OPTIMIZER DESIGN USING TRANSFORMATIONAL ATTRIBUTE GRAMMARS APPLIED TO INTERMEDIATE LOW-LEVEL TREES

by

STEPHEN V. YOUNG

B. S., Kansas State University, 1985

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

[illegible]

Major Professor

Table of Contents

List of Figures

Acknowledgements

I wish to thank Dr. Thomas Pittman for his suggestion of this topic. His patient assistance was invaluable in the completion of this work.

Secondly, I wish to thank my major advisor, Dr. Virgil Wallentine and my committee members, Dr. David Schmidt and Dr. Austin Melton for their support of this thesis.

I would also like to thank my friends for their help and encouragement. A sincere thanks goes to Scott Hammond and Steve Culp for proof reading this thesis. Their many suggestions for its improvement were greatly appreciated.

Lastly, and most of all, I thank my parents, Leonard and Wilma Young for their love and many prayers. Without their support and consistent encouragement during my graduate studies, I could never have finished.

Chapter One
Introduction

1.1. Optimization

Optimization can be defined as any attempt to improve the efficiency of an algorithm from the time it is derived to the time it is converted into a particular machine code. Although there is rarely a guarantee that resulting code produced by an optimization is the best possible, any special algorithm which performs transformations to code in an effort to improve its efficiency is called an optimizer.

1.2. Purpose of Optimization

The goals of optimization are the reduction of execution time and the reduction of code and data space. These two goals often conflict. For example, in an attempt to reduce execution time, a procedure call can be replaced with the procedure body (back substitution). Thus, the time it takes to call the procedure is eliminated. As a result however, the amount of code has been increased. This conflict is not always the case and usually some compromise is possible so that a reduction in the amount of code also results in a reduction in execution time as well.

1.3. Efficient Optimizer Design Considerations

This work is not concerned with the efficiency of different optimizations, but rather with the efficiency of the design of optimizers and the appropriateness of the intermediate language representation on which the optimizations must operate.

The representation of a program is largely constrained by the semantics of the source language and the peculiarities of the target machine. Often, the program goes through several different representations as different optimizations are applied (as in the Bliss-11 compiler [56]). This tends to complicate the task of properly ordering the optimization phases and consequently, the potential advantages of re-using optimization phases are lost. A uniform representation which is the same language for the duration of the optimization phases would preserve modularity and the ability to reapply optimization phases.

The most common representations are forms of "three-address code" (usually quadruples) as presented in several compiler texts [3] [7] [34]. A more useful representation (at least for machine-independent optimizations) is a program tree. Most of the tree languages that have been developed are no more than simple abstract-syntax trees which lack the ability to represent the target machine code (TCOL, Tree Computer Oriented Language developed for the Bliss-11 compiler and later adopted for the Production Quality Compiler Compiler (PQCC) [56]). A more useful form of tree is presented in chapter three of this thesis.

In the next chapter the standard intermediate representations of quads and trees are defined, the concepts of attributed grammars and translational grammars are defined, and the application of grammars to optimization is reviewed. Chapter three defines Intermediate Low-level Trees (ILTs) and Transformational Attribute Grammars (TAGs), which were proposed as a way of doing optimization [39]. In chapter four the optimizations of constant folding, dead code elimination, and code motion (loop constant removal) are discussed and the implementation of these optimizations in TAGs is presented.

Chapter Two
Optimization Techniques

2.1. Intermediate Representations

In a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code. Intermediate representations of the program are an "interface" between the high-level structure of the source language and the low level details of the target language. An intermediate language usually consists of a list of operations which is semantically equivalent to the source program. Optimization is applied to the program at this intermediate level and the definition of the intermediate code language will have a considerable effect on the complexity of the optimization algorithm [6].

2.1.1. Quadruples

Optimizations that are very machine-specific require an intermediate representation that encodes the necessary low-level details about the target machine hardware. The most common representation for this form is much like assembly language and is called "quads" (a form of three-address code [3]). Each individual program operation can be represented as a quad. A quad consists of one or two source operands, an operator and a destination operand. The operands are normally specified by variable names, either as specified by the original programmer or as temporaries by the compiler. High-level programming constructs, such as loops and array accesses, are decomposed in the quad notation to compares, branches and explicit address calculations. A short program segment and its quad representation are shown in figure 2.1.

Most of the program structure is lost in the quad representation and therefore, for a compiler that uses quads as an intermediate code language, much of the work of the optimizer is concerned with reconstructing information about the program structure [3] [7].

```
        .
        .
        .
X = 1
WHILE (X < 10) DO
BEGIN
    A = A * B + C * D
    X = X + 1
END
        .
        .
        .
```

$$\Longrightarrow$$

```
              .
              .
              .
      := , 1 ,   , X
L1:  < , X ,10, T0
     BRF,T0,   , L2
      * , A, B , T1
      * , C , D , T2
      + ,T1,T2, A
      + , X , 1 , X
     BR,   ,   , L1
L2:
              .
              .
              .
```

Figure 2.1. Representation of a code segment in quads.

2.1.2. Trees

The most convenient intermediate program representation for machine-independent optimizations is a tree or acyclic graph. Machine-independent optimizations require information about the program structure and in order to achieve many of these structural requirements, a compiler must take advantage of the structure available in the syntax of the source program. To preserve the structure , it is necessary that the intermediate representation be capable of representing that structure. The representation which preserves the natural hierarchical structure is an abstract-syntax tree (AST). The program segment of figure 2.1 is represented in an AST in figure 2.2. The major limitation with AST intermediate languages is that they lack the ability to represent the details of the target machine code that is inherent in quads.

An AST results from syntax analysis. With an attribute grammar (defined in the following section) an "attributed" tree (AT) can be obtained by associating attributes with nonterminal symbols in the grammar and correspondingly, with the nodes of the AST and evaluating attributes as specified by the rules of an attribute grammar.

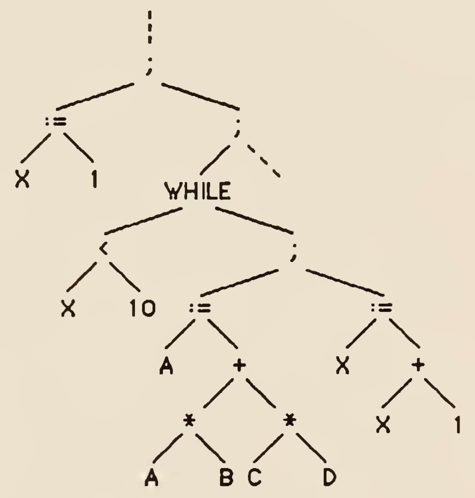


Figure 2.2. An AST for the program segment of figure 2.1.

2.2. Attribute Grammars

Attribute grammars were originally proposed as a means of defining the semantics of programming languages [26]. Since then, they have been employed for many purposes, often to give a theoretical framework for semantic analysis in compiler writing [28] [50] (in code generation [15] [16] and optimization [17] [39]).

An attribute grammar is an extension of a context-free grammar. It associates a set of *attributes* (possibly empty) with each nonterminal symbol in the vocabulary of the grammar. Each attribute represents a specific property of a nonterminal and can take on any of a specified set of values. There are two classes of attributes; inherited and derived (synthesized). Inherited attributes are passed down the parse tree to the nonterminal with which they are associated. Derived attributes are passed up the tree to productions containing references to their nonterminal. Consider Knuth's example of a grammar for binary numbers.

$$\begin{aligned}
&S \to N\\
&N \to L\\
&N \to L \text{ "." } L\\
&L \to B\\
&L \to L\ B\\
&B \to \text{"0"}\\
&B \to \text{"1"}
\end{aligned}$$

This context-free grammar generates all binary numbers in the form of a sequence of ones and zeros optionally followed by a binary point and another sequence of ones and zeros. The value of a binary number represented by this grammar is derived on the basis of the value of each digit and its position in the string. This value can be expressed by extending the above CFG into an AG as follows.

```
S                          → N ↑value

N ↑ value                  → L ↓ 0 ↑ value ↑ length
                           → L ↓ 0 ↑ value1 ↑ length1 "." L ↓ (-length2) ↑ value2 ↑ length2
                                 {value = value1 + value2}

L ↓ scale ↑ value ↑ length → B ↓ scale ↑value
                                 {length = 1}
                           → L ↓ (scale + 1) ↑ value1 ↑ length1  B ↓ scale ↑ value2
                                 {length = length1 + 1}
                                 {value = value1 + value2}

B ↓ scale ↑ value          → "0"
                                 {value = 0}
                           → "1"
                                 {value = 2 ** scale}
```

Each nonterminal symbol has associated with it a list of attributes ("↓" indicates an inherited attribute and "↑" indicates a derived attribute). In this example there are three attributes; ↓*scale* , ↑*value* and ↑*length* .

Every bit (B) inherits a scale factor corresponding to its position in the bit string and derives a value which is either 0 (if the bit is "0") or 2 raised to the power of *scale* (if the bit is "1").

A list of bits (L) inherits a scale factor and derives a value and a length. If the list is a single bit (L → B), then the length is one, the value is the value of that bit, and the scale is the scale that was inherited by the list. If the list is a list followed by a bit (L → LB), then the derived length is one greater than that of the list in the right part of the rule, the value is the sum of the values of the list in the right part and its following bit, the scale inherited by the list in the right part is one greater than that inherited by the list and the scale of the bit is the same as the scale inherited by the list.

A number (N) has no inherited attributes but it derives a value which is the value of its (single) list of bits (N → L), or the sum of the values of the two lists (N → L"."L). The scale of the list to the right of the binary point is the negative of the length of that list and the scale of the other (or only) list is zero.

2.3. Translational Grammars

Translation can be formally defined via a grammar with two right parts to each production rather than just one. A "transduction" grammar is an extension of a context-free grammar. A second right part is added to a CFG which will generate a new string of terminal and nonterminal symbols. Each occurrence of a given nonterminal in one right-part must have a corresponding occurrence of the same nonterminal in the other right-part, but not necessarily in the same order. The second right-part generates a string in either the same or possibly a different language than the underlying context-free grammar. The result is also a context-free grammar. A transduction grammar produces a translated copy of the input as output, and they can be used for such string-to-string translations as prefix, postfix or infix notation.

A transduction grammar with an underlying CFG that generates expressions containing addition and multiplication operators is shown below. This grammar defines a translation from infix to postfix notation.

$$
\begin{array}{lll}
S \rightarrow E & \Rightarrow E \\
E \rightarrow E + T & \Rightarrow E\ T + \\
E \rightarrow T & \Rightarrow T \\
T \rightarrow T * F & \Rightarrow T\ F * \\
T \rightarrow F & \Rightarrow F \\
F \rightarrow (E) & \Rightarrow E \\
F \rightarrow a & \Rightarrow a
\end{array}
$$

The translation of the string (a + a) * a is illustrated in the following parallel derivation.

E	E
T	T
T * F	T F *
F * F	F F *
(E) * F	E F *
(E + T) * F	E T + F *
(T + T) * F	T T + F *
(F + T) * F	F T + F *
(F + F) * F	F F + F *
(a + F) * F	a F + F *
(a + a) * F	a a + F *
(a + a) * a	a a + a *

Transduction grammars must follow a rule of preserving the items in the input string (every nonterminal in the right-part of the underlying CFG must appear in the translational-part of the transduction grammar). A grammar which does not preserve the input object but instead translates (transforms) the input object into another form, is called a "Transformational Grammar". Not every nonterminal in the right part of the underlying CFG is regenerated in the right-part of the transformational grammar. Rather than implementing complex reorderings (such as postfix notation) with string-to-string (transduction) grammars, strings can be translated at least partially into trees and then the trees are transformed with transformational grammars.

2.4. Grammars Applied to Optimization

Combining the power of an attribute grammar with a transformational grammar has been proposed as an effective means of implementing optimizations.

In the MUG2 compiler generator [17], a transformational grammar is applied to an attributed program graph. Optimization is done by transformation of the program graph into another graph in the same representation. The grammar they use is called an "Attributed Transformational Grammar" (ATG).

We use a "Transformational Attribute Grammar" (TAG) [39] which extends an attribute grammar into a transformational grammar. A TAG is similar to the ATG of the MUG2 project but has eliminated some of the inconsistencies for greater grammatical clarity and implementation. TAGs are applied to attributed (decorated) program trees (ILTs) and are presented in the next chapter.

Chapter Three
Introduction To TAGs

3.1. Transformational Attribute Grammars (TAGs)

TAGs were introduced as a powerful, intuitive language for the specification of optimizing transformations in a compiler [39]. A Transformational Attribute Grammar is a Transformational Grammar that is also an Attribute Grammar and is applied to attributed program trees (ILTs) to produce program trees in the same language. The power of TAGs is in their ability to pass information through attributes to the point where it is needed and their ability to specify transformations based upon the evaluation of attributes. Applying TAGs to ILTs is an effective means of implementing optimizations and demonstrating this efficiency is the intention of this work.

Optimization has been well defined in the literature, and has been broken up into several distinct transformations (One work distinguishes 27 different kinds of optimization [39]). Each optimization generally follows the same pattern and each usually requires two (maybe not distinct) phases; analysis and transformation. In a TAG, attribute grammars function to specify both the analysis and transformation phases. The transformations of the second phase are distinguished by extending the AG into a TAG, with transformational parts in some or all of the rules. Thus, TAGs are suitable for specifying most of the known optimizations.

3.2. Intermediate Low-level Trees (ILTs)

The intermediate representation of a program plays an important role in the development of compiler back-ends. The level of detail representative of the intermediate language will determine the level of effort spent in optimizing and code-generating algorithms. The information needed is the preserved structure of the original source program and a knowledge of the target code. Quads have the capability of representing the target machine but almost all of the original program structure is lost. High-level trees, on the other hand, are an excellent vehicle for preserving the program structure but rarely provide enough knowledge for generating the target machine code.

An intermediate language that is superior to representations in quads and high-level trees is "Intermediate Low-level Trees". ILTs have the combined capability of capturing the ability of quads for representing the details of the target machine and retaining the high-level structure of the program that is typical of high-level trees.

The front end of a compiler translates a source program into an intermediate language, optimizing modules perform transformations on that language and the final pass of the compiler translates the intermediate language into the target machine code. The information needed for all phases of optimization and code generation is represented in an ILT. Thus, only a single intermediate language is needed, which is contrary to other methods in which the program goes through several different representations before the final code is generated (as in the Bliss-11 compiler [56]). The use of ILTs as intermediate representations does not restrict the ordering of optimizations and the ability to reiterate optimizations is preserved. The uniformity of the program representation (ILTs) between different optimizations also provides a basis for the specification of optimization algorithms in a uniform language (TAGs).

3.2.1. ILT Decorations

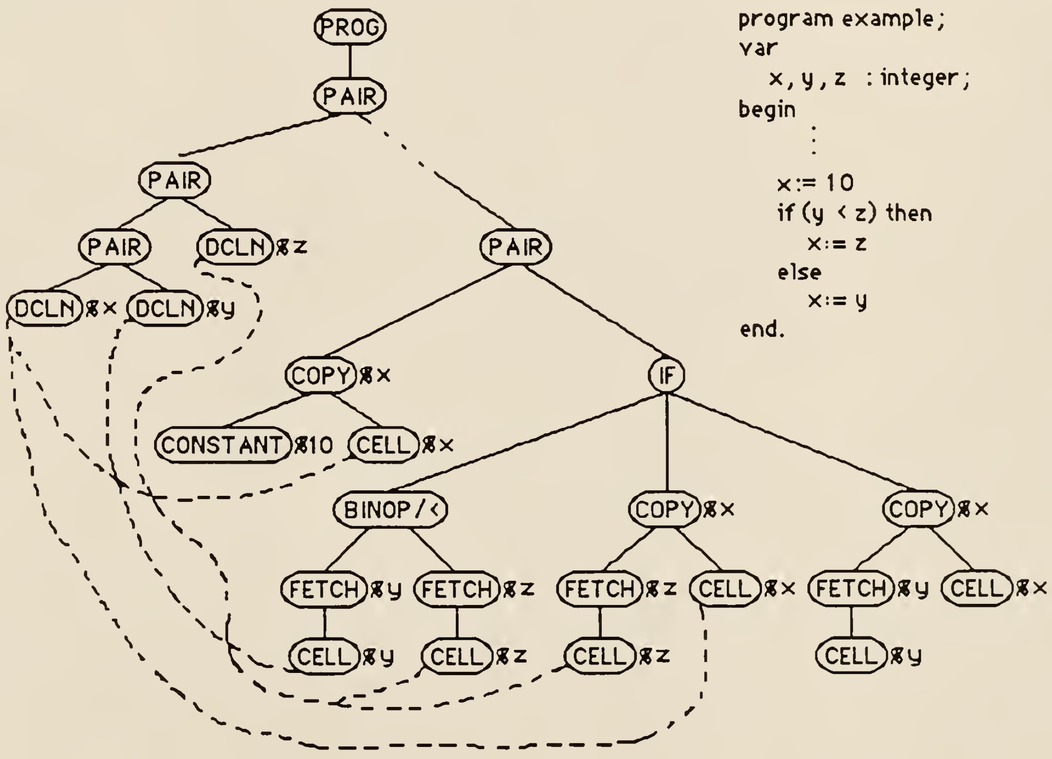
As the front end translates the source language into an ILT, syntactic and semantic information regarding the source program is associated with the nodes of the tree. The tree is (by conventional terminology) "attributed" with one or more items of information at each node and in effect, is a sort of directed graph as a result of node decorations which are links to other nodes. These links however, are not actual tree edges and are used only in the evaluation of attributes. Thus, the complexities of an arbitrary graph traversal are not present in the traversal of ILTs.

Two links which exist in an ILT are those to variable and procedure declarations. Declaration links are provided from each variable cell in the program to its declaration (DCLN) node. Information regarding the variable cell, such as its size and type, can be obtained via this link; The DCLN node is decorated with the corresponding variable's type (a TYPE node) and the TYPE node is decorated with the corresponding variable's size. A link is also provided from each procedure reference (CALL node) to its corresponding procedure tree (PROC node).

The front end places all of the initial information needed for optimization in the ILT and when applied, optimizations may add to, or change the decorations at each node in the tree. A list of the nodes and their initial front-end decorations is given in Figure 3.1. Figure 3.2 illustrates the representation of a trivial source program as an ILT with most of its initial decorations.

NODE NAME	DECORATIONS
PROG	none
PROC	procedure index
TYPE	size
DCLN	variable index, type
PAIR	none
LOOP	control variable index
IF	none
COPY	variable index, variable declaration
CALL	procedure declaration
GOTO	loop reference
BINOP/op	expression type
UNOP/op	expression type
FETCH	variable index, variable declaration
CELL	variable index, variable declaration
CONSTANT	value, type

Figure 3.1. Initial ILT nodes and decorations.



```
program example;
var
   x, y, z  : integer;
begin
      :
   x:= 10
   if (y < z) then
      x:= z
   else
      x:= y
end.
```

Figure 3.2. A program in ILT form.

3.2.2. TAG Attributes

As an optimization walks the program tree, information derived from the tree can be carried through the program tree with the attributes in a TAG and transformations can be made to the tree based upon the evaluation of these attributes. Since optimizations require context information, any data that is obtainable from the tree is a candidate to be passed by attributes to wherever it is needed in the tree. This includes not only decorations but also parts (references to nodes) of the tree. Transformations on non-contiguous parts of the program can be performed with TAGs due to the context information that is made available through the attributes of the grammar. Thus, transformations are not restricted to local parts of the tree.

3.3. Implementing Optimizations in TAGs, A Computer Language

TAGs are implemented as a data flow language. Operations are not necessarily performed as they occur sequentially in the TAG but when the values that they depend upon are available. Specific sequencing of operations can be accomplished by forcing dependencies.

Attributes are represented as identifiers but each attribute is assigned a value only once in each attribute-evaluation part of a production rule. Each production in a TAG is implemented as a (attribute-evaluation) function where the attributes associated with a given nonterminal constitute value and reference parameters. Functions which manipulate attributes but do not necessarily operate on tree nodes may be included in the grammar as pseudo-nonterminals and are not distinguishable from nonterminals. Other functions are available for attribute evaluation which (like pseudo-nonterminals) are not part of the original underlying CFG and are built into the TAG language as intrinsics encoded in a separate module (as "predeclared"s). A description of the "predeclared" functions is given in

3.3.1. TAG Syntax

Figure 3.3 illustrates the form of each production in a TAG function. The four main components which make up a TAG production are a left part, a right part, attribute evaluations and an optional transformational part.

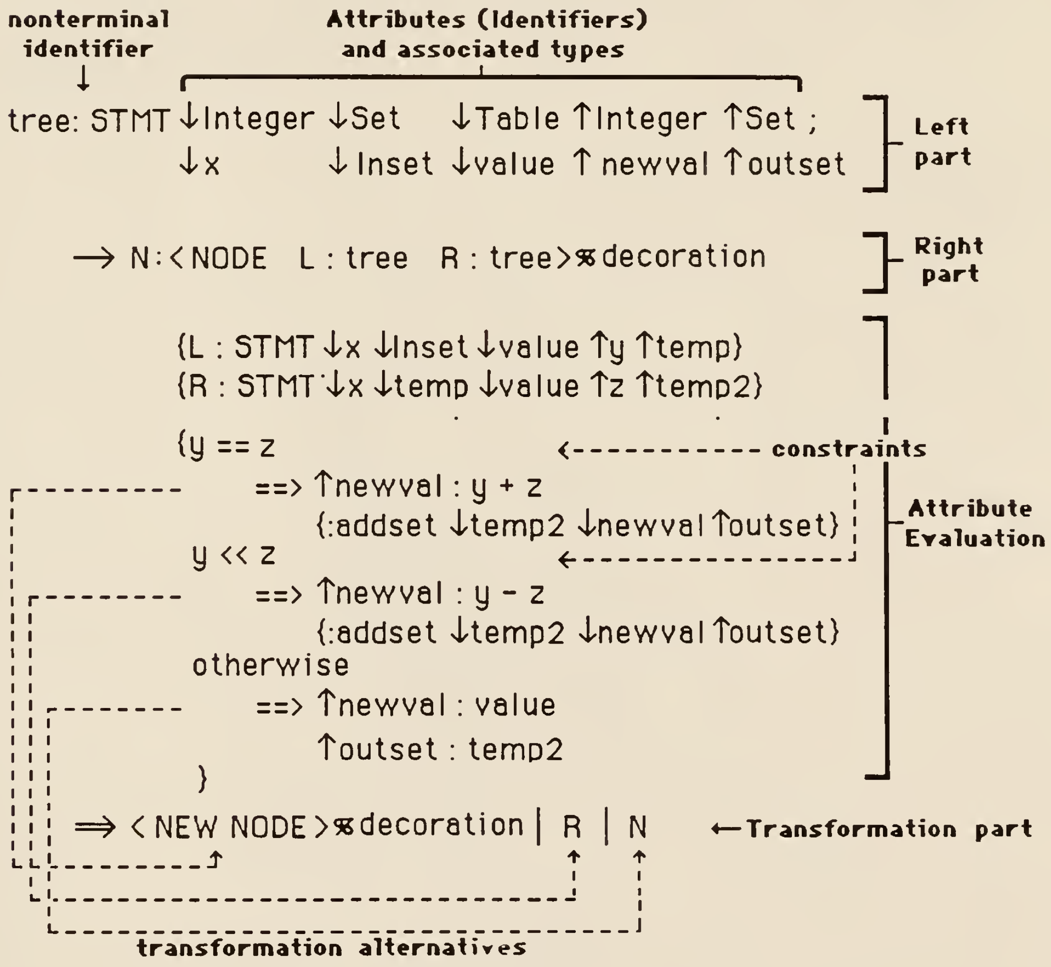
The left part (corresponding to a left part of the underlying CFG production) consists of the nonterminal identifier, and its associated list of attributes (identifiers) and their types.

The right part consists of the tree pattern which is the right part of the underlying CFG production.

Attribute evaluations are enclosed in braces and consist of function references, direct assignments to attributes, and alternatives. Alternatives distinguish the different actions to be taken based upon certain criteria (constraints), much like a high-level language CASE-statement.

The transformational part is delimited by a double arrow and indicates the new pattern which the left part will be transformed into. More than one new tree pattern may be indicated in the transformational part (seperated by "or" bars) in which case the pattern that is chosen depends upon the path (alternative) taken in the attribute evaluations.

In the example form of figure 3.3, there are three alternatives matched with three different transformations. Only one alternative is taken (the "true" one). If the first constraint is "true" then the first alternative is taken and the input tree (node) is transformed into a different node (NEW NODE). If the first is "false" but the second is "true" then the second alternative is taken and the tree is transformed into "R" which is a subtree of the original node. If neither of the constraints are "true" then the "otherwise" alternative is taken and the tree is transformed into "N" which is itself (that is, the original node is not transformed at all). Some productions may not perform transformations in which case a transformational part will not appear.



```
tree: STMT ↓Integer ↓Set    ↓Table ↑Integer ↑Set ;
           ↓x       ↓Inset  ↓value ↑newval  ↑outset

   → N:<NODE  L : tree  R : tree>∞decoration

           {L : STMT ↓x ↓Inset ↓value ↑y ↑temp}
           {R : STMT ↓x ↓temp ↓value ↑z ↑temp2}

           {y == z
                 ==> ↑newval : y + z
                     {:addset ↓temp2 ↓newval ↑outset}
            y << z
                 ==> ↑newval : y - z
                     {:addset ↓temp2 ↓newval ↑outset}
            otherwise
                 ==> ↑newval : value
                     ↑outset : temp2
            }
   ⇒ <NEW NODE>∞decoration | R | N
```

Figure 3.3. Example of a TAG production.

3.3.2. The TAG Compiler

A prototype compiler to translate TAGs into Pascal code was developed on the Macintosh computer and later ported to the Vax 11/780. It has been used in the classroom at Kansas State University and was used to compile the TAGs presented in this work.

Chapter Four
Implementations Using TAGs

4.1. Constant Folding and Dead Code Elimination

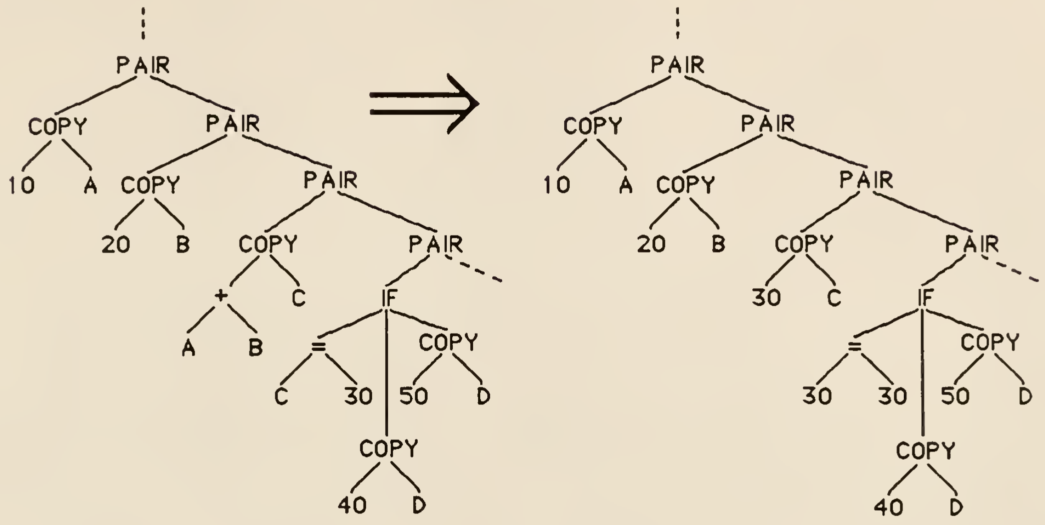
4.1.1. Definition

The constant folding optimization, also known as subsumption and constant propagation, is the evaluation of any expressions involving constant operands at compile-time. Variables which have been set to a constant value are replaced by the constant itself, and any operator node that has constant subtrees is replaced by a constant node which is the evaluation of the operator node. Consider the following example:

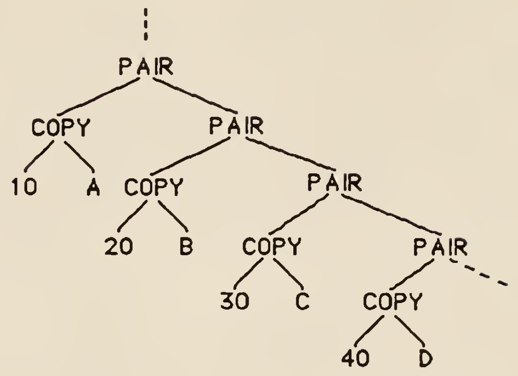
```
 .                                   .
 .                                   .
 .                                   .
A = 10                              A = 10
B = 20                              B = 20
C = A+B                             C = 30
IF (C = 30) THEN         ⟹          IF (30 = 30) THEN
   D = 40                              D = 40
ELSE                                ELSE
   D = 50                              D = 50
END IF                              END IF
 .                                   .
 .                                   .
 .                                   .
```

The constants assigned to A and B in statements 1 and 2 replace the uses of A and B in statement 3. The expression can then be evaluated at statement 3. This finds C=30 which is then propagated to the use of C in the

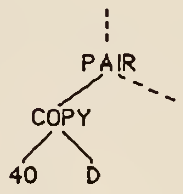
IF expression. The same example can be represented as follows in an ILT.



An immediate and obvious result of constant folding is dead (unreachable) code. Instructions are considered dead when they cannot be executed, either because they are in an area of the program which can no longer be reached, or because their results are never used. Most dead code is located in branches of a fork where the condition of the fork is constant. In the previous example the constant value for C was propagated into the IF expression. The result is that the IF condition becomes constant also (both subtrees of the BINOP/= node are constant) and is folded by evaluating the expression at the BINOP node. Once folded the condition becomes "IF (true) then". We can see that the ELSE block is now dead code and can be removed. Furthermore, the IF statement is no longer needed because of its constant-true condition and is also removed (Actually, the IF node is transformed into the THEN subtree. If the condition would have been constant-false then the IF node would have been transformed into the ELSE subtree.). The tree now appears as follows.



In addition, if it is determined that A, B and C are no longer needed (their results are no longer used), then the assignments to these variables are also considered "dead". In this case we are left with the following single statement.



Constant folding and dead code elimination have many obvious advantages and no disadvantages.

4.1.2. Implementing the Concept

As can be seen, the concept of constant folding and dead code elimination is straightforward and simple. However, the implementation of an optimizer to perform such analysis and transformations is not an easy task.

As stated previously in chapter two, the intermediate program representation upon which the optimizer works will make a big difference in the difficulty level of the design for the optimizer. The standard of quadruples was used as an intermediate representation in the design of an optimizer which performed the transformations of constant folding and dead code elimination discussed above, with Pascal being used as the implementation language (Appendix B). The design of the optimizer became very complex and nerve racking, not to mention very time consuming. As a result of the complexity of the code, the module was extremely difficult to debug.

The same optimizations were implemented using a TAG which would operate on Intermediate Low-level code Trees (Appendix C). Admittedly, the process was not trivial but the analysis was easier, transformations were straightforward and debugging was less of a problem. The overall complexity level of the algorithm is far less than that of the above optimizer designed for quads and thus, the time and effort spent in implementation was far smaller.

An effort was made to keep track of the time that it took to design and implement each of these modules. It is estimated that the Pascal/Quads version took approximately 1/3 longer to build than did the TAG/ILT version.

4.1.3. The TAG/ILT Constant Folding Optimizer, ConsFold

4.1.3.1. A Brief Description of the ConsFold Functions

ConsFold is broken up into several different functions, namely; STMT, EXPN, SCALAR, CrossFold, WalkParam, WalkArg and Exclude.

The two main functions are STMT and EXPN. STMT is used to traverse the statements in the program tree (Traversal of the program starts and ends in STMT). When an expression is encountered in a statement, EXPN is invoked to traverse it. All possible constant folding transformations are performed in the productions of the EXPN function. No other function performs transformations with the exception of STMT where the transformations performed are those of dead code elimination at an IF node.

The three functions, CrossFold, WalkArg and WalkParam are dedicated solely to the processing of subroutine calls. WalkArg is used to traverse the argument list of a subroutine call, WalkParam is used to traverse the formal parameter list in the subroutine declaration and CrossFold is used to start the traversal of the subroutine body.

The remaining two functions, SCALAR and Exclude, have nothing to do with walking the program tree. SCALAR is used to determine if a variable type is scalar (integer, boolean or character) and Exclude is used to remove an item from a set. Exclude could very well have been included in the predeclared file but as an example of TAG versatility, it was encoded in the TAG as a pseudo-nonterminal.

A complete commented listing of these functions (the ConsFold optimizer) can be found in Appendix C.

4.1.3.2. Analysis and Transformations

This section will illustrate the analysis and transformations of constant folding and dead code elimination. A description of what needs to be done will be given, followed by an explanation of how it is done in the ConsFold TAG. To simplify the discussion some attributes are left out at first and then added to productions as they are needed. The complete listing of ConsFold is in Appendix C.

The analysis required to perform constant folding can be classified as a simulated execution of the intermediate program [39]. That is; the optimizer effectively interprets the program and as much information as can be determined at compile-time is computed. The analysis is performed in the same pass as the transformations with the exception of loops (Two passes are performed on the body of a loop. The first pass performs the analysis and the second the transformations.).

Constantness of Variables

In order to perform constant folding transformations, the analysis must establish whether or not a variable is constant before that variable is referenced in the program. The "constantness" of a variable can only be determined when an assignment is made to that variable.

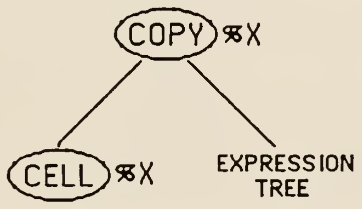


Figure 4.1. An assignment tree

At an assignment node such as that in figure 4.1, the expression subtree is traversed and a report of its constantness is returned along with its value (The value is only meaningful if the variable is constant). If it is found that the expression is constant then the variable X can be recorded as constant and a record can be made of its constant value. The following production is how this analysis at a COPY node is implemented in the ConsFold TAG.

```
    STMT ~isconin ~valin ^isconout ^valout
        -> < COPY  E:tree  < CELL > > %varid
                {E:EXPN ~isconin ~valin ^constnt ^value ^valtemp ^iscontemp}
                {constnt >< 0     #expression was constant
                    =>{:addset ~varid ~isconloop ^isconout}   #add id to iscon set
                        {:into ~valtemp ~varid ~value ^valout} #put const. value in table
                 otherwise  # variable is not constant
                    =>{:Exclude ~varid ~iscontemp ^isconout} #remove id from iscon
                        ^valout : valtemp
                }
```

Sets and tables are passed as attributes up and down through the tree as it is traversed. The two attributes *isconin* and *isconout* actually represent the same set. Isconin is the set which contains the id numbers of variables which are considered constant upon arrival at the current point (node) in the tree. Isconout represents the same set (with possible additions) to be returned from the current node to the parent node. References to this set are usually made by calling it the "iscon" set. *Valin* and *valout* also represent a common table. Valin is a table which contains variable ids along with their corresponding constant values upon arrival at the current node and valout is the same table to be returned. This table is called the "value" table.

When an identifier is determined to be constant, its id number is entered into the iscon set. The same id is also entered in the value table along with its corresponding constant value. If however, the identifier is determined to be nonconstant (the expression is not constant) then the variable's id number must be removed from the iscon set.

Constantness of Expressions

When a reference to a variable is made in an expression, evidence of its constantness can be obtained by checking the iscon set. Consider the production for the FETCH node in the EXPN function.

```
EXPN ~isconin ~valin ^isconout ^valout ^constnt ^value
        -> F: < FETCH < CELL > > % varid
             {F: examine ^decnode}      # get DCLN node
             {decnode:examine ^tipe}  # get TYPE node
             {varid in isconin   # variable is constant
                 =>  {:from ~valin ~varid ^value} # retrieve its constant value
                     ^constnt : 1              # return constant flag as true
                     {T <- value}               # decorate new node with value
                     {T:decorate ~tipe}    # decorate new node with tipe
              otherwise  # this node is not constant
                 =>  ^constnt : 0              # return constant flag as false
                     ^value : 0                 # no constant value to return
             }
             ^isconout : isconin
             ^valout : valin

        => T: < CONSTANT > | F
```

If the variable's id (varid) occurs in the iscon set then it is constant and the first alternative is taken. The variable's constant value is retrieved from the value table, the FETCH node is transformed into a CONSTANT node and decorated with its value and type, and the attribute *constnt* is returned as 1, meaning that this node is constant. Otherwise, if the variable's id does not occur in the iscon set then the variable is not constant at this node. The FETCH node is preserved and constnt is returned as 0, meaning that this node is not constant. In either case, the attribute *value* is returned. When the variable is constant, the attribute *value* will return with the variable's constant value. In the case where the variable is not constant, *value* is returned with a default value of 0.

A FETCH node may occur as an expression itself, but it will more likely be an operand for a larger expression. The operator nodes, BINOP and UNOP, are analyzed and transformed in basically the same way as a FETCH node. The difference is that the BINOP and UNOP nodes have subtrees which must be analyzed before anything can be done at the operator node. The following production shows how the analysis and transformations are performed at a typical BINOP/+ node.

```
EXPN ~isconin ~valin ^isconout ^valout ^constnt ^value
     -> B: < BINOP/add  L:tree  R:tree >
            {L:EXPN ~isconin ~valin ^iscontemp ^valtemp ^lconstnt ^lvalue}
            {R:EXPN ~iscontemp ~valtemp ^isconout ^valout ^rconstnt ^rvalue}
            {B:examine ^tipe}
            ^value : lvalue + rvalue
            {lconstnt == 1   rconstnt == 1   # both subtrees are constant
                 =>  ^constnt : 1  # this node is constant
                      {T <- value}  # decorate new node with value
                      {T : decorate ~tipe}  # decorate new node with type
             otherwise  # at least one subtree is not constant
                 =>  ^constnt : 0  # this node is not constant
            }

     => T: < CONSTANT >  |  B
```

As can be seen, the two operand subtrees are traversed first, each of which returns a report of its constantness (lconstnt and rconstnt) and a value (lvalue and rvalue). In order to consider the BINOP node constant, both subtrees must be constant (lconstnt and rconstnt must both be 1). The test is made and the appropriate actions are taken similar to those at a FETCH node. The productions for the remaining BINOP and UNOP nodes follow a similar pattern.

An operand of an expression may also be a function call. When a CALL node occurs in an expression tree, the function that is being called must be identified and walked (provided that it has not been previously walked). This must occur so that the status of variables can be updated in the iscon set and the value table. The detail concerning functions will be bypassed here because all function/procedure calls are handled in basically the same way and are discussed later in this section.

Dead Code Removal in Forks

Although there is no opportunity for constant folding at an IF node (tree fork), the analysis for constant folding which is performed on the IF expression provides the opportunity for the removal of dead code in the fork. If an IF expression is determined to be constant with a value of "true" then the statements in the ELSE tree become dead code and conversely, if the expression in found to be "constant-false" then the THEN block is dead code. In either case, if the IF expression is constant then the condition at the IF node no longer serves a purpose and can be eliminated. However, if the analysis of the IF expression reveals the fact that the expression is not constant then there is no dead code to be removed. Additionally, a record must be kept of those variables whose values are changed in the branches of the fork. These variables should no longer be considered constant at the statement following the fork. The following production is how this analysis and the above transformations are implemented in ConsFold.

```
STMT ~isconin ~valin ^isconout ^valout ^notcon
     -> I: < IF  X:branch  T:tree  E:tree >
               {X:EXPN ~isconin ~valin ^isconx ^valx ^constnt ^value ^notconx}
               {constnt == 1   value == 1  #expn is constant-true - walk then tree
                    =>  {T:STMT ~isconx ~valx ^isconout ^valout ^notcont}
                         {:union ~notconx ~notcont ^notcon}  # join notcon sets
               constnt == 1   value == 0  #expression is constant-false - walk else tree
                    =>  {E:STMT ~isconx ~valx ^isconout ^valout ^notcone}
                         {:union ~notconx ~notcone ^notcon}  # join notcon sets
               otherwise  # the expression was not constant - walk both subtrees
                    =>  {T:STMT ~isconx ~valx ^iscont ^valt ^notcont}
                         {E:STMT ~isconx ~valx ^iscone ^vale ^notcone}
                         {:union ~notconx ~notcont ^notcont}  #join notcon sets
                         {:union ~notcone ~notconf ^notcon}
                         {:difference ~isconx ~notcon ^isconout}#remove nonconstants
                         ^valout : valx
               }

     =>T | E | I
```

As previously described, when an expression tree is walked, a report of its constantness (constnt) is returned along with the expression's value (value). In the above production, if constnt is 1 and value is 1 then the expression was constant with a value of "true" and the IF node is transformed into the THEN subtree. If constnt is 1 and value is 0 then the expression was constant with a value of "false" and the IF node is transformed into the ELSE subtree. Otherwise, if constnt is 0 then the expression was not constant and the IF node is preserved. In this later case, both subtrees are traversed and each returns a set of variables (notcont and notcone). This set, known as the "notcon" set, consists of those variables that have been assigned a value in the branch of the fork from which the set is returned. These variables cannot be considered constant below the fork and are therefore combined and removed from the iscon set.

Performing Constant Folding in Loops

A loop is also considered as a fork in the program and is handled in basically the same way. When a loop is encountered in the traversal of the program, two passes are made on the body of the loop. The first pass is required to identify variables whose values are changed in the body of the loop. As in a fork, these variables must not be considered constant below the loop and additionally, they must not be considered constant in the loop. Once these variables have been identified, a second pass can be made on the loop to perform the possible transformations.

The production for the LOOP node occurs in the STMT function in ConsFold as follows.

```
STMT ~isconin ~valin ~isloop ^isconout ^valout ^notcon
    -> < LOOP  BODY:tree >   # pass 1 -- analysis,  pass 2 -- transformation
           {BODY:STMT ~isconin ~valin ~1 ^isconloop ^dontcare1 ^dontcare2}
```

```
{BODY:STMT ~isconloop ~valin ~0 ^isconout ^valout ^notcon}
```

The same function is used for both the analysis and transformation passes of the loop, and as a means of distinguishing the two passes, the attribute *isloop* is used. All productions which manipulate the analysis information or transform the tree must test isloop before taking any actions.

On the first pass, isloop is passed down as 1 to indicate that it is in the analysis pass of a loop and no transformations should yet be made. Upon returning from the analysis pass, the iscon set (isconloop) contains only those variables that are constant (not changed) in the loop. All variables whose values are changed in the loop have been removed from the iscon set. This set (isconloop) is then passed down on the second (transformation) pass of the loop and isloop is passed down as 0 to indicate that transformations may now take place.

All of the work is actually done in the production for the IF node which is the same as previously illustrated with the addition of a test on isloop in the constraints of each alternative.

```
STMT ~isconin ~valin ^isconout ^valout ^notcon
    -> I: < IF  X:branch  T:tree  E:tree >
            {X:EXPN ~isconin ~valin ^isconx ^valx ^constnt ^value ^notconx}
            {constnt == 1   value == 1   isloop == 0 #not on analysis pass of a loop
                =>  {T:STMT ~isconx ~valx ^isconout ^valout ^notcont}
                    {:union ~notconx ~notcont ^notcon}  # join notcon sets
            constnt == 1   value == 0  isloop == 0 #not on analysis pass of loop
                =>  {E:STMT ~isconx ~valx ^isconout ^valout ^notcone}
                    {:union ~notconx ~notcone ^notcon}  # join notcon sets
            otherwise  # the expression was not constant - walk both subtrees
                =>  {T:STMT ~isconx ~valx ^iscont ^valt ^notcont}
                    {E:STMT ~isconx ~valx ^iscone ^vale ^notcone}
                    {:union ~notconx ~notcont ^notconf}  #join notcon sets
                    {:union ~notcone ~notconf ^notcon}
                    {:difference ~isconx ~notcon ^isconout}#remove nonconstants
                    ^valout : valx
            }

    => T | E | I
```

On the analysis pass of the loop the "otherwise" alternative is taken (because isloop is 1) and no transformation is performed. All three subtrees are walked and each returns a notcon set. The combined notcon sets (notconx, notcont and notcone) identify those variables which should not be considered constant in or below the loop. Consequently, the variables in the notcon set are removed from the set of constants, iscon. The resulting iscon set is then returned to the parent LOOP node. On the second pass of the loop, isloop will be 0 and transformations can be performed the same as previously described for the IF node.

Processing Procedure/Function calls

The analysis and transformations applied to the code of a subroutine are the same as that applied to the main program. The difficulty of processing subroutines is in the analysis of the "interface" that the subroutine has with its caller. The analysis must be able to identify global variables whose values are changed in the subroutine and any variables changed by pass-by-reference parameters. These variables cannot be considered constant at the point in the code following the subroutine call.

Consider the following example.

```
PROGRAM EXAMPLE
VAR  A, B, C, D, E, F : INTEGER
        PROCEDURE CHANGE (VAR X:INTEGER; Y, Z:INTEGER)
        VAR  W : INTEGER
        BEGIN
             W = 15
             WHILE (X < W) DO
                Y = Y + Z
                X = X + 1
             END WHILE
             C = X * Y
        END
BEGIN                                                BEGIN
     A = 10                                               A = 10
     B = A * 10                                           B = 100
     C = A + B                                            C = 110
     CHANGE (A,B,C)                 ⟹                    CHANGE (A,B,C)
     D = A + B                                            D = A + 100
     E = B + C                                            E = 100 + C
     F = E * D                                            F = E * D
END                                                  END
```

When we reach the procedure call at statement 4 in the main program, variables A, B and C are constant. However, at statement 5 following the call, only B remains constant. The variable A is no longer constant because it is passed as an argument which is matched up with a pass-by-reference parameter and consequently, its value will be changed. C is no longer constant because, being a global variable, its value was changed in the subroutine body. Thus, at the end of the main program, the only constant variable is B.

When subroutines are called, either their effects on the program variables must be taken into account or optimization must be abandoned across procedure calls. Most methods of subroutine optimization treat each subroutine as a separate program and make notes of side effects [56]. In

ConsFold a method is used where, in effect, a subroutine is opened up in line. The subroutine is not actually "back-substituted" (although this would be a good place to perform that optimization) but rather the subroutine is traversed when a call is made to that subroutine. The argument list is matched up with the parameter list (pass-by-reference parameters are identified) and the procedure is analyzed and transformed (global variables affected in the subroutine are identified). Care is taken however, so that recursive procedure walks are avoided. The attribute *procset* is used for this purpose. Procset is a set which contains the ids of those procedures which are currently "open" (being traversed) starting with the procedure on the outer level of the calling sequence. Also, there is no need to walk a procedure (analyze and transform) more than once. In order to tell if a procedure has already been walked at some time or another in the program tree, the set attribute *walked* is used. This set is passed up and down through the entire program tree and contains the ids of those procedures which have been walked.

There are three main functions that are used to process subroutines in ConsFold. WalkArg is used to traverse the argument list of the CALL node. It walks each argument and enters the position number (where it occurs in the argument list) and the id of the actual argument into a table called *match* . If the argument is some type of expression then a default value of -99 is entered (because the argument will not be pass-by-reference and will not concern us when the arguments are matched up with the formal parameters)

The parameter list of a procedure is traversed using function WalkParam. Each parameter is analyzed and if it is pass-by-reference then the corresponding argument (variable id) is retrieved from the match table that was derived in function WalkArg. This variable id is then removed from the iscon set.

Function CrossFold receives the PROC node of the procedure that is to be traversed and is encoded in ConsFold as follows.

```
cross:CrossFold ~isconin ~valin ~isloop ~argmatch ~procset ~walkedin ^walkedout
        ^isconout ^valout ^notcon
        -> P:< PROC  PARAM:tree  BODY:tree >%pid  #PROC is only possible node
            {PARAM:WalkParam ~isconin ~valin ~argmatch ~1 ~procset ^newiscon
                ^newval ^dontcare} # walk parameter list
            {pid in walkedin  # procedure has been walked already
                =>  {P:examine ^notcon}  #retrieve the procedure's set of nonconstants
                    ^walkedout : walkedin
             otherwise  # the procedure hasn't been walked yet
                =>  {BODY:STMT ~Empty ~Empty ~isloop ^junkiscon ^junkval
                        ^notcon}
                    {P:decorate ~notcon}  # decorate the proc with its notcon set
                    {:addset ~pid ~walkedin ^walkedout}  # add proc id to walked set
            }
            {:difference ~newiscon ~notcon ^isconout} # remove from set of constants
            ^valout : newval
;
```

First, the parameter list is walked and a new iscon set is returned. Next, the walked set is checked to see if the procedure has already been walked. If it hasn't then the procedure body is traversed and a notcon set is returned, the PROC node is decorated with the notcon set and the procedure id is added to the walked set. On the other hand if the procedure id occurs in the walked set, then the set of nonconstants (notcon) can be retrieved from the PROC node. Finally, the set of nonconstants of the procedure (notcon) is removed from the set of constants (iscon) to be returned to the parent CALL node.

A subroutine can appear in a program as a statement, as a function in an expression, or as an argument. In ConsFold a production exists for each of these possibilities. They are all almost identical and for the discussion here, the production for the CALL node from function STMT is presented.

```
STMT ~isconin ~valin ~isloop ~procset ~walkedin^walkedout^isconout^valout^notcon
        -> C:< CALL/call  ARG:walka >
             {C:examine ^proc}  # retrieve PROC node
             {proc -> nextid}  # get procedure id
             {ARG:WalkArg ~isconin ~valin ~isloop ~procset ~walkedin ^walkedt
                  ^argmatch ^iscont ^valt ^notcont}  # walk argument list
             {nextid in procset  #detected a recursive call-don't walk again
                  =>  ^isconout : iscont
                      ^valout : valt
                      ^notcon : notcont
                      ^walkedout : walkedt
              otherwise  # no recursion detected yet -- walk the procedure
                  =>  {:addset ~nextid ~procset ^pidset} #add id to recursion set
                      {proc:CrossFold ~iscont ~valt ~isloop ~argmatch ~procset ~walkedt
                           ^walkedout ^isconout ^valout ^notconout}
                      {:union ~notcont ~notconout ^notcon}  # join notcon sets
             }
```

As can be seen, the procedure id is retrieved from decoration and the argument list is walked. If the procedure id occurs in procset then a recursive call loop has been detected and the subroutine is not walked. Otherwise, if no recursion is detected then the procedure id is added to procset and a call is made to function CrossFold.

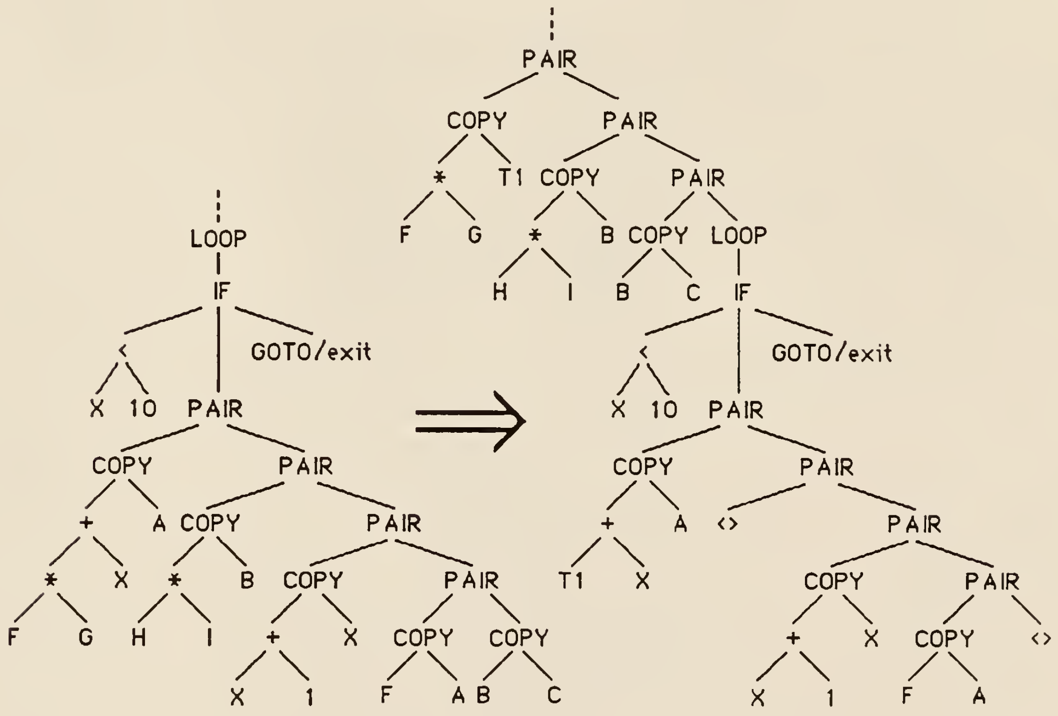
4.2. Loop Constant Removal

4.2.1. Definition

The removal of loop constants is referred to as code motion in most of the literature [4] [18] [39]. Code motion can be defined as moving a subexpression if the value available to its uses is not changed by the move and if the move will not cause side effects [4]. More simply, code motion (as applied to loops) is the process of moving out of a loop any computations that are the same for each iteration of the loop. The basic intent is to move instructions from frequently executed areas of the program to less frequently executed areas. As an example, consider the following program segment.

```
        .                          .
        .                          .
        .                          .
WHILE (X < 10) DO             T1 = F * G
BEGIN                         B = H * I
   A = F * G + X              C = B
   B = H * I        ==>       WHILE (X < 10) DO
   X = X + 1                  BEGIN
   A = F                         A = T1 + X
   C = B                         X = X + 1
END                              A = F
        .                     END
        .                          .
        .                          .
                                   .
```

The expression, F*G, in statement 1 is loop-constant and can be moved because the values of F and G never change in the loop. The same is true for H*I in the second statement. However, since the whole expression is constant and B is only assigned once in the loop, the whole statement is loop constant. In statement 3 the whole expression F is constant but is not worth moving because that would amount to replacing one variable with another and adding another assignment statement to the code above the loop. Furthermore, the whole statement is not constant because A is assigned twice in the loop. Statement 4 does not appear to be loop constant because of the assignment to B in statement 2, but once statement 2 is removed, statement 4 becomes loop-constant also. This example is represented as follows in an ILT.



It has been found that most of the execution time of a program is spent in the inner loops [27]. Thus, the primary advantage of removing loop constants is the reduction of the number of instructions executed. The disadvantage of code motion however, is that usually register usage is extended.

4.2.2. The TAG/ILT Loop Constant Optimizer, LoopCon

4.2.2.1. A Brief Description of the LoopCon Functions

LoopCon consists of 11 functions: FindCon, STMT, WalkArg, WalkParam, CrossFind, WalkLoop, rewalk, exmoveit, stmoveit, getdec and gluedec.

STMT is the main function. Traversal of the program tree starts at (and is controlled by) STMT. No pertinent analyses or transformations take place until a loop is encountered.

As in ConsFold, two passes are made on a loop. The first pass performs the analysis and the second does the transforming. Function FindCon is called to perform the analysis and function WalkLoop is called to start the transformation pass on the loop.

Functions WalkArg, WalkParam and CrossFind are used during the analysis pass to process subroutines, and perform basically the same functions as they did in ConsFold.

When items are found to be loop constant, functions exmoveit and stmoveit are called to move them out and glue them back into the tree above the loop. Exmoveit is used to move a loop-constant expression while stmoveit is used to move a loop-constant statement.

When an expression is moved out of the loop by exmoveit, it is assigned to a new variable above the loop. Function exmoveit calls function gluedec to create a DCLN node for this new variable and glue it into the tree in the current routine's declaration list. The point at which the new DCLN node is entered into the tree has been previously reserved by function getdec.

Function getdec is called at the beginning of the traversal of each subroutine (each PROC node) and at the beginning of the program (PROG node). It returns a pointer to the spot in the respective subroutine's DCLN list where new DCLNs for that routine should be entered.

The last of these functions is rewalk. This function is used to start a traversal on the code that was moved out of a loop. In the case of nested loops, loop-constant items that are moved out of the inner loop may also be loop-constant in the outer loop. It is for this reason that the moved code is "rewalked".

4.2.2.2. Analysis and Transformations

In this section, the analysis and transformations of Loop Constant Removal are illustrated. A description of what needs to be done is given, followed by an explanation of how it is done in the LoopCon TAG. To simplify the discussion some attributes are left out at first and then added to productions as they are needed. The complete listing of LoopCon is in Appendix D.

Detecting Nonconstant Variables in a Loop

When a loop is encountered in the traversal of the program tree, no transformations may take place until variables that are nonconstant in the loop are identified. Any variable which is assigned a value in the loop may not be considered constant in any expression in the loop (There is one exception to this rule as shown in the previous example). Also, a record must be kept of those variables that are assigned a value more than once in the loop. Statements that make an assignment to one of these variables are not loop constant.

In LoopCon this analysis is performed in function FindCon. Two sets are used to keep track of the variables described above. The *notcon* set contains the ids of those variables that are assigned a value in the loop and *twice* is the set that contains the ids of those variables assigned to more than once in the loop. These variables can be detected at a COPY node.

```
FindCon ~notconin ~twicein ^notconout ^twiceout
    -> < COPY  L:find  R:find > %varid
            {L:FindCon ~notconin ~twicein ^lnotcon ^ltwice}
            {R:FindCon ~lnotcon ~ltwice ^rnotcon ^rtwice}
            { varid in rnotcon  # variable has been assigned more than once
                => {:addset ~varid ~rtwice ^twiceout}
                    ^notconout : rnotcon
             otherwise  # this is the first assignment to the variable
                => {:addset ~varid ~rnotcon ^notconout}
                    ^twiceout : rtwice
            }
```

After walking the left and right subtrees, the notcon set is tested for the variable id. If it occurs in the notcon set then the variable evidently is assigned to more than once in the loop and is added to the twice set. If it is not already in the notcon set then it is added to notcon in the otherwise alternative.

When a procedure/function call is encountered during the analysis pass, the subroutine must be walked and variables whose values are changed in the subroutine (either by direct assignment or by a pass-by-reference parameter) must be recorded. Since these variables are assigned new values in the loop, they may not be considered loop-constant in any expression in the loop.

In LoopCon, the same method is used to walk a subroutine and its parameters as was used in ConsFold. The subroutine is, in effect, back-substituted at the point of call. The subroutine is walked and variables which are assigned a value in the subroutine are recorded in the notcon set. Upon return to the CALL, this notcon set from the subroutine is added to the current notcon set for the loop. If a variable already occurs in the notcon set then it is added to the twice set. Once walked, a subroutine's id is added to the *walked* set and the subroutine's notcon set is hung (as a decoration) on the PROC node. Also, as in ConsFold, a set, *procset* , is passed down each subroutine calling sequence in order to avoid a recursive walk.

Detecting Constant Items in a Loop

Part of the analysis is performed in the actual transformation pass of the loop in LoopCon. Constant weights are assigned to each node based on the constantness of its branches.

Before an expression may be moved to the outside of a loop, it must of course be constant and additionally, it must be big enough to warrant that moving it will result in optimization. In the example of section 4.2.1, the expression of statement 3 is indeed constant. However, moving this expression would result in adding instructions as illustrated in figure 4.2.

In order to avoid this problem, expressions should not be moved on the basis of their constantness alone, but also based on their size. The leaf nodes, CELL and CONSTANT, are given a minimum constant weight of 2. Thus, any expression which is determined constant must have a constant weight greater than 2 in order to be moved. This constant weight is derived up the tree using the attribute *iscon* . Each node derives a constant weight (iscon) from each of its subtrees. Each node can then test the constant weights of its subtrees to determine if a subtree should be moved.

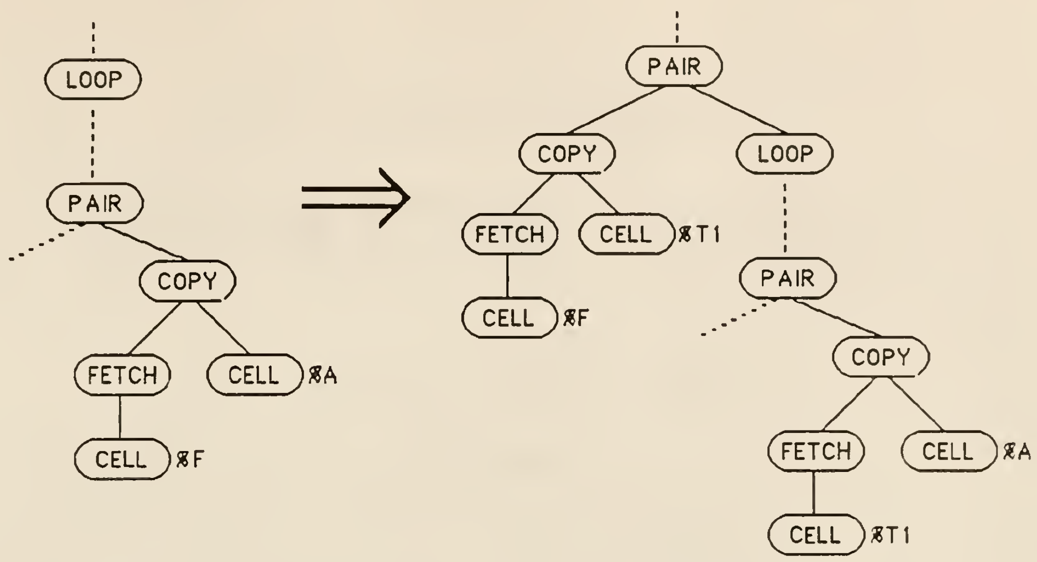


Figure 4.2. Moving an expression that is too small out of a loop.

Reserving Insertion Points

To move a constant item out of a loop, information is needed to indicate where in the tree that it should be glued back in. Figure 4.3 illustrates the transformation that takes place. As can be seen, each loop constant item is spliced into the tree at a new PAIR node above the loop.

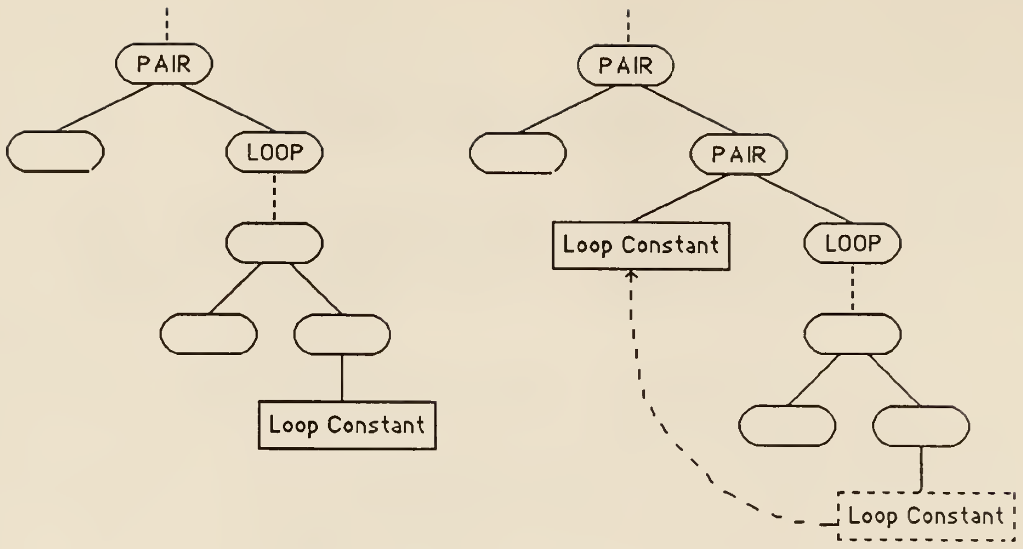


Figure 4.3. Moving a loop-constant item.

Each node in the loop inherits a link to the PAIR node where the loop-constant code should be moved to. As code is moved out of the loop this pointer is updated to point to a new insertion point. The attribute *gluept* carries this pointer.

Moving a loop-constant expression is a little more involved than moving a statement. Figure 4.4 illustrates the moving out of a loop-constant expression. Not only is the expression moved out, but it is also assigned to a new variable cell. Thus, for this new cell, a DCLN node must be created and glued into the tree. In LoopCon, instead of inserting this new DCLN node right into the middle of the program (as in figure 4.4), a pointer to an insertion point in the declaration block (of the current routine) is inherited and the new DCLN is glued in there. This pointer is derived at the beginning of each subroutine by function getdec and is carried by the attribute *decpt* . Like gluept, decpt is inherited by each node. As new DCLNs are created this pointer is updated to point to a new insertion point.

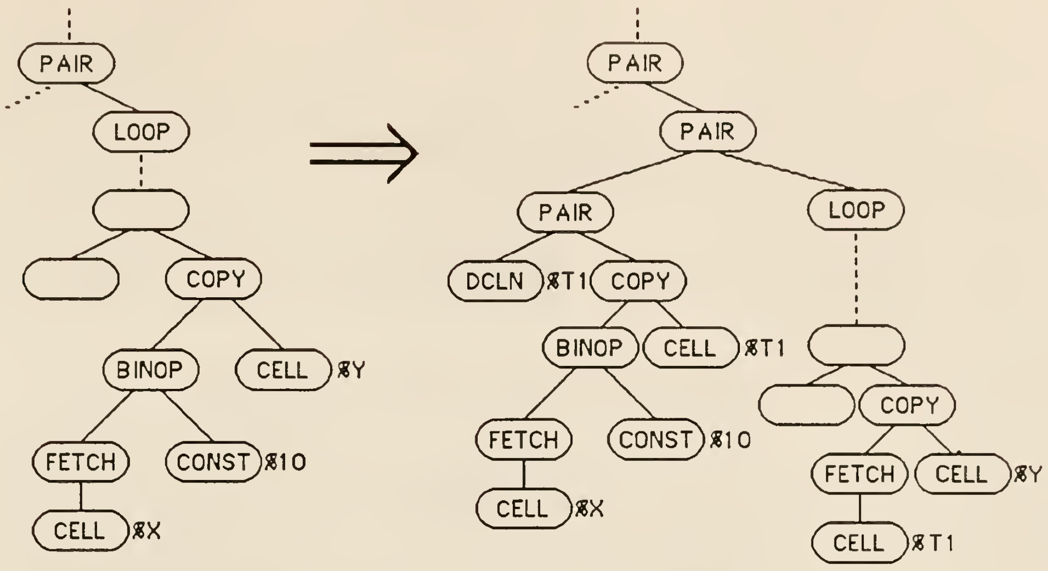


Figure 4.4. Moving a loop-constant expression.

Moving Loop-Constant Expressions

The simplest of expressions is an expression consisting of a single datum. This item is a leaf node in the tree (a CONSTANT node or a FETCH to a CELL). As previously described, by itself it is not large enough to move and is given a constant weight of 2. However, a constant expression may exist in the address calculation subtree of the FETCH node.

```
STMT ~notcon ~twice ~gluept ~decpt ^newgluept ^newdecpt ^iscon
    -> F: < FETCH  V:tree > %varid
            {V:STMT ~notcon ~twice ~gluept ~decpt ^vgluept ^vdecpt ^viscon}
            {viscon >> 2   varid in notcon  vgluept >< NULL
                => # variable is not loop constant but cell calculation is
                    {vgluept:exmoveit ~<TYPE <CONSTANT> <CONSTANT>>
                        ~V ~vdecpt ^newdecpt ^newgluept ^decnode}
                    {decnode -> varno}  # get new variable id
                    {C <- varno}  # decorate new cell with its variable id
                    {C: decorate ~decnode}  # decorate new cell with its DCLN
                    {E <- varno}  # decorate new fetch with its variable id
                    {E: decorate ~decnode}  # decorate new fetch with its DCLN
                    ^iscon : 0   # constantness of this node is returned as zero
            viscon == 2  # the expression was just a CELL
                => {varid in notcon  # variable is not loop constant
                        => ^iscon : 0  # this node is not constant
                        otherwise # this node is constant with a CELL
                            => ^iscon : 2
                        }
                      ^newdecpt : vdecpt
                      ^newgluept : vgluept
             otherwise  # this whole node may be loop constant
                   => ^iscon : viscon  # return constant weight
                        ^newgluept : vgluept
                        ^newdecpt : vdecpt
             }

    => < FETCH  E: < FETCH  C: < CELL >>> |  F | F
```

The expression tree (address calculation) is walked and its constant weight (viscon) is returned. If the constant weight is greater than 2 then the subtree can be moved. However, it can only be moved out if the variable is not loop constant and of course only if currently in a loop (gluept will not be NULL if in a loop). If the variable's id appears in the notcon set that was created in the analysis pass, then it is not loop constant. If the variable is loop constant (the otherwise alternative is taken) then the FETCH node itself is loop-constant and will be moved out at a node higher up in the tree. If viscon is 2 then the subtree was a CELL node and we must test to see if that variable is loop-constant. Otherwise, the subtree is constant or doesn't weigh enough or the whole node is loop-constant.

If the first alternative is taken, the expression subtree is moved with function exmoveit and a pointer to the new DCLN node is returned. The new variable id is retrieved and the new FETCH and CELL nodes created in the transformation part are decorated with the new DCLN and the new variable id.

An operator node is handled a little differently. A BINOP node has two subtrees whose constant weights must be checked and whether or not a subtree is moved depends on not only its own weight, but also on the constant weight of the other subtree.

```
STMT ~notcon ~twice ~gluept ~decpt ^newgluept ^newdecpt ^iscon
    -> B: < BINOP  L:tree  R:tree >
         {B: examine ^tipe}
         {L: STMT ~notcon ~twice ~gluept ~decpt ^lgluept ^ldecpt ^liscon}
         {R: STMT ~notcon ~twice ~lgluept ~ldecpt ^rgluept ^rdecpt ^riscon}
         {liscon >> 2  riscon == 0  gluept >< NULL #left const, right not, in loop
             =>  {rgluept:exmoveit ~tipe ~L ~rdecpt^newgluept^newdecpt^dec}
                 {dec -> varno} # retrieve varible id from new DCLN node
                 {C <- varno} # decorate new cell with id
                 {C: decorate ~dec} # decorate new cell with DCLN node
                 {F <- varno} # decorate new FETCH with variable id
                 {F: decorate} # decorate new FETCH with DCLN node
         riscon >> 2  liscon == 0  gluept >< NULL #right const, left not, in loop
             =>  {rgluept:exmoveit ~tipe ~R ~rdecpt^newgluept^newdecpt^dec}
                 {dec -> varno} # retrieve varible id from new DCLN node
                 {C <- varno} # decorate new cell with id
                 {C: decorate ~dec} # decorate new cell with DCLN node
                 {F <- varno} # decorate new FETCH with variable id
                 {F: decorate} # decorate new FETCH with DCLN node
         otherwise
             =>  ^newgluept : rgluept
                 ^newdecpt : rdecpt
         }
         ^iscon : liscon*riscon # increase expression weight

    =>   < BINOP  F:< FETCH  C:< CELL >>  R > |
         < BINOP  L  F:< FETCH  C:< CELL >>> |  B
```

In order for one subtree to be moved, its constant weight must be large enough and the other subtree must not be constant. If both subtrees have a constant weight then the BINOP node becomes constant and the constant weight is increased and returned. The transformations performed in the first two alternatives are the same as the transformation performed at the FETCH node.

A UNOP node has only one subtree and no significant analysis is required. Because a unary operator will not change the constantness of its operand, the constant weight of the UNOP node is simply returned, reflecting the constantness of its subtree (The weight does increase though if the subtree is constant).

Moving Loop-Constant Statements

Moving a loop-constant statement out of a loop is not quite as involved as moving an expression. However, the analysis involved is a little more detailed. In order for the statement to be loop-constant, all expressions in the statement must be loop-constant, the statement must not be in a branch of a fork and if it is an assignment statement, the variable it is assigning cannot be assigned by any other statement in the loop.

In LoopCon, since the constant weight (iscon) of a subtree is returned to the parent node, it is easy to test each subtree of a statement to determine if they are all constant. To determine if a variable being assigned to, is assigned to more than once in the loop, the twice set which was derived in the analysis pass of the loop, can be checked. If the variable's id occurs in the set then the statement is not constant. Finally, to determine if a statement is in a fork branch, the attribute *inIF* is used. InIF is passed down as 1 in the subtrees of an IF, and in any other case is passed as 0. The following production is an abbreviated version of the production for the COPY node in LoopCon to illustrate the analysis.

```
STMT ~notcon ~twice ~gluept ~inIF ~decpt ^newdecpt ^newgluept ^iscon
    -> CY: < COPY  L:tree  R:tree > %varid
        {L:STMT ~notcon ~twice ~gluept ~0 ~decpt ^ldecpt ^lgluept ^liscon}
        {R:STMT ~notcon ~twice ~lgluept ~0 ~ldecpt ^rdecpt ^rgluept ^riscon}
        {liscon >> 2   riscon == 0   gluept >< NULL
            => # left subtree is constant but right isn't and in loop
               . . . move left subtree (expression)
              ^iscon : 0
         liscon == 0   riscon >> 2   gluept >< NULL
            => # right subtree is constant but right isn't and in loop
               . . . move right subtree (cell calculation)
              ^iscon : 0
         liscon >> 2   riscon == 2   varid in twice   gluept >< NULL
            => # both trees const but right not big enough & var assigned twice
               . . . move right subtree (cell calculation)
              ^iscon : 0
         liscon >> 2   riscon >> 2   varid in twice   gluept >< NULL
            => # both subtrees constant and var assigned twice
```

```
                . . .  move right subtree (cell calculation)
                . . .  move left subtree (expression)
               ^iscon : 0
        otherwise
            => {inIF == 0   # not in branch of a fork
                  => ^iscon : liscon * riscon
                otherwise  # in a fork branch
                  => ^iscon : 0
               }
        }
```

In the first four alternatives, either one of the subtree expressions is not constant, or the variable being assigned to occurs in the twice set. Consequently, the constant weight for the COPY node (iscon) is returned as 0. In the last alternative (otherwise), inIF is tested to determine if this node is in a fork branch. If so, the constant weight is returned as 0. If all analysis tests are passed (both subtrees are loop-constant, varid is not assigned more than once in the loop and this node is not in an IF subtree) then a constant weight is returned and the statement (COPY node) will be moved from somewhere higher up in the tree.

Loop-constant statements (blocks) are detected and moved at a PAIR node.

```
STMT ~notcon ~twice ~gluept ~inIF ~decpt ^newdecpt ^newgluept ^iscon
    -> P: < PAIR  L:tree  R:tree >
          {L:STMT ~notcon ~twice ~gluept ~inIF ~decpt ^ldecpt ^lgluept ^liscon}
          {R:STMT ~notcon ~twice ~lgluept ~inIF ~ldecpt ^ndecpt ^rgluept
              ^riscon}
          {liscon >> 2   riscon == 0   gluept >< NULL
              => {rgluept: stmoveit ~L ^newgluept}
           liscon == 0   riscon >> 2   gluept >< NULL
              => {rgluept: stmoveit ~R ^newgluept}
           otherwise
              => ^newgluept : rgluept
          }
          ^iscon : liscon* riscon   # increase expression weight
          ^newdecpt : rdecpt
    => < PAIR  <>  R >  |  < PAIR  L  <>>  |  P
```

The actions taken at a PAIR node are similar to those at a BINOP node. The constant weights of the subtrees are tested and a subtree is only moved if the other subtree is not loop-constant. If both subtrees have some weight or neither subtree was big enough or neither subtree was loop-constant, then no transformation is made and iscon is returned reflecting the constantness of the PAIR subtrees (^iscon : liscon*riscon).

Rewalking Moved Code

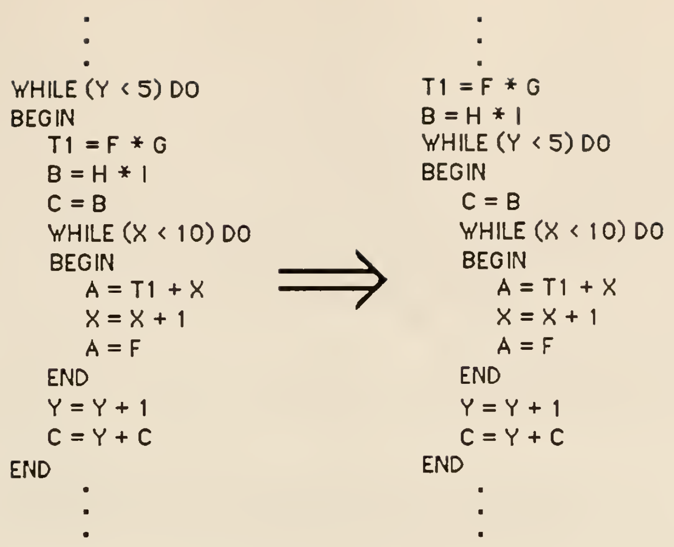
In the case of nested loops, items that are loop-constant in the inside loop may also be loop-constant in the outside loop. In figure 4.5(a) loop-constant items of the inside loop are moved out. The constantness of the items that were moved can then be checked in the outside loop. If they are constant in the outside loop they can also be moved to the outside of it, as illustrated in figure 4.5(b).

```
        .
        .
        .
WHILE (Y < 5) DO
BEGIN
    WHILE (X < 10) DO
    BEGIN
        A = F * G + X
        B = H * I
        X = X + 1
        A = F
        C = B
    END
    Y = Y + 1
    C = Y + C
END
        .
        .
        .
```

$\Longrightarrow$

```
        .
        .
        .
WHILE (Y < 5) DO
BEGIN
    T1 = F * G
    B = H * I
    C = B
    WHILE (X < 10) DO
    BEGIN
        A = T1 + X
        X = X + 1
        A = F
    END
    Y = Y + 1
    C = Y + C
END
        .
        .
        .
```

(a)



```
        .
        .
        .
WHILE (Y < 5) DO
BEGIN
   T1 = F * G
   B = H * I
   C = B
   WHILE (X < 10) DO
   BEGIN
      A = T1 + X
      X = X + 1
      A = F
   END
   Y = Y + 1
   C = Y + C
END
        .
        .
        .
```

$\Longrightarrow$

```
        .
        .
        .
T1 = F * G
B = H * I
WHILE (Y < 5) DO
BEGIN
   C = B
   WHILE (X < 10) DO
   BEGIN
      A = T1 + X
      X = X + 1
      A = F
   END
   Y = Y + 1
   C = Y + C
END
        .
        .
        .
```

(b)

Figure 4.5. Moving constant items out of nested loops.

In LoopCon, after a loop has been analyzed and all loop-constant items have been moved out, the items that were moved are rewalked for the level outside the loop.

```
STMT ~notcon ~twice ~gluept ~inIF ~decpt ^newgluept ^newdecpt ^iscon
    -> < LOOP  B:tree >
        # analysis pass
        {B: FindCon ~Empty ~Empty ~Empty ^notconset ^twiceset}

        # transformation pass
        {B: STMT ~notconset ~twiceset ~P ~0 ~decpt ^ldecpt ^lgluept ^liscon}

        # rewalk the moved code
        {P: rewalk ~notcon ~twice ~gluept ~inIF ~ldecpt ^newdecpt ^newgluept}
        ^iscon : 0
    => P: < PAIR <> < LOOP  B >>
```

As shown in the production for the LOOP node, the tree (starting at node P) that results from moving loop constant code out of the loop is rewalked before returning from the LOOP node.

4.3. Notes On Implementation

4.3.1. Tools Built in TAGs

Since some of the tools that are needed when testing TAG modules perform manipulations on ILTs, they were also designed in TAGs.

A front end was available which would produce an ILT from a source program. However, this front-end (a prototype) did not fully decorate nodes to fit the specifications of ILTs and would only accept integer and boolean types. To fill this gap, a module was designed in a TAG to walk the program tree and "redecorate" its nodes to fit the specifications. Types such as character and real were manually created after the front-end had generated the source ILT. The redecorator would simply change designated variables in the program tree into another type (character, real, etc.). This was a matter of changing the TYPE node decorations on DCLN nodes.

The actual file representation of an ILT consists of a list of septuples of integer numbers. In order to be able to read the output of a TAG, it was converted to a readable form. A tool (pretty printer) was built in a TAG that would walk an ILT and produce a representation of the tree which would show each node and the hierarchical structure in indented or "outline" form. This tool proved to be very useful in determining if nodes were decorated correctly and other precise details.

Once one is no longer concerned with low-level details such as node names and decorations, the output can be transformed into a higher-level representation. To perform this conversion another tool was designed in a TAG (a decompiler) that would simply convert the ILT back into its original high-level language (in this case; Pascal). With this representation it is easy to tell if the proper optimizations were performed on the program by comparing it with a listing of the original source program.

4.3.2. The Testing Process

The process of compiling and testing each TAG involved several different modules. Figure 4.6 illustrates the process of compiling and testing a TAG.

Compiling a TAG actually consists of two compilations. First, the TAG is compiled by the TAG compiler which produces Pascal code. The Pascal code is then compiled into object code. The resulting module is the executable optimizer.

To test each optimizer, a test program was converted by the front end into an ILT. The source ILT was then redecorated with the redecorating module and the resulting ILT could then be pretty-printed into a readable form before optimization. The optimizer being tested would then perform optimization on the test program and produce an optimized ILT. This final ILT was then converted into a readable form (using the pretty-printer and/or the decompiler) and compared with the representation before optimization to determine if the optimizer performed correctly.

Debugging a TAG was not hard due to the clarity of the notation. Logic errors were the hardest to find. When a TAG is executed it generates a rather large file in which information is recorded concerning the execution (provided that the debug flag is set when the TAG is compiled). The information recorded in this "optimizer-trace-information" file consists of node names as they are visited, predeclared function names when they are called, and pseudo-nonterminal names as they are invoked (and exited) along with the value of their attributes. Also, "DEBUGging" functions (in the predeclared module) could be called to print various items, such as the contents of sets or attribute values, at any point during execution. This information could be used to determine what actions were taken at particular points in the tree and together with the resulting tree (pretty-printed ILT), errors could be detected.

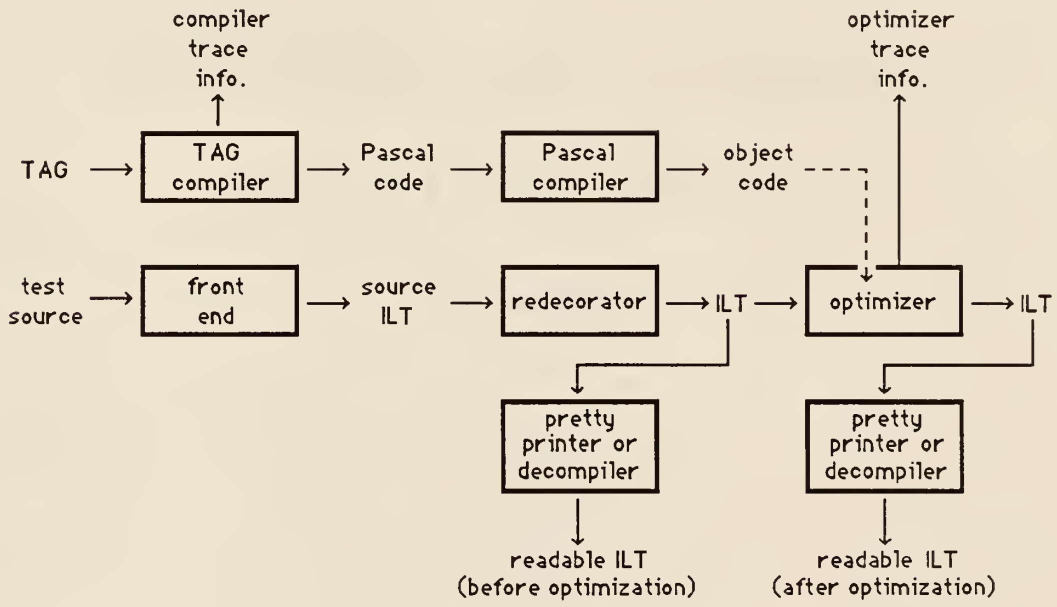


Figure 4.6. Compiling and testing an optimizer.

Chapter Five
Conclusion

5.1. Findings

The application of TAGs to ILTs is presented in this work as an efficient method for the specification and implementation of optimizations in a compiler. Each optimization typically requires phases of analysis and transformation, and TAGs function to specify both phases. TAGs utilize the power of an attribute grammar to pass information through attributes and the power of a transformational grammar to make transformations based upon attribute evaluation.

The TAG notation is unique. Understanding the concept of the underlying grammar and how transformations are performed on ILTs are the only prerequisites to writing optimizers in a TAG. The TAG language is versatile. Not only are nonterminals of the underlying CFG included as functions, but also pseudo-nonterminals (nonterminals which are not part of the underlying CFG) needed for attribute evaluation can be.

Any information which is derivable from the program tree (including nodes of the tree) can be passed by attributes to wherever it is needed. Thus, remote transformations on noncontiguous parts of the program can be performed with TAGs. As in the removal of loop constants (illustrated in section 4.2), a reference to a node in the tree can be passed as an attribute to a different point in the tree where transformations on that node can be performed.

The level of detail included in the intermediate program representation has a significant effect on the amount of effort spent in optimization algorithms. The information contained in the intermediate representation of ILTs makes it superior to those of quads and other forms of abstract syntax trees. The high-level structure of the source program required for machine independent optimizations and the low-level detail needed for machine dependent optimizations are both preserved in an ILT. Very little effort is spent on reconstructing the original program in optimizers (designed with TAGs) that use ILTs as an intermediate language and consequently, the complexity of the optimization algorithm is much less than that of algorithms which use other methods.

A consistent program representation between different optimization phases is necessary for providing the ability to reapply optimization phases. Since ILTs can represent the information needed for most types of optimization, each optimization phase operates on an ILT and produces a tree in the same representation. The initial ordering of optimization phases and the ability to reiterate phases are at the discretion of the compiler designer.

TAGs provide the necessary framework for the compiler writer and decompose the optimization process into easily manageable components. The application of TAGs to ILTs provides an efficient method for the timely development of highly optimizing, correct compilers.

5.2. Future Work

The optimizations of constant folding, dead code elimination and loop constant removal were implemented in this project using TAGs. A complete optimizer of course would include several other optimization phases. The opportunity therefore exists to implement other optimizations in TAGs as well.

Other optimizations could also be added to the optimizers implemented in this work. In ConsFold there is a unique opportunity to include the optimization of back substitution (replacing a procedure call with the procedure body). The optimizations of loop unrolling (eliminating the loop overhead by unrolling a small loop into sequential code) and loop fusion (joining two or more loops together which have the same number of iterations) could also be included in LoopCon.

Bibliography

1. Aho, A. V., Sethi, R., Ullman, J. D., "A Formal Approach to Code Optimization", SIGPLAN Symposium on Compiler Opimization, Vol 5, No. 7, 1970. New York NY.

2. Aho, A. V., Sethi, R., Ullman, J. D., "Code Optimization and Finite Church-Rosser Systems", Design and Optimization of Compilers, (1972), Rustin, R. (ed.), pp 89-106.

3. Aho, A. V., Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, New York, NY 1985.

4. Allen, F., Cocke, J., "A Catalogue of Optimizing Transformations", Design and Optimization of Compilers, (1972), Rustin, R. (ed.), pp 1-30.

5. Barrett, W. A., Bates, R. M., Gustafson, D. A, Couch, J. D., *Compiler Construction Theory and Practice* , 2nd edition, Science Research Associates, 1986.

6. Bassett, S., "Multipass Compilers Produce Tight Code", Computer Design, Vol 23, No. 1, Jan 1984, pp 44-47.

7. Bauer, F. L., Eickel, J., *Compiler Construction: An Advanced Course*, 2nd ed. Lecture Notes in Computer Science, Springer-Verlag, New York 1976.

8. Burkhard, N. A., "Machine-Independent C Optimizer", SIGPLAN Notices, Vol 20, No. 11, Nov 1985, pp 23-26.

9. Cattell, R. G. G., Newcomer, J. M., Leverett, B. W., "Code Generation in a Machine Independent Compiler", SIGPLAN Symposium on Compiler Construction, 1979, Denver, Colorado, pp 65-75.

10. Cocke, J., Markstein, P. W., "Measurement of Program Improvement Algorithms", Information Processing, North-Holland, Amsterdam 1980, pp 221-228.

11. DeRemer. F. L., "Transformational Grammars", Bauer, F. L., Eickel,J. (editors), Compiler Construction: An Advanced Course, 2nd ed., Lecture Notes in Computer Science, Springer-Verlag, New York 1974, pp 121-145.

12. Fairman, R. N., Kortesoja, A. A., "An Optimizing Pascal Compiler", Proceedings of COMPSAC, IEEE 1979, New York, pp 624-628.

13. Feign, D., "A Note on Loop Optimization", SIGPLAN Notices, Vol 14, No. 11, Nov 1979, pp 23-25.

14. Frailey, D. J., "An Intermediate Language for Source and Target Independent Code Optimization", SIGPLAN Notices, Vol 14, No. 8, Aug 1979, pp 188-200.

15. Ganapathi, M., Fischer, C. N., "Description-Driven Code Generation Using Attribute Grammars", Ninth Annual ACM Symposium on Principles of Programming Languages, 1980, Albequerque, NM, pp 108-119.

16. Ganzinger, H., Giegrich, R., "Attribute Coupled Grammars", SIGPLAN Symposium on Compiler Construction, 1984, Montreal, pp 157-170.

17. Ganzinger, H., Ripken, K., Wilhelm, R., "Automatic Generation of Optimizing Multipass Compilers", Proceedings of the IFIP 1977 Congress, American North-Holland, 1977, pp 535-540.

18. Goldgerg, P. C., "A Comparison of Certain Optimization Techniques", Design and Optimization of Compilers, (1972), Rustin, R. (ed.), pp 31-50.

19. Goss, C. F., Meretzky, M. S., Pollak, B., "Optimizing Compilers", UNIFORUM 1986 Conference Proceedings, 4-7 February, Anaheim, CA., (Santa Clara, CA, /usr/group 1986), pp 125-140.

20. Harbison, S. P., "Architectural Alternative to Optimizing Compilers", SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, Vol 17, No. 4, April 1982, pp 57-65.

21. Harrison, W., "A New Strategy For Code Generation - The General Purpose Optimizing Compiler", IBM Watson Research Center, 1976.

22. Hunter, R., *Compilers : Their Design and Construction Using Pascal*, Wiley, New York, 1985.

23. Jazayer, M., Hayden, M., "Optimizing Compilers Are Here (Mostly)", SIGLAN Notices, Vol 21, No. 5, May 1986, pp 61-63.

24. Jullig, R. K., DeRemer, F. L., "Regular Right-Part Attribute Grammars",SIGPLAN Symposium on Compiler Construction, Vol 19, No. 6, June 1984, pp 171-178.

25. Keller, S. E., Perkins, J. A., Payton, T. F., Mardinly, S. P., "Tree Transformation Techniques and Experiences", SIGPLAN Symposium on Compiler Construction, Vol 19, No. 6, June 1984, pp 190-200.

26. Knuth, D. E., "Semantics of Context-Free Languages", Mathematical Systems Theory, Vol 2, No. 2, Springer-Verlag, New York, 1968, pp 127-145.

27. Knuth, D. E., "An Empirical Study of Fortran Programs", Software Practice and Experience, Vol 1, No. 2, 1971, pp 105-133.

28. Koskimies, K., Raika, K. J., Sarjakoski, M., "Compiler Construction Using Attribute Grammars", SIGPLAN Notices, Vol 17, No. 6, June 1982, pp 153-159.

29. Landwehr, R., Jansohn, H., Goos, G., "Experience with an Automatic Code Generator Generator", SIGPLAN Notices, Vol 17, No. 6, June 1982, pp 56-66.

30. Lee, J. A., *The Anatomy of a Compiler* (2nd ed.), Van Nostrand Reinhold, 1974.

31. Lewi, J., DeVlaminick, K., Huens, J., Steegmans, E., *A Programming Methodology in Compiler Construction -- Part II : Implementation* , North-Holland, 1983.

32. Lewis, P. M., Rosenkrantz, D. J., Stearns, R. E., *Compiler Design Theory* , Addison-Wesley, 1976.

33. Markstein, V., Cocke, J., Markstein, P., "Optimization of Range Checking", SIGPLAN Symposium on Compiler Construction, Vol 17, No. 6, June 1982, pp 114-119.

34. McKeeman, W. M., "Compiler Construction", Compiler Construction, Bauer, F. L., Eickel, J. (editors), Springer-Verlag, 1976, pp 1-36.

35. Mintz, R. J., Fisher, G. A., Sharir, M., "The Design of a Global Optimizer", SIGPLAN Symposium on Compiler Construction, Denver, Colorado, 1979, pp 226-234.

36. Nakata, I., Sassa, M., "L-attributed LL(1) Grammars are LR-attributed", INF. PROCESS. LETT. (Netherlands), Vol 23, No. 6, 3 Dec 1986, pp 325-328.

37. Peters, P. S., Ritchie, R. W., "On the Generative Power of Transformational Grammars", Information Science, Vol 6, 1973.

38. Pittman, T., "Using Transformational Attribute Grammars For Code Optimization", Kansas State University, 1987.

39. Pittman, T., "Practical Code Optimization by Transformational Attribute Grammars Applied to Low-Level Intermediate Code Trees, PhD Thesis, Univ. of California at Santa Cruz, 1985.

40. Pollak, B. W., *Compiler Techniques* , Averbach, 1972.

41. Powell, M. L., "A Portable Optimizing Compiler for Modula-2", SIGPLAN Sumposium on Compiler Construction, Vol 19, No. 6, June 1984, pp 310-318.

42. Reif, J. H., Lewis, H. R., "Efficient Symbolic Analysis of Programs", Journal of Computer and System Sciences, Vol 32, pp 280-314.

43. Rosen, B. K., "Tree-Manipulation Systems and Church-Rosser Theorms", Journal of ACM, Vol 20, No. 1, 1973, pp 160-187.

44. Rudmick, A., Lee, E. S., "Compiler Design for Efficient Code Generation and Program Optimization", SEGPLAN Symposium on Compiler Construction, Denver, Colorado, 1979, pp 127-138.

45. Rustin, R. (editor), *Design and Optimization of Compilers* (Courant Computer Science Symposium #5), Prentiss-Hall, 1972.

46. Sassa, M., "ECLR-Attributed Grammars: A Practical Class of LR-Attributed Grammars", INF. PROCESS. LETT. (Netherlands), Vol 24, No. 1, 15 Jan 1987, pp 31-41.

47. Spector, D., Turner, P. K., "Limitations of Graham-Glanville Style Code Generation", SIGPLAN Notices, Vol 22, No. 2, Feb 1987, pp 100-108.

48. Symposium on Compiler Optimization, SIGPLAN Notices, Vol 5, No. 7, 1970.

49. Teuris, P. M., Rosenkrantz, D. J., Stearns, R. E., *Compiler Design Theory* , Addison-Wesley, 1976.

50. Tiennari, M., "On the Definition of an Attribute Grammar", Semantics Directed Compiler Generation (Lecture Notes in Computer Science), Springer-Verlag, 1980, pp 408-414.

51. Turner, P. K., "Up-Down Parsing With Prefix Grammars", SIGPLAN Notices, Vol 21, No. 12, Dec 1986, pp 167-174.

52. Waite, W. M., "Optimization", Bauer, F. L., Eickel, J. (ed) Compiler Construction, Springer-Verlag, 1976, pp 549-602.

53. Weber, J., Program Transformation With Attributed Transformational Grammars. Tech. Univ. München 7604, 1976.

54. Wolfe, M., Macke, T., "Where are the Optimizing Compilers?", SIGPLAN Notices, Vol 20, No. 11, Nov 1985, pp 64-68.

55. Wolfgang, Polak, *Compiler Specification & Verification* . (Lecture Notes in Computer Science), Springer-Verlag, 1981.

56. Wulf, W. A., Johnsson, R. K., Weinstock, C. B., Hobbs, S. Q., *The Design Of An Optimizing Compiler* . American Elsevier, New York, 1975.

57. Wyrostek, P., "On the Size of Unambiguous Context-Free Grammars", THEOR. COMPUT. SCI. (Netherlands), Vol 47, No. 1, 1986, pp 107-110.

Appendix A

Predeclared Values and Functions

Predeclared values and functions are encoded seperately from a TAG in a "predeclared file". These functions and values are needed by the TAG compiler and for attribute evaluations in a TAG. When a TAG is compiled, the predeclared-module is included in the compilation. The values and functions defined below are those generally needed by the TAG designer. The TAG designer may however, define his own values and functions in the predeclared file.

Values/attributes

Empty	-- May be used as an empty set, an empty table or a reference to no tree.
Universe	-- A special set that contains every set as a subset
True, False	-- Distinctive values of true and false.

Functions (Pseudo-nonterminals)

General

- NEWVARNO

    definition : {:newvarno ^varno}

    This function will return the next available variable id number. Generally used when creating a new variable cell.

Set Functions

- UNION

    definition : {:union ~seta ~setb ^unionset}

    This function receives two sets (seta and setb) and returns a set (unionset) which is the union of the two sets.

- INTERSECT

  definition : {:intersect ~seta ~setb ^intersectset}

  Intersect receives two sets (seta and setb) and returns a set (intersectset) which contains the items which were members of boths sets, seta and setb.

- DIFFERENCE

  definition : {:difference ~seta ~setb ^differenceset}

  Difference receives two sets and returns a set (differenceset) which is the difference of the two sets (members of setb are removed from seta).

- ADDSET

  definition : {:addset ~varno ~oldset ^newset}

  Addset receives an item (possibly a variable id number, varno) and a set (oldset) and returns a set (newset) which is the old set with the item (varno) added.

<u>Table Functions</u>

- OPENFRAME

  definition : {:openframe ~Empty ^newtable}

  Openframe is used to initialize a new symbol table.

- INTO

  definition : {:into ~oldtable ~varno ~value ^newtable}

  Into receives a symbol table (oldtable), some item (typically a variable id number, varno) and a value to associate with that item and returns a new symbol table (newtable) which is the result of adding the item and its value to the old table.

- FROM

  definition : {:from ~symtable ~varno ^value}

  From receives a symbol table (symtable) and an item (varno) to be searched for in the table. If the item is in the table then the value which was entered (associated) with that item (value) is returned. If the item was not in the table then the function fails.

<u>Decoration Functions</u>

- DECORATE

  definition : {N:decorate ~item}

  Decorate will attach an item to the specified node (N).

- EXAMINE

  definition : {N:examine ^decoration}

  Examine will retrieve a decoration from a node (N) which was originally placed there with function Decorate.

- PACK4

  definition : {:pack4 ~item1 ~item2 ~item3 ~item4 ^packed}

  Pack4 receives four items, "packs" them into one and returns the compressed item (packed). Because a node may only be decorated with one item (using function Decorate), several items may be packed together using this function when it is desirable to decorate with more than one item.

- UNPACK4

  definition : {:unpack4 ~packed ^item1 ^item2 ^item3 ^item4}

  Unpack4 is the opposite of Pack4. It receives the "packed" item and returns the "unpacked" items (in the same order that they were packed).

NOTE: Decorating tree nodes may be done in three different ways. A node may be decorated using the Decorate function defined above and/or by using one of the following two methods:

to decorate a node with "item"	to retrieve decoration "item"
=> . . . <node . . . > %item	-> . . . < node . . . > %item
{N <- item}	{N -> item}

These two methods can be used interchangably. They may NOT however, be used in conjunction with functions Decorate and Examine.

Debugging Functions

NOTE: The following functions have attributes "~in" and "^out". These attributes are dummy attributes provided only for use in forcing dependencies.

- PRINTNODE

  definition : {N:printnode ~in ^out}

  Printnode will print a picture of the specified node (N) to the screen/trace file. (i. e. if N is a FETCH node then "< FETCH >" will be printed.

- DEBUGNL

  definition : {:DEBUGnl ~in ^out}

  DEBUGnl will send a new line to the screen/trace file.

- DEBUGINT

  definition : {:DEBUGint ~value ~in ^out}

  DEBUGint will print the specified value to the screen/trace file.

- DEBUGCHR

  definition : {:DEBUGchr ~ordinal ~in ^out}

  DEBUGchr will print chr(ordinal) to the screen/trace file.

- DEBUGNODE

  definition : {:DEBUGnode ~node ~in ^out}

  DEBUGnode will print "< integer form of node >" to the screen/trace file.

- DEBUGSET

  definition : {DEBUGset ~theset ~in ^out}

  DEBUGset will print "[all members of theset]" to the screen/trace file.

Appendix B

Pascal/Quad Constant Folder

```
(*****************************************************************)
(* Pascal/Quad Constant Folder and Dead Code Eliminator.                     *)
(* This program is designed to read in an intermediate program  representation in    *)
(* the form of quads and perform the following constant folding optimizations:      *)
(*         1. Constant Folding                                               *)
(*              - constant expression evaluation                             *)
(*              - constant propagation                                       *)
(*         2. Dead Code Removal                                              *)
(*                                                                           *)
(* Optimization across procedure/function calls is not attempted.           *)
(*                                                                           *)
(* The input file (in the form of quads) should be in the following format:         *)
(*          lable, dest., opcode, s1 constant flag, source1, s2 constant flag, source2  *)
(*                                                                           *)
(* This program will produce an output file called "after" which will also be in the  *)
(* form of quads.                                                            *)
(*****************************************************************)

program constfold(input,output,after)
;const
    n = 100       (* block list size *)
   ;m = 100       (* variable list size *)
   ;NOP = 0       (* op codes *)
   ;BRA = 1
   ;BRF = 2
   ;RETN = 4
   ;assign = 5
   ;less = 15
   ;greater = 16
   ;equal = 17
   ;lsoreq = 13
   ;groreq = 14
   ;unequal = 18
   ;plus = 6
   ;minus = 7
   ;star = 8
   ;divv = 9
   ;modd = 10
   ;neg = 19
   ;nott = 20
   ;andd = 11
```

```
    ;orr =  12
    ;call = 3
;

 type
    quadfile = text
   ;quadptr = ^quadtype

   ;quadtype = record        (* a quad *)
       labl    : integer       (* label field *)
      ;dest    : integer       (* destination field *)
      ;opcode  : integer       (* op code *)
      ;sflag1  : boolean       (* source1 constant flag *)
      ;source1 : integer       (* source1 field *)
      ;sflag2  : boolean       (* source2 constant flag *)
      ;source2 : integer       (* source2 field *)
      ;qptr    : quadptr       (* pointer to next quad *)
    end (* record quadtype *)

   ;predptr = ^member        (* pointer to predecessor list *)

   ;blocktype = record       (* a block -- [head, tail, left, right] *)
       top     : quadptr       (* pointer to head quad of block *)
      ;bottom  : quadptr       (* pointer to tail quad of block *)
      ;left    : integer       (* block # of left branch - not always used *)
      ;right   : integer       (* block # of right branch *)
      ;list    : predptr       (* pointer to predecessor list *)
    end (* record blocktype *)

   ;array1 = array[1..n] of blocktype   (* the block list *)

   ;member = record          (* one member of the predecessor list *)
       bl      : integer       (* the block number of predecessor *)
      ;nextbl  : predptr       (* link to next pred. *)
    end (* record member *)

   ;vecttype = record                           (* a const/value vector for block *)
       value    : array[1..m] of integer     (* value of variable in block *)
      ;iscon    : array[1..m] of boolean     (* is the var definitly const? *)
      ;notcon   : array[1..m] of boolean     (* is the var definitly not const? *)
    end (* record vecttype *)

   ;array2 = array[1..n] of vecttype
   ;array3 = array[1..n] of boolean
;

 var
    after     : quadfile       (* output file *)
   ;head      : quadptr        (* pointer to head quad (first quad of prg.) *)
   ;blocks    : array1         (* program blocks *)
   ;numblocks : integer        (* the number of basic blocks *)
   ;vect      : array2          (* const/value vector for analysis *)
   ;visited   : array3          (* tells if block has been visited in analysis *)
   ;curblock  : integer         (* the current block number *)
   ;change    : boolean         (* false until program analysis stabilizes *)
;

(********************************************************************)
(* The following procedure is designed to read in the list of quads from the input file. *)
(* It reads them into a linked list.  This procedure is called by the main program.       *)
(********************************************************************)

 procedure readquads
;
 var
    dumb1   : integer          (* dummy for input *)
   ;dumb2   : integer          (* dummy for input *)
   ;curptr  : quadptr          (* pointer to current quad *)
   ;newptr  : quadptr          (* another quad pointer *)
;

 begin
    new(head)
   ;with head^ do
    begin
       readln(labl,dest,opcode,dumb1,source1,dumb2,source2)
      ;sflag1:= odd(dumb1)
      ;sflag2:= odd(dumb2)
    end (* with *)
   ;new(curptr)
   ;head^.qptr:= curptr
   ;while not eof do
    begin (* read in quads *)
       with curptr^ do
       begin
          readln(labl,dest,opcode,dumb1,source1,dumb2,source2)
         ;sflag1:= odd(dumb1)
         ;sflag2:= odd(dumb2)
       end (* with *)
      ;if not eof then
       begin
          new(newptr)
         ;curptr^.qptr:= newptr
         ;curptr:= newptr
       end (* if then *)
    end (* while *)
 end (* procedure readquads *)
;

(********************************************************************)
(* The following procedure is designed to identify the basic blocks in the quad.  It is   *)
(* called by the main program.                                              *)
(**************************************************************************)

 procedure identblocks
;
 var
    curquad    : quadptr  (* the current quad pointer *)
   ;endfound  : boolean (* true if the end of a block is found *)
;

 begin
    curblock:= 1
   ;curquad:= head
   ;blocks[1].top:= head  (* start of first block is first quad *)
   ;while curquad^.qptr <> nil do
    begin
       endfound:= false
      ;while not endfound do
       begin (* find end quad of block *)
          if curquad^.qptr = nil then
             endfound:= true
          else
             if (curquad^.qptr^.labl <> 0)or(curquad^.opcode in [0,BRA,BRF]) then
                endfound:= true
             else
                curquad:= curquad^.qptr
       end (* while *)
      ;blocks[curblock].bottom:= curquad (* record pointer to tail of block *)
      ;if curquad^.qptr <> nil then
       begin (* we have not found the last quad *)
          curquad:= curquad^.qptr
         ;curblock:= curblock+1
         ;blocks[curblock].top:= curquad
       end (* if then *)
    end (* while *)
   ;blocks[curblock].bottom:= curquad (* record end of last block *)
   ;numblocks:= curblock
 end (* procedure identblocks *)
;

(**************************************************************************)
(* The following procedure is called upon by procedure buildtree to record a pred-   *)
(* ecessor to the current block.                                            *)
(**************************************************************************)

 procedure recpred(x,pred:integer)
;
 var
    ptr1  : predptr (* pointer in the predecessor list *)
;
```

```
begin
   if blocks[x].list^.bl = 0 then
      blocks[x].list^.bl:= pred (* record predecessor *)
   else
   begin (* there is already a pred. recorded *)
      ptr1:= blocks[x].list
     ;while ptr1^.nextbl <> nil do
         ptr1:= ptr1^.nextbl (* find the next available slot *)
     ;new(ptr1^.nextbl)
     ;ptr1^.nextbl^.bl:= pred (* record predecessor *)
   end (* if else *)
end (* procedure recpred *)
;

(*************************************************************** *)
(* The following procedure is designed to analize the block list and link it togetherby *)
(* following the flow of execution.  It is called by the main program and calls pro-    *)
(* cedure recpred to record predecesors for each block                                  *)
(*************************************************************** *)

procedure buildtree
;
var
   tail      : quadptr (* pointer to the end quad of the current bl. *)
  ;jump      : integer (* holds the block number dest. of a branch *)
  ;i         : integer (* index and induction var. *)
;

begin
   curblock:= 1
  ;while curblock <= numblocks do
   begin (* go through each block *)
      blocks[curblock].left:= 0   (* intitialize *)
     ;blocks[curblock].right:= 0   (* intitialize *)
     ;tail:= blocks[curblock].bottom
     ;if (tail^.opcode = BRA) or (tail^.opcode = BRF) then
      begin
         jump:= tail^.dest
        ;if tail^.opcode = BRF then
         begin (* record pred. *)
            blocks[curblock].right:= curblock+1 (*record con. to next blk*)
           ;recpred(curblock+1,curblock) (* curblock is pred to curblck+1*)
         end (* if then *)
        ;i:= 1
        ;while jump <> blocks[i].top^.labl do
            i:= i+1
        ;blocks[curblock].left:= i  (* record block number of branch *)
        ;recpred(i,curblock) (* records i as predecessor to curblock *)
      end (* if then *)
      else (* fall through to next block *)
         if curblock <> numblocks then
```

```
          begin (* don't do this for last block *)
             blocks[curblock].left:= curblock+1
            ;recpred(curblock+1,curblock)
          end (* if else *)
      ;curblock:= curblock+1
   end (* while *)
end (* procedure buildtree *)
;


(******************************************************************* *)
(* The following procedure is called by the main program to initialize the const-       *)
(* ant/value vectors for each block before the constant analysis begins in procedure  *)
(* traverse.                                                                             *)
(******************************************************************* *)

procedure initvects
;
var
   i    : integer (* loop and index var *)
;

begin
   curblock:= 1
  ;while curblock <= numblocks do
   begin
      new(blocks[curblock].list) (* used later in buildtree *)
     ;blocks[curblock].list^.bl:= 0
     ;for i:= 1 to m do
      begin
         vect[curblock].iscon[i]:= false (* initialize for traverse *)
        ;vect[curblock].notcon[i]:= false (* initialize for traverse *)
      end (* for to do *)
     ;curblock:= curblock+1
   end (* while *)
end (* procedure initvects *)
;


(******************************************************************* *)
(* The following procedure is called by procedure traverse and procedure foldconst   *)
(* to calculate a working vector for the current block from its predecessor's vectors.  *)
(******************************************************************* *)

procedure calcvect(curblock:integer;var blvect:vecttype)
;
var
   ptr1    : predptr (* pointer in predecessor list *)
  ;i       : integer (* loop and index var *)
;
```

```
begin
   if blocks[curblock].list^.bl <> 0 then
   begin (* there is a predecessor *)
      ptr1:= blocks[curblock].list
     ;for i:=1 to m do
      begin (* transfer first pred to working vector *)
         blvect.value[i]:= vect[ptr1^.bl].value[i]
        ;blvect.iscon[i]:= vect[ptr1^.bl].iscon[i]
        ;blvect.notcon[i]:= vect[ptr1^.bl].notcon[i]
      end (* for to do *)
     ;ptr1:= ptr1^.nextbl
     ;while ptr1 <> nil do
      begin (* for each pred. do *)
        for i:= 1 to m do
        begin
          case blvect.iscon[i] of
            true :case vect[ptr1^.bl].iscon[i] of
                   true :if blvect.value[i]<>vect[ptr1^.bl].value[i] then
                        begin
                           blvect.iscon[i]:= false
                          ;blvect.notcon[i]:= true
                        end (* if then *)(*else iscon=T & already mark*)
                  ;false:if vect[ptr1^.bl].notcon[i] then
                        begin
                           blvect.iscon[i]:= false
                          ;blvect.notcon[i]:= true
                        end (* if then *)(*else iscon=T for now & al- *)
                  end (* case *)                (* ready marked *)
           ;false:case vect[ptr1^.bl].iscon[i] of
                   true :if not blvect.notcon[i] then
                        begin
                           blvect.iscon[i]:= true
                          ;blvect.value[i]:= vect[ptr1^.bl].value[i]
                        end (* if then *)(*can't have iscon=T & not=T *)
                  ;false:if vect[ptr1^.bl].notcon[i] then
                           blvect.notcon[i]:= true
                  end (* case *)
          end (* case *)
        end (* for to do *)
       ;ptr1:= ptr1^.nextbl
      end (* while *)
   end (* if then *)
end (* procedure calcvect *)
;

(*****************************************************************)
(* The following function is called by procedure whichop to calculate the value of an *)
(* expression when it has been determined that source1 and source2 of a quad are      *)
(* both constant. This function receives the operator and both source values in order *)
(* to do the calculation and returns the calculated value.                            *)
(*****************************************************************)
```

```
 function calc(code,s1,s2:integer): integer
;

 begin
    case code of
        star,andd : calc:= s1*s2
       ;plus       : calc:= s1+s2
       ;minus      : calc:= s1-s2
       ;divv       : calc:= s1 div s2
       ;modd       : calc:= s1 mod s2
       ;orr        : calc:= ord(s1+s2 > 0)
       ;less       : if s1 < s2 then
                          calc:= 1 (* true *)
                       else
                         calc:= 0 (* false *)
       ;greater   : if s1 > s2 then
                         calc:= 1 (* true *)
                     else
                         calc:= 0 (* false *)
       ;equal     : if s1 = s2 then
                         calc:= 1 (* true *)
                    else
                         calc:= 0 (* false *)
       ;lsoreq    : if s1 <= s2 then
                         calc:= 1 (* true *)
                    else
                         calc:= 0 (* false *)
       ;groreq   : if s1 >= s2 then
                         calc:= 1 (* true *)
                   else
                         calc:= 0 (* false *)
       ;unequal : if s1 <> s2 then
                         calc:= 1 (* true *)
                    else
                         calc:= 0 (* false *)
    end (* case *)
 end (* function calc *)
;




(******************************************************************* *)
(* The following procedure is designed to determine if source1 and source2 of the     *)
(* current quad are constant.  If they are then it calls function calc to calculate the    *)
(* value for the destination.  This procedure is called by procedure findconst.        *)
(******************************************************************* *)

procedure whichop(code:integer;curquad:quadptr;var blvect:vecttype)
```

```
;
var
   checked : boolean (* true if sources have been checked previously *)
;

begin
   checked:= false
  ;with curquad^ do
   begin
     if not sflag1 or not sflag2 then
        if not sflag1 and blvect.notcon[source1] then
        begin (* source1 is definitely not const. *)
           blvect.notcon[dest]:= true
          ;blvect.iscon[dest]:= false
          ;checked:= true
        end (* if then *)
        else (* see below *)
     else (* source1 is a constant *)
        if not sflag2 and blvect.notcon[source2] then
        begin (* source2 is definitely not const. *)
           blvect.notcon[dest]:= true
          ;blvect.iscon[dest]:= false
          ;checked:= true
        end (* if then *)
    ;if not checked then
```

```
          if sflag1 then (* source1 is a const. *)
             if sflag2 then
             begin (* source2 is a const. *)
                blvect.iscon[dest]:= true
               ;blvect.notcon[dest]:= false
               ;blvect.value[dest]:= calc(code,source1,source2)
             end (* if then *)
             else (* source2 is a variable *)
                if blvect.iscon[source2] then
                begin (* source2 var. is const. *)
                   blvect.iscon[dest]:= true
                  ;blvect.notcon[dest]:= false
                  ;blvect.value[dest]:= calc(code,source1,blvect.value[source2])
                end (* if then *)
                else (* source2 not known as const -- notcon check done above*)
          else (* source1 is a variable *)
             if sflag2 then (* source2 is a const. *)
                if blvect.iscon[source1] then
                begin (* source1 is a var. const. *)
                   blvect.iscon[dest]:= true
                  ;blvect.notcon[dest]:= false
                  ;blvect.value[dest]:= calc(code,blvect.value[source1],source2)
                end (* if then *)
                else (* source1 not known as const -- notcon check done above*)
             else (* source2 is a variable *)
                if blvect.iscon[source1] and blvect.iscon[source2] then
                begin (* source1 and source2 are const. vars. *)
                   blvect.iscon[dest]:= true
                  ;blvect.notcon[dest]:= false
                  ;blvect.value[dest]:= calc(code,blvect.value[source1],blvect.value[source2])
                end (* if then *)
                else (* source1 and 2 are not known as const. -- notcon check*)
    end (* with *)                    (* was done above *)
 end (* procedure whichop *)
;


(********************************************************************* *)
(* This procedure is called by procedure traverse and procedure foldconst to determine   *)
(* if the current quad contains a constant expression.  This procedure calls procedure    *)
(* whichop depending upon which opcode is sent from the caller.                          *)
(********************************************************************* *)

 procedure findconst(code:integer;curquad:quadptr;var blvect:vecttype)
;


 begin
    case code of
       assign  :
```

```
     begin (* assignment statement *)
        with curquad^ do
        begin
           if sflag1 then
           begin (* source1 is a constant *)
              blvect.iscon[dest]:= true
             ;blvect.notcon[dest]:= false
             ;blvect.value[dest]:= source1
           end (* if then *)
           else (* source1 is a variable *)
              if blvect.iscon[source1] then
              begin (* var source1 is definitely const. *)
                 blvect.iscon[dest]:= true
                ;blvect.notcon[dest]:= false
                ;blvect.value[dest]:= blvect.value[source1]
              end (* if then *)
              else (* var source1 is not known to be const. *)
                 if blvect.notcon[source1] then
                 begin (* definitely not const. *)
                    blvect.notcon[dest]:= true
                   ;blvect.iscon[dest]:= false
                 end (* if then *)
        end (* with *)
     end (* case of assign *)
    ;star     : whichop(star,curquad,blvect)
    ;divv     : whichop(divv,curquad,blvect)
    ;modd     : whichop(modd,curquad,blvect)
    ;plus     : whichop(plus,curquad,blvect)
    ;minus    : whichop(minus,curquad,blvect)
    ;neg      :
     begin (* assignment statement *)
        with curquad^ do
        begin
           if sflag1 then
           begin (* source1 is a constant *)
              blvect.iscon[dest]:= true
             ;blvect.notcon[dest]:= false
             ;blvect.value[dest]:= source1*-1
           end (* if then *)
           else (* source1 is a variable *)
              if blvect.iscon[source1] then
              begin (* var source1 is definitely const. *)
                 blvect.iscon[dest]:= true
                ;blvect.notcon[dest]:= false
                ;blvect.value[dest]:= blvect.value[source1]*-1
              end (* if then *)
              else (* var source1 is not known to be const. *)
                 if blvect.notcon[source1] then
                 begin (* definitely not const. *)
                    blvect.notcon[dest]:= true
                   ;blvect.iscon[dest]:= false
                end (* if then *)
          end (* with *)
       end (* case of neg *)
      ;nott    :
       begin (* assignment statement *)
          with curquad^ do
          begin
             if sflag1 then
             begin (* source1 is a constant *)
                 blvect.iscon[dest]:= true
                ;blvect.notcon[dest]:= false
                ;blvect.value[dest]:= 1-source1
             end (* if then *)
             else (* source1 is a variable *)
                if blvect.iscon[source1] then
                begin (* var source1 is definitely const. *)
                    blvect.iscon[dest]:= true
                   ;blvect.notcon[dest]:= false
                   ;blvect.value[dest]:= 1-blvect.value[source1]
                end (* if then *)
                else (* var source1 is not known to be const. *)
                   if blvect.notcon[source1] then
                   begin (* definitely not const *)
                       blvect.notcon[dest]:= true
                      ;blvect.iscon[dest]:= false
                   end (* if then *)
          end (* with *)
      end (* case of nott *)
      ;andd      : whichop(andd,curquad,blvect)
      ;orr       : whichop(orr,curquad,blvect)
      ;less      : whichop(less,curquad,blvect)
      ;greater   : whichop(greater,curquad,blvect)
      ;equal     : whichop(equal,curquad,blvect)
      ;lsoreq    : whichop(lsoreq,curquad,blvect)
      ;groreq    : whichop(groreq,curquad,blvect)
      ;unequal   : whichop(unequal,curquad,blvect)
    end (* case *)
 end (* procedure findconst *)
;


(************************************************************************)
(* The following procedure is a recursive procedure which is called by the main program*)
(* to follow the tree structure and in the end, determine the constant variables.  It is      *)
(* continually called until no more changes are made in the const/value vectors and the    *)
(* analysis stabilizes.  Not only does this procedure call itself but also the procedures   *)
(* calcvect and findconst.                                                                      *)
(************************************************************************)

 procedure traverse(curblock:integer; var change:boolean)
```

```
;
 var
    blvect    : vecttype (* the working vector for the current block *)
   ;curquad : quadptr  (* pointer to the current quad *)
   ;i          : integer   (* loop and index var *)
;

 begin
    if curblock <> 0 then
    begin
       if not visited[curblock] then
       begin (* haven't been to this block yet *)
          calcvect(curblock,blvect)
         ;curquad:= blocks[curblock].top
         ;while curquad <> blocks[curblock+1].top do
          begin (* iterate through entire block *)
             if not (curquad^.opcode in [BRA,BRF,NOP,call,RETN]) and
                not ((curquad^.opcode=0) and (curquad^.labl=0)) then
                (* screen out BRs, sub calls, nop's and returns *)
                 findconst(curquad^.opcode,curquad,blvect) (* & last quad *)
            ;curquad:= curquad^.qptr
          end (* while *)
         ;for i:= 1 to m do
          begin (* transfer working vector to block vectors *)
             if (blvect.iscon[i] <> vect[curblock].iscon[i]) or
                (blvect.notcon[i] <> vect[curblock].notcon[i]) then
             begin (* changes have been made *)
                 change:= true (* not stable *)
                ;vect[curblock].iscon[i]:= blvect.iscon[i]
                ;vect[curblock].notcon[i]:= blvect.notcon[i]
             end (* if then *)
            ;vect[curblock].value[i]:= blvect.value[i]
          end (* for to do *)
         ;visited[curblock]:= true
         ;traverse(blocks[curblock].left,change)
         ;traverse(blocks[curblock].right,change)
       end (* if then *)
    end (* if then *)
 end; (* procedure traverse *)

(************************************************************************)
(* The following function is called by procedure foldconst to check on a certain block     *)
(* to see if it is no longer needed.  It returns true if the current block is the only          *)
(* predecessor to the block in question and false otherwise.  It also returns the block      *)
(* number of the dead code.                                                                                              *)
(************************************************************************)

 function deadcode(i:integer;var dead:integer):boolean
;
 var
    target    : integer  (* the block in question *)
   ;pred      : predptr (* pointer in predecessor list *)
   ;marker : predptr (* marks where the current block is in pred list *)
   ;found     : boolean  (* true if current block is not only pred. *)
;

 begin
    target:= blocks[i].left (* get left branch *)
   ;pred:= blocks[target].list (* get list of predecessors *)
   ;found:= false
   ;while (pred <> nil) and not found do (* see if there is a block in *)
       if pred^.bl <> i then (* the pred list other than the current *)
          found:= true (* there is *)
       else
       begin
          marker:= pred (* keep location of current block in pred list *)
         ;pred:= pred^.nextbl
       end (* if else *)
   ;if not found then
    begin (* found that current block was the only pred to target *)
       deadcode:= true
      ;dead:= target
    end (* if then *)
    else (* found that current block was not the only pred to target *)
    begin
       deadcode:= false
      ;if blocks[target].list = marker then (* rm cur block from list *)
          blocks[target].list:= marker^.nextbl (* first in list *)
       else (* rm current block from list -- somewhere else in list *)
       begin
          pred:= blocks[target].list
         ;while pred^.nextbl^.bl <> i do
             pred:= pred^.nextbl
         ;pred^.nextbl:= pred^.nextbl^.nextbl
       end (* if else *)
    end (* if else *)
 end; (* function deadcode *)


(********************************************************************* *)
(* The following procedure is called upon by the main program once all of the analysis   *)
(* is done.  It determines how the quads are to be output depending upon the result of    *)
(* the analysis.  It writes to the file "after".  This procedure calls the procedures calcvect *)
(* and findconst and the function deadcode.                                                  *)
(********************************************************************* *)

 procedure foldconst
;
 const
    zero  = 0
   ;one   = 1
;
```

```
type
   sett = set of 1..100
;

var
   i              : integer    (* index and loop control *)
  ;curquad     : quadptr   (* pointer to current quad being processed *)
  ;jumpquad  : quadptr   (* pointer to quad which is BRAing *)
  ;blvect       : vecttype (* working vector for the current block *)
  ;deadblock  : integer    (* a returned value from function deadcode *)
  ;deadlist    : sett         (* set of blocks determined to be unreachable *)
;


begin
   rewrite(after)
  ;deadlist:= [] (* initalize to no dead blocks found yet *)
  ;deadblock:= 0
  ;i:= 1 (* start at first block *)
  ;while i <= numblocks do
   begin (* process all blocks *)
      calcvect(i,blvect) (* must recalculate a working vector *)
     ;curquad:= blocks[i].top
     ;while curquad <> blocks[i+1].top do
      begin (* repeat for entire block *)
          with curquad^ do
          begin
              if not ((opcode=0) and (labl=0)) then (* not at last quad *)
                  if not (opcode in [BRA,BRF,call,RETN,NOP]) then
                  begin (*screen out ops - no const. will exist except BRF*)
                      findconst(opcode, curquad,blvect) (* redeterm. value *)
                     ;if blvect.iscon[dest] then (* destination is const *)
                          writeln(after,labl,dest,assign,one,blvect.value[dest],zero,zero)
                      else (* check see if source1 is const. *)
                         if blvect.iscon[source1] then (* replace ref. with value *)
                             writeln(after,labl,dest,opcode,one,blvect.value[source1],
                                        ord(sflag2), source2)
                         else (* check see if source2 is constant *)
                             if source2 <> 0 then (* make sure we have a source first *)
                                 if blvect.iscon[source2] then (* replace ref. with value *)
                                     writeln(after,labl,dest,opcode,ord(sflag1),source1,one,
                                                 blvect.value[source2])
                                 else (* print quad -- it has no const. opt. *)
                                     writeln(after,labl,dest,opcode,ord(sflag1),source1,
                                                 ord(sflag2),source2)
                             else (* print quad -- we have no source2 *)
                                 writeln(after,labl,dest,opcode,ord(sflag1),source1,zero,zero)
                  end (* if then *)
                  else
                      if (opcode = BRF)then
          if blvect.iscon[source1] and (vect[i].value[source1]=0) then
          begin (* always false-change to BRA and remove unreachable code*)
               jumpquad:= curquad (* save pointer to BRAing quad *)
             ;while (curquad^.qptr^.labl=0)or(i+1 in deadlist) do
              (*no lable on next block*)
              begin (* code to end of block is unreachable *)
                   i:= i+1 (* move to next block *)
                 ;curquad:= blocks[i].bottom (* skip to bottom *)
                 ;if curquad^.opcode in [BRA,BRF] then (*if branch*)
                       if deadcode(i,deadblock) then (*branch was to dead code*)
                            deadlist:= deadlist+[deadblock]
              end (* while *)
            ;if curquad^.qptr^.labl <> jumpquad^.dest then
                  with jumpquad^ do
                     (* we aren't just jumping to next stmt so print branch *)
                     writeln(after,labl,dest,BRA,zero,zero,zero,zero)
            else (* we are just jumping to next stmt *)
                 if not (curquad^.labl = 0) then
                     (* must print label on BRAing quad*)
                     writeln(after,jumpquad^.labl,zero,zero,zero,zero,zero,zero)
         end (* if then *)
         else (* check for useless branch *)
              if blvect.iscon[source1] and (vect[i].value[source1] = 1) then
              begin (* don't print useless branch and check for dead code *)
                   if deadcode(i,deadblock) then
                        deadlist:= deadlist+[deadblock] (* block is dead *)
              end (* if then *)
              else
                   writeln(after,labl,dest,opcode,ord(sflag1),source1,ord(sflag2,
                                source2)
     else
     begin
          jumpquad:= curquad (* save ptr to current quad *)
         ;if opcode = BRA then
          begin (* remove dead code here too *)
               while (curquad^.qptr^.labl=0)or(i+1 in deadlist) do
               (* no lable on next block *)
               begin (* code of next block is unreachable *)
                    i:= i+1 (* move to next block *)
                  ;curquad:= blocks[i].bottom (* skip to bottom *)
                  ;if curquad^.opcode in [BRA,BRF] then (*if branch*)
                       if deadcode(i,deadblock) then (*branch was to dead code*)
                            deadlist:= deadlist+[deadblock]
               end (* while *)
              ;if curquad^.qptr^.labl <> jumpquad^.dest then
                    with jumpquad^ do (* we aren't just jumping to next stmt *)
                         writeln(after,labl,dest,BRA,zero,zero,zero,zero)
              else (* we are just jumping to next stmt *)
                   if not (curquad^.labl = 0) then
                   (*must print labl on BRAing quad*)
                        writeln(after,jumpquad^.labl,zero,zero,zero,zero,zero,
                                          zero)
                           end (* if then *)
                           else (* just print out the quad *)
                                writeln(after,labl,dest,opcode,ord(sflag1),source1,ord(sflag2,
                                          source2)
                        end (* if else *)
                     else (* prints out last quad *)
                        writeln(after,labl,dest,opcode,ord(sflag1),source1,ord(sflag2),source2)
               end (* with *)
              ;curquad:= curquad^.qptr
          end (* while *)
         ;i:= i+1 (* increment to next block *)
         ;while i in deadlist do (* while we have a block which was deter- *)
              i:= i+1 (* mined unreachable -- move to next block *)
       end (* while *)
 end (* procedure fold *)
;



begin (*** M A I N   P R O G R A M ***)
   readquads        (* input quad list *)
  ;identblocks      (* identify and set up basic blocks *)
  ;initvects        (* initialize vectors for locating constants *)
  ;buildtree        (* identify & build tree struct. of prog's logic flow *)
  ;change:= true
  ;while change do
   begin (* continue until stablized (not change) *)
        change:= false
       ;for curblock:= 1 to numblocks do
            visited[curblock]:= false (* initialize *)
       ;curblock:= 1
       ;traverse(curblock,change)
   end (* while *)
  ;foldconst        (* fold constants & output optimized quads *)
end. (*** M A I N   P R O G R A M **)
```

Appendix C
ConsFold TAG

```
# CONSFOLD -- A constant folding and dead code removing optimizer.

   # The following TAG analyzes the program tree and optimizes all nodes which are
   # determined to be constant by folding the node (performing the operation of the
   # node) into a constant node.,  All unreachable tree branches, (dead code) resulting
   # from constant folding, are removed (Dead Code Elimination).

   # Tree node and node-subtitle specifications:
   #
   #          1  PROG       BINOP     UNOP      CALL       GOTO
   #          2  PROC        0 +       0 -       0 call     0 exit
   #          3  UNOP        1 -       1 not     1 return   1 again
   #          4  BINOP       2 *
   #          5  PAIR        3 div
   #          6  COPY        4 mod
   #          7  FETCH       5 and
   #          8  CONSTANT    6 or
   #          9  CELL        7 <=
   #         10  CALL        8 >=
   #         11  LOOP        9 <
   #         12  GOTO       10 >
   #         13  IF         11 =
   #         14  DCLN       12 <>
   #         15  TYPE


predefined  Empty, Dummy, Ref, Val,
        openframe ~VALSET ^VALSET;
        into ~VALSET ~Integer ~Integer ^VALSET;
        from ~VALSET ~Integer ^Integer;
        union ~CONSET ~CONSET ^CONSET;
        difference ~CONSET ~CONSET ^CONSET;
        addset ~Integer ~CONSET ^CONSET;
        pack4  ~NODE ~Integer ~Integer ~Integer ^DECRTN;
        unpack4 ^DECRTN  ^NODE ^Integer ^Integer ^Integer;

transformer ConsFold:
```

```
          # The following function is the start function for the TAG

start: STRT;

     -> N:start
        {:STMT ~Empty ~0 ~Empty ~Empty ~Empty ^dontcare0 ^dontcare1 ^dontcare2
                 ^dontcare3}
;


          # The following function is the main driver of the TAG.  Traversal of the
          # program tree starts here and ends here.

tree : STMT ~CONSET  ~Integer ~IDSET  ~VALSET ~IDSET   ^IDSET      ^VALSET
          ^CONSET  ^CONSET;
             ~isconin     ~isloop  ~procset  ~valin       ~walkedin ^walkedout ^valout
          ^notcon      ^isconout

          # WHERE:
          #      ~isconin - is the set of variables constant upon entering this function.
          #      ~isloop - is 1 if we are in a loop and 0 if we are not in a loop (isloop==1
          #           during the analysis pass of the loop and isloop==0 on the next
          #           (transformation) pass of the loop).
          #      ~procset - a set which is carried on the downward traversal of a tree branch.
          #           It contains the ids of procedures called during the traversal of this path.
          #           It is used solely to detect (prevent) a recursive walk.
          #      ~valin - is the symbol table of variables and their current values upon
          #           entering this function.
          #      ~walkedin - is the set of procedure ids which have been walked upon
          #           entering this function.  If a procedure id occurs in this set then it has
          #           been walked and has a decoration of the set of var ids which are not
          #           constant in the particular procedure.
          #      ^walkedout - is the set of procedure ids which have been walked upon
          #           exiting this function.
          #      ^valout - is the symbol table of variables and their current values upon
          #           exiting this function.
          #      ^notcon - is the set of vars. which have been assigned to in the branches of
          #           a fork (loop) or the body of a procedure.  Variables in this set are
          #           known to be NON-constant. Notcon is always returned and is only
          #           important upon returning from branches in an IF node or from a CALL
          #           to a procedure.
          #      ^isconout - is the set of variables constant upon exiting this function.

          # The PROG node is the first node of the tree and the first visited.  At the start we
          # ignore declarations, initialize a new symbol table and begin traversal of the
          # program body.  All attributes are insignificant at this point (inhereted and
          # derived).

     -> <PROG PARAM BODY:tree>  # Root node
        {:openframe ~Empty ^table}  # initialize table
        {BODY:STMT ~Empty ~0 ~Empty ~table ~walkedin ^walkedout ^valout ^notcon
```

```
                ^isconout}

-> <DCLN> | <GOTO> | <>  # nothing happens at these nodes
     ^valout : valin    # pass everything on through
     ^isconout : isconin
     ^notcon : Empty
     ^walkedout : walkedin

-> <PROC>   # don't walk procedures until they are called
     ^valout : valin    # pass everyting on through
     ^isconout : isconin
     ^notcon : Empty
     ^walkedout : walkedin

      # If a CALL node is a CALL/retn then we have arrived at the end of a particular
      # segment of code -- pass everything on through.  If the node is a CALL/call
      # however, then we have a call to a procedure and we must analyze it.  The
      # corresponding PROC node decoration is retrieved along with its id and the
      # argument list is walked.  If it is determined that we are on a recursive path
      # (nextid in procset) then we are done here.  If not then we must walk the
      # procedure and add its set of nonconstants to the nonconstant set.

-> <CALL/sub> # "sub" is really "retn" -- see subtitles defin.
      ^valout : valin    # pass everything on through
      ^isconout : isconin
      ^notcon : Empty
      ^walkedout : walkedin

-> C:<CALL/add ARG:walka> # "add" is really "call" - see subtitles defin.
      {C:examine ^proc} # retrieve decoration (PROC node)
      {proc -> nextid}  # get procedure id and then walk the arg. list
      {ARG:WalkArg ~isconin ~valin ~isloop ~Empty ~1 ~procset ~walkedin ^walkt
      ^argmatch ^dontcare ^iscont ^valt ^notcont}
      {nextid == nextid | nextid in procset # if the proc id is in
            =>  ^valout : valt         # procset, it must be recursive
                ^notcon : notcont   # Don't walk it again!
                ^isconout : iscont
                ^walkedout : walkt
        otherwise  # no recursion detected yet -- walk the procedure
            => {:addset ~nextid ~procset ^pidset} # add it to recurs. set
               {proc:CrossFold ~iscont ~valt ~isloop ~argmatch ~pidset ~walkt
                   ^walkedout ^valout ^notconout ^isconout}
               {:union ~notcont ~notconout ^notcon} # join nonconst. sets
      }

      # The first thing checked at the IF node is the expression. If it is constant
      # (^constnt returns 1) then only one branch of the fork needs to be traversed (the
      # other will disappear).  The branch to be walked is determined by the attribute
      # valexpn.  If valexpn returns 1 then the IF expn was contant-true, the
      # THEN-tree is walked and the ELSE-tree is thrown away (the IF node is
      # transformed into the THEN-tree).  If the IF expn was constant-false (valexpn
```

```
        # returns 0) then the ELSE-tree is walked and the THEN-tree is thrown away (IF
        # node is transformed into the ELSE-tree).  If constnt returns 0 then the IF
        # expression was not constant and both the THEN and ELSE trees must be
        # walked.  If isloop is 1, then we are in the analysis pass of a loop and both trees
        # must be walked disregarding the fact that the IF expn may have been constant.
        # The next time through (the transformation pass) isloop will be 0 -- see
        # production for LOOP node.  It is also taken into account here that the THEN
        # and ELSE trees may be NULL.

-> I:<IF X:branch T:tree E:tree>  # walk the expression first
        {X:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^constnt ^valexpn
            ^valx ^isconx ^notconx}
        {constnt==1  isloop==0  valexpn==1  T><<> #expression always true
              =>  {T:STMT ~isconx ~isloop ~procset ~valx ~walkt ^walkedout ^valout
                     ^notcont ^isconout}
                  {:union ~notconx ~notcont ^notcon} # join notcon sets
                 # TRANSFORM IF node to T subtree.

         constnt==1  isloop==0  valexpn==1  T==<> #expression always true
              => {T:STMT ~isconx ~isloop ~procset ~valx ~walkt ^walkedout ^valout
                     ^notcont ^isconout}
                 {:union ~notconx ~notcont ^notcon} # join notcon sets
                 # TRANSFORM IF node to NULL -- T was null

         constnt==1  isloop==0  valexpn==0  E><<> #expression always false
              => {E:STMT ~isconx ~isloop ~procset ~valx ~walkt ^walkedout ^valout
                     ^notcont ^isconout}
                 {:union ~notconx ~notcont ^notcon} # join notcon sets
                 # TRANSFORM IF node to E subtree.

         constnt==1  isloop==0  valexpn==0  E==<> #expression always false
              =>  {E:STMT ~isconx ~isloop ~procset ~valx ~walkt ^walkedout ^valout
                     ^notcont ^isconout}
                  {:union ~notconx ~notcont ^notcon} # join notcon sets
                  # TRANSFORM IF node to NULL -- E was null


         otherwise   => #are on the analysis pass of a loop or the expn.was not constant.
                  {T:STMT ~isconx ~isloop ~procset ~valx ~walkt ^walke ^valthen
                     ^notcont ^tiscon}
                  {E:STMT ~isconx ~isloop ~procset ~valx ~walke ^walkedout ^valelse
                     ^notcone ^eiscon}
                  {:union ~notcont ~notcone ^notconf} #combine known non-constants
                  {:union ~notconf ~notconx ^notcon} #combine known non-constants
                  {:difference ~isconx ~notcon ^isconout} #remove from set of constants
                  ^valout : valx

        }
=> T | <PAIR <> <>> | E | <PAIR <> <>> | I

        # At a PAIR node we simply go down the left tree, go down the right tree, and
```

```
            # combine the set of known non-constants to be returned.

        -> P:<PAIR L:tree R:tree>
            {L:STMT ~isconin ~isloop ~procset ~valin ~walkedin ^walkt^valtemp
                ^lnot^iscontemp}
            {R:STMT ~iscontemp ~isloop ~procset ~valtemp ~walkt ^walkedout ^valout
                ^rnot ^isconout}
            {:union ~lnot ~rnot ^notcon} #combine sets of known non-constants

# At the COPY node the var. being assigned to is added to the notcon set.  The CELL
# node is examined to obtain the cell's DCLN node (decnode). The DCLN node (decnode)
# is then examined to get the class (tipe) of the variable.  The expression tree is
# traversed and the returned data is analyzed.  If the expression was constant
# (econstnt=1) and we are not in the analysis pass of a loop (isloop=0) and the
# variable is scalar (tipe=scalar) then we can add the var to table and the constant set
# isconin. Otherwise (if we are analyzing a loop or if the exp. wasn't constant or if the
# var. is not scalar) we remove the var from the set of constants and return constnt as
# 0 (false).

        -> C:<COPY E:branch CL:branch>%v
            {C:examine ^decnode} # retrive DCLN node
            {decnode:examine ^tipe} # get variable type
            {:addset ~v ~Empty ^notconv} # put var id in nonconstant set
            {CL:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^gg ^aa ^bb ^dd ^ee^ff}
            {E:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkedout ^econstnt
                ^value ^valt ^iscont ^notconx}
            {:union ~notconv ~notconx ^notcon} # join non-constant sets
            {econstnt >< 0  isloop == 0  # not analyzing loop and const. expn
                =>  {tipe : SCALAR} #if node not scalar then this al-
                    {:addset ~v ~iscont ^isconout} # ternative fails
                    {:into ~valt ~v ~value ^valout} # put const value in value table
             otherwise # expn wasn't constant or we are analyzing a loop
                => {:Exclude ~v ~iscont ^isconout} # remove var from set of constants
                    ^valout : valt
            }

# The LOOP tree is traversed twice.  Once to analyze and find the true constants (any
# vars. assigned to in the loop are not const) and second to make any possible
# transformations.  The first time isloop is passed down as 1 to distinguish that we are
# the analysis pass of a loop and the next time it is passed down as 0 so that other
# productions can make transformations.  After the analysis pass of the loop, we have
# obtained the set of constants (isconloop) from the first pass and it is passed down as
# the set of legal constants (as isconin) in the second pass.

-> L:<LOOP BODY:tree> # pass 1 -- analysis.  pass 2 -- transformation
    {BODY : STMT ~isconin ~1 ~procset ~valin ~walkedin ^walkloop ^valoutloop
        ^notconloop ^isconloop}
    {BODY : STMT ~isconloop ~0 ~procset ~valin ~walkedin ^walkedout ^valout ^notcon
        ^isconout}
;
```

```
# The following function is used to check a variable type to see if it is scalar (integer,
# boolean or character).  This function will fail if the type does not match this pattern.

typpe: SCALAR;

        -> <TYPE <CONSTANT> <CONSTANT>>  # Scalar type node pattern
;


# Function EXPN is used to traverse expressions in the program tree.

branch:EXPN ~CONSET ~VALSET ~Integer ~IDSET ~IDSET     ^IDSET      ^Integer
        ^VARVAL ^VALSET ^CONSET  ^CONSET;
                    ~isconin    ~valin        ~isloop  ~procset ~walkedin ^walkedout ^constnt
        ^value        ^valout      ^isconout    ^notcon

# WHERE:
#      ~isconin - is the set of variables constant upon entering this function.
#      ~valin - is the symbol table of variables and their current values upon entering
#           this function.
#      ~isloop - is 1 if we are in a loop and 0 if we are not in a loop (isloop==1 during
#           the analysis pass of the loop and isloop==0 on the next (transformation) pass
#           of the loop).
#      ~procset - a set which is carried on the downward traversal of a tree branch.  It
#           contains the ids of procedures called during the traversal of this path.  It is
#           used solely to detect (prevent) a recursive walk.
#      ~walkedin - is the set of procedure ids which have been walked upon entering this
#           function.  If a procedure id occurs in this set then it has been walked and has a
#           decoration of the set of var ids which are not constant in the particular
#           procedure.
#      ^walkedout - is the set of procedure ids which have been walked upon exiting this
#           function.
#      ^constnt - is used to tell if the expn. was constant.  It returns 1 if the expn. was
#           constant and 0 if not.
#      ^value - returns the value of the expression and is only meaningful if constnt
#           returns as 1 (true).
#      ^tipe  - returns the tipe node of the expression
#      ^valout - is the symbol table of variables and their current values upon exiting
#           this function.
#      ^isconout - is the set of variables constant upon exiting this function.
#      ^notcon - is the set of vars. which have been assigned to in the branches of a fork
#           (loop) or the body of a procedure.  Variables in this set are known to be
#           NON-constant. Notcon is always returned and is only important upon
#           returning from branches in an IF node or from a CALL to a procedure.


# The following 12 functions are basically the same.  The left subtree is traversed and
# then the right subtree is traversed. ^Lcon and ^rcon both are received as either 1 or
# 0 depending on whether the left or right expn was constant. If they both come back 1
# then both trees were constant, the BINOP node is transformed into a CONSTANT node
# and ^constnt is returned as 1 (true).  The values received from the left and right
# subtrees are computed (depending on the operation) and the result is returned in
# ^value. ^value is meaningless unless ^constnt returns true.

-> B:<BINOP/add L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon} # combine sets of nonconstants
    ^value : leftvalue + rightvalue # perform operation
    {lcon == 1  rcon == 1  isloop == 0 # both operands constant and
       =>  ^constnt : 1            # not analyzing loop
           {T <- value} # decorate new CONSTANT node with value
           {T:decorate ~tipe} # decorate new node with its type
     lcon == 1  rcon == 1  isloop == 1 # both operands constant but
       =>  ^constnt : 1            # we are analyzing a loop
     otherwise # at least one operand wasn't constant
       =>  ^constnt : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/sub L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    ^value : leftvalue - rightvalue
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
           {T <- value}
           {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
       =>  ^constnt : 1
     otherwise
       =>  ^constnt : 0
    }

=> T:<CONSTANT> | B | B
-> B:<BINOP/mpy L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    ^value : leftvalue * rightvalue
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
       =>  ^constnt : 1
     otherwise
       =>  ^constnt : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/modd L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    ^value : leftvalue - ((leftvalue / rightvalue) * rightvalue)
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
       =>  ^constnt : 1
     otherwise
       =>  ^constnt : 0
    }

 => T:<CONSTANT> | B | B


-> B:<BINOP/dvd L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    ^value : leftvalue / rightvalue
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
       =>  ^constnt : 1
     otherwise
```

```
        =>  ^constnt : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/great L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
        ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        =>  ^constnt : 1
             {leftvalue >> rightvalue =>  ^value : 1
              otherwise                =>  ^value : 0
             }
             {T <- value}
             {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        =>  ^constnt : 1
             ^value : 0
     otherwise
        =>  ^constnt : 0
             ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/less L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
        ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        =>  ^constnt : 1
             {leftvalue << rightvalue  =>  ^value : 1
              otherwise                 =>  ^value : 0
             }
             {T <- value}
             {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        =>  ^constnt : 1
             ^value : 0
     otherwise
        =>  ^constnt : 0
             ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/equal L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
            {leftvalue == rightvalue  =>  ^value : 1
             otherwise                =>  ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
       =>  ^constnt : 1
            ^value : 0
     otherwise
       =>  ^constnt : 0
            ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/noteq L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
       ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
       ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
       =>  ^constnt : 1
            {leftvalue >< rightvalue  =>  ^value : 1
             otherwise                =>  ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        => ^constnt : 1
            ^value : 0
     otherwise
        => ^constnt : 0
            ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/greq L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
        ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        => ^constnt : 1
            {leftvalue >> rightvalue | leftvalue == rightvalue  => ^value : 1
             otherwise                                         => ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        => ^constnt : 1
            ^value : 0
     otherwise
        => ^constnt : 0
            ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/lseq L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
        ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        => ^constnt : 1
            {leftvalue << rightvalue | leftvalue == rightvalue  => ^value : 1
             otherwise                                         => ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        =>  ^constnt : 1
            ^value : 0
     otherwise
        =>  ^constnt : 0
            ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/andd L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^leftvalue ^valt
        ^iscont ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rightvalue ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        =>  ^constnt : 1
            {leftvalue >< 0   rightvalue >< 0  =>  ^value : 1
             otherwise                         =>  ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        =>  ^constnt : 1
            ^value : 0
     otherwise
        =>  ^constnt : 0
            ^value : 0
    }

=> T:<CONSTANT> | B | B


-> B:<BINOP/orr L:branch R:branch>
    {L:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkt ^lcon ^lval ^valt ^iscont
        ^notconl}
    {R:EXPN ~iscont ~valt ~isloop ~procset ~walkt ^walkedout ^rcon ^rval ^valout
        ^isconout ^notconr}
    {B:examine ^tipe}
    {:union ~notconl ~notconr ^notcon}
    {lcon == 1  rcon == 1  isloop == 0
        =>  ^constnt : 1
            {lval >< 0 | rval >< 0  =>  ^value : 1
             otherwise              =>  ^value : 0
            }
            {T <- value}
            {T:decorate ~tipe}
     lcon == 1  rcon == 1  isloop == 1
        => ^constnt : 1
             ^value : 0
     otherwise
        => ^constnt : 0
             ^value : 0
    }

=> T:<CONSTANT> | B | B


# The UNOP nodes are basically the same as the BINOP nodes except there is only one
# operand tree to analyze.

-> U:<UNOP/add E:branch>  # "add" is really "sub" -- see subtitles def.
    {E:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkedout ^constnt ^expvalue
        ^valout ^isconout ^notcon}
    {U:examine ^tipe} # retreive expression type
    ^value : expvalue * (1-2) # perform operation
    {constnt == 1  isloop == 0 # expn was const and not analyzing loop
        =>  {T <- value} # decorate new CONSTANT node with value
             {T:decorate ~tipe} # decorate new CONSTANT node with type
     otherwise  # expn was not constant or we are analyzing a loop
        =>  # do nothing
    }

=> T:<CONSTANT> | U


-> U:<UNOP/sub E:branch> # "sub" is really "not" -- see subtitles defin.
    {E:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^walkedout ^constnt ^expvalue
        ^valout ^isconout ^notcon}
    {U:examine ^tipe} # retreive expression type
    {constnt == 1  isloop == 0 # expn was const and not analyzing loop
        =>  {expvalue == 1  =>  ^value : 0 # perform node operation
              otherwise         =>  ^value : 1
             }
             {T <- value} # decorate new CONSTANT node with value
             {T:decorate ~tipe} # decorate new node with type
     otherwise # expn was not constant or we are analyzing a loop
        =>  ^value : 0  # do nothing
    }

=> T:<CONSTANT> | U


# At a FETCH node we must check to see if the variable (v) that we are fetching is in
# the set isconin.  If it is then we can retrieve that variable's value from the symbol
# table valin and return ^constnt as 1 (true). Otherwise we return ^constnt as 0
# (false).  If we are analyzing a loop and the variable was constant (in isconin) then
# we can't make a transformation.  But if we were not in the analysis pass of a loop we
# can transform the FETCH to a CONSTANT node.

-> F:<FETCH C:branch>%v
    {F:examine ^decnode} # retrieve DCLN node
    {decnode:examine ^tipe} # get var type
    {C:EXPN ~isconin ~valin ~isloop ~procset ~walkedin ^garb ^sowhat ^dontcare ^aa
        ^bb ^cc}
    {isloop == 1  v in isconin # analysing loop and var in constant set
        =>  {:from ~valin ~v ^value} # get var's value from table
             ^constnt : 1
     isloop == 0  v in isconin # not analyzing and var in constant set
        =>  {:from ~valin ~v ^value} # get var's value from table
             ^constnt : 1
             {T <- value} # decorate new CONSTANT node with value
             {T:decorate ~tipe} # decorate new node with its type
     otherwise # var was not in the constant set
        =>  ^constnt : 0
             ^value : 0
    }
    ^valout : valin
    ^isconout : isconin
    ^notcon : Empty
    ^walkedout : walkedin

=> F | T:<CONSTANT> | F


# At a cell node we must examine the node to obtain the DCLN node for the cell. We can
# then examine the DCLN node and get the var. type. Next, we check to see if the var. is
# constant (v in isconin) and of scalar type. If it is then we can retrieve its value
# from the table and send back a report that it is constant (^constnt : 1). Otherwise it
# is not constant (^constnt:0). No transformation here.

-> C:<CELL>%v
    {C:examine ^decnode} # retrieve DCLN node
    {decnode:examine ^tipe} # get var type
    {isloop == isloop   v in isconin   # var is constant
        =>  {tipe : SCALAR}  # if node not scalar this alter-
             {:from ~valin ~v ^value}    # native will fail
             ^constnt : 1
     otherwise   # var is not constant
        =>  ^constnt : 0
    }
    ^valout : valin
    ^isconout : isconin
    ^notcon : Empty
    ^walkedout : walkedin


# At a CONSTANT node we must first examine the node and get the decoration so we can
# check to see if the variable is scalar. If so, we can return its value. If not, we return
# ^constnt as 0 (false).

-> C:<CONSTANT>%value
    {C:examine ^tipe} # get node type
    {isloop == isloop
       =>  {tipe : SCALAR} # make sure its scalar
            ^constnt : 1
      otherwise  # this constant was not scalar type
       =>  ^constnt : 0
    }
    ^valout : valin
    ^isconout : isconin
    ^notcon : Empty
    ^walkedout : walkedin

# It is possible that an expression will be a function call.  It is handled basically the
# same way as in the STMT function.

-> C:<CALL/add ARG:walka>  # "add" is really "call" -- see subtitles def.
    {C:examine ^proc} # retrieve corresponding PROC node
    {proc -> nextid} # get procedure id and walk argument list
    {ARG:WalkArg ~isconin ~valin ~isloop ~Empty ~1 ~procset ~walkedin ^walkt
       ^argmatch ^dontcare ^iscont ^valt ^notcont}
    {nextid == nextid | nextid in procset  # if the proc id is in procset
       =>  ^valout : valt                # it must be recursive
            ^notcon : notcont              # don't walk it again!
            ^isconout : iscont
            ^walkedout : walkt
      otherwise  # we have called a non-recursive procedure... walk it
       =>  {:addset ~nextid ~procset ^pidset} # add it to recurs. set
            {proc:CrossFold ~iscont ~valt ~isloop ~argmatch ~pidset ~walkt ^walkedout
            ^valout ^notconout ^isconout}
            {:union ~notconout ~notcont ^notcon} # join notcon sets
    }
    ^constnt : 0  # we have to
    ^value : 0    # send these back
    ^tipe : 0     # with something
;


# The following function is called when we have a subroutine to walk.  PROC is the only
# possible node for this function.



cross:CrossFold ~CONSET ~VALSET ~Integer ~VALSET ~IDSET ~IDSET
    ^IDSET      ^VALSET ^CONSET ^CONSET;
                     ~isconin    ~valin      ~isloop   ~argmatch ~procset ~walkedin
    ^walkedout ^valout     ^notcon     ^isconout
```

```
# WHERE:
#     ~isconin - is the set of variables constant upon entering this function.
#     ~valin - is the symbol table of variables and their current values upon entering
#          this function.
#     ~isloop - is 1 if we are in a loop and 0 if we are not in a loop (isloop==1 during
#          the analysis pass of the loop and isloop==0 on the next (transformation) pass
#          of the loop).
#     ~argmatch - the table of argument numbers and their corresponding var ids (var
#          id will be -99 if arg was an expn).
#     ~procset - a set which is carried on the downward traversal of a tree branch.  It
#          contains the ids of procedures called during the traversal of this path.  It is
#          used solely to detect (prevent) a recursive walk.
#     ~walkedin - is the set of procedure ids which have been walked upon entering this
#          function.  If a procedure id occurs in this set then it has been walked and has a
#          decoration of the set of var ids which are not constant in the particular
#          procedure.
#     ^walkedout - is the set of procedure ids which have been walked upon exiting this
#          function.
#     ^valout - is the symbol table of variables and their current values upon exiting
#          this function.
#     ^notcon - is the set of vars. which have been assigned to in the branches of a fork
#          (loop) or the body of a procedure.  Variables in this set are known to be
#          NON-constant.  Notcon is always returned and is only important upon
#          returning from branches in an IF node or from a CALL to a procedure.
#     ^isconout - is the set of variables constant upon exiting this function.

# The parameter list is walked first.  The set of walked procedures (walkedin) is
# checked to see if the current proc. has been walked yet.  If the proc. id occurs in the
# set then it has been walked and is decorated with the set of non-constants derived in
# the proc.  This set is retrieved and removed from the set of known constants.  If the
# procedure id does not occur in procset then it has not yet been walked.  If this is the
# case, the procedure is walked, the PROC node is decorated with the set of non-
# constants derived from the procedure and the proc. id is added to the walked set.

-> P:<PROC PARAM:walkp BODY:tree>%pid  # only possible node here
    {PARAM:WalkParam ~isconin ~valin ~argmatch ~1 ~procset ^newiscon ^newval
        ^junk}
    {pid == pid   pid in walkedin  #proc has been walked already
        =>  {P:examine ^notcon}  # get decoration (set of non-constants)
             ^walkedout : walkedin
      otherwise  # proc has not been walked yet -- walk it
        =>  {BODY:STMT ~Empty ~isloop ~procset ~Empty ~walkedin ^walkt ^junk1
                 ^notcon ^junk2}
             {P:decorate ~notcon}  # decorate with set of non-constants
             {:addset ~pid ~walkt ^walkedout} #add proc id to walked set
    }
    {:difference ~newiscon ~notcon ^isconout} # remove set of non-const
    ^valout : newval                          # from returning set of constants
;
```

```
# WalkArg is called when a CALL/call (procedure call) is encountered in the program
# tree.  This function walks the argument list and returns a symbol table (matchset)
# containing the argument numbers (order of occurance) and their corresponding
# variable ids (id = -99 if the argument was an expression).  When the proc. is
# walked (in function CrossFold) this table is passed to function WalkParam so that
# arguments can be matched with their corresponding parameters.

walka:WalkArg ~CONSET ~VALSET ~Integer ~VALSET ~Integer  ~IDSET  ~IDSET
        ^IDSET     ^VALSET ^Integer     ^CONSET ^VALSET ^CONSET;
                    ~isconin     ~valin       ~isloop  ~matchin    ~pnumin ~procset ~walkedin
        ^walkedout ^matchset  ^pnumout ^isconout   ^valout       ^notcon

# WHERE:
#     ~isconin - is the set of variables constant upon entering this function.
#     ~valin - is the symbol table of variables and their current values upon entering
#          this function.
#     ~isloop - is 1 if we are in a loop and 0 if we are not in a loop (isloop==1 during
#          the analysis pass of the loop and isloop==0 on the next (transformation) pass
#          of the loop).
#     ~matchin - the argument list table upon entering this function
#     ~pnumin - the number of the current argument in the list being traversed.
#     ~procset - a set which is carried on the downward traversal of a tree branch.  It
#          contains the ids of procedures called during the traversal of this path.  It is
#          used solely to detect (prevent) a recursive walk.
#     ~walkedin - is the set of procedure ids which have been walked upon entering this
#          function.  If a procedure id occurs in this set then it has been walked and has a
#          decoration of the set of var ids which are not constant in the particular
#          procedure.
#     ^walkedout - is the set of procedure ids which have been walked upon exiting this
#          function.
#     ^matchset - the table of arguments upon exit from the function
#     ^pnumout - the number for the next argument (pnumin + 1)
#     ^isconout - is the set of variables constant upon exiting this function.
#     ^valout - is the symbol table of variables and their current values upon exiting
#          this function.
#     ^notcon - is the set of vars. which have been assigned to in the branches of a fork
#          (loop) or the body of a procedure.  Variables in this set are known to be
#          NON-constant. Notcon is always returned and is only important upon
#          returning from branches in an IF node or from a CALL to a procedure.

-> <BINOP> | <UNOP> | <CONSTANT> # argument is an expression
    {:into ~matchin ~pnumin ~(1-100) ^matchset} # place in arg list table
    ^pnumout : pnumin + 1   # increment argument counter
    ^isconout : isconin
    ^valout : valin
    ^notcon : Empty
    ^walkedout : walkedin

# It is possible that an argument will be a function call.  It is handled basically the
# same way as in the STMT function.
```

```
-> C:<CALL/add ARG:walka> # "add" is really "call" -- see subtitles def.
   {C:examine ^proc}  # "call" is only possibility for arguments
   {proc -> nextid} # get procedure id and walk the argument list
   {ARG:WalkArg ~isconin ~valin ~isloop ~Empty ~1 ~procset ~walkedin ^walkt
      ^argmatch ^dontcare ^iscont ^valt ^notcont}
   {nextid == nextid | nextid in procset # if proc id is in procset,
      =>  ^valout : valt              # then it must be recursive!
          ^notcon : notcont           # don't walk it again
          ^isconout : iscont
          ^walkedout : walkt
    otherwise  # we have called a non-recursive procedure... walk it
      =>  {:addset ~nextid ~procset ^pidset} # add it to recurs. set
          {proc:CrossFold ~iscont ~valt ~isloop ~argmatch ~pidset ~walkt ^walkedout
             ^valout ^notconproc ^isconout}
          {:union ~notcont ~notconproc ^notcon} # join notcon sets
   }
   {:into ~matchin ~pnumin ~(1-100) ^matchset} # record arg as (pass-by-val)
   ^pnumout : pnumin + 1   # increment argument counter

-> <FETCH>%vn  # argument is a varible -- possibly pass-by-ref.
   {:into ~matchin ~pnumin ~vn ^matchset} # record arg no. and vn in table
   ^pnumout : pnumin + 1  # increment argument counter
   ^isconout : isconin
   ^valout : valin
   ^notcon : Empty
   ^walkedout : walkedin

-> <PAIR L:walka R:walka>
   {L:WalkArg ~isconin ~valin ~isloop ~matchin ~pnumin ~procset ~walkedin ^walkt
      ^matcht ^pnumt ^iscont ^valt ^notl}
   {R:WalkArg ~iscont ~valt ~isloop ~matcht ~pnumt ~procset ~walkt ^walkedout
      ^matchset ^pnumout ^isconout ^valout ^notr}
   {:union ~notl ~notr ^notcon} # join non-constant sets

-> <>
   ^matchset : matchin
   ^pnumout : pnumin
   ^isconout : isconin
   ^valout : valin
   ^notcon : Empty
   ^walkedout : walkedin
;


# WalkParam is called from function CrossFold to walk a procedure's parameter list.
# The parameters are walked and if the parameter is pass-by-ref then the
# corresponding var id is retrieved from the argument list table (passed in as
# argmatch) and that var id is removed from the set of constants (iscon).

walkp:WalkParam ~CONSET ~VALSET ~VALSET   ~Integer  ~IDSET  ^CONSET
   ^VALSET ^Integer;
```

```
                    ~isconin    ~valin      ~argmatch    ~pnumin ~procset ^isconout
   ^valout     ^pnumout

# WHERE:
#     ~isconin - is the set of variables constant upon entering this function.
#     ~valin - is the symbol table of variables and their current values upon entering
#          this function.
#     ~argmatch - the table of argument ids and the corresponding arguments obtained
#          from walking the argument list of the CALL which invoked the function being
#          traversed. (table from function WalkArg)
#     ~pnumin - current parameter id upon entering this function.
#     ~procset - a set which is carried on the downward traversal of a tree branch.  It
#          contains the ids of procedures called during the traversal of this path.  It is
#          used solely to detect (prevent) a recursive walk.
#     ^isconout - is the set of variables constant upon exiting this function.
#     ^valout - is the symbol table of variables and their current values upon exiting
#          this function.
#     ^pnumout - number of the next parameter (pnumin + 1)

-> <DCLN/kind>%vn
   {kind == Ref  # argument was pass-by-ref
      =>  {:from ~argmatch ~pnumin ^argid} #get var id from arg table
          {:Exclude ~argid ~isconin ^isconout}  # remove from const set
    otherwise # argument was not pass-by-ref.
      =>  ^isconout : isconin
   }
   ^valout : valin
   ^pnumout : pnumin + 1  # increment parameter counter

-> <PAIR L:walkp R:walkp>
   {L:WalkParam ~isconin ~valin ~argmatch ~pnumin ~procset ^iscont ^valt ^pnumt}
   {R:WalkParam ~iscont ~valt ~argmatch ~pnumt ~procset ^isconout ^valout ^pnumout}

-> <>
   ^isconout : isconin
   ^valout : valin
   ^pnumout : pnumin
;



# The following function will remove an item (varno) from a set (inset) and return
# the resultant set (outset).

excl : Exclude ~Integer ~VARSET ^VARSET;
               ~varno   ~inset      ^outset

-> N:excl
   {:addset ~varno ~Empty ^tempset}
   {:difference ~inset ~tempset ^outset}
```

```
;


end ConsFold
;
```

Appendix D
LoopCon TAG

```
# LOOPCON -- A Loop Constant Optimizer

# The following TAG analyzes the program tree and optimizes its loops.  All items
# which are determined to be constant in the loop are moved out of the loop and glued
# into the tree just above the loop.  Such items may be a constant statement or a
# constant expression.  When a statement is found to be constant, the entire statement
# is clipped from its place in the loop and glued back into the tree just above the loop
# node.  When an expression is found to be constant, that expression is removed from
# its place in the loop and replaced with a fetch to a new variable cell.  The removed
# expression is then glued into the tree just above the loop and assigned to the new
# variable which replaced it in the loop.  The declaration for this new variable is also
#  glued into the tree in the declaration list of the subroutine in which the loop-constant item
#  occures.

# Tree node and node-subtitle specifications:
#
#        1  PROG       BINOP     UNOP      CALL       GOTO
#        2  PROC        0 +       0 -       0 call     0 exit
#        3  UNOP        1 -       1 not     1 return   1 again
#        4  BINOP       2 *
#        5  PAIR        3 div
#        6  COPY        4 mod
#        7  FETCH       5 and
#        8  CONSTANT    6 or
#        9  CELL        7 <=
#       10  CALL        8 >=
#       11  LOOP        9 <
#       12  GOTO       10 >
#       13  IF         11 =
#       14  DCLN       12 <>
#       15  TYPE

predefined  Empty, Dummy, Universe, Min, Max, Ref,
        openframe ~VALSET ^VALSET;
        into ~VALSET ~Integer ~Integer ^VALSET;
        from ~VALSET ~Integer ^Integer;
        union ~CONSET ~CONSET ^CONSET;
        difference ~CONSET ~CONSET ^CONSET;
        addset ~Integer ~CONSET ^CONSET;
        pack4  ~NODE ~Integer ~Integer ~Integer ^DECRTN;
        unpack4 ~DECRTN  ^NODE ^Integer ^Integer ^Integer;
        newvarno ^Integer;

transformer LoopCon:

        # The following function is the start function for the TAG
start: STRT;

    -> N:start
        {:STMT ~Empty ~Empty ~Empty ~0 ~0 ~Empty ^sowhat0 ^sowhat1 ^sowhat2
            ^sowhat3 ^sowhat4}
;


# The following function takes care of the analysis of a loop.  It is invoked when a
# LOOP node is encountered in the traversal of the program tree.  What we are
# interested in (in the analysis) are the variables which are changed in the loop.
# All of the action in this function takes place at the COPY node (assignment) and
# the CALL node (subroutine call).

find:FindCon ~VARSET ~VARSET ~IDSET ~IDSET  ^IDSET    ^VARSET
     ^VARSET;
          ~notconin  ~twicein    ~procset ~walkedin ^walkedout ^notconout
     ^twiceout

# WHERE:
#    ~notconin - the set of variable ids known to be non-constant upon entering
#        this function.  These vars take on new values in the loop (have been
#        assigned a value at least once).
#    ~twicein - the set of variable ids known to be non-constant upon entering
#        this function.  These vars have been assigned a value in the loop at least
#        twice.
#    ~procset - a set which is carried on the downward traversal of a tree branch.
#        It contains the ids of procedures which have been called during the
#        traversal of this path. It is used solely to detect (prevent) a recursive
#        walk.
#    ~walkedin - is the set of procedure ids which have been walked upon entering
#        this function.  If a procedure id occurs in this set then it has been walked
#        and has a decoration of the set of var ids which are not constant in that
#        particular procedure.
#    ^walkedout - is the set of procedure ids which have been walked  upon exiting
#        this function.
#    ^notconout - the set of non-constant var ids upon exiting this function.
#    ^twiceout - the set of non-constant var ids which have been assigned to at
#        least twice upon exiting this function.

-> <PAIR L:find R:find> | <BINOP L:find R:find>
    {L:FindCon ~notconin ~twicein ~procset ~walkedin ^walkedt^notcontemp^twicetemp}
    {R:FindCon ~notcontemp ~twicetemp ~procset ~walkedt ^walkedout ^notconout
        ^twiceout}

-> <COPY L:find R:find>%varno
```

```
    {L:FindCon ~notconin ~twicein ~procset ~walkedin ^walkedt ^lnotcon ^ltwice}
    {R:FindCon ~lnotcon ~ltwice ~procset ~walkedt ^walkedout ^rnotcon ^rtwice}
    {varno==varno   varno in rnotcon # var is already in notcon set
       => {:addset ~varno ~rtwice ^twiceout} # put it in twice set
              ^notconout : rnotcon
     otherwise # var doesn't occur in notcon set -- this is its first
       => {:addset ~varno ~rnotcon ^notconout} # put it in notcon set
              ^twiceout : rtwice
    }

-> <IF X:find T:find E:find>
    {X:FindCon ~notconin ~twicein ~procset ~walkedin ^walkedx ^xnotcon ^xtwice}
    {T:FindCon ~xnotcon ~xtwice ~procset ~walkedx ^walkedt ^tnotcon ^ttwice}
    {E:FindCon ~tnotcon ~ttwice ~procset ~walkedt ^walkedout ^notconout ^twiceout}

-> <CELL> | <CONSTANT> | <DCLN> | <GOTO> | <>
    ^notconout : notconin  # nothing interesting at these nodes
    ^twiceout  : twicein   # pass everything on through
    ^walkedout : walkedin

# If a CALL node is a CALL/retn then we have arrived at the end of a particular
# segment of code -- pass everything on through.  If the node is a CALL/call
# however, then we have a call to a procedure and we must analyze it.  The
# corresponding PROC node decoration is retrieved along with its id and the
# argument list is walked.  If it is determined that we are on a recursive path
# (nextid in procset) then we are done here.  If not then we must walk the
# procedure and add its set of non-constants to the notcon set.

-> <CALL/sub>  # "sub" is really "return" -- see subtitles definition
    ^notconout : notconin  # pass everything on through
    ^twiceout  : twicein
    ^walkedout : walkedin

-> C:<CALL/add ARG:walka>  # "add" is really "call" -- see subtitles def.
    {C:examine ^proc} # retrieve PROC node
    {proc -> nextid}  # get procedure id and walk argument list
    {ARG:WalkArg ~Empty ~procset ~walkedin ~Empty ~1 ^matchset ^pnumout ^walkedt
       ^notcont}
    {:intersect ~notconin ~notcont ^twicet}
    {:union ~twicet ~twicein ^twicex} # join twice sets
    {:union ~notconin ~notcont ^notconx} # join notcon sets
    {nextid == nextid   nextid in procset  # if the proc id is in
       =>  ^notconout : notconx           # procset, it must be recur-
           ^walkedout : walkedt           # sive. Don't walk it again!
           ^twiceout : twicex
     otherwise  # no recursion detected yet -- walk the procedure
       =>  {:addset ~nextid ~procset ^pidset} #add it to recurs. set
           {proc:CrossFind ~Empty ~pidset ~walkedt ~matchset ^walkedout ^notcony}
           {:intersect ~notconx ~notcony ^twicey}
           {:union ~twicex ~twicey ^twiceout} # join twice sets
           {:union ~notconx ~notcony ^notconout} # join notcon sets
    }


-> <FETCH T:find> | <UNOP T:find>
    {T:FindCon ~notconin ~twicein ~procset ~walkedin ^walkedout ^notconout ^twiceout}

-> L:<LOOP B:find> # a nested loop
    {B:FindCon ~notconin ~twicein ~procset ~walkedin ^walkedout ^notconout^twiceout}
;


# WalkArg is called when a CALL/call (procedure call) is encountered in the
#  program tree.  This function walks the argument list and returns a symbol table
# (matchset) containing the argument numbers (order of occurance) and their
#  corresponding variable ids (id = -99 if the argument was an expression).  When
# the proc. is walked (in function CrossFind) this table is passed to function
#  WalkParam so that arguments can be matched with their corresponding
# parameters.

walka: WalkArg ~CONSET  ~IDSET  ~IDSET   ~VALSET  ~Integer  ^VALSET
        ^Integer   ^IDSET    ^CONSET;
                  ~notconin   ~procset ~walkedin ~matchin    ~pnumin ^matchset
        ^pnumout ^walkedout ^notconout

# WHERE:
#     ~notconin - the set of variable ids known to be non-constant upon entering
#          this function.  These vars take on new values in the procedure.
#     ~procset - a set which is carried on the downward traversal of a tree branch.
#          It contains the ids of procedures which have been called during the
#          traversal of this path. It is used solely to detect (prevent) a recursive
#          walk.
#     ~walkedin - is the set of procedure ids which have been walked upon entering
#          this function.  If a procedure id occurs in this set then it has been walked
#          and has a decoration of the set of var ids which are not constant in that
#          particular procedure.
#     ~matchin - the argument list table upon entering this function.
#     ~pnumin - the number of the current argument in the list being traversed.
#     ^matchset - the table of arguments upon exit from this function
#     ^pnumout - the number for the next argument (pnum + 1).
#     ^walkedout - is the set of procedure ids which have been walked upon exiting
#          this function.
#     ^notconout - the set of non-constant var ids upon exiting this function.

-> <BINOP> | <UNOP> | <CONSTANT>
    {:into ~matchin ~pnumin ~(1-100) ^matchset} # put in arg list table
    ^pnumout : pnumin + 1  # increment argument counter
    ^notconout : notconin
    ^walkedout : walkedin
```

```
# It is possible that an argument will be a function call.
# It is handled basically the same way as in the Findcon function.

-> C:<CALL/add ARG:walka>  # "add" is really "call" -- see subtitles def.
    {C:examine ^proc}  # "call" is only possibility for arguements
    {proc -> nextid}  # get procedure id and walk argument list
    {ARG:WalkArg ~Empty ~procset ~walkedin ~Empty ~1 ^argmatch ^pnumout
        ^walkedt ^notcont}
    {:union ~notconin ~notcont ^notconx} # join notcon sets
    {nextid == nextid   nextid in procset  # if proc id is in procset,
        =>  ^notconout : notconx             # then it must be recursive!
            ^walkedout : walkedt             # Don't walk it again.
     otherwise  # no recursion detected yet -- walk the procedure
        =>  {:addset ~nextid ~procset ^pidset} # add it to recurs set
            {proc:CrossFind ~Empty ~pidset ~walkedt ~argmatch ^walkedout ^notcony}
            {:union ~notconx ~notcony ^notconout} # join notcon sets
    }
    {:into ~matchin ~pnumin ~(1-100) ^matchset} # put in arg list table
    ^pnumout : pnumin + 1  # increment argument counter

-> <FETCH>%vn  # argument is a variable -- possibly pass-by-ref
    {:into ~matchin ~pnumin ~vn ^matchset} #put arg no. and vn in table
    ^pnumout : pnumin + 1  # increment argument counter
    ^walkedout : walkedin
    ^notconout : notconin

-> <PAIR L:walka R:walka>
    {L:WalkArg ~notconin ~procset ~walkedin ~matchin ~pnumin ^matcht ^pnumt
        ^walkedt ^notcont}
    {R:WalkArg ~notcont ~procset ~walkedt ~matcht ~pnumt ^matchset ^pnumout
        ^walkedout ^notconout}

-> <>
    ^matchset : matchin
    ^pnumout : pnumin
    ^walkedout : walkedin
    ^notconout : notconin
;


# WalkParam is called from function CrossFind to walk a procedure's parameter
# list.  The parameters are walked and if the parameter is pass-by-ref then the
# corresponding var id is retrieved from the argument list table (passed in as
# argmatch) and that var id is added to the set of non-constants (notcon).

walkp: WalkParam ~CONSET ~VALSET ~Integer  ~IDSET ^Integer    ^CONSET;
                   ~notconin   ~argmatch ~pnumin ~procset ^pnumout ^notconout

# WHERE:
#    ~notconin - the set of variable ids known to be non-constant upon entering
#        this function.  These vars take on new values in the loop (have been
#          assigned a value at least once).
#      ~argmatch - the argument list table upon entering this function
#      ~pnumin - the number of the current parameter in the list being traversed.
#      ~procset - a set which is carried on the downward traversal of a tree branch.
#          It contains the ids of procedures which  have been called during the
#          traversal of this path. It is used solely to detect (prevent) a recursive
#          walk.
#      ^pnumout - number of the next parameter (pnumin + 1)
#      ^notconout - the set of non-constant var ids upon exiting this function.

 -> <DCLN/kind>%vn
    {kind == Ref  # argument is pass-by-ref
       =>  {:from ~argmatch ~pnumin ^argid} # get var id from arg table
           {:addset ~argid ~notconin ^notconout} # add to notcon set
     otherwise # argument was not pass-by-ref
       =>  ^notconout : notconin
    }
    ^pnumout : pnumin + 1  # increment parameter counter

 -> <PAIR L:walkp R:walkp>
    {L:WalkParam ~notconin ~argmatch ~pnumin ~procset ^pnumt ^notcont}
    {R:WalkParam ~notcont ~argmatch ~pnumt ~procset ^pnumout ^notconout}

 -> <>
    ^notconout : notconin
    ^pnumout : pnumin
 ;

# The following function is called when we have a subroutine to walk. PROC is the only
# possible node for this function.

 cross:CrossFind ~CONSET ~IDSET  ~IDSET    ~VALSET ^IDSET     ^CONSET;
                 ~notconin   ~procset  ~walkedin ~argmatch  ^walkedout ^notconout

 # WHERE:
 #     ~notconin - the set of variable ids known to be non-constant upon entering
 #         this function.  These vars take on new values in the loop (have been
 #         assigned a value at least once).
 #     ~procset - a set which is carried on the downward traversal of a tree branch.
 #         It contains the ids of procedures which have been called during the
 #         traversal of this path. It is used solely to detect (prevent) a recursive
 #         walk.
 #     ~walkedin - is the set of procedure ids which have been walked upon entering
 #         this function.  If a procedure id occurs in this set then it has been walked
 #         and has a decoration of the set of var ids which are not constant in that
 #         particular procedure.
 #     ~argmatch - the argument list table upon entering this function
 #     ^walkedout - is the set of procedure ids which have been walked upon exiting
 #         this function.
 #     ^notconout - the set of non-constant var ids upon exiting this function.
# The parameter list is walked first. The set of walked procedures (walkedin) is
# checked to see if the current proc. has been walked yet. If the proc. id occurs in
# the set then it has been walked and is decorated with the set of nonconstants
# derived in the procedure. This set is retrieved and added to the returning set of
# non-constants. If the procedure id does not occur in procset then it has not yet
# been walked. If this is the case, the procedure is walked, the PROC node is
# decorated with the set of non-constants derived from the procedure and the
# proc id is added to the walked set.

-> P:<PROC PARAM:walkp BODY:find>%pid # only possible node here
    {PARAM:WalkParam ~notconin ~argmatch ~1 ~procset ^junk ^argnotcon}
    {pid == pid   pid in walkedin # proc has already been walked
        => {P:examine ^notcon} # get set of non-constants
            ^walkedout : walkedin
      otherwise # proc. has not been walked yet -- walk it
        => {BODY:FindCon ~Empty ~Empty ~procset ~walkedin ^walkedt ^notcon
                ^twiceout}
            {P:decorate ~notcon} # decorate with set of non-constants
            {:addset ~pid ~walkedt ^walkedout} # add proc id to walked set
    }
    {:union ~argnotcon ~notcon ^notconout} # add set of non-constants
;                                          # to returning set of constants.


# The following function is the main driver of the TAG. Traversal of the program
# tree starts here and ends here. Nothing happens until a LOOP is encountered.
# Two passes are made on each loop encountered. On the first pass function
# Findcon is called to analyze the loop. On the second pass any possible
# transformations are made on items which are determined to be loop constant.

tree : STMT ~CONSET ~CONSET ~NODE ~Integer ~NODE ~IDSET   ^IDSET
    ^NODE     ^NODE      ^Integer ^CONSET;
            ~notcon    ~twice     ~gluept ~inIF    ~decpt   ~walkedin ^walkedout
    ^newdecpt  ^newgluept ^iscon   ^newcons

# WHERE:
#    ~notcon - the set of variable ids known to be non-constant upon entering this
#         function. These vars take on new values in the loop (have been assigned a
#         value at least once).
#    ~twice - the set of variable ids known to be non-constant upon entering this
#         function. These vars have been assigned a value in the loop at least
#         twice.
#    ~gluept - The spot reserved above the loop for putting the next  item found to
#         be loop constant upon entering this function.
#    ~inIF - An attribute used to distinguish whether or not we are in a branch of
#         a fork. It is 1 if we are in a fork and  0 otherwise (always 0 in an
#         expression tree).
#    ~decpt - The spot reserved in the DCLN block for putting the next new DCLN
#         node upon entering this function.
#    ~walkedin - is the set of procedure ids which have been walked upon entering
#         this function. If a procedure id occurs in this set then it has been walked
#         and has a decoration of the set of var ids which are not constant in that
#         particular procedure.
#     ^walkedout - is the set of procedure ids which have been walked upon exiting
#         this function.
#     ^newdecpt - The spot reserved in the DCLN block for putting the next new
#         DCLN node upon exiting this function.
#     ^newgluept - The spot reserved above the loop for putting the next item found
#         to be loop constant upon exiting this function.
#     ^iscon - Returns the subtree's constant weight.
#     ^newcons - A set containing var ids which were created for loop constant
#         expressions that were moved out somewhere below the current level of
#         the tree traversal.

# At both the PROG node and the PROC nodes the first thing done is a place is
# made in the DCLN block for inserting new var. DCLNs.  The program/procedure
# tree is then walked.

-> <PROG PARAM BODY:tree>%jj | <PROC PARAM BODY:tree>%jj
   {BODY:getdec ~jj ^bdecpt} #get pointer for glueing in new dcln nodes
   {BODY:STMT ~notcon ~twice ~<> ~0 ~bdecpt ~walkedin ^walkedout ^newdecpt
      ^newgluept ^iscon ^newcons}

-> <TYPE> | <>
   ^iscon : 2
   ^newgluept : gluept
   ^newdecpt : decpt
   ^newcons : Empty
   ^walkedout : walkedin

# Each time a DCLN node is encountered in the program/procedure tree a counter
# must be incremented so that when we need to create new var ids we will have a
# new one (unused consecutive number).  The predeclared function "newvarno",
# each time called, increments a counter for this purpose.

-> <DCLN>
   {:newvarno ^dontcare} # increment so we can create new ones
   ^iscon : 2
   ^newgluept : gluept
   ^newdecpt : decpt
   ^newcons : Empty
   ^walkedout : walkedin

-> <GOTO> | <CALL>
   ^iscon : 0
   ^newgluept : gluept
   ^newdecpt : decpt
   ^newcons : Empty
   ^walkedout : walkedin

-> <UNOP T:tree>
   {T:STMT ~notcon ~twice ~gluept ~0 ~decpt ~walkedin ^walkedout ^newdecpt
```

```
        ^newgluept ^expiscon ^newcons}
    ^iscon : expiscon*expiscon  # increase expression weight

# At a PAIR node, the left subtree is walked, and then the right subtree is walked.
# The sets of new constants returned are joined together and then the constant
# weight of each subtree is checked.  1) If the left tree's constant weight is great
# enough (iscon >> 2) and the right tree has no constant weight then the left
# subtree is loop constant and can be moved out (provided that we are in a loop
# (gluept not equal to NULL)).  2) If the right subtree's constant weight is large
# enough and the left has no constant weight then the right subtree is moved out.
# 3) Otherwise, we are not in a loop, both subtrees had some constant weight, or
# niether subtree had a constant weight.  In cases 1 and 2 the PAIR node is
# transformed by removing the subtree which was determined to be constant and
# replacing it with a NULL.  The constant subtree is passed to function "stmoveit"
# to be glued in above the loop.  In case 3 no transformation takes place.

-> P:<PAIR L:tree R:tree>
    {L:STMT ~notcon ~twice ~gluept ~inIF ~decpt ~walkedin ^walkedt ^lnewdec
        ^lnewglue ^liscon ^lnewcons}
    {R:STMT ~notcon ~twice ~lnewglue ~inIF ~lnewdec ~walkedt ^walkedout ^newdecpt
        ^rnewglue ^riscon ^rnewcons}
    {:union ~lnewcons ~rnewcons ^newcons} # join sets of new consts
    {liscon>>2  riscon==0  gluept><<> #left const, right not, in a loop
        =>  {rnewglue:stmoveit ~L ^newgluept} # glue branch back in
     riscon>>2  liscon==0  gluept><<> #right const, left not, in a loop
        =>  {rnewglue:stmoveit ~R ^newgluept} # glue branch back in
     otherwise # not in loop, both trees were constant or both weren't
        =>  ^newgluept : rnewglue # keep the same gluept
    }
    ^iscon : (liscon*riscon)  # increase expression weight

=> <PAIR <> R>  |  <PAIR L <>>  |  P

# Basically the same thing happens at the BINOP node as does at the PAIR node.
# The difference is that at a BINOP node the constant expression is moved out and
# replaced with a FETCH to a new CELL.  Function "exmoveit" is called to glue the
# expression back in above the loop.  Exmoveit returns the DCLN node of the new
# variable cell so that the new <FETCH <CELL>> can be decorated here.

-> B:<BINOP/op L:tree R:tree>
    {B:examine ^tipe} # retrieve expression type
    {L:STMT ~notcon ~twice ~gluept ~inIF ~decpt ~walkedin ^walkedt ^lnewdec
        ^lnewglue ^liscon ^lnewcons}
  {R:STMT ~notcon ~twice ~lnewglue ~inIF ~lnewdec ~walkedt ^walkedout ^rnewdec
        ^rnewglue ^riscon ^rnewcons}
    {:union ~lnewcons ~rnewcons ^tempnewcons} # join new constant sets
```

```
     {liscon>>2  riscon==0  gluept><<> # left const, right not, in loop
        =>  {mewglue:exmoveit ~tipe ~L ~mewdec ^newdecpt ^newgluept ^decnode
                 ^lmnewcons}
             {:union ~lmnewcons ~tempnewcons ^newcons}
             {decnode -> varno} # retrieve new variable id
             {C <- varno}   # hang variable id on new cell
             {C:decorate ~decnode} # decorate new cell with DCLN node
             {F <- varno}   # hang variable id on FETCH to new cell
             {F:decorate ~decnode} # decorate FETCH with DCLN node
      riscon>>2  liscon==0  gluept><<> # left const, right not, in loop
        =>  {mewglue:exmoveit ~tipe ~R ~mewdec ^newdecpt ^newgluept ^decnode
                 ^rmnewcons}
             {:union ~rmnewcons ~tempnewcons ^newcons}
             {decnode -> varno} # retrieve new var id.
             {C <- varno}   # hang variable id on new cell
             {C:decorate ~decnode} # decorate new cell with DCLN node
             {F <- varno}   # hang variable id on FETCH to new cell
             {F:decorate ~decnode} # decorate FETCH with DCLN node
      otherwise
        =>  ^newgluept : mewglue  # keep the same loop glue point
             ^newdecpt : decpt  # keep the same dcln glue point
             ^newcons : tempnewcons
     }
    ^iscon : (liscon*riscon)  # increase expression weight

=> <BINOP F:<FETCH C:<CELL>> R>  |  <BINOP L F:<FETCH C:<CELL>>>  |  B

# At an IF node the three subtrees are walked but only the expression subtree's
# constant weight is tested for transformation.  The expression is moved out
# only if one of the other trees (THEN or ELSE) has a constant weight of 0
# liscon*eiscon = 0).  This is so because if both subtrees, THEN and ELSE, have
# some constant weight then it is possible that the whole IF statement may be
# loop constant.  If it is then the whole statement will be moved out at the next
# level up in the  program tree.

-> I:<IF X:tree T:tree E:tree>
    {X:STMT ~notcon ~twice ~gluept ~0 ~decpt ~walkedin ^walkedx ^xdec ^xgluept
        ^xiscon ^xnewcons}
    {T:STMT ~notcon ~twice ~xgluept ~1 ~xdec ~walkedx ^walkedt ^tdec ^tgluept ^tiscon
        ^tnewcons}
    {E:STMT ~notcon ~twice ~tgluept ~1 ~tdec ~walkedt ^walkedout ^edec ^egluept
        ^eiscon ^enewcons}
    {:union ~xnewcons ~tnewcons ^holdnewcons} # join new constant sets
    {:union ~holdnewcons ~enewcons ^tempnewcons}
    {xiscon>>2  tiscon*eiscon==0  gluept ><<> # exp. const, t & e not, in loop
        =>  {egluept:exmoveit ~<TYPE <CONSTANT>%0 <CONSTANT>%1> ~X ~edec
                 ^newdecpt ^newgluept ^decnode ^xmnewcons}
             {:union ~xmnewcons ~tempnewcons ^newcons}
             {decnode -> varno} # retrieve new var id
             {C <- varno} # decorate new cell with var id
             {C:decorate ~decnode} # decorate new cell with its DCLN
```

```
            {F <- varno} # decorate new FETCH with var id
            {F:decorate ~decnode} # decorate new FETCH with var DCLN
            ^iscon : 0
     otherwise # not in loop or exp not const or t & e had some weight
       =>  {inIF == 0 # not in branch of fork
               => ^iscon : (xiscon*tiscon*eiscon) # increase weight
             otherwise # in branch of fork -- can't move this stmt
               => ^iscon : 0
            }
            ^newgluept : egluept  # keep the same loop glue point
            ^newdecpt : decpt     # keep the same dcln glue point
            ^newcons : tempnewcons
    }

=> <IF F:<FETCH C:<CELL>> T E> | I
```

# At a COPY node we, again just like the PAIR and BINOP nodes, check the constant
# weight of the subtrees and make the appropriate transformations. If the right
# subtree is to be moved, we must send an Integer type node to function exmoveit.

```
-> CY:<COPY L:tree R:tree>%varid
    {L:STMT ~notcon ~twice ~gluept ~0 ~decpt ~walkedin ^walkedt ^ldecpt ^lnewglue
       ^liscon ^lnewcons}
    {R:STMT ~notcon ~twice ~lnewglue ~0 ~ldecpt ~walkedt ^walkedout ^rdecpt
       ^rnewglue ^riscon ^rnewcons}
    {:union ~lnewcons ~rnewcons ^tempnewcons} # join new constant sets
    {liscon>>2  riscon==0  gluept><<> # left const, right not, in loop
       =>  {CY:examine ^dec} # retrieve var DCLN
            {dec:examine ^tipe} # get var type
            {rnewglue:exmoveit ~tipe ~L ~rdecpt ^newdecpt ^newgluept ^decnode
               ^lmnewcons}
            {:union ~lmnewcons ~tempnewcons ^newcons}
            {decnode -> varno} # retrieve var id of new DCLN
            {C <- varno}   # hang variable id on new cell
            {C:decorate ~decnode} # decorate new cell with DCLN node
            {F <- varno}   # hang variable id on FETCH to new cell
            {F:decorate ~decnode} # decorate FETCH with DCLN node
            ^iscon : 0
     liscon==0  riscon>>2  gluept><<> # right const, left not, in loop
       =>  {rnewglue:exmoveit ~<TYPE <CONSTANT>%Min <CONSTANT>%Max>
               ~R ~rdecpt ^newdecpt ^newgluept ^decnode ^rmnewcons}
            # send a numeric type because its an address calculation
            {:union ~rmnewcons ~tempnewcons ^newcons}
            {decnode -> varno} # retrieve new var id
            {C <- varno}   # hang variable id on new cell
            {C:decorate ~decnode} # decorate new cell with DCLN node
            {F <- varno}   # hang variable id on FETCH to new cell
            {F:decorate ~decnode} # decorate FETCH with DCLN node
            ^iscon : 0
    liscon>>2  riscon==2  varid in twice  gluept><<>
       =>  {rnewglue:exmoveit ~<TYPE <CONSTANT>%Min <CONSTANT>%Max>
```

```
                ~R ~rdecpt ^newdecpt ^newgluept ^decnode ^rmnewcons}
             # send a numeric type because its an address calculation
             {:union ~rmnewcons ~tempnewcons ^newcons}
             {decnode -> varno} # retrieve new var id
             {C <- varno}  # hang variable id on new cell
             {C:decorate ~decnode} # decorate new cell with DCLN node
             {F <- varno}  # hang variable id on FETCH to new cell
             {F:decorate ~decnode} # decorate FETCH with DCLN node
             ^iscon : 0
      liscon>>2  riscon>>2  id in twice  gluept><<>
         => # move expression tree out
             {CY:examine ^dec} # retrieve var DCLN
             {dec:examine ^tipe} # get var type
             {rnewglue:exmoveit ~tipe ~L ~rdecpt ^tdecpt ^lmnewglue ^ldecnode
                ^lmnewcons}
             {:union ~lmnewcons ~tempnewcons ^cellnewcons}
             {ldecnode -> varno} # retrieve var id of new DCLN
             {C <- varno}  # hang variable id on new cell
             {C:decorate ~ldecnode} # decorate new cell with DCLN node
             {F <- varno}  # hang variable id on FETCH to new cell
             {F:decorate ~ldecnode} # decorate FETCH with DCLN node
             # move cell calculation out
             {lmnewglue:exmoveit ~<TYPE <CONSTANT>%Min <CONSTANT>%Max>
                ~R ~tdecpt ^newdecpt ^newgluept ^rdecnode ^rmnewcons}
             # send a numeric type because its an address calculation
             {:union ~rmnewcons ~cellnewcons ^newcons}
             {rdecnode -> id} # retrieve new var id
             {D <- id}  # hang variable id on new cell
             {D:decorate ~rdecnode} # decorate new cell with DCLN node
             {E <- id}  # hang variable id on FETCH to new cell
             {E:decorate ~rdecnode} # decorate FETCH with DCLN node
             ^iscon : 0
      otherwise  # not in loop or niether tree was const or both were
         => {inIF == 0 # not in a fork branch
                => ^iscon : liscon*riscon
             otherwise # in a fork branch -- cant move this stmt
                => ^iscon : 0
             }
             ^newgluept : rnewglue # keep the same loop glue point
             ^newdecpt : rdecpt    # keep the same DCLN glue point
             ^newcons : tempnewcons
      }

   =>  <COPY F:<FETCH C:<CELL>> R>  |  <COPY L F:<FETCH C:<CELL>>>
      | <COPY L F:<FETCH C:<CELL>>>
      | <COPY F:<FETCH C:<CELL>> E:<FETCH D:<CELL>>> | CY


   -> F:<FETCH V:tree>%varid
      {V:STMT ~notcon ~twice ~gluept ~0 ~vdecpt ~walkedin ^walkedout ^vdecpt ^vgluept
         ^viscon ^vnewcons}
```

```
    {viscon>>2  varid in notcon  gluept><<> # address const, var not
       => {vgluept:exmoveit ~<TYPE <CONSTANT>%Min <CONSTANT>%Max> ~V
              ~vdecpt ^newdecpt ^newgluept ^decnode ^tnewcons}
           # send a numeric type because its an address calculation
           {:union ~vnewcons ~tnewcons ^newcons}
           {decnode -> varno} # retrieve new var id
           {C <- varno}   # hang variable id on new cell
           {C:decorate ~decnode} # decorate new cell with DCLN node
           {E <- varno}   # hang variable id on FETCH to new cell
           {E:decorate ~decnode} # decorate FETCH with DCLN node
           ^iscon : 0
    viscon==2  # the subtree was just a CELL
       =>  {varid==varid  varid in notcon  # var not loop-constant
              => ^iscon : 0
            otherwise # the cell is const but not big enough
              => ^iscon : 2
           }
           ^newdecpt : vdecpt
           ^newgluept : vgluept
           ^newcons : vnewcons
   otherwise # this FETCH is not loop constant
       =>  ^iscon : 0
           ^newdecpt : vdecpt
           ^newgluept : vgluept
           ^newcons : vnewcons
   }

=> <FETCH E:<FETCH C:<CELL>>> | F | F


-> <CONSTANT> | <CELL>
   ^iscon : 2    # return minimum constant expression weight
   ^newgluept : gluept  # pass everything else on through
   ^newdecpt : decpt
   ^newcons : Empty
   ^walkedout : walkedin

# When a LOOP node is encountered, two passes are made on it.  Function FindCon
# is called to analyze the loop and return the sets of non-constants (notconset
# and twiceset).  These sets are passed to function STMT when the loop body is
# walked for transformation.  However, before the body is walked, the LOOP node
# is transformed so that there is a spot above the loop for glueing the items
# which are found to be loop constant.  This node (P) is passed to function STMT
# as "gluept" and also indicates to functions which perform transformations that
# we are indeed in a loop.  After returning from the transformation pass on the
# loop we check to see if the whole loop was constant.  If so, then the whole loop
# body is moved out.  After all loop constant items are moved out we must rewalk
# them (moved items) for the next level.  This may have been a nested loop, so we
# must now check to see if these items are also loop constant on the next level out.
```

```
-> <LOOP B:tree>
    {B:FindCon ~Empty ~Empty ~Empty ~walkedin ^walkedt ^notconset ^twiceset}
    {B:WalkLoop ~notconset ~twiceset ~P ~0 ~decpt ~walkedt ^walkedx ^ldec ^loopgluept
        ^loopiscon ^lnewcons}

    # lnewcons is received back containing the new vars created to
    # hold the values of expressions moved out of this loop        .

    {:union ~lnewcons ~notcon ^newnotcon} # add them to nonconst. set

    # The items which were moved out of this loop are rewalked
    # in case this loop node is in a loop.

    {P:ReWalk ~newnotcon ~twice ~gluept ~inIF ~ldec ~walkedx ^newdecpt ^newgluept
        ^newcons}
    ^iscon : 0
    ^walkedout : walkedx

=> P:<PAIR <> <LOOP B>>

;


# This function is called from function STMT to start a transformation pass on a loop.

walkl:WalkLoop ~CONSET ~CONSET ~NODE ~Integer ~NODE ~IDSET    ^IDSET
        ^NODE    ^NODE      ^Integer ^CONSET;
                  ~notcon    ~twice      ~gluept  ~inIF     ~decpt   ~walkedin ^walkedout
        ^newdecpt ^newgluept ^iscon   ^newcons


# WHERE:
#    ~notcon - the set of variable ids known to be non-constant upon entering this
#        function.  These vars take on new values in the loop (have been assigned a
#        value at least once).
#    ~twice - the set of variable ids known to be non-constant upon entering this
#        function.  These vars have been assigned a value in the loop at least
#        twice.
#    ~gluept - The spot reserved above the loop for putting the next item found to
#        be loop constant upon entering this function.
#    ~inIF - An attribute used to distinguish whether or not we are in a branch of
#        a fork.  It is 1 if we are in a fork and 0 otherwise (always 0 in an
#        expression tree).
#    ~decpt - The spot reserved in the DCLN block for putting the next new DCLN
#        node upon entering this function.
#    ~walkedin - is the set of procedure ids which have been walked upon entering
#        this function.  If a procedure id occurs in this set then it has been walked
#        and has a decoration of the set of var ids which are not constant in that
#        particular procedure.
#    ^walkedout - is the set of procedure ids which have been walked upon exiting
#        this function.
#    ^newdecpt - The spot reserved in the DCLN block for putting the next new
#         DCLN node upon exiting this function.
#    ^newgluept - The spot reserved above the loop for putting the next item found
#         to be loop constant upon exiting this function.
#    ^iscon - Returns the subtree's constant weight.
#    ^newcons - A set containing var ids which were created for loop constant
#         expressions that were moved out somewhere below the current level of
#         the tree traversal.

-> I:<IF X:tree T:tree E:tree>
    {X:STMT ~notcon ~twice ~gluept ~0 ~decpt ~walkedin ^walkedx ^xdec ^xgluept
       ^xiscon ^xnewcons}
    {T:STMT ~notcon ~twice ~xgluept ~0 ~xdec ~walkedx ^walkedt ^tdec ^tgluept ^tiscon
       ^tnewcons}
    {E:STMT ~notcon ~twice ~tgluept ~0 ~tdec ~walkedt ^walkedout ^edec ^egluept
       ^eiscon ^enewcons}
    {:union ~xnewcons ~tnewcons ^holdnewcons} # join new constant sets
    {:union ~holdnewcons ~enewcons ^tempnewcons}
    {xiscon>>2  # loop expression is constant
       =>  {egluept:exmoveit ~<TYPE <CONSTANT>%0 <CONSTANT>%1> ~X ~edec
              ^newdecpt ^newgluept ^decnode ^xmnewcons}
           {:union ~xmnewcons ~tempnewcons ^newcons}
           {decnode -> varno} # retrieve new var id
           {C <- varno} # decorate new cell with var id
           {C:decorate ~decnode} # decorate new cell with its DCLN
           {F <- varno} # decorate new FETCH with var id
           {F:decorate ~decnode} # decorate new FETCH with var DCLN
     otherwise   # expression is not loop-constant
       =>  ^newgluept : egluept  # keep the same loop glue point
           ^newdecpt : decpt     # keep the same dcln glue point
           ^newcons : tempnewcons
    }
    ^iscon : 0

=> <IF F:<FETCH C:<CELL>> T E>  | I

;


# Obviously there may be nested loops.  Items which are loop constant in the
# inside loop may also be constant in the outside loop.  Function ReWalk is called
# from function STMT after the transformation pass of a LOOP is completed.  This
# function rewalks the items which were moved out of the loop so that their loop
# constantness can be checked for the level which they are now on.


levl:ReWalk ~CONSET ~CONSET ~NODE ~Integer ~NODE ~IDSET    ^NODE
       ^NODE     ^CONSET;
              ~notcon     ~twice     ~gluept ~inIF    ~decpt    ~walkedin ^newdecpt
       ^newgluept ^newcons
```

```
# WHERE:
#    ~notcon - the set of variable ids known to be non-constant upon entering this
#         function. These vars take on new values in the loop (have been assigned a
#         value at least once).
#     ~twice - the set of variable ids known to be non-constant upon entering this
#         function. These vars have been assigned a value in the loop at least
#         twice.
#    ~gluept - The spot reserved above the loop for putting the next item found to
#         be loop constant upon entering this function.
#    ~inIF - An attribute used to distinguish whether or not we are in a branch of
#         a fork. It is 1 if we are in a fork and 0 otherwise (always 0 in an
#         expression tree).
#    ~decpt - The spot reserved in the DCLN block for putting the next new DCLN
#         node upon entering this function.
#    ~walkedin - is the set of procedure ids which have been walked upon entering
#         this function. If a procedure id occurs in this set then it has been walked
#         and has a decoration of the set of var ids which are not constant in that
#         particular procedure.
#    ^newdecpt - The spot reserved above the loop for putting the next item found
#         to be loop constant upon exiting this function.
#    ^newgluept - The spot reserved in the DCLN block for putting the next new
#         DCLN node upon exiting this function.
#    ^newcons - A set containing var ids which were created for loop constant
#         expressions that were moved out somewhere below the current level of
#         the tree traversal.


-> P:<PAIR M:tree L:tree> #M = the constants moved out of loop L
                          #L  = the loop we just worked on
   {M:STMT ~notcon ~twice ~gluept ~inIF ~decpt ~walkedin ^walkedout ^newdecpt
      ^tgluept ^tiscon ^newcons}
   {tiscon>>2  # all items were constant for this level
      =>  {tgluept:stmoveit ~M ^newgluept} # glue branch back in
    otherwise
      =>  ^newgluept : gluept
   }

=> <PAIR <> L> | P

;


# The following function is used to glue an expression which has been determined
# to be loop constant back into the tree at the "gluept" above the loop. A new var
# id is created to assign the expression to, a DCLN node for the new var is created
# using function gluedec, the appropriate transformations are made to glue in the
# constant expression, the new nodes are decorated with the necessary data and a
# new loop gluept is returned.
```

```
exmove:exmoveit ~NODE    ~NODE ~NODE ^NODE      ^NODE       ^NODE
        ^CONSET;
                       ~twigtype ~twig    ~decpt    ^newdecpt  ^newgluept ^newdecnode
        ^newcons

# WHERE:
#      ~twigtype - The TYPE node for the expression being glued.
#      ~twig - The expression tree which is being glued into the tree
#      ~decpt - The spot reserved in the DCLN block for putting the next new DCLN
#           node upon entering this function.
#      ^newdecpt - The spot reserved in the DCLN block for putting the next new
#           DCLN node upon exiting this function.
#      ^newgluept - The spot reserved above the loop for putting the next item found
#           to be loop constant upon exiting this function.
#      ^newdecnode - The new DCLN node which was created for the new var to hold
#           this expression.
#      ^newcons - A set containing var ids which were created for loop constant
#           expressions that were moved out somewhere below the current level of
#           the tree traversal.

-> <PAIR <> R:tree>  # only structure possible for this function
    {:newvarno ^varno} # get a new id number
    {decpt:gluedec ^newdecpt ^newdecnode} # glue in the new DCLN
    {:addset ~varno ~Empty ^newcons} # add the var to the notcon set
    {newdecnode:decorate ~twigtype} # decorate new DCLN node with type
    {newdecnode <- varno} # decorate the new DCLN node with its var id
    {C <- varno} # decorate the new CELL with the var id
    {CY <- varno} # decorate the new COPY node with the var id
    {C:decorate ~newdecnode} # decorate new cell with DCLN node
    {CY:decorate ~newdecnode} # decorate new COPY with DCLN node

=> <PAIR CY:<COPY twig C:<CELL>> newgluept:<PAIR <> R>>

;


# The following function is used to glue a statement which has been determined
# to be loop constant back into the tree at the "gluept" above the loop.  The
# appropriate transformations are made to glue in the constant expression and a
# new gluept is returned.

 stmove:stmoveit ~NODE  ^NODE;
                       ~twig     ^newgluept

# WHERE:
#       ~twig - The constant statement to be glued in.
#       ^newgluept - The spot reserved above the loop for putting the next item found
#            to be loop constant upon exiting this function.
```

```
-> <PAIR <> R:tree>  # only structure possible for this function

=> <PAIR twig newgluept:<PAIR <> R>>

;


# Getdec is called before the beginning of the program tree traversal and each
# procedure tree traversal.  It is used to find and reserve a spot at the DCLN block
# of the procedure for placing new DCLNs.

getglu:getdec ~Integer ^NODE;
               ~junk    ^newdecpt

# WHERE:
#      ~junk - An attribute used to force dependencies.
#      ^newdecpt - the new DCLN gluept

-> <PAIR guts:tree return:tree>
      ^newdecpt : guts
;

 # Gluedec is used to create a new DCLN node.  The node is created and its
 # reference along with a new DCLN gluept are returned.

adddec:gluedec ^NODE       ^NODE;
        ^newdecpt ^newdecnode

# WHERE:
#       ^newdecpt - a new DCLN gluept.
#       ^newdecnode - a reference to the new DCLN node just created.

-> <PAIR L:tree R:tree>

 => <PAIR newdecpt:<PAIR L newdecnode:<DCLN>> R>

;


end LoopCon
;
```

OPTIMIZER DESIGN USING TRANSFORMATIONAL ATTRIBUTE GRAMMARS APPLIED TO INTERMEDIATE LOW-LEVEL TREES

by

STEPHEN V. YOUNG

B. S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

A TAG (Transformational Attribute Grammar) is used to specify program optimizations. TAGs operate on an intermediate program representation called ILTs (Intermediate Low-level Trees). ILTs are superior to other intermediate languages in that they preserve the high-level structure of the source program while including low-level details for the target machine code. Analysis is accomplished by passing information contained in the tree with the attributes of the grammar to wherever it is needed. Optimization is accomplished by manipulating the tree with transformations specified by the grammar. Remote optimizations can be performed with TAGs utilizing the context information which is available through its attributes. TAGs can be used to implement most types of optimization and with the domain of ILTs only one intermediate representation is needed for all phases of optimization. To illustrate the efficiency of designing optimizers with TAGs, the optimizations of constant folding, dead code elimination and code motion (loop constant removal) are discussed and the implementation of these optimizations in TAGs is presented.