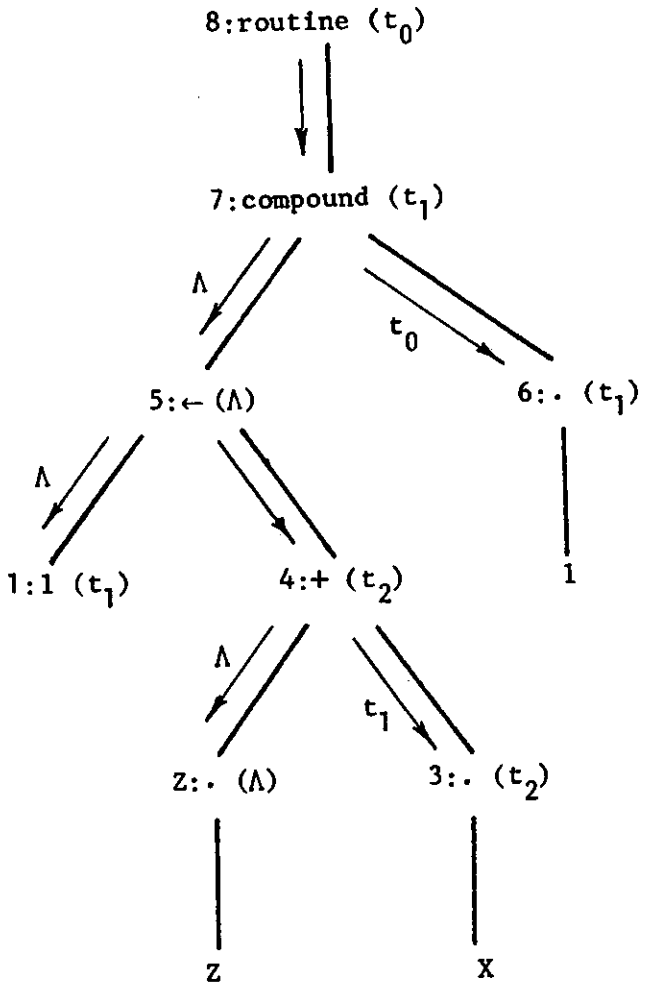


Figure 23. Tree with temporary names assigned.

```
if u(node) then
    begin
    if c(node) then target ← Λ
    {perform node-specific action, place actual binding in tn};
    if c(node) then binduses (node,tn);
    end
```

where

u(node) =	true iff the node is a common-sub-expression use.
c(node) =	true iff the node is a common-sub-expression creation.
binduses	binds all uses of a given common-sub-expression creation to the same TN as the creation.

Nodes which represent neither uses nor creations of common-sub-expressions are handled

as described previously. Uses of cse's, on the other hand, are simply ignored since their temporary names are assigned when the creation node is handled. Creation nodes ignore their target in order to insure that later ranking and packing treat the cse as an independent entity.

It is impossible to treat all of the node-specific actions; however, two will be discussed as examples - the "typical" binary operator and if-then-else. These were chosen to illustrate the difference between arithmetic and control operators. In order to present these examples some functions and predicates are needed:

functions

tla(node,target) invoke the appropriate node-specific action on the specified node, passing it the specified target TN. TLA returns a (possibly different) TN.

gettn create a new temporary name and return it.

wantpref(t1,t2) form the union of the preference classes of the temporary names t1 and t2.

bindset(set) perform the "tla" operation on each node which is a member of the specified set; this function is applied, for example, to the α and ω sets associated with an if-then-else.

predicates

δ(node) true iff the result specified by the node is "destroyable," i.e., is not a user-defined variable or cse with remaining uses.

$\tau(node_i)$ true iff the ith subnode of the specified node is its "target path."

tnneeded(node) true iff a result temporary is needed for the specified node.

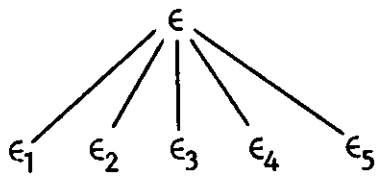
Now consider a binary node, ϵ, with subnodes ϵ_1 and ϵ_2. The node-specific action is:

```
TN←target;
l←tla (ε1, if τ(ε1) then TN else Λ);
r←tla (ε2, if τ(ε2) then TN else Λ);
temp←Λ;
if TN=Λ then
    if τ(ε1) then (if δ(ε1) then TN←l else temp←l) else
    if τ(ε2) then (if δ(ε2) then TN←r else temp←r);
if tnneeded(ε) then
    (if TN=Λ then TN←gettn( ); wantpref(TN,temp));
```

Thus the action is first to perform "tla" on the subnodes, passing the target down along the target path. Next, if there was no target, the resultant TN for the target path is closen if it

is destroyable. If, on the other hand, there was no target and the TN on the target path is not destroyable, a new temporary name is created and added to the preference class of the target-path result.

Now consider an if-then-else node, ϵ. Such a node actually has five subnodes



where ϵ_1 is the α-set, ϵ_2 the boolean, ϵ_3 and ϵ_4 are the then and else parts, and ϵ_5 is the ω-set. The node specific action is:

```
bindset(ϵ1);
tla(ϵ2,Λ);
tt←tla(ϵ3,Λ);
te←tla(ϵ4,Λ);
if tnneeded (ϵ) then (TN←gettn( ); wantpref(TN,tt); wantpref (TN,te));
bindset (ϵ5);
```

This coding is, perhaps, surprising in that it makes no use of the target. Perhaps one might have expected something of the form

```
TN←target
bindset(ϵ1);
tla(ϵ2,Λ);
tt←tla (ϵ3,TN);
te←tla (ϵ4,TN);
if TN = Λ ∧ tnneeded (ϵ) then TN←gettn( );
if TN ≠ tt then wantpref (TN,tt);
if TN ≠ te then wantpref (TN,te);
bindset (ϵ5);
```

This form would, indeed, be more consistent with the targeting philosophy. However, experience has shown that the latter form will tie up a temporary name over too great a span of the program.

IV.3.1b Cost Determination

Each temporary name may be used more than once and may be used in each instance in one of several ways - e.g. as a simple value, as an index quantity, etc. In the case that there are insufficient registers to hold all the temporary values needed at one time we would like those in registers to be the "most important" ones and those in memory to be of lesser importance. We have chosen to define the importance in terms of a cost measure. The measure currently used is:

$$C = \sum_{\text{uses}} (1 + m(\text{node}) * \sigma(\text{node}) * (\text{loopdepth} + 1))$$

where

- m(node) is a measure of the number of memory references implied in accessing the result represented by the node and is implied by the addressing mode determined by DELAY. For example, register mode has a cost of Ø, deferred register mode has a cost of 1, deferred indexed mode has a cost of 3, etc.

- σ(node) is a measure of the number of instructions which will be generated to implement a given node; this measure is usually 1, but can be considerably larger for a bit-field extraction (for example).

Actually two such measures are accumulated during TLA - a "max" and "min" version. The two versions are identical in form, but differ in that one (max) assumes the temporary will be allocated in memory, and the other (min) assumes the temporary will be allocated to a register.

Notice the term "loopdepth + 1" tends to heavily weight the importance in favor of temporary names which occur inside loops.

IV.3.1c Lifetime Determination

The characterization of the lifetime of a temporary is one of those extremely important problems for which no truly satisfactory solution exists in the literature. The simplest scheme is to characterize the lifetime in terms of a linear span from its earliest to latest use in the program. This scheme is unsatisfactory in the context of forked constructs. Consider, for example,

$$\ldots \epsilon \ldots \text{ if } \beta \text{ then } \gamma \text{ else } (\ldots \epsilon \ldots)$$

The linear measure would prevent the use of the location of the expression "ϵ" throughout the entire body of "γ" even though this might be legitimate. The other extreme is to characterize the lifetime of a location along each possible flow path. While this affords a complete solution, it entails recording a great deal of data. The scheme used in the Bliss/11 compiler is a compromise which has worked quite satisfactorily.

During TLA each node is labeled with two values, a "lon", or "linear order number," and "fon", or "flow order number". The lon is a unique, monotonically increasing value assigned in execution order. Thus the lon specifies a relative position in the final program where the code for a specific node will go. The fon, on the other hand, monotonically increases along a flow path but is the same for the head of each branch eminating from a fork. The <lon,fon> pairs for a hypothetical case expression are illustrated in Figure 24.

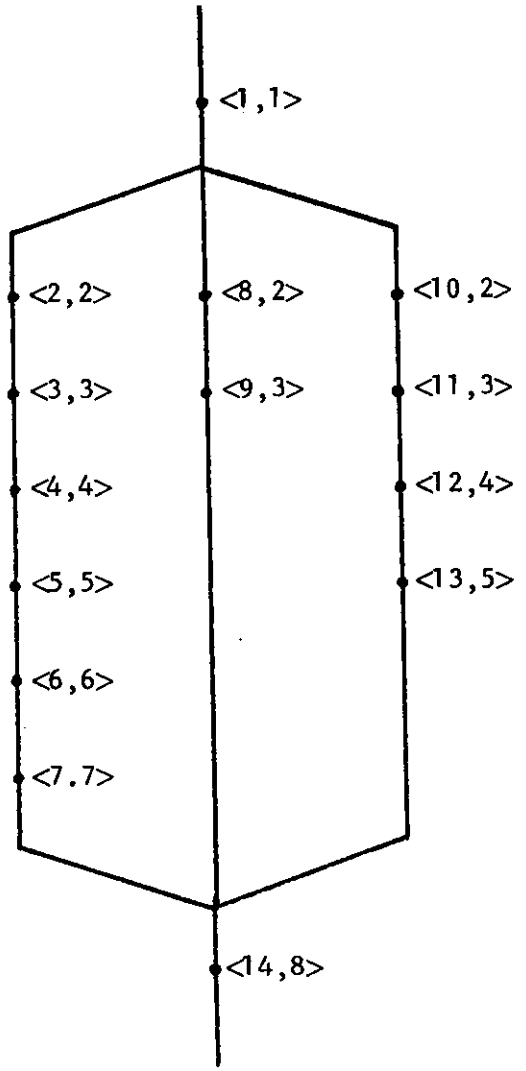


Figure 24. Lon-fon values for a case expression.

Since fon≤lon, the set of possible <lon,fon> pairs is representable by the set of points on or below the diagonal in the first quadrant of 2-space as shown in Figure 25a. We choose to represent the lifetime of a temporary name by four points which define a rectangle in this space - its lon and fon values of first and last use (lonfu, lonlu, fonfu,

fonlu). This lifetime may be seen graphically in Figure 25b. Two temporary names are considered contemporaneous if the rectangles representing their lifetimes intersect. In such a case they cannot be allocated to the same location.

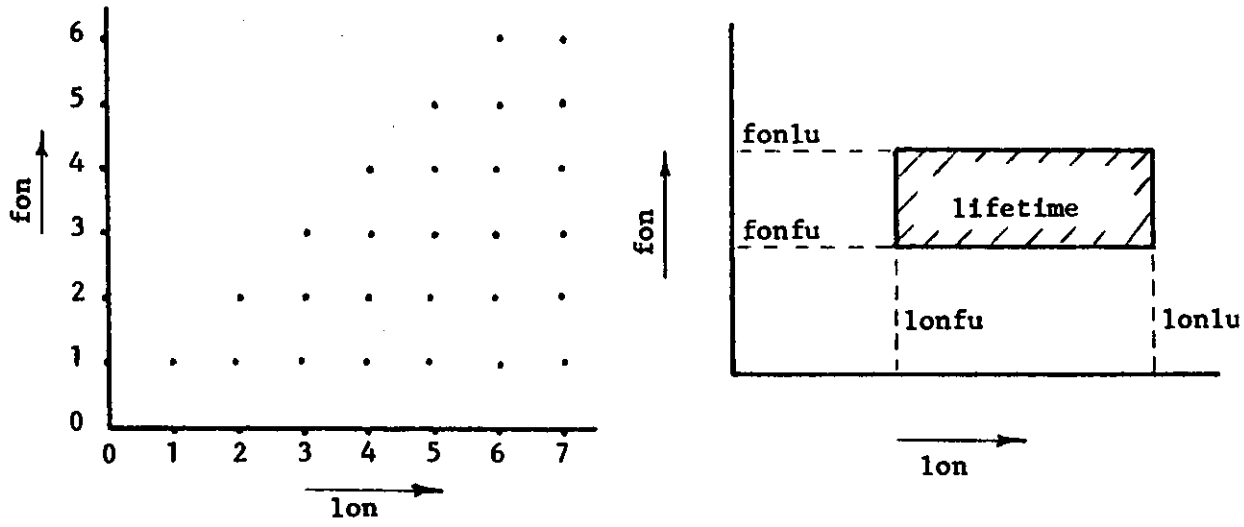


Figure 25. Graphical representation of the lon-fon space.

The lonfu, lonlu, fonfu, and fonlu are noted during TLA as the temporary names are assigned to nodes by a function "span" invoked by each node-specific action routine. In particular, "span" is invoked on the subnodes of a given node in order to establish that the result represented by the subnode is still "in-use" at the parent.

IV.3.1d The Run-Time Stack

Generally this paper has avoided the description of the run-time environment for compiled programs. This section will deviate from that practice in order to describe one of the more important issues treated by TNBIND. The environment of a routine consists of a stack containing (possibly) some local variables, saved registers, return address, and actual parameters of the routine as illustrated in Figure 26. A single register, SP, points to the current top of this stack; no other pointers into the stack are maintained, hence all items in the stack are indexed relative to this register. A typical routine call "f(.x,.y)" for example, produces

```
MOV X,-(SP)      ; PUSH X
MOV Y,-(SP)      ; PUSH Y
JSR PC,F         ; CALL F
```

This places the actual parameters and return address on the stack and transfers control to the routine "f". The body of "f", in turn, saves registers, creates and deletes locals, etc., and pops the return address off of the stack when it returns. Now, the point of all this is that neither the called routine nor the caller removes the actual parameters - not right away

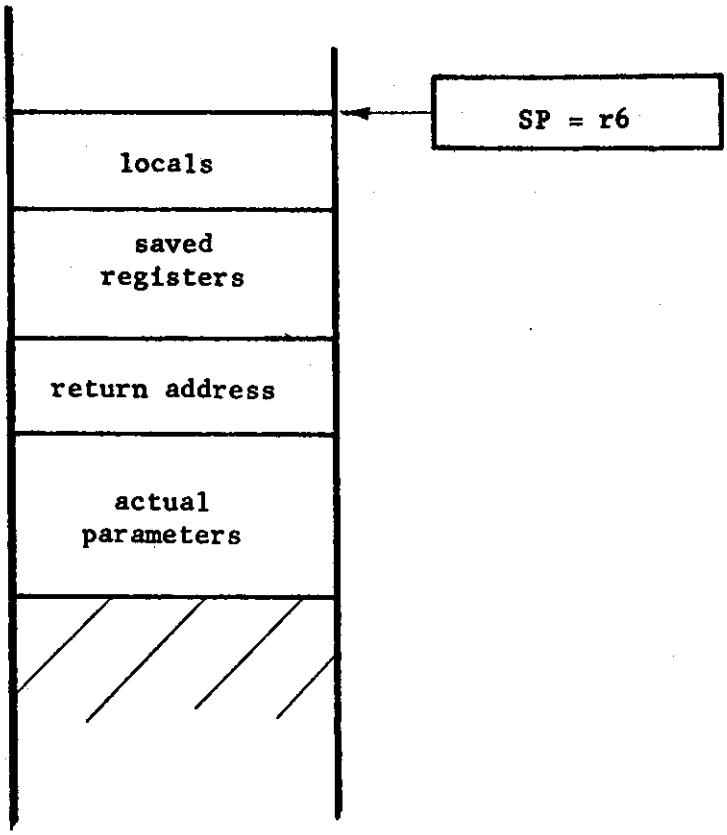


Figure 26. The runtime stack.

---

at least. Rather, the caller leaves these two locations on the stack for possible use for either temporaries or locals. The effect of this strategy is illustrated by the following example:

```
routine r(x)=
    begin local a,b;
    a←f(x); b←g( );
    return .a+.b
    end;
```

The code produced for this routine is:

```
R: MOV  X,-(SP)    ; PUSH X
   JSR  PC,F       ; CALL F
   MOV  R0,@SP     ; STORE VALUE OF F IN A
   JSR  PC,G       ; CALL G
   ADD  (SP)+,R0   ; FORM .A+.B
   RTS
```

In this particular case the local "a" is allocated to the location "created" by the parameter push from "f(.x)." the local "b" is allocated (because of the targeting/preferencing mechanism described earlier) to RØ. Notice that in this case the created local is also deallocated "for free" by the auto-increment addressing in "ADD (SP)+,RØ." The mechanism which achieves this is described later in the section on FINAL.

In order to exploit this optimization the compiler must keep track of the number of "dynamic temporaries" created by pushing parameters and must occassionally generate code to remove some or all of them. In particular, dynamic temporaries must be removed: (1) at the "bottom" of a loop - so that the stack depth will be the same for the next iteration, (2) at the "join" of a forked construct - so that the stack depth will be the same through all paths leading through the join,* (3) whenever the number of such temporaries seems "unreasonably large."

The "dynamic temps" are implemented by simulating the stack at compile time. The node-specific routine for actual parameters "opens" dynamic temps for parameters. These are "closed" again at the points mentioned above, e.g. at the bottom of loops, etc. "Open" and "closed" periods for the dynamic temporaries are represented in a manner to be described below in the section on PACK. Information is passed to the code phase to effect run-time stack adjustment in a field in each node. TLA sets this field, the "dynamic-temp-depth" field, to the depth of the simulated stack at that point in the program. The code phase emits instructions to adjust the stack whenever its simulated stack depth differs from that set in the node.

IV.3.1e Label Assignment

Label assignment has nothing whatsoever to do with the main topic of this section, namely temporary name binding. However, label assignment is performed during TLA and therefore will be discussed here - albeit briefly. Label assignment is another targeting process. Consider, for example, as if-then-else expression of the form

$$\underline{\text{if}}\ (\sim\epsilon_1)\ \underline{\text{and}}\ \epsilon_2\ \underline{\text{then}}\ \epsilon_3\ \underline{\text{else}}\ \epsilon_4$$

The node-specific routine for if passes to the and node two labels - a "true" label which is that of the then part, ϵ_3, and a false label which is that of the else part, ϵ_4. The and node generates no code; but the node-specific routine for and passes the false label to both of its operands: the true label passed to its left operand is that of its right operand, while the true label passed to the right operand is the true label input to the and node. The "~" node-specific routine also does not generate code, it merely inverts the input labels. The effect of label targeting may be seen in the following example:

$$\text{if } .x \le y \text{ and } (.x \ge 1 \ \underline{\text{or}}\ .y \neq 0) \text{ then } x \leftarrow 1 \text{ else } y \leftarrow 1$$

*The number is reduced to the minimum of the number generated along any single path on those paths which generate more than this minimum number.

which produces the following code

```
    CMP X,Y
    BGT L2
    CMP X,#1
    BGE L1
    TST Y
    BEQ L2
L1: MOV #1,X
    BR  L3
L2: MOV #1,Y
L3:
```

IV.3.2 RANK

The function of the RANK phase is simply to order all temporary names so that the PACK phase may process them in order of their importance. At the same time the temporary names are separated into four catagories depending on their individual requirements for binding: (1) specific register,* (2) some register,* (3) a memory (stack) location, and (4) either a register or a memory location. The majority of TN's fall into the fourth catagory.

Ranking is based on both the cost measure and lifetime characterizations described earlier. Specifically, the ranking is based on

$$\frac{C}{(\text{fonlu-fonfu})}$$

where C is the "max" cost measure discussed earlier. The effect of this ranking criterion is to treat TN's whose uses are distributed over a large span as less important than those with similar cost measures but more compact uses.

IV.3.3 PACK

Before proceeding to a description of the pack algorithm itself we must discuss the mechanism by which the availability of various locations, whether registers or in memory, are represented. Specifically, each location, l_i, is represented by a set, L_i; the elements of L_i are temporary names. Initially all L_i are empty. The following functions and predicates are defined on temporary names and then sets:

*Bliss programs may declare that certain variables must be allocated to a register or to a specific register.

predicates

empty(L)	true iff the set L is empty
interset(t_1,t_2)	true iff the lifetimes of the temporary names t_1 and t_2 do not intersect
fits(t,L)	true iff $\forall t_1 \in L(\sim$ intersect(t,t_1));

functions

open(L) =	if empty(L) then L←{z,i} else ERROR where z and i are special "dummy" temporary names that have lieftimes of <0,0,0,0> and <∞,∞,∞,∞> respectively.
close(L,lon,fon) =	L←L ∪ {c} where c is another "dummy" temporary name specially created with a lifetime of <lon,∞,fon,∞>.
reopen(L,lon,fon)=	if empty(L) then open(L) else if ∃ t ∈ L lifetime(t) = <x,∞,y,∞> then L←(L-t)ut$_1$ where lifetime (t_1) = <x,lon,y,fon>
tryfit(t,L) =	if fits (t,L) then (L←Lu{t}; thru) else false

Note: in terms of these predicates, the "stack simulation" alluded to in Section IV.3.1d is actually a stack of these sets. A set is reopened each time a simulated parameter push is done, and one or more sets are closed each time the simulated stack is cut back.

The packing algorithm consists of considering each temporary name and attempting to fit it into various sets. The order of consideration is

1st) those TN's that must be bound to a specific register
2nd) those TN's that may be bound to any register
3rd) those TN's that must be bound to a memory (stack) location
4th) the remaining TN's; these are treated in order of "importance" as defined by the RANK phase.

For those TN's which must be bound to a specific register the register is opened if this has not already been done; then "tryfit" is attempted. If "tryfit" succeeds everything is fine; otherwise a fatal error is signaled.

For those TN's which may be bound to any register, tryfit is attempted on all open registers. If no open register can hold the TN an attempt is made to open another one - if no register can be found a fatal error is signaled.

For those TN's which must be bound to a memory (stack) location "tryfit" is attempted on all the dynamic temporaries created during the TL phase. (The dummy TN's inserted by successive "close" and "reopens" will prevent these TN's from fitting unless the dynamic

temporary is available over its entire lifetime.) If no dynamic temporary is suitable, an attempt is made to fit the TN into an open "static temporary." (The static temporaries are allocated and deallocated at routine entry and exit respectively.) If the TN won't fit into an open static temporary, a new one is opened.

For the remaining TN's, which constitute the majority, either registers or memory may be used. These TN's are considered in order of their importance and for each the following sequence of attempts is tried:

1) try to use the TN's preference, if any.
2) try an open register.
3) if the difference between the maximum and minimum cost measure computed in TLA is sufficiently small it isn't worth opening a new register (which will have to be saved and restored) so try to use currently open dynamic and static locals.
4) try to open another register.
5) if locals were not attempted in step 3, try them now.
6) open a static local.

IV.4. Code

It might appear that after the previous phases have completed code generation would be trivial; after all, redundant computations have been detected and eliminated, (near) optimal execution order has been determined, temporary variables have been carefully allocated to the available registers, etc. It would appear that the only remaining task is to perform an execution-order tree walk and generate the code appropriate to each node (if, indeed, any is required). This is in fact what must be done, but would that it were as easy as this glib statement suggests!

There are few descriptions of code generators in the literature [Low69, Hop69, Gri71] for a very good reason. It falls to the code generator to cope with, and hopefully exploit, the idiosyncracies of the target machine. While not conceptually difficult, an honest attempt to exploit the target machine involves a great deal of special case analysis. Performing that analysis is both tedious to do and to explain, yet it is extremely important. In the final analysis the quality of the local code has a greater impact on both the size and speed of the final program than any other optimization.

Lest we bore the reader (and ourselves) too much, we shall present only two aspects of the code generator, namely generating the code to move a value from one arbitrary field to another arbitrary field and generating the code for the control of an incr loop (the Bliss counterpart of the Algol for or Fortran DO). Neither example will be presented in full detail, nor do we claim that either example, or the two together, is representative of the problems faced by other compilers or even the remainder of this one. The purpose in presenting these examples is merely to illustrate the level of detailed analysis necessary in order to generate good quality local code.

The general strategy of code generation is to perform an execution-order tree walk invoking node-type specific routines. The mechanism used to implement this tree-walk is identical to that used in DELAY and TNBIND. The responsibility of each specific routine is to generate the optimal local code for that node given that the code for its subnodes has already been generated. In the exposition we shall use the following functions, predicates, and conventions:

(1) functions

$\gamma(op,a_1,a_2)$	emit a PDP-11 instruction "op" with addresses a_1 and a_2; a_2 is omitted in the case that "op" is a unary.
α(node)	generate an address of the form used by γ which is the location of the result represented by the node "node."
gl	get a "compiler generated" label which is distinct from all previously generated labels.
pl(lab)	place a label, "lab," at the current point in the instruction stream.
p(node) s(node)	retrieve the "position," p, and "size," s, of the result represented by "node" within the word whose address is given by "α(node)."
ν(opnd)	in the case that "opnd" describes a literal value, get that value; undefined otherwise.
τ(node)	extracts the temporary location (if any) in which the result of "node" will reside at execution time; the composite function $\alpha(\tau(node))$, denoted $\alpha\tau$(node), generates the appropriate address of this temporary.
C(e)	causes the code associated with expression e, if any, to be generated.

(2) predicates

δ(node):	true iff the result represented by "node" is contained in a "destroyable" temporary, i.e., is <u>not</u> either a user-defined variable or a common-sub-expression with remaining uses.
λ(opnd):	true iff the operand is a literal
z(opnd):	true iff the operand is the literal zero, i.e., $\lambda(opnd) \wedge \nu(opnd) = \emptyset$
o(opnd,s):	true iff the operand is a literal all of whose last s bits are one, $\lambda(opnd) \wedge [((\nu(opnd) \wedge (2\uparrow s-1)) - 2\uparrow s-1]$

(3) conventions

(a) It will be convenient to adopt a more compact node/subnode naming convention than is used in the compiler itself. In particular we shall refer to the node itself as e_0 and its subnodes as e_1, e_2, etc. (left-to-right) respectively. We shall, further, extend this convention to the functions and predicates described above. Thus, for example, $\alpha_1 \equiv \alpha(e_1)$, $\nu_2 = \nu(e_2)$, $p_1 = p(e_1)$, etc.

(b) Brackets, { }, are used to enclose instruction to be generated, thus
{ADD x,y; ASH y} ≡ γ(ADD,α(x),α(y)); γ(ASH,αy)

IV.4.1 Generating Data Moves

The problem we wish to consider is that of generating code to move the contents of some arbitrary field in one word to an arbitrary field of another word; in terms of Bliss source code, such moves arise in contexts such as

x<5,3> ← .y<9,2>

In terms of the tree shown in Figure 27, we are concerned with generating code for e_0. No code will have been generated for either e_1 or e_2, but DELAY will set fields in e_1 and e_2 such that

α_1 is the address of x, α_2 is the address of y
$p_1 = 5$, $s_1 = 3$, $p_2 = 9$, $s_2 = 2$
etc.

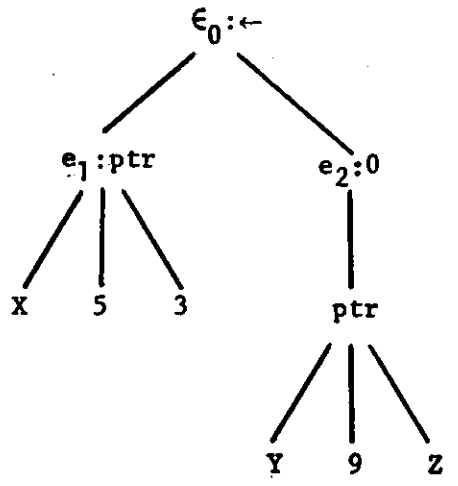


Figure 27. Tree for x<5,3> ← .y<9,2> .

Since the PDP-11 does not contain instructions for general sub-field moves, or even for field extraction and insertion, the effect must be achieved by an appropriate sequence of shifting, masking, etc. The variety of sequences possible for doing this is expanded significantly by the presence of the byte manipulation instructions and by the lack of instructions for multiple position shifts. Therefore, before proceeding to the description of the move generation itself we shall introduce several functions which handle the special cases of field isolation and positioning.

(1) clearfield (node, posn, size):

This function clears (sets to zero) the subfield of the operand described by "node." The following cases are treated:

(a) full word (posn = 0, size = 16): use the CLR instruction.
(b) byte (posn = 0 or 8, size = 8): use the CLRB instruction.*
(c) otherwise: use the BIC instruction with an appropriate literal mask as the source operand.**

The rationale for these cases arises from the observation that since CLR and CLRB are unary operators and BIC is a binary the latter will occupy one additional word of code space and cause one additional memory reference at execution time.

(2) isolate (node, posn, size):

This function is complementary to "clearfield" in that it clears (sets to zero) everything but the specified subfield; it considers the same cases.

(3) dial (node, fp, fs, tp, ts):

This function moves a subfield of the result represented by "node" to another subfield in the same word.*** The source subfield is described by a position and size, fp and fs respectively, and the destination subfield is similarly described by tp and ts. No special cases are treated by "dial" per se, but rather by another function "shift" described below. The action of "dial" is simply:

shift(node,fp,tp,min(fs,ts)); isolate(node,tp,min(fs,ts))

---

*Here, and elsewhere in this exposition, we will ignore a level of complexity introduced by the fact that byte operations do not operate in quite the same way when the destination is a register as they do when the destination is memory.

**Here we choose to ignore another level of complexity arising from the restriction that word instructions must specify an even word address.

***Here we are ignoring the complexity introduced by the possibility that the result represented by "node" is not "destroyable" - i.e., it is a common-sub-expression whose value will be needed again (or it is a user-defined variable).

(4) shift (node, fp, tp, size):

This function shifts a subfield of a specified size from one position, fp, to another, tp. The lack of multiple bit shift or rotate instructions together with the presence of the byte operations, especially SWAB, make this a fairly complex decision based on the distance of the shift, the size of the field, and the relative and absolute positions of the source and destination subfields. The following special cases are recognized:

(a) $-5 \leq tp-fp \leq 4$: The straightforward option of generating a series of shifts in the intended direction is best.

(b) $8 \leq abs(tp-fp) \leq 12$: Best here is to generate a SWAB followed by a series of shifts in the intended direction.

(c) $abs(tp-fp) > 12$: Here we would like to use the rotate instructions to rotate the field out one end of the word and in the other. We must however, remember that all the rotate operations move the carry bit around as if it were part of the word or byte being rotated; one more instruction must be generated than we would otherwise generate.

(d) $5 \leq abs(tp-fp) \leq 7$: We wish to generate a SWAB and a series of shifts in the direction opposite to the one intended. Complications are introduced here by the possibility that the source or destination field overlaps both bytes of the word, in which case we should either wait to generate the SWAB until enough shifts have been generated to move the field wholly within one byte, or if the field is too large for one byte, we should use rotates rather than shifts and worry a little harder than in part (c) about the carry bit. The exact algorithm we use is:

(1) if size > 8, generate a SWAB, then a byte rotate (RORB or ROLB) to accommodate the carry bit by putting it in a harmless place, and then a series of full word rotates

(2) if size ≤ 8, generate a series of shifts (in the prescribed direction), inserting a SWAB as soon as the field is wholly within one byte (which condition must occur sometime, or the intended shift would move the field past a word boundary).

In all the above discussion it should be noted that, strictly from considerations of programming convenience, the routine SHIFT in the compiler does not distinguish situations when it could either generate a shift (ASR or ASL) or a rotate (ROR or ROL) from situations when it must generate a rotate, and simply always generates rotate instructions. A few examples of generated code might clarify the above description; suppose A is the machine address generated for node:

fp	tp	Size	code
2	5	9	ROL A; ROL A; ROL A (three shifts)
2	1	3	SWAB A; ROL A; ROL A; ROL A

1	14	2	ROR A; ROR A; ROR A; ROR A
1	7	9	SWAB A; RORB A; ROR A; ROR A
7	1	9	SWAB A; ROLB A+1; ROL A; ROL A
1	7	8	ROR A; SWAB A; ROR A
2	8	8	ROR A; ROR A; ROR A; SWAB A

We shall introduce one more function before proceeding to a description of the move operation. This function tests a single bit in a word and generates a branch to a specified label if that bit is set to zero.

(5) bittest (node, bit, lab):

This function generates a test of the specified bit in the result specified by a node and will generate a branch to the specified label which will be taken if the bit is zero at execution time. The following cases are treated:

(a) bit = 15: {TST α_0; BPL lab}
(b) bit = 7: {TSTB α_0; BPL lab}
(c) bit = $\emptyset \wedge \delta$(node): {ROR α_0; BHIS lab}
(d) bit = $14 \wedge \delta$(node): {ASL α_0; BGT lab}
(e) bit = $6 \wedge \delta$(node): {ASLB α_0; BGT lab}
(f) otherwise: {BIT #1↑bit,α_0; BEQ lab}

As with several of the special cases treated earlier, the rationale is that the unary operators (TST, ROR, etc.) require one fewer words of code and one fewer memory reference at execution time then the default BIT sequence.

Given the code generation functions described above we can now describe the operation of generating code for moving a field of one word to a field of another. In analyzing the kind of code one would like to produce for any particular such move it becomes apparent that various subcases arise in connection with the position and size of the subfields of each of the words involved. These cases are:

full word:	p = $\emptyset$, s = 16
byte:	p = $\emptyset$ or p = 8, s = 8
bit:	p = anything, s = 1
other:	all other fields

The various cases of movement from one of these types of fields are treated separately by functions m_0-m_7 as specified by the following table:

	destination field type			
	word	byte	bit	other
word	m_0	m_7	m_1	m_4
byte	m_2	m_1	m_7	m_4
bit	m_6	m_5	m_7	m_7
other	m_3	m_4	m_7	m_4

The functions are explained below:

m_0 (full-word to full word case):
  $z(e_2)$: {CLR α_1}
  otherwise: {MOV α_2,α_1}

m_1 (full word or byte to byte case):*
  $z(e_2)$: {CLRB α_1}
  otherwise: {MOVB α_2,α_1}

m_2 (byte to full word cases)*:
  {CLR α_1; BISB α_2,α_1}

m_3 (arbitrary field to full word):*
  {MOV α_2,α_1}; shift($e_1,p_2,\emptyset,s_2$); isolate($e_1,\emptyset,s_2$);

m_4 (any field to arbitrary field):*
  $z(e_2)$: clearfield(e_1,p_1,s_1)
  $o(e_2)$: {BIS #(2↑s_1-1)*2↑p_1,α_1}
  otherwise: clearfield(e_1,p_1,s_1); dial(e_2,p_1,s_1,p_2,s_2); {BIS α_2,α_1}

m_5 (single bit to byte):*
  clearfield(e_1,p_1,8); l←gl; bittest(e_2,p_2,l); {INCB α_1}; pl(l)

m_6 (single bit to full word):*
  clearfield($e_1,\emptyset$,16); l←gl; bittest(e_2,p_2,l); {INC α_1}; pl(l)

m_7 (move to/from arbitrary one bit field):*
  $\lambda(e_2) \wedge [(e_2) \bmod 2=1]$: {BIS #2↑$p_2\alpha,\alpha_1$}
  $\lambda(e_2) \wedge [(e_2) \bmod 2=\emptyset]$: {BIC #2↑$p_2,\alpha_1$}
  otherwise: clearfield(e_1,p_1,1); l←gl; bittest(e_2,p_2,l);
  if p_1-∅ or p_1=8 then {INCB α_1} else {BIS #2↑p_2,α_1};
  pl(l)

Some examples of various assignment expressions and the code they generate are given in Figure 28 .

*Once again we are ignoring the complexities which arise due to: (1) the non-symmetric properties of byte operators with respect to registers and memory, and (2) word boundary problems with word instructions.

Source	Object (all constants are octal)
X .Y;	MOV @#Y,X
X<0,8>.Y<8,8>;	MOVB @#Y+1,@#X
X<0,8> 0;	CLRB @#X
x<5,3> 7;	BISB #340,@#X
X<13,2>.Y<3,2>;	MOVB @#Y,R5 SWAB R5 ROL R5 ROL R5 BIC #117777,R5 BIC #60000,@#X BIS R5,@#X
X<3,1>.Y<4,1>;	BICB #10,@#X BITB #20,@#Y BEQ L$1 BIS #10,@#X L$1:
X<0,8>.Y<5,1>;	CLRB @#X BITB #40,@#X BEQ L$2 INCB @#X L$2:
X<14,2>.Y<0,2>;	MOVB @#Y,R5 ROR R5 ROR R5 ROR R5 BIC #37777,R5 BIC #140000,@#X BIS R5,@#X

Figure 28. Code produced for some assignment expressions.

IV.4.2 Generating the incr Loop Code

In this section we wish to examine the generation of the code for the incr expression - which is the Bliss counterpart of the Algol for statement or Fortran DO statement. The form of the incr expression is:

incr i from e_1 to e_2 by e_3 do e_4

The semantics of Bliss specify that the counter variable, i, is to be initialized to the value of e_1. Then the expression e_4 is to be repeatedly evaluated (with i being increased by the value of e_3 after each such execution of e_4) so long as the value of i is not greater than e_2. The values of e_2 and e_3 are computed only once - before e_4 is evaluated the first time. Associated with an incr expression are also the χ and ρ sets described in Section IV.1.5; the expressions in these sets must be evaluated prior to entering the loop body.

The "general case" code for the incr expression is shown below. The opportunities for optimizing this code arise when certain of these are constants.

evaluate e_1 and store into i
evaluate e_2
evaluate e_3
evaluate χ,ρ expressions
branch to L2
L1: evaluate e_4 (the body)
increment i by e_3
L2: compare i and e_2 and conditional branch to L1

Consider, for example, the following loop:

```
INCR I FROM .X TO .Y BY .Z do (.Y+3)←.(Y+3)+1;
```

This loop produces the following code:

```
        MOV     @#X,R4      Load e1 into i (R4 in this case)
        MOV     @#Y,R5      Evaluate the X set (".Y+3" in this case)
        ADD     #3,R5
        BR      L3          Branch to the end condition test
L2:     INC     @R5         The body!
        ADD     @#Z,R4      Increment the loop index
L3:     CMP     R4,@#Y      Compare and branch back if not done
        BLE     L2
```

(In this particular case FLO noted that both the end condition, ".Y", and the step size, ".Z", were invarient in the loop; therefore neither was explicitly loaded into a temporary. This observation is purely the result of FLO and will not be discussed here.)

Now consider a similar loop with all of e_1, e_2, and e_3 being constants:

```
INCR I FROM -10 TO -1 BY 1 DO (.Y+3)←.(.Y+3)+1;
```

The code produced for this loop is:

```
        MOV    #177766,R4
        MOV    @#Y,R5
        ADD    #3,R5
L$2:    INC    @R5
        INC    R4
        BLT    L$2
```

Two things have happened. First since both the beginning and ending values of the index variable are known at compile time, the branch to the conditional test preceding the first iteration is eliminated. Second, since the data manipulation instructions set the condition codes the comparison itself can often be avoided. In this case the identity $i \leq -1 \equiv i < 0$ was exploited since the condition code settings resulting from the "INC R4" may then be used directly.

In order to fully explain the code generation for the incr expression it is convenient to introduce a few more functions and predicates:

(1) notecode: returns a unique identification of the most recent instruction generated; used in conjunction with "anycodesince" below.

(2) anycodesince(c): the parameter must be the value returned by some previous call on "notecode." The value of "anycodesince" will be true iff any code has been generated since the corresponding call on notecode.

(3) genmove(e_1,e_2): generates code to move the result of the expression e_1 to the location named by the expression e_2; this is precisely the function described in the previous section.

The algorithm for generating the code for the incr expression is then:

```
l1←gl; l2←gl; C(e1); genmove(e1,i); C(e2); c←notecode; C(e3); C(χ); C(ρ);
if z(e2) ∧ λ(e1) ∧ anycodesince(c) then {TST i; BR l2} else
if   λ(e1) ∨ λ(e2) ∨ [ (e1) >  (e2)] then {BR l2};
pl(l1); C(e4);
{ADD α3,i};
pl(l2);
if z(e2) then {BLE l1} else
    if  λ(e2) ∧  (e2)=-1 then {BLT l1} else {CMP α2,i; BLE l1}
```

One of the cases covered by this algorithm was not covered by the previous discussion. Consider the case of a loop whose end condition is zero but the starting point is not known at compile time; for example

INCR I FROM .X TO 0 BY 1 DO <body>

In this case the action of incrementing the index variable at the end of the loop will set the condition codes and therefore an explicit "TST" instruction is not needed when this path is followed. However, before the first time the <body> is executed it is necessary to insure the condition codes are set properly. Therefore the following code is generated:

```
        MOV     X, R5
        <any χ and ρ code>
        TST     R5*
        BR      L$2
L$1:    <body>
        INC     R5
L$2:    BLE     L$1
```

IV.5. FINAL

Ah, finally...

FINAL is the last of the several phases of the compiler and has two responsibilities: (1) performing a collection of ad hoc optimizations on the code produced by CODE, and (2) preparation of the final listing and relocatable code. The latter aspect is irrelevant to the purpose of this paper and will not be discussed. The optimizations performed at this stage all result from looking only at the object code and make no use of the more global context available in the tree representation; these optimizations include those which have been called "peephole" optimizations [McK65].

It is reasonable to ask why further optimizations should be necessary at this stage, especially such ad hoc, low level ones as will be described, given that reasonable care has been taken in the earlier phases. There are three reasons. First, some information isn't known until the code is generated. We shall see several examples of this, but, for example, the code generator will happily create a branch instruction whose target is another branch. It is better, of course, for the first instruction to branch directly to its ultimate target. Second, adjacent instructions in the code stream may have been generated from widely distant points in the tree; what seemed like good local code at each of those points may not look so clever when juxtaposed. Third, the operation of the FINAL optimizations themselves create new adjacency relations which may then admit reapplication of one or more of its other optimizations.

---

*This instruction is generated only if any code intervened between the "MOV x,R5" and this point.

IV.5.1 The Final Data Structure

FINAL receives its input from the CODE module in the form of a doubly linked list. This data structure allows FINAL to add and/or delete instructions anywhere in the code stream and to scan the code in both the forward and reverse directions. The items on this list are of two types: code cells and label cells. Each code cell contains a description of a complete PDP-11 instruction. Each label cell marks a position in the object code addressed by an instruction in one of the code cells. Label cells have a sublist consisting of items called reference cells. There is a unique reference cell for each instruction that refers to a given label and that reference cell contains a back pointer to the instruction. This data structure is illustrated in Figure 29.

IV.5.2 The First Subphase

The first phase of FINAL is a loop that examines each item in the code stream and calls an appropriate routine to perform optimizations for that type of item. If during this loop a change is made to the code stream, a flag is set and the loop will be repeated. Only when a pass is completed in which no changes are made will FINAL progress to its second phase. The discussion below is keyed to the actions taken for the various types of cells encountered during the loop.

Label Cells. The first thing that is done when a label cell is encountered is to delete adjacent labels in the code stream and attach their reference cells to the remaining label cell. If the set of reference cells for a particular label is empty, this label is not needed and is deleted. Otherwise, the instructions named by these reference cells are scanned for unconditional branch instructions that address this label. All possible pairs of these unconditional branches are examined by a "back-over" procedure, called "cross-jumping," which exploits the fact that the code sequences preceding the two branches will merge at the label. The sequence of code cells preceding both branches (any preceding label cells are ignored) are compared and, if identical sequences are found, each code cell in one sequence is replaced by a branch to the corresponding code cell in the other sequence. Later optimization will remove unnecessary branch instructions which may be introduced by this procedure.

Consider, for example, the following example:

if .x then x ← .y + .z else x ← .w + .z

which might produce the following code (before and after this portion of FINAL)

Before Final				After Cross-jumping		
	BIT	#1,X			BIT	#1,X
	BEQ	L1			BEQ	L1
	MOV	Y,X	FINAL		MOV	Y,X
	ADD	Z,X			BR	L3
	BR	L2			BR	L2
L1:	MOV	W,X		L1:	MOV	W,X
	ADD	Z,X		L3:	ADD	Z,X

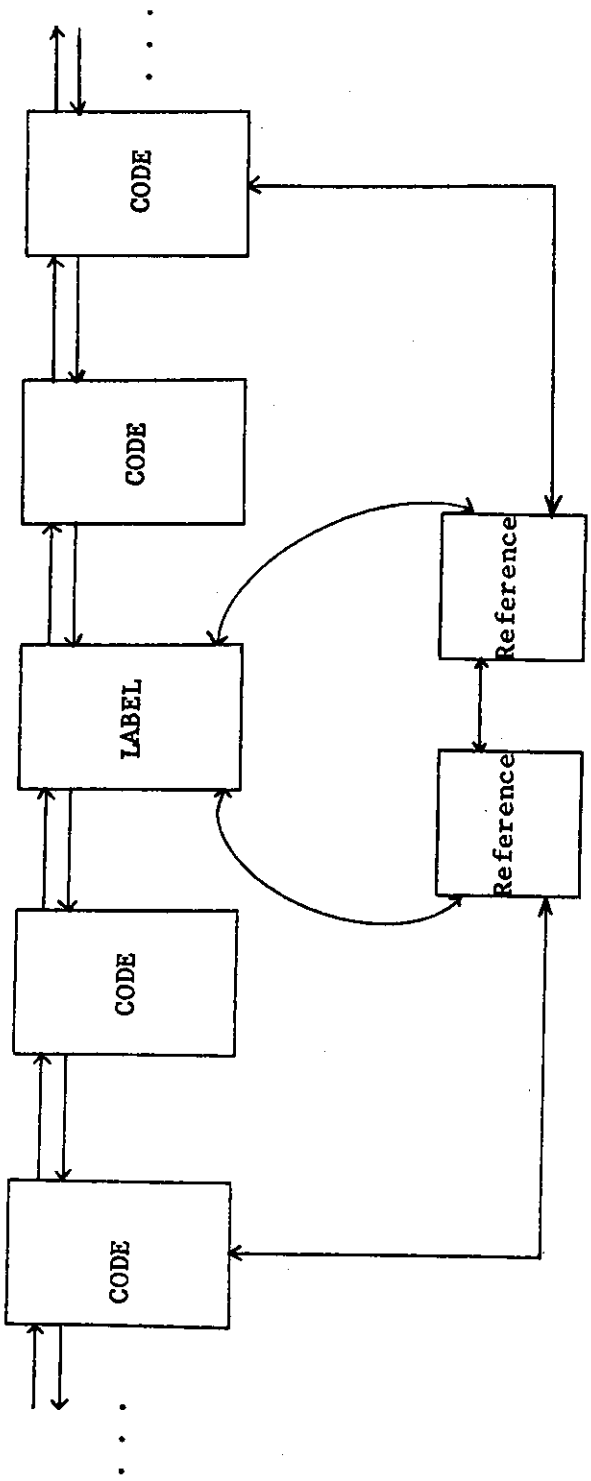


Figure 29. The FINAL data structure.

L2: L2:

This optimization is one of those mentioned in the introduction as being impossible until the object code is produced. It is, to us, surprisingly effective.

Unconditional Branch Instructions. The first thing done when optimizing an unconditional branch is to delete all code cells following it until the next label cell is reached since any such code is unreachable. This will, for example, eliminate the redundant "BR L2" in the previous example. The label referenced by a branch is then compared with the label following the branch. If they are the same, the branch instruction is deleted. Otherwise, a "branch-chaining" procedure is executed. "Branch-chaining" consists of testing for a branch that addresses an internal label which precedes another branch which in turn may address a label which precedes another branch, etc. The entire length of such a chain is searched and each branch in the chain is changed to use the same addressing as the last branch in the chain.

Finally, after the branch-chaining procedure, the cross jumping procedure described above is applied to the code sequences preceding the branch and the label referenced by the branch.

Conditional Branch Instructions. First, the item following the conditional branch is checked to determine whether it is the label addressed by the branch. If this happens, the branch is deleted. Otherwise, this item following the conditional branch is checked to see if it is an unconditional branch immediately followed by the label addressed by the conditional. In this case the unconditional branch is deleted and the sense of the conditional branch is reversed. If this fails to occur, the next code cell (intervening label cells may now be ignored) is found and tested for being an unconditional branch to the same label; if it is, the conditional branch is deleted.

If the conditional branch is not deleted by one of the above criteria, "branch-chaining" is performed on it. Notice that conditional branches may only appear as the first element of a branch chain.

The utility of the optimizations of both conditional and unconditional branches arises from two sources: (1) the effect of the optimizations themselves sometimes create the conditions in question, and (2) since the optimizations are desirable due to (1), and therefore present, the CODE module can be somewhat simpler if it need not worry about creating such conditions. To see the effect of these optimizations taken together, consider another (fake) example:

if .x then x ← .y + .z + .w else (y ← .q; w ← .r; x ← .y + .z + .w)

Notice that ".x + .y + .z" is not a common-sub-expression or eligible for α-motion due to the assignments in the else branch; it is eligible for ω-motion - a fact which we wish to ignore here for expository purposes. The code generated would then be:

```
BIT     #1,X
BEQ     L1
MOV     Y,R0
```

```
        ADD     Z,R0
        ADD     W,R0
        MOV     R0,X
        BR      L2
L1:     MOV     Q,Y
        MOV     R,W
        MOV     Y,R0
        ADD     Z,R0
        ADD     W,R0
        MOV     R0,X
L2:
```

Although the optimizations will not happpen in the order described below, for purposes of exposition we will describe them as though each is performed as a separate pass. First cross jumping will produce

```
        BIT      1,X
        BEQ     L1
        BR      L6
        BR      L5
        BR      L4
        BR      L3
        BR      L2
L1:     MOV     Q,Y
        MOV     R,W
L6:     MOV     Y,R0
L5:     ADD     Z,R0
L4:     ADD     W,R0
L3:     MOV     R0,X
L2:
```

Eliminating unreachable instructions (the branches) and then removing unreferenced labels will yield the following:

```
        BIT     #1,X
        BEQ     L1
        BR      L6
L1:     MOV     Q,Y
        MOV     R,W
L6:     MOV     Y,R0
        ADD     Z,R0
        ADD     W,R0
        MOV     R0,X
```

Finally, the "condition reversal" applied to the "BEQ" followed by removal of the unreferenced label will produce:

```
        BIT     #1,X
        BNE     L6
```

```
        MOV   Q,Y
        MOV   R,W
L6:     MOV   Y,RØ
        ADD   Z,RØ
        ADD   W,RØ
        MOV   RØ,X
```

Although the example above is somewhat contrived it does serve to illustrate that the action of some of the FINAL optimizations create situations where others become operable.

Test and Compare Instructions. Test and compare instructions do not modify their operands, but are executed solely for their effect of setting condition codes. Any test or compare instruction where the next code cell is neither a conditional branch, nor the start of a branch-chain sequence leading to a conditional branch may be deleted, since all other instructions will change the condition codes.

Literal Operands. Some machine operations, such as "add," "subtract," "or," and "nand," often have literal constants as one operand. When such an instruction is located, the literal constant may be one of a set of special cases. For example, an "add" of a literal zero can be deleted (provided the next instruction is not a conditional branch) or a "move" of a literal zero can be made into the shorter instruction "clear."

If the literal is not one of the special cases, the items following in the code stream are examined until a label, a branch instruction or an instruction accessing the data operated on by the original literal is found. If the item on which this scan terminates is another literal operation of the same type then this literal is modified appropriately and the original instruction is deleted. Literal adds and subtracts to registers scan backwards as well as forwards in order to attempt to make use of the PDP-11's auto increment and auto decrement addressing modes.

Addressing Optimization. The PDP-11 hardware recognizes eight different addressing modes. However, FINAL recognizes 38 different addressing constructs. These constructs are differentiated by such considerations as whether the register being used for indexing is the program counter, push down stack pointer or some other general purpose register and whether or not the indexing base contains a reference to an own variable location. Some of these constructs are more desirable than others because they execute faster, or take less space, or are more amenable to future optimization. FINAL attempts to use what it considers the best form of addressing.

In particular, whenever FINAL is asked to optimize a "move" or "clear" instruction it considers the source and destination operands as containing identical data. The code stream following the instruction is scanned. Scanning stops if a label cell, unconditional branch instruction or an instruction that might modify either operand or might destroy the addressing of either operand (e.g., a register used in an index operation) is encountered. If either of the operands appears in a scanned instruction, that operand is replaced by the other if the resulting form is "better."

Redundant Store Optimization. One other optimization also occurs with "move" and

"clear" instructions. In this case, a scan is made backwards from the data store until a branch or an instruction that might use the destination location is found. During this scan any instruction whose sole purpose is to modify the destination location of the data move is a "redundant store" and is deleted.

IV.5.3 The Second Subphase

The above completes the description of FINAL's first phase. When no more first phase optimizations are possible, then the second phase begins; this phase makes a single pass over the code stream and performs various bookkeeping operations, as well as modifying certain instructions to equivalent, simpler forms. (For example, "increment" m,nstructions are formed from "ADD #1,x.")

The major bookkeeping function is to compute the length and relative location of each instruction. Unfortunately this is not yet completely possible. The PDP-11 branch instruction takes two forms; there is a one word branch instruction, BR, which can only branch to instructions located within 128 words of itself, and there is a two word branch instruction, JMP, which can transfer to any location in memory. Conditional branches are the short form; hence if the destination is more than 128 words away, the conditional must be fabricated from the reverse conditional and a JMP. This requires a total of three words. As a result, the second phase of FINAL can only determine the minimum and maximum length of each instruction; it then uses these to compute a minimum and maximum relative position for each instruction and label.

IV.5.4 The Third Subphase

FINAL now enters its third phase in order to resolve the length and relative position of each instruction. It does this by scanning for the instructions whose minimum and maximum lengths are different. By taking differences between the minimum and maximum locations of the instruction and the label it references, it is possible to compute the minimum and maximum possible distances between the instruction and the label. If the maximum possible distance is less than 128, then the short form branch is formed and the maximum length is set the same as the minimum length. If the minimum possible distance is greater than or equal to 128, then the long form branch is generated and minimum length is set to the maximum length. As FINAL scans for those instructions whose minimum and maximum lengths differ, it also updates the minimum and maximum locations of all the instructions and labels. The code is repeatedly scanned until a pass is completed during which no changes are made. The correct location of each instruction and label is its minimum location and all instructions of unresolved length will take their short form and minimum length. It follows that this is safe from the fact all unresolved instructions between any still unresolved branch-label pair will take their short form.

V. CONCLUSION

As stated in the introduction, our goal in this paper has been to describe the various optimizations used in a specific, real compiler and to show how these optimizations interact to produce high quality code. We hope that a secondary effect of the exposition has been to pinpoint areas in which further research is needed. While no single aspect of the compiler has been described at the level of detail of the implementation itself, we expect that most knowledgable systems programmers can infer the implementation from the material which is presented.

In this conclusion, rather than recap what has already been said explicitly, we would like to focus on what has only been said implicitly -- namely, those areas of compiler optimization which lack adequate formalization and/or for which complete algorithms are not currently known. Our experience has shown that, in some intuitive sense, the size, complexity, and bug-proneness of each of the compiler phases is inversely proportional to the degree of formalization to which that area has been subjected.

In listing areas for further work there is a great temptation to order them by their "importance" in terms of their effect on the quality of the resultant code. We have resisted that temptation. The fact is that they are all important. Given our initial goal of producing code which is better than that produced by human programmers, a change of even a few percent is significant. Moreover, even a single extra instruction in a deeply nested loop may have an effect on execution time completely disproportionate with its static effect on program size. Hence, the following list is simply ordered by the phases in the Bliss/11 compiler within which the problem arises.

(1) DELAY

(a) Evaluation order: As noted in the text, Bliss/11 uses a heuristic extension of a well-known algorithm for choosing the evaluation order of expressions in order to minimize the number of in-use registers. A complete algorithm, or at least a better heuristic, is badly needed which works in the presence of globally redundant expressions (including those involved in code-motion optimizations).

(b) Desirable vs. feasible optimizations: This is currently done on a completely ad hoc basis, and requires a very detailed knowledge of the target machine. It is, in fact, an instance of a more general problem of deciding when to commit to an optimization.

(c) Exploiting addressing hardware: The mechanisms used in DELAY, and discussed in Section IV.2.4, are critical to producing object code which exploits the target machine. We believe that these ideas can be generalized and expressed in a machine-independent way.

(2) TNBIND

(a) Targeting and preferencing are ideas which we have not seen in the literature before. They have a significant effect in Bliss/11, and we believe they can be generalized in a machine-independent way.

(b) Floating temporaries: Bliss/11, like most compilers, dedicates a location (usually a register) to hold the value of a temporary (or user variable) over its entire lifetime. There are many contexts in which this is non-optimal, but we know of no computationally reasonable formulation which avoids this commitment.

(c) Lifetime characterization: The lon/fon characterization of the lifetime of a temporary is relatively inexpensive, but incomplete. A scheme is needed which is both computationally reasonable and (more) complete.

(3) CODE

When all is said and done, the quality of highly local code is absolutely essential to the overall performance of the object program produced by a compiler. No matter how clever the other phases of the compiler, if the local code is sloppy, the compiler will be a disaster. Because of the highly machine specific nature of CODE it is possibly inevitable that the construction of this phase will be relatively ad hoc. However, it is not the ad hoc nature of the implementation of this phase which disturbs us, but rather the generation of the special cases which should be considered by the implementation. The set of cases treated in the text for general data moves, for example, represents a substantial investment of intellectual effort, has been modified many times as new cases were uncovered, and still has no guarantee of being exhaustive. It seems reasonable to us that the generation of these cases can be mechanized.

(4) FINAL

All of FINAL is ad hoc, yet its effect is significant. It is very hard to determine whether FINAL would be so important if more complete algorithms existed at earlier stages of the compilation process; nevertheless, the actual implementation is highly regular -- suggesting to the authors that its effects could be both formalized and extended. It is also difficult to avoid drawing analogies between certain aspects of FLO and FINAL (notably cross-jumping); we would like to believe that the analogy is more than accidental.

We would like to conclude by presenting our view on the importance of efficiency. It is fashionable in some circles, especially some academic ones, to depreciate the importance of efficiency. The argument goes: "It is programming time, not execution time, that matters. What difference does it make if the program runs one minute or two?" This argument is often used, for example, to encourage the use of some of the newer, often less efficient, programming languages.

It is difficult to argue with this. In fact we don't for our own programs. Our time is much more important than the machine's. However, the other fellow's program (yours) is an entirely different matter! Every second that his program executes ours can't. If his program is inefficient, we are the ones who suffer, not him. The problem of efficiency is not one of how long it takes to run a program, but rather one of obtaining the maximum benefit from a finite resource.

The reason that compiler optimization is important is that programmer efficiency and execution efficiency need not be a choice we must make. Optimization is a technological device to let us have our cake and eat it, too -- to have both convenient programming and efficient programs.

APPENDIX A

PRIMER ON THE PDP-11

In order to understand and appreciate some of the optimizations described in this paper some understanding of the target machine, the PDP-11, is necessary. This appendix is not a definitive description of the PDP-11. It is, rather, a brief description of some of the addressing features and enough of the instruction set to allow the reader to understand the examples; in particular the i/o and interrupt structure of the machine are totally ignored.

Basic Properties

The PDP-11 is a "mini-computer" with a word size of 16 bits; memory is byte (8 bits) addressable. Because a memory address is constrained to exist in a single word, directly addressable memory is limited to 2^{16} (64K) bytes or 2^{15} (32K) words. The processor contains eight "general purpose" registers (r0-r7) two of which are assigned specific uses (r6 = SP and is used as a stack pointer for subroutine calls and r7 = PC and is the program counter). A ninth register in the processor, called the "program status register", PS, contains a set of condition code bits which are set by the action of data manipulation instructions. The condition code bits are

Z = 1 iff the result of the operation was zero
N = 1 iff the result of the operation was negative
C = 1 iff the result of the operation produced a carry out of its most significant bit
V = 1 iff the result of the operation produced an arithmetic overflow

The data manipulation instructions of the PDP-11 are classified as either unary (monadic) or binary (diadic). In both cases the instructions may be used as zero address (i.e., stack oriented with reverse polish operators) or single address (conventional general register) architecture. Binary (diadic) instructions may, in addition, be viewed as two address (memory-to-memory).

Addressing Structure

In this section we shall use one of the binary operators to illustrate the addressing options, namely "MOV src, dst" which moves a word from the location specified by the source address, "src", to the location specified by the destination address, "dst".

The group of binary instructions are encoded in a 16-bit word as shown below.

OPERATOR	SOURCE	DESTINATION
(4 bits)	(6 bits)	(6 bits)

The source and destination fields are coded identically as shown below. As should be obvious, these six bit fields are not adequate to hold memory addresses; rather some encodings of these fields specify that words following the instruction contain operands or addresses.

mode d reg

The "reg" field always specifies one of the eight processor registers. The "d" bit, when set, specifies one level of indirection or "deferral". The meanings of the mode field values are given below for d = Ø (one level of indirection should be added to these descriptions for d = 1).

m=Ø	register mode	The operand to be used in the instruction is contained in the register specified by the "reg" field.
m=1	auto-increment mode	The register specified by the "reg" field contains the address of the operand to be used. After being used the specified register is incremented. (The register is incremented by either one or two depending upon the nature of the instruction - "byte" instructions increment the register by one and "word" instructions increment the register by two.)
m=2	auto-decrement mode	The register specified by the "reg" field specifies the address of the operand to be used. However, before being used the value of the register is automatically decremented by either one or two as in the m=1 case.)
m=3	index mode	The register specified by the "reg" field is used as an index register and is added to the 16-bit word following the instruction in order to obtain the address of the operand to be used in the instruction. The PC is automatically incremented (by 2) to point beyond the index word.

The following notations, used by the PDP-11 assembler, will be used to illustrate the wide range of addressing constructs possible on the PDP-11:*

r	register mode (m=Ø, d=Ø)
@r	indirect register mode (m=Ø, d=1)
(r)+	auto-increment mode (m=1, d=Ø)
@(r)+	indirect auto-increment mode
-(r)	auto-decrement mode
@-(r)	indirect auto decrement mode

*In this table the symbol "r" is used to denote any of the eight processor registers and "x" denotes a constant.

x(r) indexed mode
@x(r) indirect indexed mode

Three special cases involving the use of the program counter as the register named in an address specification are worthy of special note:

(1) literal operands: The notation "#x" denotes +(PC) addressing where the literal, "x", has been placed in the word following the instruction; the effect is to use the literal "x" itself as the operand.

(2) relative addressing: The location of a data item may be expressed as a displacement of the current instruction. This is the preferred method of accessing data. Hence a symbol, e.g., "A", is generally interpreted as "α(PC)" addressing where α is some appropriate constant. Deferred relative addressing, "@A", is, of course, possible.

(3) absolute addressing: In a few cases relative addressing, as in (2), is undesirable; in these cases absolute addressing, denoted @#A, is possible. Absolute addressing is actually "@+(PC)" mode.

The following list illustrates some of the variety possible with the single MOV instruction given the variety of addressing modes; in the center column of this list, memory is abbreviated to "m", register to "r", and literal to "l". The lengths (in words) of the various forms are also given.

Instruction	Effect	Length
MOV r0,r1	r-to-r	(1 word)
MOV r0,X	r-to-m	(2)
MOV X,r0	m-to-r	(2)
MOV 5(r0),r1	m (indexed)-to-r	(2)
MOV r0,5(r1)	r-to-m (indexed)	(2)
MOV X,Y	m-to-m	(3)
MOV 3(r1),-1(r0)	m (indexed)-to-m (indexed)	(3)
MOV #17,r3	l-to-r	(2)
MOV #17,X	l-to-m	(3)
MOV #17,5(r2)	l-to-m (indexed)	(3)
MOV r1,-(SP)	push r contents onto stack*	(1)
MOV (SP)+,r0	pop stack contents into r*	(1)
MOV #6,-(SP)	push l value onto the stack*	(2)
MOV X,-(SP)	push m contents onto the stack*	(2)
MOV (SP)+,X	pop stack contents into m*	(2)

*Note that by convention the PDP-11 stack grows "down" toward lower addresses, thus, for example, a "push" operation involves decrementing the stack r.

Of course, many more combinations are possible; these are sufficient to illustrate the flexibility of the addressing mechanism and to illustrate the various optimizations.

The Instruction Set

In this section we shall enumerate the PDP-11 instructions needed for the examples in the text. There are four instruction groups in the PDP-11 of interest to us, each with its own instruction format.

BinaryGroup:	op source	destination
UnaryGroup:	op	destination
BranchGroup:	op	offset
Jump Group:	op reg, or	destination

Binary Group

The "binary group" instructions specify two addresses - a source address and a destination address. In general these instructions perform an operation between source and destination operands and place the result in the destination. The relevant instructions are:

MOV*	move source to destination
ADD	add source to destination
SUB	subtract source from destination
CMP*	compare source and destination; set condition codes; no operands modified
BIS*	"bit set", form inclusive or of source and destination in destination
BIC*	"bit clear", form nand of source and destination (src $\wedge$ dst) in destination
BIT*	"bit test", form and of source and destination; set condition codes; no operands modified

Unary Group

The "unary group" instructions specify a single address - the destination. In general these instructions perform an operation on the destination operand and return it to the destination location. The relevant instructions are:

CLR**	clear destination word to zero
COM**	complement the destination operand
NEG**	negate the destination operand
INC**	increment the destination operand by one
DEC**	decrement the destination operand by one
TST**	compare the destination operand to zero; set the condition codes

*These instructions also have a "byte form", e.g., MOVB which moves a byte (8 bits) from source to destination.

**These instructions also have a "byte form", e.g., CLRB which clears a single byte.

ROR**	rotate right
ROL	rotate left
ASR	arithmetic shift right
ASL	arithmetic shift left
SWAB	swap the two bytes of the destination (word) operand

It should be noted that all the shift instructions (ROR, ROL, ASR, ASL) are single bit shifts of the specified type; also, the rotate instructions involve 17 (not 16) bits - the "carry bit", C, of the condition codes participates in these shifts.

Branch Group Instructions

These instructions branch conditionally on the setting of the condition codes. The instructions do not specify the address of the branch, but rather, an offset from their own location (i.e., the value of PC). Since the offset is eight bits and denotes a word displacement, branches are limited to -128 or +127 words from their own location. The relevant instructions are:

BR	branch (unconditional)
BEQ	branch on equal (to zero)
BNE	branch on not equal
BMI	branch on minus
BPL	branch on plus
BLT	branch on "less than"
BGE	branch on "greater or equal"
BLE	branch on "less than or equal"
BGT	branch on greater than
BHI	branch on higher
BLOS	branch on lower or same
BLO	branch on lower
BHIS	branch on higher or same

Two things should be noted about these instructions. First, all relationals (e.g., BLE) are comparisons against zero. However, since the compare instruction, CMP, performs an implicit subtract, a sequence such as "CMP A,B; BLE LABEL" corresponds to a branch on the relative magnitudes of A and B. Second, the branches BHI, BLO, etc. are termed "unsigned comparisons" since they treat the items as unsigned (positive) 16-bit numbers.

Jump Group Instructions

This group includes three instructions: (1) the unconditional jump, JMP, (2) the subroutine call instruction (JSR), and (3) the subroutine return instruction, RTS.

The JMP and JSR instructions specify normal addressing and, thus, are not constrained as to the distance which may be jumped as the branch instructions are.

The JSR and RTS instructions are more involved than is necessary for our purposes. The only forms used are "JSR PC,SUBR" and "RTS PC" which together have the effect of

transferring control to a subroutine leaving the return address on the stack, and of returning from a subroutine.

JMP	jump
JSR	jump to subroutine
RTS	return from subroutine

APPENDIX B

A SHORT PRIMER ON BLISS

The Bliss language is used throughout this paper for examples. This appendix has been included for those who may not be familiar with the language. This primer is not a complete description of Bliss - it is, rather, an introduction to those aspects of the language used in the examples. More complete information on the language may be found in [Wul71, Wul72a].

Bliss is a descendent of Algol 60 in that it has block structure, a similar expression format - including operator precedence, similar control constructs, and potentially recursive procedures (called "routines"). It differs from Algol in its interpretation of identifiers, the omission of a goto, and the fact that it is expression-oriented rather than statement-oriented. Because of its resemblance to Algol, this primer will concentrate on the differences.

INTERPRETATION OF NAMES

A Bliss program operates with and on a number of storage "segments." A segment consists of a fixed and finite number of "words." A word may be "named;" the value of a name is called a "pointer" to the word. Identifiers are bound to names by declarations. Thus the value of an instance of an identifier, say x, is not the value of the word named by x, but rather a pointer to x. This interpretation requires a "contents of" operator for which the symbol "." has been chosen.

This context independent interpretation of identifiers as pointers is maintained consistently throughout the language. It is the operators of Bliss which place an interpretation on the value of an expression. So, for example, the assignment operator "←" interprets its right hand operand as a value which is to be stored in the word pointed to by the value of the left hand operand. As a result the effect of the Algol assignment statement "A:=B+C" is identical to the Bliss assignment "A←.B+.C". This interpretation of names also allows the computation of pointers in Bliss so that the effect of the assignment "(A+3)←.(A+5)" is to store the value of the fifth location past A into the third location past A.

CONTROL STRUCTURES

Bliss is a block-structured, go-to-less, "expression language." That is, every executable construct, including those which manifest control, is an expression and computes a value. Expressions may be concatenated with semicolons to form expression sequences. An expression sequence is evaluated in strictly left-to-right order and its value is that of its last (rightmost) component expression. A pair of symbols, begin and end, or left and right parentheses, may be used to embrace such an expression sequence to form a simple expression. A block is a special case of the construction which contains declarations.

Other than expressions and functions, control mechanisms in Bliss fall into four classes: conditional, selection, looping, and leave. The conditional expression

$$\underline{\text{if}}\ e_0\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$$

is defined to have the value e_1 just in the case that e_0 evaluates to true (a one in the low order bit) and e_2 otherwise. The abbreviated form "if e_0 then e_1" is considered to be "if e_0 then e_1 else 0."

The conditional expression provides two-way branching. The case and select expressions provide n-way branching:

$$\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{set}}\ e_1;\ e_2;\ \ldots;\ e_n\ \underline{\text{tes}}$$

$$\underline{\text{select}}\ e_0\ \underline{\text{of}}\ \underline{\text{nset}}\ e_1:\ e_2;\ \ldots;\ e_{2n-1}:\ e_{2n}\ \underline{\text{tesn}}$$

The case expression is executed as follows: (1) the expression e_0 is evaluated, (2) the value of e_0 is used as an index to choose one of the e_j's ($0 \le j \le n$). The value of e_0 is constrained to lie in the range $0 \le e_0 \le n$. The value of the case expression is e_i ($i = e_0$). The select expression is similar to the case expression except that e_0 is used in conjunction with the e_{2j-1}'s to choose among the e_{2j}'s. The execution of the select expression above is described by the following equivalent Bliss expression.

(T←e; V←-1; if e_1 eql .T then V←e_2; ... if e_{2n-1} eql .T then V←e_{2n}; .V)

Hence the value of the select expression is that of the last e_{2j} to be executed or -1 if none of them is executed.

The loop expressions imply repeated execution (possibly zero times) of an expression until a specific condition is satisfied. There are several forms, some of which are:

$$\underline{\text{do}}\ e_0\ \underline{\text{while}}\ e_1$$

$$\underline{\text{incr}}\ \langle\text{id}\rangle\ \underline{\text{from}}\ e_0\ \underline{\text{to}}\ e_1\ \underline{\text{by}}\ e_2\ \underline{\text{do}}\ e_3$$

In the first form the expression e_0 is repeated so long as e_1 satisfies the Bliss definition of true. The second form is similar to the "step ... until" construct of Algol, except (1) the control variable, <id>, is implicitly declared to be local to the incr expression, and e_0, e_1, and e_2 are evaluated only once (before the evaluation of the loop body, e_3). Except for the possibility of a leave expression within e_3 (see below) the value of a loop expression is uniformly taken to be -1.

The control mechanisms described above are either similar to, or slight generalizations of constructs in many other languages. Of themselves they do not remove the inconveniences generated by removing the goto. Another mechanism is desirable -- the leave mechanism. A leave is a highly structured form of forward branch which is constrained to terminate coincidentally with some control environment in which the leave is nested. The general form is:

leave <label> with <expression>

where <label> must be attached to a control environment within which the leave expression is nested. A leave expression causes control to immediately exit from a specified control environment. The <expression> defines the value of the environment.

Finally, functions are defined and called in Bliss in a manner similar to that in Algol, except that there are no specifications and all parameters are implicitly call-by-value. The value of a function is the value of the expression forming its body.

APPENDIX C

A COMPLETE EXAMPLE

In this Appendix we will attempt to present a complete example, that is, to take a short Bliss program and to show the effects of some of the various optimizations in the order in which they are applied. There are inherent difficulties in this attempt: in order to keep the example to tolerable length the program must be short; in order to illustrate several optimizations it must be fairly rich. The result of these two considerations is a contrived program. We must also lie a bit in the presentation in order to avoid tedious detail.

The example we shall consider is the simple program shown below:

```
begin
external f,g;
structure bytearray[i,j] = 1i*j](.bytearray+(.i-1)*j-1+.j)<0,8>;
own bytearray a[8,8];

decr i from 8 to 1 do
    decr j from 8 to 1 do
        if .i eql .j then a[.i,.j]←0 else
        if .i gtr .j then a[.i,j]←f(.i+.j) else a[.i,.j]←g(.i+.j);
end;
```

If you care, and it certainly isn't essential to an understanding of the following material, the effect of this program is to initialize a two-dimensional, Fortran-style (1 origin indexing) array. The upper triangular elements are initialized so that $a_{i,j} = f(i+j)$, the lower triangular elements are initialized so that $a_{i,j} = g(i+j)$, and the diagonal elements are set to zero. Thus the resulting matrix is shown in Figure C-1.

Since we shall be primarily concerned with the tree representation of the program it may be simpler to think in terms of the program as it would appear after the structure accesses have been expanded and defaults supplied, namely:

```
begin
external f,g;
own a[64];

decr i from 8 to 1 by 1 do
    decr j from 8 to 1 by 1 do
        if .i eql .j then (a+(.i-1)*8-1+.j)<0,8>←0 else
        if .i gtr .j
            then (a+(.i-1)*8-1+.j)<0,8>←f(.i+.j)
            else (a+(.i-1)*8-1+.j)<0,8>←g(.i*.j);
end;
```

	1	2	3	4	5	6	7	8
1	0	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	f(9)
2	g(3)	0	f(5)	f(6)	f(7)	f(8)	f(9)	f(10)
3	g(4)	g(5)	0	f(7)	f(8)	f(9)	f(10)	f(11)
4	g(5)	g(6)	g(7)	0	f(9)	f(10)	f(11)	f(12)
5	g(6)	g(7)	g(8)	g(9)	0	f(11)	f(12)	f(13)
6	g(7)	g(8)	g(9)	g(10)	g(11)	0	f(13)	f(14)
7	g(8)	g(9)	g(10)	g(11)	g(12)	g(13)	0	g(15)
8	g(9)	g(10)	g(11)	g(12)	g(13)	g(14)	g(15)	0

Figure C-1. Array produced by sample program.

The final code produced for this example is given in Figure C-2; the purpose of this appendix is to demonstrate how this code is generated.

Figure C-3 shows the form of display we shall use to illustrate the tree representation of this program. Indentation is used to denote levels in the tree; vertical lines connect subnodes of a given node. The numbers along the left edge of the diagram identify the nodes. The symbol "φ" is used to denote the identical subtrees formed by the structure accesses. Subsequent figures will show various types of information in the tree nodes; no one figure contains all the information in order to avoid cluttering up the diagrams with too much detail.

Figures C-4a through C-4d illustrate the tree after LEXSYNFLO but before delay. In particular, the CSTHREAD, CSPARENT, etc. fields have been attached to those nodes for which they are interesting. (A field is "interesting" here if it doesn"t name itself or contain a null value.) In particular notice that

1. The subtree corresponding to "a+(.i-1)*8-1" has been recognized as constant in the inner loop (that is, the node corresponding to this subtree is in the χ-list attached to the inner loop).

```
        MOV     #10,R1      ; initialize outer loop index in R1
L1:     MOV     R1,R5
        ASL     R5
        ASL     R5
        ASL     R5
        ADD     #A-9,R5     ; form (A + (.i-1)*8-1) in R5
L3:     MOV     #10,R3      ; initialize inner loop index in R3
L5:     MOV     R5,R4
        ADD     R3,R4       ; form entire address (A + (.i-1)*8-1+.j) in R4
        CMP     R1,R3
        BNE     L8          ; branch if .i ≠ .j
        CLRB    @R4         ; zero the diagonal element
        BR      L9
L8:     MOV     R1,R2
        ADD     R3,R2       ; form (.i+.j) outside the if expression
        CMP     R1,R3
        BLE     L13         ; branch if .i ≤ .j
        MOV     R2,-(SP)    ; push the argument and call "f"
        JSR     PC,F
        BR      L15         ; branch to end of conditional
L13:    MOV     R2,-(SP)    ; push the argument and call "g"
        JSR     PC,G
L15:    MOVB    R0,@R4      ; perform assignment for either branch
        TST     (SP)+       ; restore stack state at end of loop
L9:     DEC     R3          ; decrement inner loop index
        BGT     L5          ; loop if ≥ 1
        DEC     R1          ; decrement outer loop index
        BGT     L1          ; loop if ≥ 1
```

Figure C-2. Final version of code with annotations.

2. The subtree corresponding to ".i+.j", which is a routine call parameter on both forks of the last if expression, has been recognized as a candidate for -motion and has been attached to the α-list of that if node.

In this simple case, no other common subexpressions or interesting feasible optimizations were detected by FLO.

Figure C-5 illustrates the subtree for the expression "(a+(.i-1)*8-1+.j)" after DELAY but before TNBIND (most of the interesting effects of DELAY occur in this subtree). Note the following obvious changes.

(1) In i(4) the multiplication by eight has been changed to a shift by three.

(2) In i(3) the evaluation order of the operands has been reversed (by reversing the subnodes themselves). This will reduce the number of registers needed to compute "a+(.i-1)↑3".

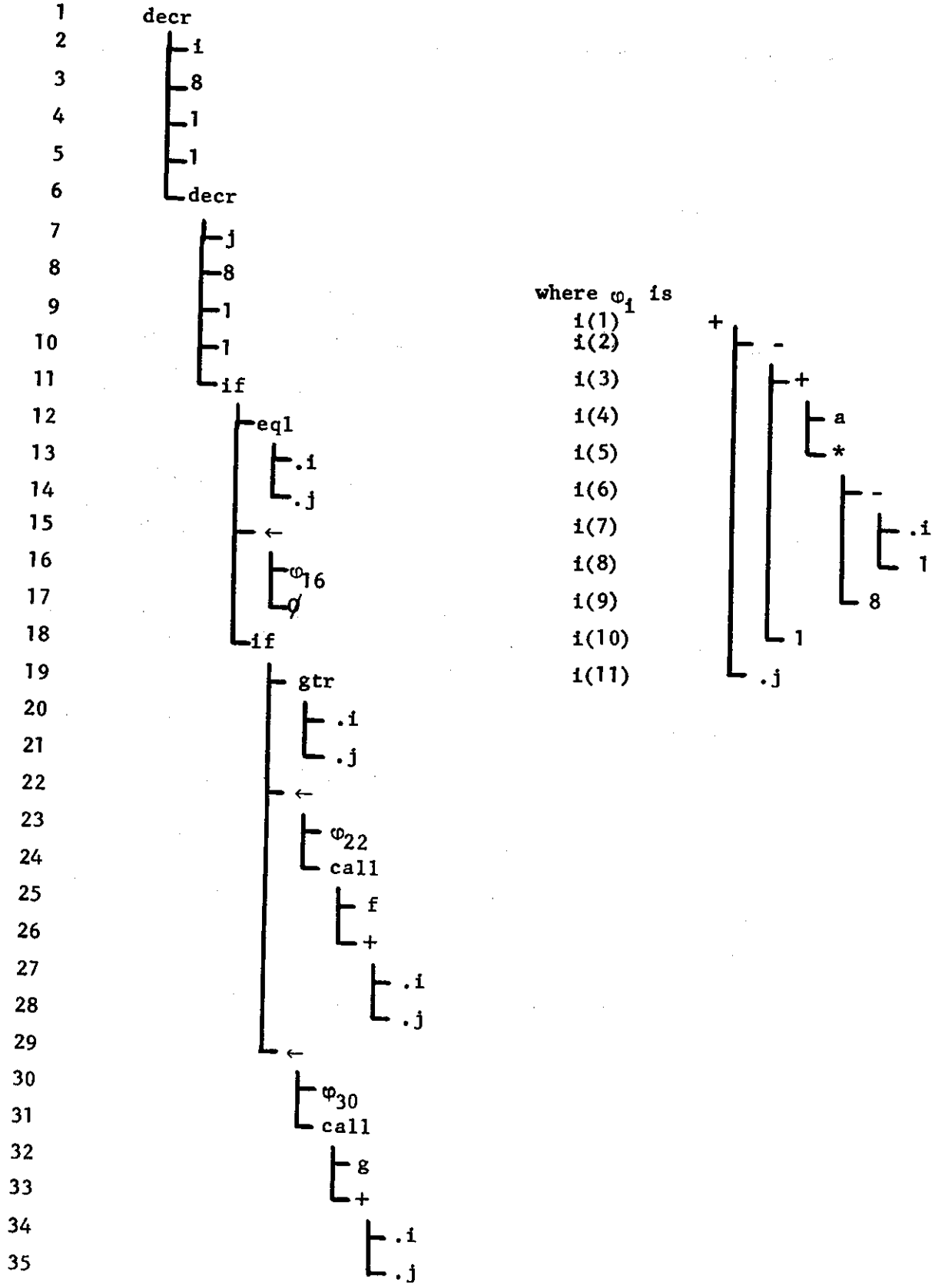


Figure C-3. Tree representation of the example.

Less obvious, perhaps, is the fact that the field settings shown in C-5 have effectively converted the expression

```
a+(.i-1)*8-1
into
(a-9)+.i*3
```

Figure C-6 shows (partially) the temp name assignments and lon/fon values resulting from TLA (the lon/fon of "uninteresting" nodes have been deleted to emphasize those of greater importance). Figure C-7 illustrates the rectangular regions in the lon/fon space which characterize the "lifetime" of the various temporary names. In this particular case the lifetimes all overlap and hence the diagram is a bit uninteresting -- all the temporary names are assigned to different registers. Packing is also rather uninteresting since all the temporaries may be bound to registers.

Figure C-8 illustrates the code as generated by CODE. Several things should be noted:

1. Since the loop bounds are fixed, no code has been omitted to check them on the first time through.

2. The loop termination conditions, e.g. ".i < 1", have been modified to compare against zero; this is always more efficient on the PDP-11, but especially so when the condition codes have been implicitly set by an arithmetic operation.

3. Stores of zero have been performed by the more efficient CLRB instruction.

Comparison of Figures C-2 and C-8 shows the effect of FINAL; in particular note:

1. The sense of some conditionals has been reversed to save a BR instruction.

2. Operations involving constant arithmetic have been modified. In particular, "ADD #2,SP" has been modified to "TST (SP)+" and "SUB #1,R" has been changed to "DEC R" in two places.

3. Cross-jumping has eliminated one "MOVB R0,@R4" instruction.

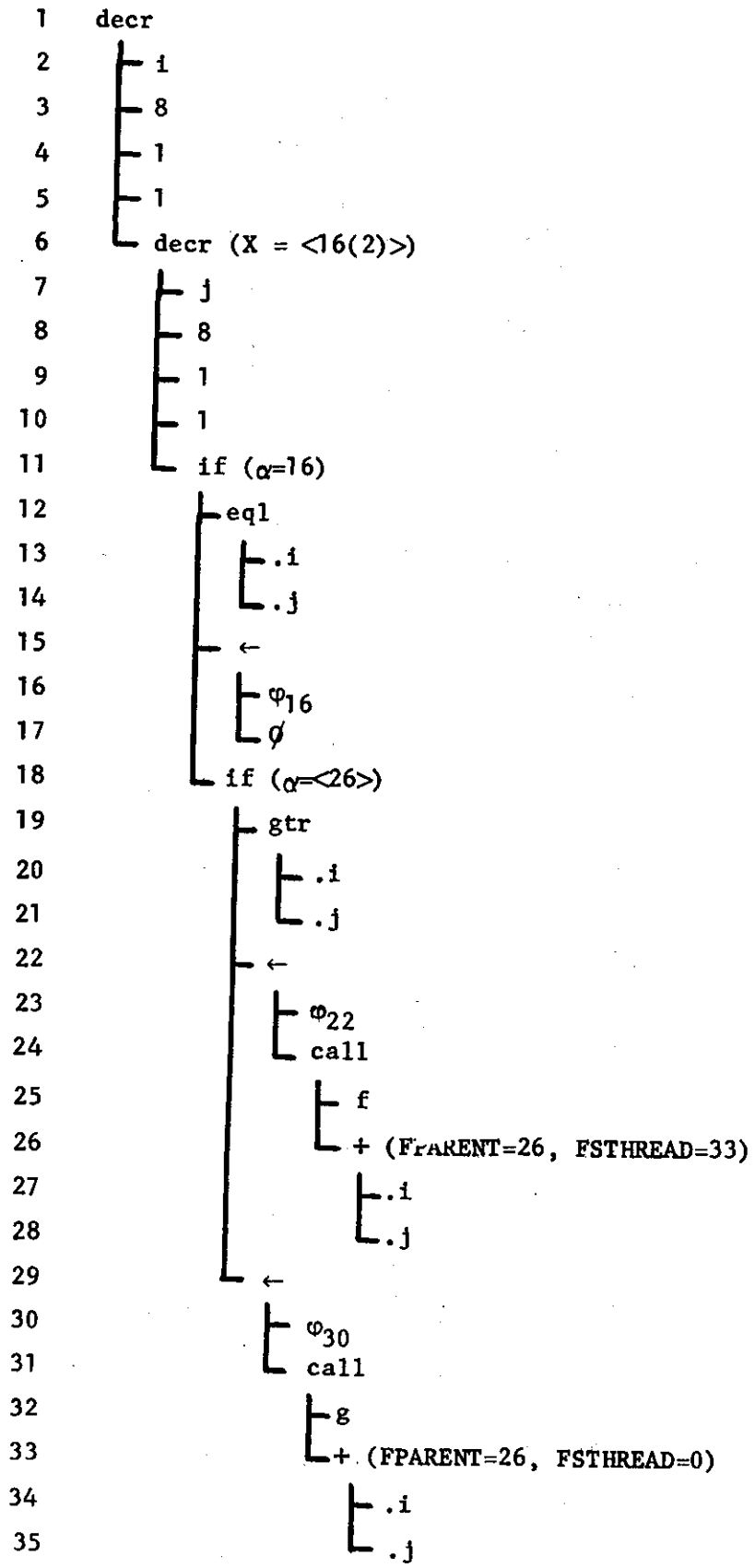

1 decr
2 i
3 8
4 1
5 1
6 decr (X = <16(2)>)
7 j
8 8
9 1
10 1
11 if (α=16)
12 eql
13 .i
14 .j
15 ←
16 φ16
17 α
18 if (α=<26>)
19 gtr
20 .i
21 .j
22 ←
23 φ22
24 call
25 f
26 + (FPARENT=26, FSTHREAD=33)
27 .i
28 .j
29 ←
30 φ30
31 call
32 g
33 + (FPARENT=26, FSTHREAD=0)
34 .i
35 .j

Figure C-4a.

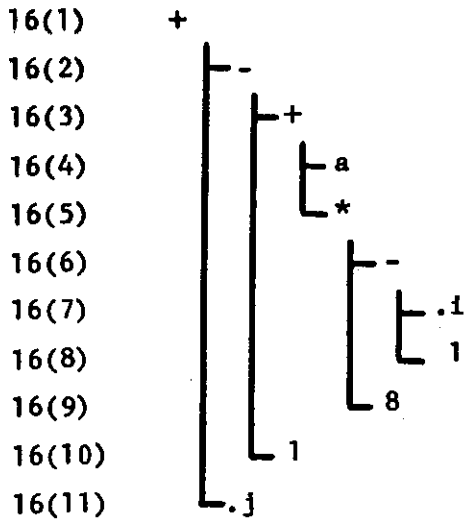


Figure C-4b.

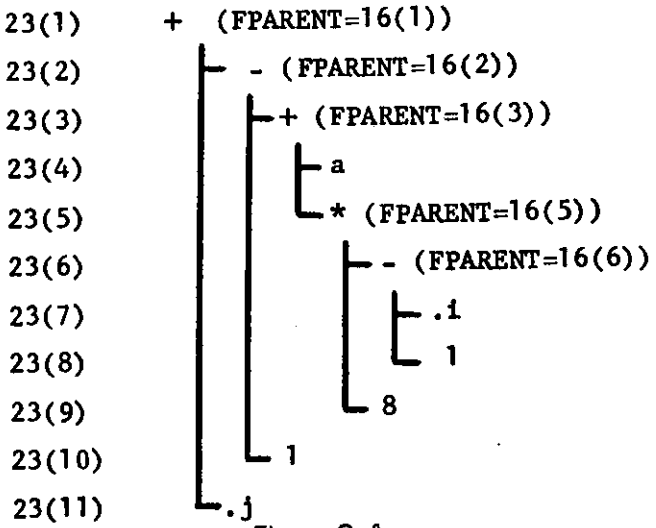


Figure C-4c.

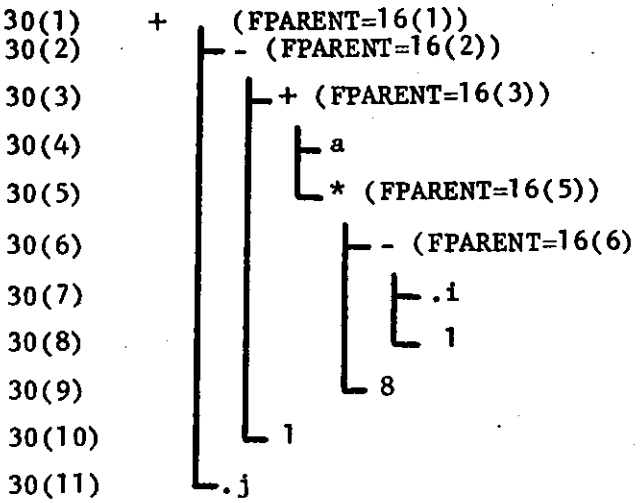


Figure C-4d.

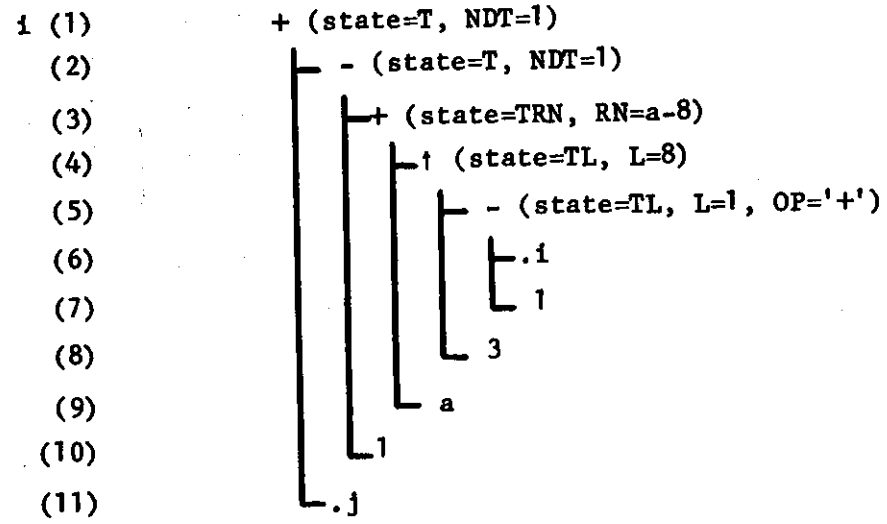

i (1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
+ (state=T, NDT=1)
- (state=T, NDT=1)
+ (state=TRN, RN=a-8)
↑ (state=TL, L=8)
- (state=TL, L=1, OP='+')
.i
1
3
a
1
.j

Figure C-5.

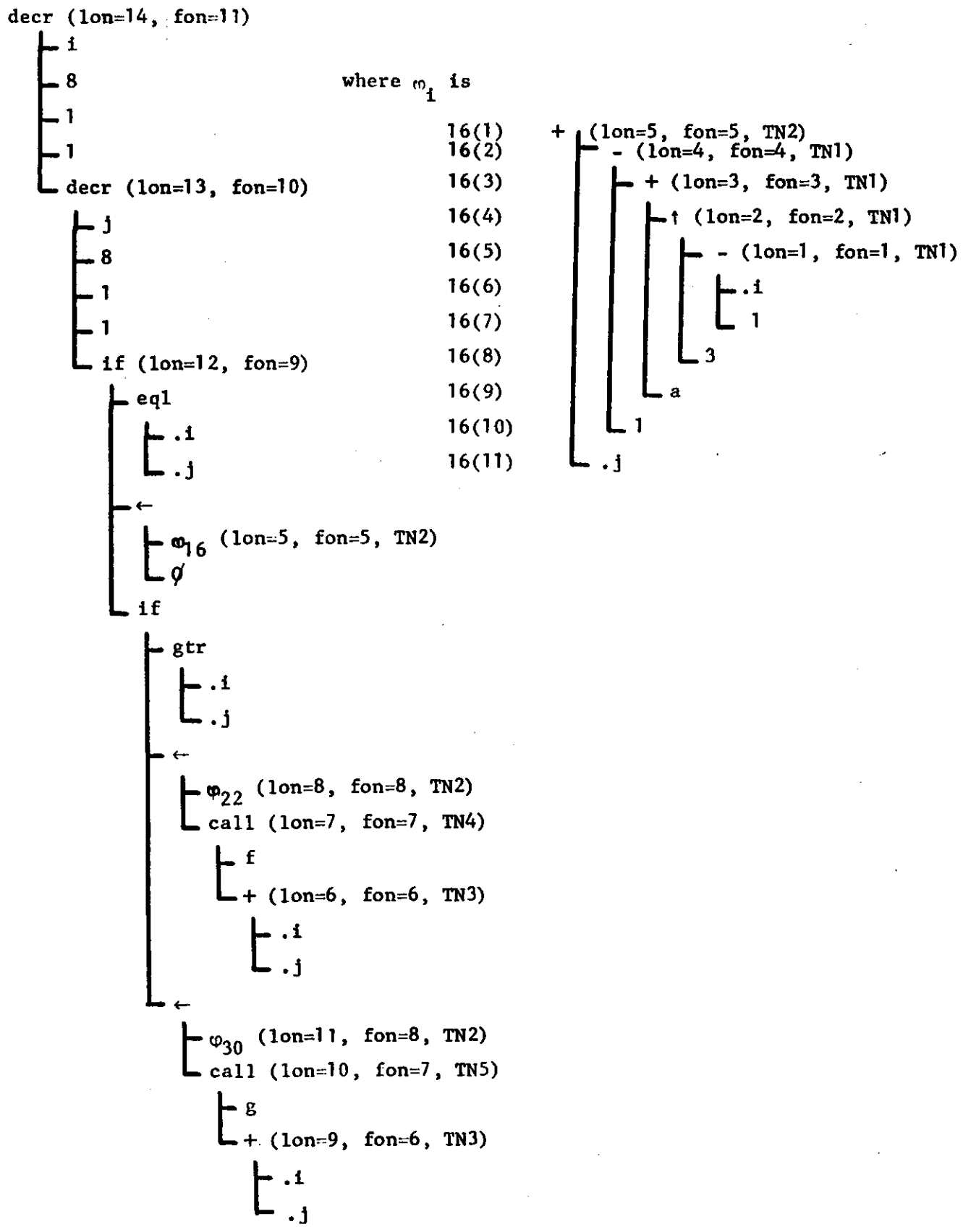

decr (lon=14, fon=11)
i
8
1
1
decr (lon=13, fon=10)
j
8
1
1
if (lon=12, fon=9)
eql
.i
.j
←
φ16 (lon=5, fon=5, TN2)
0
if
gtr
.i
.j
←
φ22 (lon=8, fon=8, TN2)
call (lon=7, fon=7, TN4)
f
+ (lon=6, fon=6, TN3)
.i
.j
←
φ30 (lon=11, fon=8, TN2)
call (lon=10, fon=7, TN5)
g
+ (lon=9, fon=6, TN3)
.i
.j
where φi is
16(1) + (lon=5, fon=5, TN2)
16(2) - (lon=4, fon=4, TN1)
16(3) + (lon=3, fon=3, TN1)
16(4) ↑ (lon=2, fon=2, TN1)
16(5) - (lon=1, fon=1, TN1)
16(6) .i
16(7) 1
16(8) 3
16(9) a
16(10) 1
16(11) .j

Figure C-6.

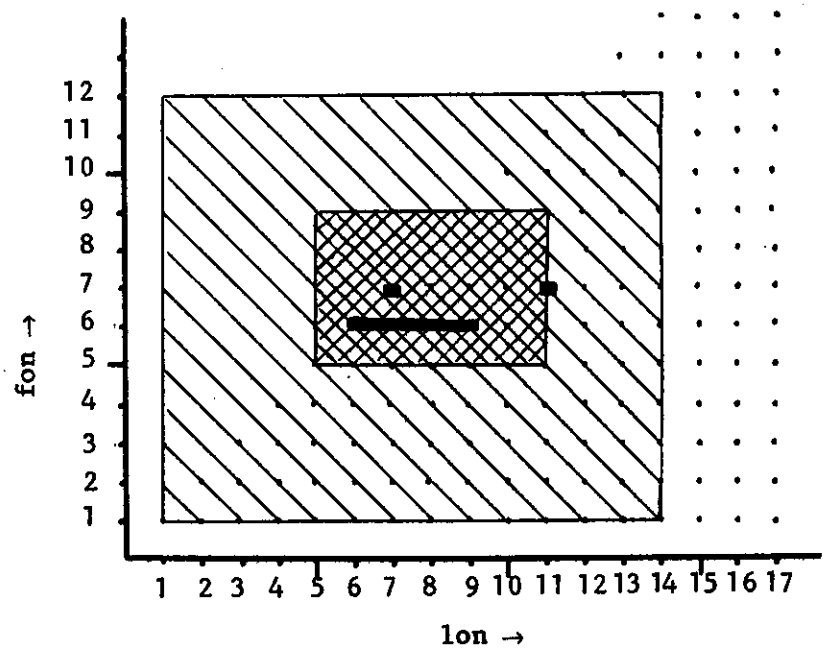


Figure C-7. lon/fon space for the example.

```
        MOV     #10,R1
L1:     MOV     R1,R5
        ASL     R5
        ASL     R5
        ASL     R5
        ADD     #A-9,R5
L3:     MOV     #10,R3
L5:     MOV     R5,R4
        ADD     R3,R4
        CMP     R3,R1
        BEQ     L6
        BR      L8
L6:     CLRB    @R4
        BR      L9
        MOV     R1,R2
        ADD     R3,R2
        CMP     R1,R3
        BGT     L10
        BR      L13
L10:    MOV     R2,-(SP)
        JSR     PC,F
        MOVB    R0,@R4
        BR      L16
L13:    MOV     R2,-(SP)
        JSR     PC,G
        MOVB    R0,@R4
L16:    ADD     #2,SP
        SUB     #1,R3
        BGT     L5
        SUB     #1,R3
        BGT     L1
```

Figure C-8. Code produced by CODE.

BIBLIOGRAPHY

[Bea72] Beatty, James C., "An Axiomatic Approach to Code Optimization for Expressions," **JACM 19**,4 (October 1972), 613-640.

[Fra70] Frailey, Dennis, "Expression Optimization Using Unary Complement Operators," **Proceedings of the Symposium on Compiler Optimization, SIGPLAN 5**,7 (July 1970).

[Hop69] Hopgood, F. R. A., **Compiling Techniques**, American Elsevier, New York, 1969.

[Ges72] Geschke, Charles M., "Global Program Optimizations," Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1972.

[Gri71] Gries, David, **Compiler Construction for Digital Computers**, John Wiley and Sons, New York, 1971.

[Low69] Lowery, E. S. and C. W. Medlock, "Object Code Optimization," **CACM 12**,1 (January 1969), 13-22.

[McK65] McKeeman, W., "Peephole Optimization," **CACM 8**,7 (July 1965), 443-444.

[Nak67] Nakata, Ikuo, "On compiling Algorithms for Arithmetic Expressions," **CACM 10**,8 (August 1967), 492-94.

[Red69] Redziejowski, R. R., "On Arithmetic Expressions and Trees," **CACM 12**,2 (February 1969), 81-84.

[Set70] Sethi, R. and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," **JACM 17**,4 (October 1970), 715-728.

[Wul71] Wulf, W. A., D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," **CACM 14**,12 (December 1971), 780-790.

[Wul72a] Wulf, W. A., et al., **BLISS-11 Programmer's Manual**, Digital Equipment Corporation, Maynard, Mass., 1972.

[Wul72b] Wulf, W. A., "A Case Against the goto," **Proceedings of the ACM National Conference**, Boston, August 1972.