Recognition and Selection of Idioms for Code Optimization*

Lawrence Snyder

Technical Report #132, September 1978

Abstract

Idioms are frequently occurring expressions that programmers use for *logically* primitive operations for which no primitive construct is available in the language. For example, in ALGOL-60 the expression abs(X-X÷2×2) is idiomatic for *parity of X*. With optimization as a motive, two problems, idiom recognition and selection, are defined. Recognition is solved in O(n log n) time (worst case), O(n) time (average case) on a uniform cost RAM. Selection is solved in O(n) time. Ambiguity is solved in $O(n^2)$ time and is related to resolution theorem proving.

*This work was supported in part by National Science Foundation Grant MCS78-04749.

1. *Introduction*

In natural languages an *idiom* is "the syntactical, grammatical or structural form peculiar to a language" [1]. Perlis [2] has observed that programming language usage also encourages the development of idioms and with Rugaber has compiled an impressive list of idioms used in APL [3]. For programming languages we may define an idiom as a construction used by programmers for a logically primitive operation for which no language primitive exists. For example, ALGOL-60 programmers use

$$\text{abs}(X-X\div 2\times 2) \tag{1.1}$$

to test the *parity* of an integer X.

Idioms probably arise in every programming language. In APL, with its rich operator repertoire and weak control structures, idioms tend to arise at the "expression level." In scalar languages such as ALGOL-60, they tend to be found at the "statement level." The current discussions of structured programming may be viewed as discussions about statement level idioms.

The importance of idioms is embodied in two properties:

(i) idioms are a vehicle by which experienced programmers can pass their knowledge of the language to beginners, and,

(ii) idioms often admit important optimizations.

The first point manifests itself in the classroom, textbooks and in much of the current discussion of programming style. The second point motivates this report.

As an example of a potential optimization, we note that expression

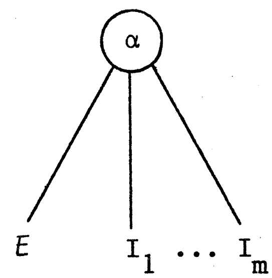
(1.1) can be realized by a masking operation rather than the more expensive arithmetic. This may not be a large savings, but it is comparable with the improvements using classical optimizations [4]. For a language like APL, however, where a single expression can represent a prodigious amount of computation, Miller [5] has shown that the savings from optimizing idioms can be very handsome. For example,

$$(V \neq C)/V \tag{1.2}$$

is the *delete-C* idiom of APL for removing occurrences of the element C from a vector V. The two loops of the literal translation (to form a mask of occurrences of V and to compress them out) can be combined into one loop saving code, instruction executions and the allocation/deallocation of a vector temporary.

The first task in realizing idiom optimizations is to *identify* a list of idioms and their corresponding code segments. Idiom identification is obviously language sensitive and is beyond the scope of this investigation. (For APL Perlis and Rugaber [3] have identified idioms, Holls [6] has compiled another set, and Miller [5] has begun finding efficient code segments.) Hereafter we will assume that a list of idioms $I_1,\ldots,I_m$ has been identified and names $C_1,\ldots,C_m$ for their code pieces have been assigned. We represent idioms by their parse tree.

The second task is to *recognize* the idioms $I_1,\ldots,I_m$, if any, in a given arithmetic expression E. That is, find for each node v in E which idioms match a subtree rooted at v. One *might* abstract this as a common subexpression recognition task for a new composit expression



(where α is a dummy operator), but this is *not* correct because the variable operands of idioms are "free variables" that match constants, variables or expressions*. Thus, unlike common subexpressions that occur at the "bottom" of parse trees, idioms can occur throughout. For example, in Figure 1 the *parity* idiom (I_1) is recognized in the midst of the parse tree, and is replaced by the name of the code segment. The free variable X matches the expression B+C.

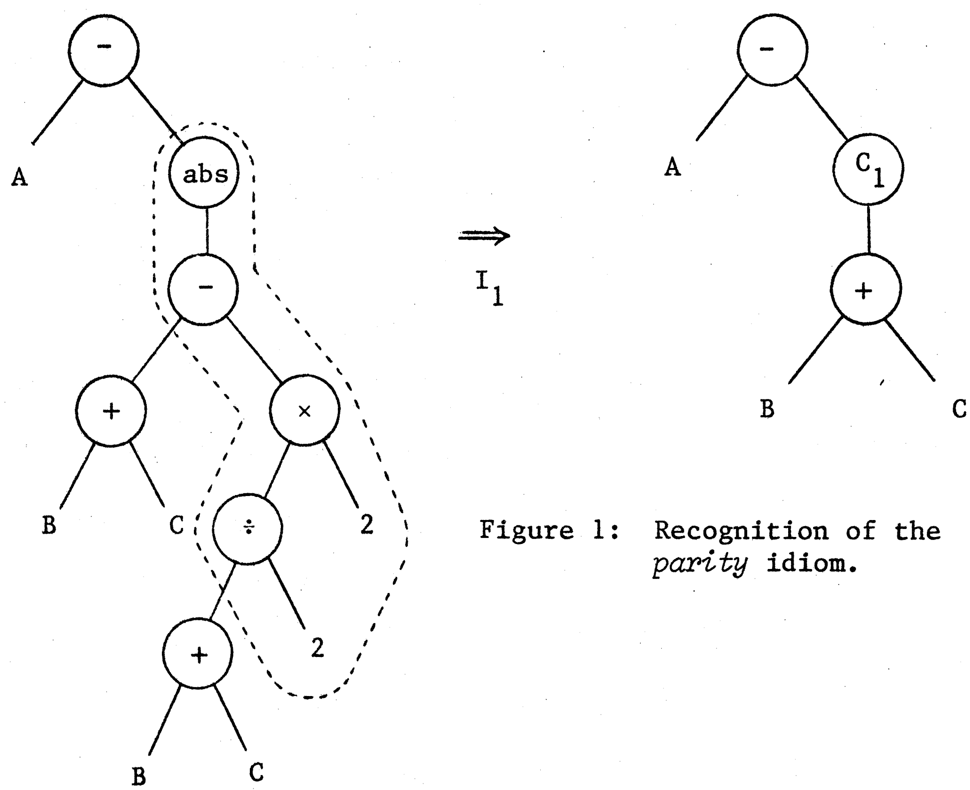


Figure 1: Recognition of the *parity* idiom.

----

*Of course, explicitly given constants (e.g. 2 in (1.1)) must match exactly.

The important consequence of the fact that idioms match throughout a parse tree is that two instances can *overlap*. To illustrate this phenomenon, consider two APL idioms. The *zero-C* idiom

$$I_2\text{: } (X{\neq}C)\backslash(X{\neq}C)/X \tag{1.3}$$

sets all occurrences of the element C in vector X to zero*. The *merge-by-B* idiom

$$I_3\text{: } (B\backslash U)+(\sim B)\backslash W \tag{1.4}$$

uses a boolean vector B to control the merging of elements from vectors U and W according as $B[I]$ is 1 or 0, respectively.

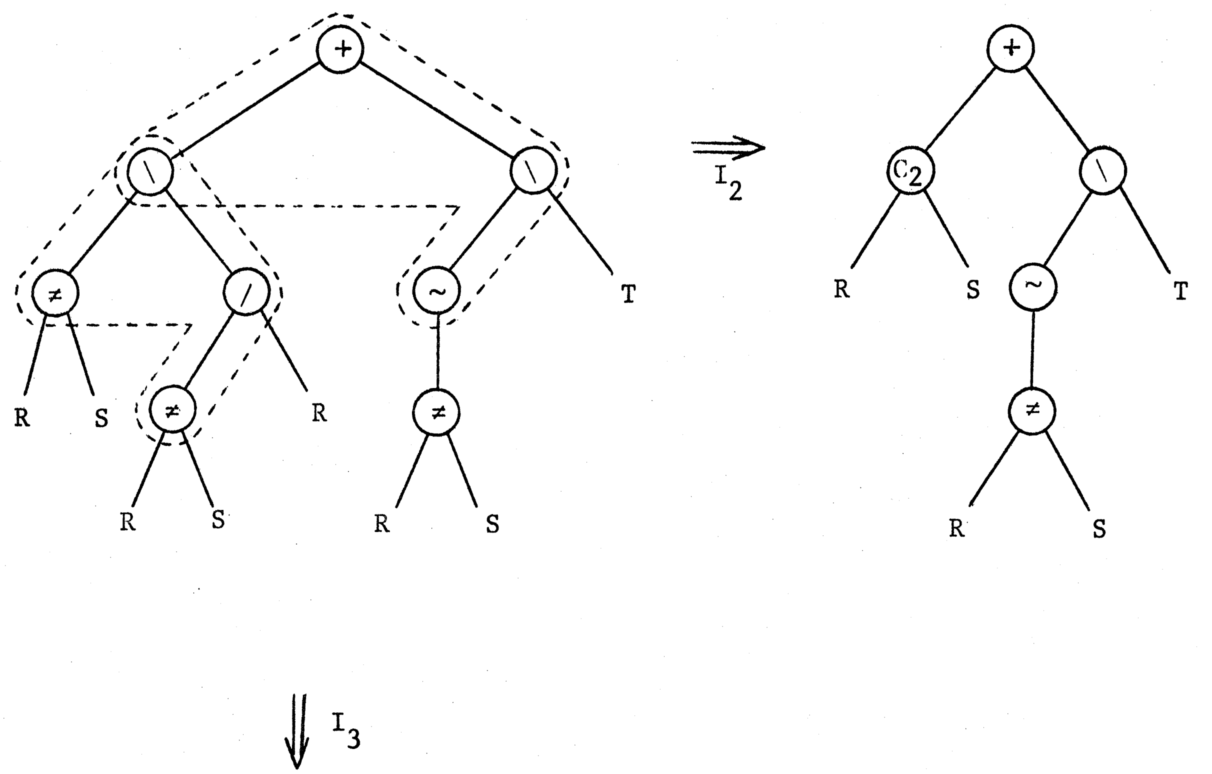
Now, consider the APL expression

$$((R{\neq}X)\backslash(R{\neq}S)/R)+(\sim R{\neq}S)\backslash T \tag{1.5}$$

whose parse tree is given in Figure 2**. The recognized instances of idioms I_2 and I_3 share an expansion operation (\) and this means that they cannot both be replaced. This is obviously true from a syntactic point of view, since once the instance of I_2 has been replaced by the C_2 vertex, I_3 no longer matches, and vice versa. But there is a more fundamental reason why both optimizations cannot be realized: the logically primitive nature of idioms. In general what occurs in the hybrid idiom code is a loss of the usual correspondence between the nodes of the parse tree and instruction sequence in the code segment. Thus, it is usually difficult to decompose the code and recombine pieces of the two segments to accomplish the overlap.

Zero-C would probably be written $A\backslash(A{\leftarrow}X{\neq}C)/X$ or as $A{\leftarrow}X{\neq}C$ and $A\backslash A/X$ on separate lines [3]; we repeat the operands simply as a visual aid. Note too, that *delete-C* is a subidiom of *zero-C*.

**We write $R{=}S$ as $\sim R{\neq}S$; use of such identities is discussed in section 5.



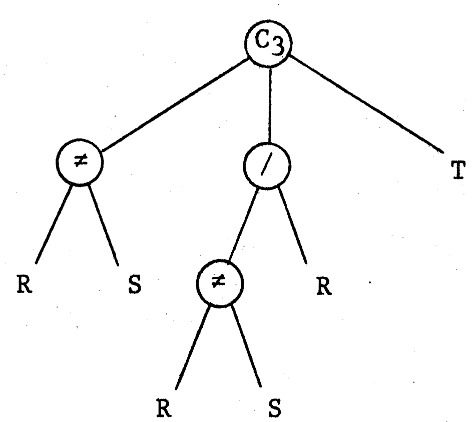


Figure 2: Recognition of idioms I_2 and I_3 in expression (1.5), where they overlap.

This inability to decompose and recombine is obvious for the one (mask) instruction implementing the *parity* operation, but it is less clear for *zero-C* and the *merge-by-B* idioms. To understand the difficulty, we recognize

```
C2(T,X,C):
comment implements T ← (X≠C)\(X≠C)/X;
for i←1 step 1 until ρX do
    T[i] ← if X[i]≠C then X[i] else 0;
```

and

```
C3(T,B,U,W):
comment implements T ← (B\U)+(~B)\W;
j←0; k←0;
for i←1 step 1 until ρB do
    T[i] ← if B[i]=1 then U[j←j+1] else W[k←k+1];
```

as reasonable code segments for *zero-C* and *merge-by-B*, respectively. The apparent way to combine these two pieces of text is to leave the expansion operation that is common to I_2 and I_3 uncompleted by the C_2 routine and then complete it as part of the C_3 routine. But as can easily be seen in Figure 3, one of the required operands to C_3, the value U(3 4 5 in the figure) resulting from $(R{\neq}S)/R$ is never explicitly produced by C_2! The difficulty of merging C_2 and C_3 is now apparent. Since both idioms cannot lead to optimizations it is probably wise to chose the one with the biggest payoff.

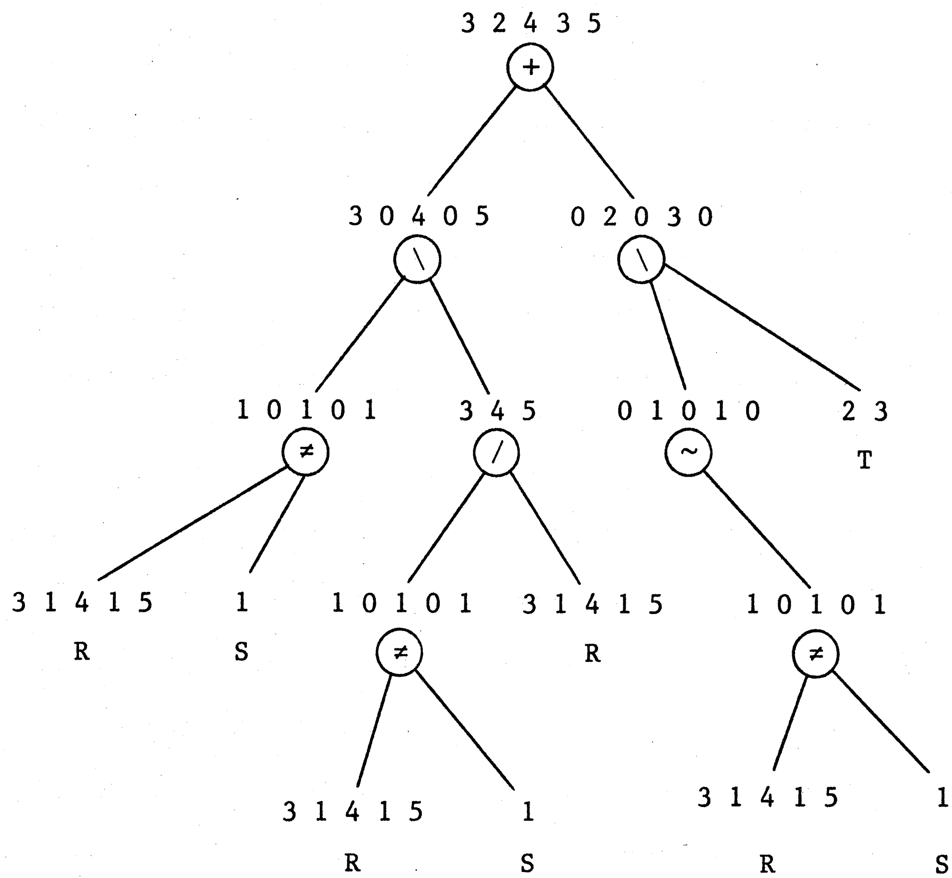


Figure 3: Evaluation of expression (1.5) where
$R \leftrightarrow 3\ 1\ 4\ 1\ 5$, $S \leftrightarrow 1$, $T \leftrightarrow 2\ 3$.

Thus the third task (in addition to identification and recognition) is *selection* of a nonoverlapping subset of the recognized idioms of a parse tree that maximizes the benefit of the optimizations. For a scalar language, *size* is probably an adequate benefit measure since the savings tend to be proportional to the number of nodes eliminated. Thus the selection problem is equivalent to finding the idioms forming a maximal cover. For a language like APL the payoff is usually sensitive to the size of the operands and complexity of the idiom code segment as compared to the corresponding naive code. Thus, we hypothesize a more general payoff function $\pi(E)$ that measures the benefits of each of the recognized idioms of E. This allows different occurrences of the same idiom to be sensitive to their particular operands. Selection is solved by finding the maximum global benefit with respect to E.

Finally, we note that the idiom recognition and selection problems appear to be related to the algebraic simplification problem. The key difference is that when viewed as rewriting problems, algebraic simplification has both left- and right-hand sides chosen from the same domain while the idiom case has left- and right-hand sides chosen from different domains. Thus, substitutions can be iterated in the former case, but not in the latter.

The plan for the remainder of the paper is to give definitions in Section 2, ancillary theorems in Section 3, recognition algorithms in Section 4, selection algorithms in Section 5, and conclusions in Section 6.

2. *Preliminaries*

In this section the terminology of the introduction is restated in more precise terms. It is assumed that the reader is acquainted with standard tree terminology; if not, details can be found in Knuth [7].

2.1 *Definition of the idiom recognition problem*

Hereafter, a *tree* T will be finite, rooted, oriented, and of bounded degree, d. The vertex set is denoted $V_T = \{v_1,\dots,v_n\}$; the notation $|T|$ will denote the cardinality n of V_T. The degree of a vertex v, *degree(v)*, is the number of its descendents.

Let Σ be a countable set of *operand* or *variable* symbols and Φ a countable set of *operator symbols** for which an *arity function* $\alpha:\Phi \rightarrow \{0,\dots,d\}$ is defined. Elements of Φ with zero arity are *constants*. A *labelling* of a tree T is a function

$$\lambda_T : V_T \rightarrow \Sigma \cup \Phi \qquad (2.1)$$

satisfying for all $v \in V_T$

$$\text{(i)} \quad \lambda_T(v) \in \Phi \Rightarrow \alpha(\lambda_T(v)) = \text{degree}(v) \qquad (2.2)$$

$$\text{(ii)} \quad \lambda_T(v) \in \Sigma \Rightarrow \text{degree}(v) = 0. \qquad (2.3)$$

Thus, the operator arity must match the degree of the vertex it labels and variables can only label leaves.

The notion of an idiom matching a portion of an expression requires first that operators match and secondly, that multiple

----

*In examples, the usual arithmetic operators and constants are used but there is no intended significance to any of the expressions used in the examples.

occurrences of the same operand of the idiom must match identical subtrees. These conditions are formalized as

Definition 2.1: Let I and E be trees, $v \in V_E$ and T be a connected subgraph containing v of the subtree rooted at v. *I matches E at v* if and only if

(i) there is an isomorphism ϵ from V_I to T called the *external match* preserving orientations and labellings of the operator nodes, and

(ii) for all $u, w \in V_I$ such that $\lambda_I(u) = \lambda_I(w) \in \Sigma$ there is an isomorphism ι, called the *internal match at u,w,* between the subtrees rooted at $\epsilon(u)$ and $\epsilon(w)$ that preserves orientations and all labellings.

Operator labeled vertices in the range of ϵ are said to be *externally matched;* vertices in the range of ι are said to be *internally matched.* If U is the subtree rooted at u, we say that the variable $\lambda_I(u)$ *matches* U. See Figure 4 for an example.

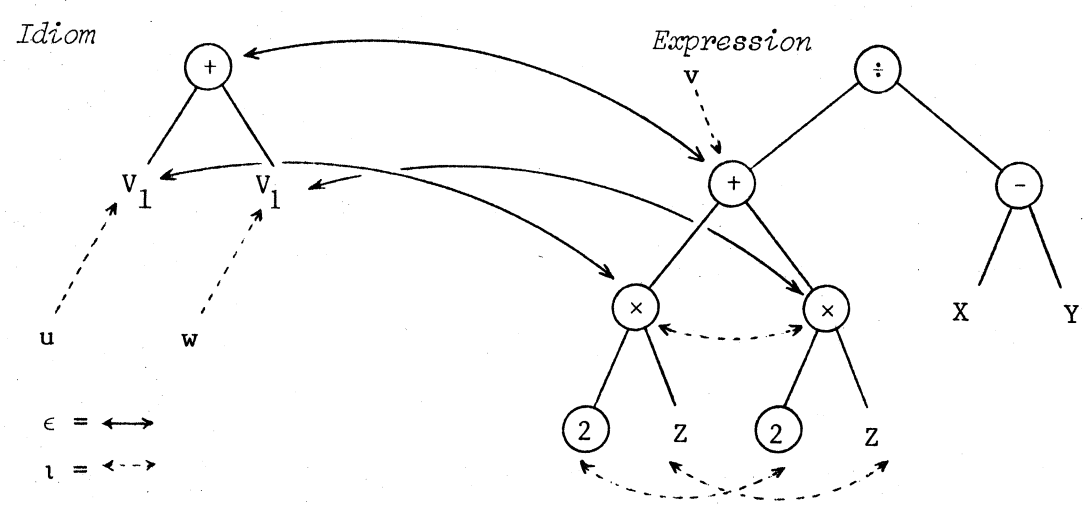


Figure 4: Idiom I matches E at v. ϵ is the external match, ι the internal match. V_1 matches $2 \times Z$.

> *Idiom Recognition Problem:* Given a tree E, the *expression tree*, and a finite set of trees $I = \{I_1,\dots,I_m\}$, the *idiom set*, mark each node v of V_E with an integer i if and only if I_i matches E at v.

A solution to this problem is a function μ called the *I marking* of E and defined for all $v \in V_E$ by

$$\mu(v) = \begin{cases} \{i,\dots,j\} & \text{if } I_i \text{ matches } E \text{ at } v,\dots,I_j \text{ matches } E \text{ at } v, \\ \{0\} & \text{if no idiom matches } E \text{ at } v. \end{cases}$$

2.2 Definition of the selection problem

The following presentation is greatly simplified by introducing an extension to the notion of idiom. The *trivial idiom* I_f for an operator $f \in \Phi$ is an $\alpha(f)+1$ node tree with root labeled with f and leaves labeled by distinct variable symbols. No confusion should result if vertices marked with 0 are interpreted to mean that the trivial idiom $I_{\lambda(v)}$ matches E at v. Thus, trivial idioms are introduced to match only those vertices that do not match conventional idioms.

Let μ be an I marking of E. A *selection from* μ is a function

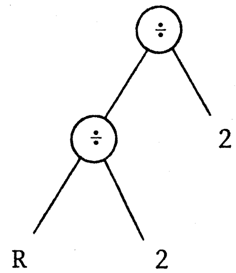
$\sigma: V_E \to \{0,\dots,m\}$

such that for all $v \in V_E$, $\sigma(v) \in \mu(v)$.

A selection σ from μ is said to be *overlapping* if there exist $i \in \sigma(v)$ and $j \in \sigma(v')$ such that ε is the external match of I_i at v, ε' is the external match of I_j at v', $\epsilon(u) = \epsilon'(w)$ for some u in I_i and some w in I_j and $\lambda_{I_i}(u) \in \Phi$, $\lambda_{I_j}(w) \in \Phi$.

Note that some of the operators of the overlap region must match *operators* in both idioms. Thus, two instances of the *shift-*

right idiom of ALGOL-60, X÷2, do not overlap in



since the vertex in common matches the operand in the upper instance and an operator in only the lower instance. A nonoverlapping selection is called a *true selection.*

In order to choose among the various idioms recognized in an expression, it is necessary to know how great the savings are from each idiomatic optimization. Thus, a *payoff* for an I marking of $\mathcal{E}$ is a function

$$\pi : V_{\mathcal{E}} \times \{0,\ldots,m\} \to \mathbb{N} \qquad (2.5)$$

such that for all $v \in V_{\mathcal{E}}$ $\pi(v,0) = 0$. Note that this definition permits different occurrences of the same idiom to have different payoffs. Let σ be a true selection of μ and let π be a payoff, then

$$\mathit{benefit}(\sigma) = \sum_{v \in V_{\mathcal{E}}} \pi(v,\sigma(v)). \qquad (2.6)$$

Benefit is the total improvement of the selected optimizations over the entire expression.

Idiom Selection Problem: Given μ, an I marking of $\mathcal{E}$, find a true selection σ such that

$$\text{benefit}(\sigma) = \max_{\substack{\sigma' \text{ a true} \\ \text{selection of } \mu}} \{\text{benefit}(\sigma')\} \qquad (2.7)$$

Idiom selection is discussed in Section 5.

3. *Comparison bounds for idiom recognition*

In this section the idiom recognition problem is analyzed in order to discover which characteristics of the problem contribute to its complexity.

3.1 *Internal and external matches*

It is useful to introduce additional vocabulary for speaking about matches. Let *ex(v)* (resp. *in(v)*) denote the number of times over all idiom matches in the I marking of E that vertex v is externally (resp. internally) matched.

Theorem 3.1: Let $I = \{I_1,\ldots,I_m\}$ be an idiom set. There are constants c and c' such that for all expression trees E ($|E| = n$),

(i) $ex(v) \le c \le |I_1| + \ldots + |I_m|$,

(ii) $in(v) \le c' \log_2 n$

for all $v \in V_E$.

Proof: (i) Immediate since $\epsilon_1^{-1}(v) = \epsilon_2^{-1}(v)$ implies $\epsilon_1 = \epsilon_2$ and thus each v can, at most, be the image of every vertex of every idiom once. (ii) Assuming there is just one idiom I_1, we will establish a bound of $c_1 \log_2 n$ from which the general bound

$$c' \log_2 n = \sum_{i=1}^{m} c_i \log_2 n \qquad (3.1)$$

will follow, where each c_i depends on the characteristics of I_i. Select a vertex v in E and let $H = \{<u_1,w_1>,\ldots,<u_r,w_r>\}$ be such that v internally matches u_i in an instance of I_1 rooted at w_i, $1 \le i \le r$. If $k \ge 2$ is the maximum number of occurrences of any symbolic operand of

I_1, then there are at most k-1 pairs in H with the same second term, i.e. there are at most $s \leq r \div (k-1)$ distinct vertices at which I_1 matches So, let $\{<u_1,w_1>,\ldots,<u_s,w_s>\} \subseteq H$ be pairs with all w_i distinct. Observe that each w_i is above v in E and so $\{<u_1,w_1>,\ldots,<u_s,w_s>\}$ may be ordered by distance from v (w_1 closest). Denote by $T_{v,i}$ and $T_{u,i}$ the domain and range, respectively, of ι_i the internal matching function establishing the internal match (i.e. $\iota_i(v) = u_i$). Clearly, $T_{v,i} \subset T_{v,i+1}$ and $T_{u,i}$ and $T_{u,i+1}$ are distinct. If h is the height of I_1, then

$$2|T_{v,i}| \leq |T_{v,i+h}|$$

because w_i is a common ancestor of both v and u_i and thus by choice of h, $T_{v,i+h}$ contains w_i and therefore $T_{v,i}$ and $T_{u,i}$. Since

$$|T_{v,s}| < n,$$

we have that

$$2^{\lfloor \frac{s}{h} \rfloor} |T_{v,1}| < n.$$

Solving for s and multiplying by k-1 to bound r, we have

$$r \leq c_1 \log_2 n.$$

Having established our claim, we have from (3.1) the desired bound.

□

The bound of $in(v) = \log_2 n-1$ is achievable as the reader can easily verify with the idiom v+v and an expression tree that is a complete binary tree with + operator labels on the internal nodes and all leaf nodes labeled with the same symbolic operand, say A. For this case each leaf has $\log_2 n$ internal matches. (See Figure 5.)

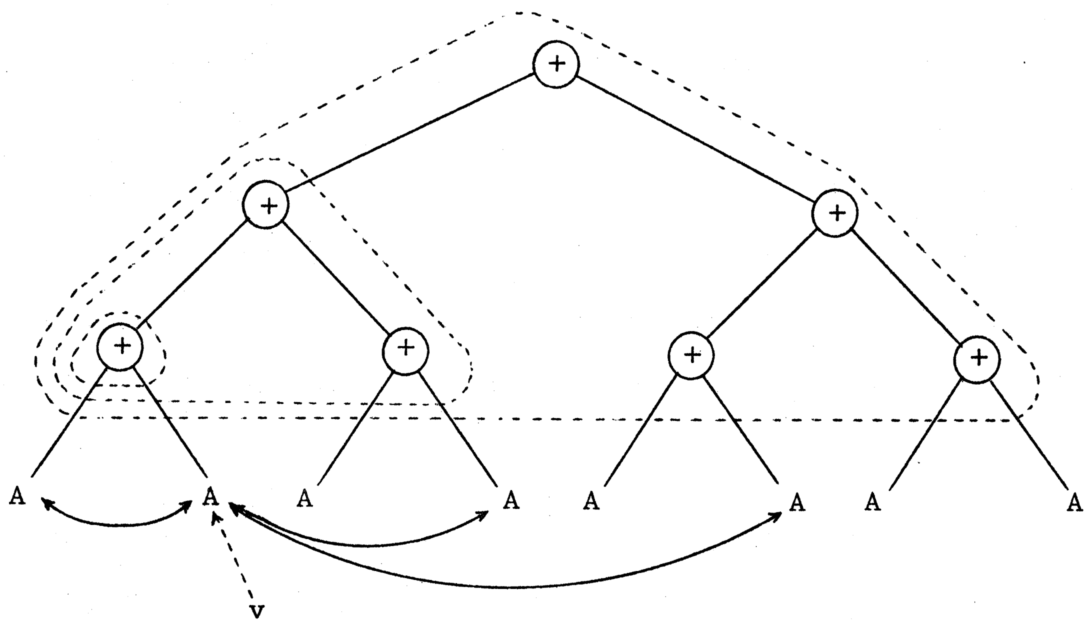


Figure 5: Internal matches of vertex v for idiom V+V. The matched instances of the idiom requiring internal matches of v are indicated by dotted lines.

3.2 Ambiguity

A set of idioms I is said to be *unambiguous* if for every expression E, the marking μ of I in E satisfies $|\mu(v)| = 1$ for $v \in V_E$. Otherwise the marking is *ambiguous*.

Determining whether or not an idiom set is ambiguous can be done in $O(n^2)$ in the size n of the idiom list. To see this, observe that the ambiguity requirement may be restated as

> there exists a tree T and two idioms I_1 and I_2 both matching T at its root.

This is equivalent to the statement from resolution theorem proving [8],

> there exists a unifier T for I_1 and I_2,

provided that the trees be thought of as well-formed formulae in the obvious way. Recall that a unifier is a well-formed formula resulting from the simultaneous substitution of formulae for variables in I_1 and I_2. A "most general unifier" is a smallest formula unifying I_1 and I_2. In the context of the ambiguity of idiom sets, the most general unifier is a smallest witness to the ambiguity of I_1 and I_2. The existence of efficient procedures to find a most general unifier enables one to prove:

Theorem 2.2: Let $\mathcal{I} = \{I_1,\ldots,I_m\}$ be a set of idioms and let $|I_1| +\ldots+ |I_m| = n$. There is an $O(n^2)$ algorithm that determines whether or not $\mathcal{I}$ is ambiguous.

Proof: From the previous remarks, the algorithm must only compute

$$\bigvee_{1\le i<j\le m} \text{unifiable}(I_i,I_j) \tag{3.2}$$

where unifiable (I_i,I_j) is the unification predicate. Then $\mathcal{I}$ is ambiguous if and only if (3.2) is true. The time bound follows from the linearity of the Paterson-Wegman linear time unification algorithm [9].

Our interest in ambiguity derives first from the possibility that the presence of an ambiguous pair may signal the presence of a "hybrid" idiom (the unifier) that subsumes both special cases, and from the unexpected link it provides with resolution theorem proving.

4. *Linear-time idiom recognition*

Theorem 3.1, giving a constant bound on the number of external matches and a logarithmic bound on the internal matches, suggests a worst case bound of O(n log n) for recognition. We establish this and give an algorithm with a linear expected case bound using an approach suggested by R. E. Ladner and M. R. Brown.

The general approach of the algorithm is to use two passes over the tree. The first (bottom up pass) is to mark all nodes with integers. The second pass then performs the actual recognition from top down. Viewed more abstractly, the numbering phase prepares for the internal matching operations by identifying with the same unique number all roots of identical subtrees. This operation is performed uniformly without regard to whether the subtree is actually involved in an internal match. The second pass performs the matching, and uses the numbering to perform the internal matches. In order to simplify the presentation, we assume all trees are binary; generalization to arbitrary degree is a straight forward matter.

4.1 *The numbering phase*

The numbering phase assumes (1) that the expression tree E has been "threaded" with h = height(E) "threads" in such a way that all nodes at the same height are on the same linked list, and (2) the nodes at height 0, the leaves, have been numbered with integers 1,...,t, so that like operands are assigned the same number. (See Section 4.3 for a discussion of this operation.)

At step k, the algorithm will number the nodes at height k. All nodes at height k with the same numbers assigned to their left and right descendants will receive the same number. In order to do this numbering efficiently, we employ two bucket sorting operations. The first sorts all nodes at height k on the number of their left descendant. The second sorts all elements that landed in the same bucket on the number of their right descendant. Elements that land in the same bucket as a result of the second sort have left and right descendants numbered the same and they are all assigned the same unique number.

With proper attention to chaining the algorithm works in linear time. We use the following forward-linked lists in the algorithm, all of which are terminated by NIL:

4.1 (a) HEIGHT[k], all nodes of E at height k (using LINK field of TREE);

(b) BCHAIN1, all "in use" buckets for the first bucket sort (using LINK field of BUCKET);

(c) BCHAIN2, all "in use" buckets for the second bucket sort (using LINK field of BUCKET);

(d) BUCKET1[i].HEAD, all nodes in BUCKET1[i] (using LINK field of TREE);

(e) BUCKET2[i].HEAD, all nodes in BUCKET2[i] (using LINK field of TREE).

The actual text of the algorithm is given in Figure 6. The outer loop processes each height of the tree. For the nodes at each height, lines 3-15 perform the first bucket sort. Lines 16-46 perform the second bucket sort (20-33) and assignment of unique numbers (34-45) for each nonempty bucket produced by the first bucket sort. The linear

```
Algorithm: NUMBER
    input:  TREE[1:n] a vector containing n node entries with
               fields
               LEFT   contains TREE indices referring to left
                      descendant, NIL for leaves
               RIGHT  contains TREE indices referring to right
                      descendant, NIL for leaves
               LINK   contains TREE indices for chain (4.1a)
                      initially, later it is used for (4.1d),
                      (4.1e)
               NUMBER contains an integer i, 1≤1≤UNIQUE≤n.
                      Initially only leaves assigned such that
                      TREE[r].NUMBER = TREE[s].NUMBER
                      ⇔ TREE[r].OP = TREE[s].OP.
               OP     contains the label (operator or operand)
                      for the node
               BENEFIT benefit value of subtree rooted at
                      this node (used in Section 5)
               SELECTCHAIN header of list of operand vertices
                      of idiom selected here (used in Section 5).
            HEIGHT[1:d] a vector containing TREE indices used
               as header for (4.1a) chains, d = depth of the
               tree
            UNIQUE, integer, initially the largest value used
               to initialize number field of leaves.
   output:  the TREE vector with all entries in NUMBER field
            filled such that for nonleaves TREE[i].NUMBER
            = TREE[j].NUMBER ⇔ TREE[TREE[i].LEFT].NUMBER
            = TREE[TREE[j]. LEFT ].NUMBER
            ∧ TREE[TREE[i].RIGHT].NUMBER
            = TREE[TREE[j].RIGHT].NUMBER
   used:    BUCKET1[1:n] a vector containing elements with
               fields
               LINK   contains BUCKET1 indices to implement
                      (4.1b), initially NIL
               HEAD   contains TREE indices as header for
                      (4.1d), initially NIL
            BUCKET2[1:n] a vector like BUCKET1, with LINK
               implementing (4.1c) and HEAD as header to (4.1e).
            BCHAIN1, BCHAIN2, headers for implementing (4.1b),
               (4.1c), initially NIL
```

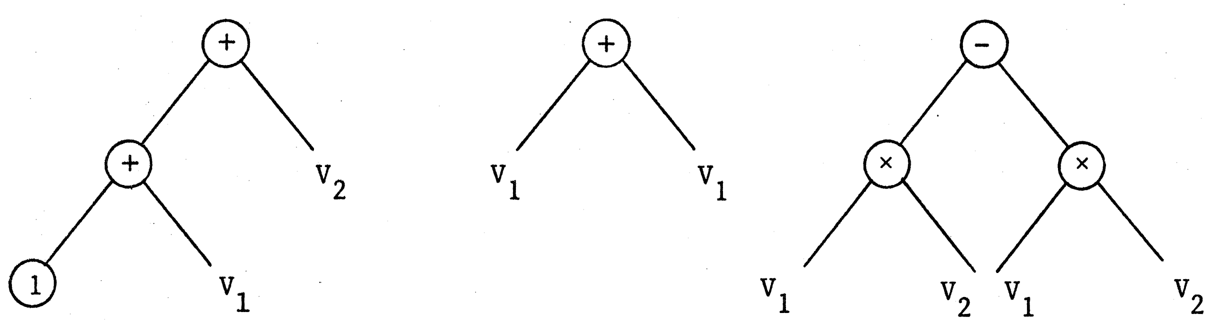
```
for i ← 1 step 1 until d do
    begin
    while HEIGHT[i] ≠ NIL do
        begin
            NODE ← HEIGHT[i];
            HEIGHT[i] ← TREE[NODE].LINK;
            LNUM ← TREE[TREE[NODE].LEFT].NUMBER;
            if BUCKET1[LNUM].HEAD = NIL
                then begin
                    BUCKET1[LNUM].LINK ← BCHAIN1;
                    BCHAIN1 ← LNUM
                    end;
            TREE[NODE].LINK ← BUCKET1[LNUM].HEAD
            BUCKET1[LNUM].HEAD ← NODE
        end;
    while BCHAIN1 ≠ NIL do
        begin
            B1 ← BCHAIN1;
            BCHAIN1 ← BUCKET1[B1].LINK;
            while BUCKET1[B1].HEAD ≠ NIL do
                begin
                    NODE ← BUCKET1[B1].HEAD;
                    BUCKET1[B1].HEAD ← TREE[NODE].LINK;
                    RNUM ← TREE[TREE[NODE].RIGHT].NUMBER;
                    if BUCKET2[RNUM].HEAD = NIL
                        then begin
                            BUCKET2[RNUM].LINK ← BCHAIN2;
                            BCHAIN2 ← RNUM
                            end;
                    TREE[NODE].LINK ← BUCKET2[RNUM].HEAD;
                    BUCKET2[RNUM].HEAD ← NODE
                end;
            while BCHAIN2 ≠ NIL do
                begin
                B2 ← BCHAIN2;
                BCHAIN2 ← BUCKET2[B2].LINK;
                UNIQUE ← UNIQUE + 1;
                while BUCKET2[B2].HEAD ≠ NIL do
                    begin
                    NODE ← BUCKET2[B2].HEAD;
                    TREE[NODE].NUMBER ← UNIQUE;
                    BUCKET2[B2].HEAD ← TREE[NODE].LINK
                    end
        end
    end
end;
```

Figure 6: Algorithm for numbering an expression tree.

complexity follows trivially since each node must be removed from the height chain, placed in a bucket for the first sort, removed from the bucket, placed in a bucket for the second sort, and assigned a unique number. These are all constant time operations and except for a pinch of overhead (e.g. loop control) accounts for all of the activity of the algorithm.

4.2 *The matching phase*

The idioms are stored in a table with each entry linearized in parenthesis-free prefix notation. The entries are grouped and ordered lexicographically so that they may be easily referenced by indexing. Operands will be represented by the notation V_i to indicate that this is the i^{th} distinct operand of the idiom and $\bar{V}_i$ denotes that this operand is a repeated instance of the i^{th} distinct operand. Accordingly,



would appear as

$+ + 1\ V_1\ V_2$

$+ V_1\ \bar{V}_1$

$- \times V_1\ V_2 \times \bar{V}_1\ \bar{V}_2$

when represented in tabular form. The tabular format will allow for simpler indexing to be used to traverse the idioms in a depth-first discipline. Thus, a position in a idiom can be denoted by a pair (i,j) of indices and T(i,j) refers to the j^{th} position of the i^{th} idiom.

All of the external matches will be performed simultaneously in a single depth-first traversal of the tree. In order to keep track of the matches in progress, *match descriptors* will be used and will have the general form

$$(v,i,j,v_1,\ldots,v_p)$$

where v is the vertex at which the idiom will be rooted (if it matches), i,j are indices into the idiom table, and $v_1,\ldots,v_p$ are vertices of E at which the first p (distinct) operands of the idiom I_i are rooted.

On the recursive depth-first traversal, the algorithm has two operations to perform before processing descendant nodes, and one operation to perform afterwards. Before visiting and descendant nodes, it must *initiate* descriptors for all idioms that could be rooted at this vertex. Secondly, it must *update* all descriptors for matches in progress and create a new list of the matches that are continuing. After processing all descendant nodes, it must be determined which idioms initiated at this node matched and *mark* them.

The text of the algorithm is given in Figure 7. The initiate descriptors operation (line 4) compares TREE[v].OP with the first column in the idiom table T and for each match, say on row i, constructs a descriptor (v,i,1) indicating that the root of idiom i matches at v. The descriptor is temporarily saved in list A and will later become part of the ACTIVE list (line 14).

The update operation (lines 6-13) continues those matches in progress as indicated by the existence of descriptors on the ACTIVE list. The operations performed are indicated in Table 1.

```
Algorithm:  MATCH
     input:  TREE as described in NUMBER algorithm
     output:  marked tree
     initial call:  match(root(TREE))
     legend:  -ACTIVE is a global list of match descriptors,
              -A is a global temporary,
              -the list operation concatenate is denoted by
               a comma as in A ← A, (v,i,1) ⇔ concate-
               nate(A,((v,i,1))),
              -for each <sublist> in LIST do is an iteration
               statement that removes successive instances of
               <sublist> from list and instaniates the
               generic parameters.
```

```
     procedure match(v):
1.      begin
2.      local list SUSPEND;
3.      A ← SUSPEND ← NIL;
4.      for i ← 1 step 1 until m do
5.         if T[i,1] = TREE[v].OP then A ← A, (v,i,1);
6.      for each (u,i,j,v1,...,vp) in ACTIVE do
7.         if T[i,j+1]∈Φ
8.            then if T[i,j+1] = TREE[v].OP
9.               then A ← A, (u,i,j+1,v1,...,vp)
10.           else if T[i,j+1] = 'v̄k'
11.              then if TREE[v].NUMBER = TREE[vk].NUMBER
12.                 then SUSPEND ← SUSPEND, (v,i,j+1,v1,...,vp)
13.              else SUSPEND ← SUSPEND, (u,i,j+1,v1,...,vp,v)
14.     ACTIVE ← A;
15.     if TREE[v].LEFT ≠ NIL then match (TREE[v].LEFT);
16.     if TREE[v].RIGHT ≠ NIL then match (TREE[v].RIGHT);
17.     A ← NIL;
18.     for each (u,i,j,v1,...,vp) in ACTIVE do
19.        if u = v then mark(v,i)
20.           else A ← A, (u,i,j,v1,...,vp);
21.        ACTIVE ← A, SUSPEND
22.     end;
```

Figure 7: match algorithm

If v is an operator or constant that matches the indicated idiom (line 8) then the idiom index, j, in the descriptor is updated and it remains in the ACTIVE list. If the idiom symbol is a variable operand, then it is either the first occurrence of that operand (line 13) in which case the j entry of the descriptor is updated and the value of v is included as well, or else it is a repeated occurrence (line 12) in which case the value in TREE[v].NUMBER is compared with that of the corresponding operand. If it matches or if it was a first occurrence, the descriptor is added to a local list called SUSPEND. In all other cases the descriptors did not match and they are discarded.

The role of SUSPEND is to save the descriptors of those idiom matches in progress that have reached a leaf of the idiom. Their removal then allows lower regions of the tree to be processed without the ACTIVE list being cluttered with unnecessary overhead.

During the marking phase after the descendants have been visited, (lines 15-16) the ACTIVE list is scanned for elements initiated at this vertex, (line 19). Any that are found represent instances of matching idioms. (We are vague about the actual marking here in anticipation of incorporation of the selection algorithm; see Section 5.) In addition all elements that were SUSPENDed are reactivated (line 21).

value of TREE[v].OP	value of T(i,j+1): operator	constant	operand V_k	operand $\bar{V}_k$
operator	1	MM	2	3
constant	MM	1	2	3
variable operand	MM	MM	2	3

1 = match ⇒ update j; descriptor remains ACTIVE (8)
2 = update descriptor with v; SUSPEND (13)
3 = match v_k of descriptor with v; SUSPEND (12)
MM = mismatch

Table 1: Summary of updating operations; numbers in parentheses refer to lines of the match algorithm.

It is clear that the algorithm halts since without lines 2-14, 17-21 it is a standard depth-first-traversal algorithm. That the external matches are performed properly is clear from line 8. The internal matches rely on the fact that matching is an equivalence relation, that the descriptor contains the number corresponding to each of its distinct operands and that two vertices with the same numbers are roots of identical subtrees.

4.3 Recognition complexity

An antecedent condition to the numbering algorithm is that the leaves of the expansion tree E be numbered by an integer from 1 to UNIQUE with like-labeled leaves receiving the same number. This requirement permits us to use the bucket sorting procedures. In general use, this is a realistic assumption since the leaves will actually refer to symbol table and/or constant table entries. Standard hashing techniques augmented with some straightforward bookkeeping in the symbol table will enable the required leaf numbering to be realized in linear expected time [7]. This, together with the guaranteed linearity of the numbering and matching operations, allows us to conclude for a random access machine model,

Theorem 4.1: The recognition problem for an expression tree E has expected running time of $O(n)$, where $n = |E|$.

□

Even though these assumptions are realistic in practice, an adversary could present us with an expression tree that causes the hashing algorithm to achieve its worst case performance. Thus, we may prefer a balanced search tree scheme, say AVL or B-tree [10], giving a guaranteed $O(n \log n)$ bound to number the leaves in worst case. By considering the worst case behavior, we can also dispense with the assumption of random access machine model. In particular, we use the random access feature for the numbering operation, e.g., bucket sort, but with $O(n \log n)$ time available, the numbering is not needed. A direct test of both external and internal matches by an augmented version of the match routine can be used. (Create (in line 11) a second kind of descriptor for internal matches when suspension takes place and make them ACTIVE; details are left to the reader.) This strategy must make all of the comparisons required by Theorem 3.1 (but no more) and thus operates in $O(n \log n)$. We conclude

Theorem 4.2: The recognition problem for an expression tree E has worst case running time of $O(n \log n)$ where $n = |E|$.

□

Finally, we observe that this direct method achieves the worst case only for the pathological cases (as illustrated in Figure 5); as long as the amount of overlap tends to be small, the direct method operates efficiently, independent of the distributional characteristics of the leaves.

5. *Idiom selection*

As the results of Section 3 established, the amount of "overlap" of idioms can be substantial. But idioms are logically primitive and thus the target code cannot be easily factored and recombined to merge parts of idioms. Thus a true selection of the recognized idioms, i.e., a nonoverlapping subset, must be found.

The trouble with finding true selections is that there are potentially a large number of arrangements of a given set of idioms into nonoverlapping subsets. For example, consider the schematic expression tree of Figure 8 for which eight idioms have been recognized.

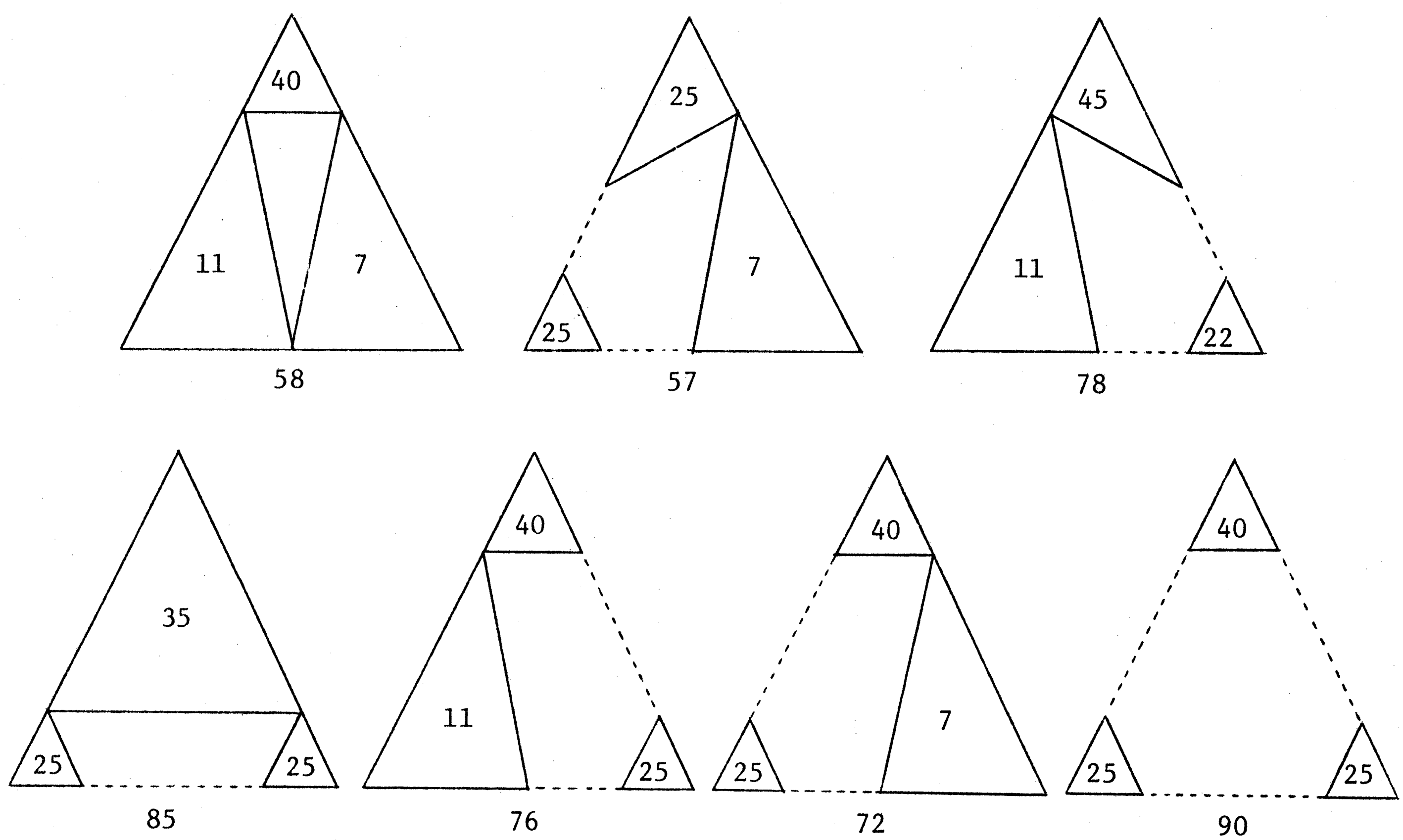


Figure 8: Schematic of nonoverlapping subsets of 8 recognized idioms.

Of course only maximal nonoverlapping subsets need to be considered since if a subset is not maximal the global benefit can be improved by adding idioms to make it maximal.

In this section, we concentrate on computing the maximal benefit rather than finding the selection that yields the maximum benefit. It will be seen from the algorithms given later that it is a simple matter to accumulate the information about the optimal selection as the benefit is being computed. Then when the maximal value is known, the selection(s) that realize it can be fetched directly.

5.1 *A different formulation of benefit*

In this section the function benefit(σ) of a true selection of an I marking of E will be formulated as an easily computed recursive function $\beta(\mathrm{root}_E)$. Recall from Section 2 that when $\mu(v) = 0$, it is assumed that this refers to the trivial idiom for the label $\lambda_E(v)$.

Suppose that I_i matches E at v and let ϵ be the external match. Define

$$L_i(v) = \{v \in E \mid \mathrm{degree}(\epsilon^{-1}(v)) = 0\}.$$

Thus, $L_i(v)$ contains all vertices in E that correspond to leaves in I_i. In addition, define recursively,

$$\beta(v) = \max_{i \in \mu(v) \cup \{0\}} \pi(v,i) + \sum_{u \in L_i(v)} \beta(u)$$

where for the trivial idiom $L_0(v)$ = the immediate descendants of v.

Theorem 5.1: Let μ be an I marking of E and σ a true selection that maximizes the benefit of μ in E. Then

$$\text{benefit}(\sigma) = \beta(\text{root}_E) \tag{5.1}$$

Proof: Induction on the height of the tree E.

(Basis) When height is 1, only one nondegree 0 vertex is involved, i.e., $V_E = \{\text{root}_E\}$. Thus

$$\beta(\text{root}_E) = \max_{i\in\mu(\text{root}_E)\cup\{0\}} \pi(\text{root}_E,i) + \sum_{u\in L_i(\text{root}_E)} \beta(u)$$

$$= \max_{i\in\mu(\text{root}_E)\cup\{0\}} \pi(\text{root}_E,i)$$

$$= \max_{i\in\mu(\text{root}_E)} \pi(\text{root}_E,i).$$

Moreover, for each

$$\text{benefit}(\sigma') = \sum_{v\in V_E} \pi(v,\sigma'(v))$$

$$= \pi(\text{root}_E, \sigma'(\text{root}_E)).$$

Since any selection is a true selection

$$\text{benefit}(\sigma) = \max_{i\in\mu(\text{root}_E)} \pi(\text{root}_E,i)$$

and the equality holds, since by definition $\pi(v,0) = 0$.

(Induction) Suppose the theorem holds for all trees of height h or less, and let E be an expression tree of height h+1. Let $d_1,\ldots,d_p$ be the immediate descendants of root_E, let $T_1,\ldots,T_p$ be the subtrees of E rooted at $d_1,\ldots,d_p$, and for any idiom $k\in\mu(\text{root}_E)$, let $L_k(\text{root}_E) = \{u_1,\ldots,u_q\}$ and let $H_1,\ldots,H_q$ be the subtrees of E rooted at $u_1,\ldots,u_q$. Suppose $\sigma(\text{root}_E) = 0$. Then

$$\text{benefit}(\sigma) = \sum_{v\in T_1} \pi(v,\sigma(v)) + \ldots + \sum_{v\in T_p} \pi(v,\sigma(v)). \tag{5.2}$$

Since $\text{height}(T_i) \le h$ $(1 \le i \le p)$, by hypothesis, (5.2) can be written

$$\beta(d_1) + \ldots + \beta(d_p)$$

$$= \pi(\text{root}_E, 0) + \sum_{d \in L_0(\text{root}_E)} \beta(d)$$

$$\le \beta(\text{root}_E) \tag{5.3}$$

Alternatively, suppose $\sigma(\text{root}_E) = k$, then

$$\text{benefit}(\sigma) = \pi(\text{root}_E, k) + \sum_{v \in H_1} \pi(v, \sigma(v)) +$$

$$\ldots + \sum_{v \in H_q} \pi(v, \sigma(v)) \tag{5.4}$$

and again $\text{height}(H_i) \le h$, $(1 \le i \le q)$. The hypothesis permits (5.4) to be written

$$\pi(\text{root}_E, k) + \beta(u_1) + \ldots + \beta(u_q)$$

$$\le \beta(\text{root}_E).$$

Thus (5.3) and (5.4) imply

$$\text{benefit}(\sigma) \le \beta(\text{root}_E). \tag{5.5}$$

Now suppose that

$$\beta(\text{root}_E) = \pi(\text{root}_E, 0) + \sum_{d \in L_0(\text{root}_E)} \beta(d) \tag{5.6}$$

The $\{d_1, \ldots, d_p\} = L_0(\text{root}_E)$ are roots of $T_i (1 \le i \le p)$ and each has height less than h. Thus, by hypothesis, (5.6) is

$$\pi(\text{root}_E, 0) + \text{benefit}(\sigma_1) + \ldots + \text{benefit}(\sigma_p)$$

where the $\sigma_i (1 \le i \le p)$ are benefit maximal selections for the T_i. Define $\sigma'(\text{root}_E) = 0$ and $\sigma'(v) = \sigma_i(v)$ for all $v \in T_i$ and all $i (1 \le i \le p)$. Note that σ' is a true selection, and thus

$$\beta(\text{root}_E) \le \text{benefit}(\sigma).$$

Finally, suppose

$$\beta(\text{root}_E) = \pi(\text{root}_E, k) + \sum_{u \in L_k(\text{root}_E)} \beta(u) \tag{5.7}$$

$k \neq 0$. Then the $\{u_1, \ldots, u_q\} = L_k(\text{root}_E)$ are roots of trees $H_1, \ldots, H_q$ all of height less than or equal to h and so by hypothesis (5.8) may be written as

$$\pi(\text{root}_E, k) + \text{benefit}(\sigma_1) + \ldots + \text{benefit}(\sigma_q)$$

where $\sigma_1, \ldots, \sigma_q$ are the benefit-maximal true selections of $H_1, \ldots, H_q$, respectively. Define $\sigma'(\text{root}_E) = k$ and $\sigma'(v) = \sigma_i(v)$ for all $v \in H_i$ and all $i (1 \le i \le q)$. Note that this is a true selection from μ. Thus

$$\beta(\text{root}_E) \le \text{benefit}(\sigma).$$

From this and (5.5) and (5.7) it follows that (5.1) holds in general.

□

5.2 *The linear selection algorithm*

The function β of the previous section easily computes the maximal benefit by a recursive routine that is almost a translation of the definition. But because we anticipate combining this algorithm with the recognition algorithm, a version is presented that is synchronized to a depth-first traversal of the tree.

The algorithm takes a root of a subtree as a value. After computing the benefit for all subtrees and accumulating these values in b, a local variable, each idiom benefit is computed. This computation is facilitated by assuming that the benefit of each descendant vertex has been *saved* in the BENEFIT field associated with each tree vertex.

Again, we assume the trees are only binary and leave the easy generalization to the reader. The text of the algorithm is given in Figure 9. Since the summation is bounded by the number of leaves in idiom u_i, a constant, and k is bounded by m the number of idioms, the entire algorithm is time bounded by O(n).

```
Algorithm:  BENEFIT
     input:  marked tree represented as in  NUMBER
    output:  value of maximum benefit

benefit(v):
   begin
   local b;
   if TREE[v].LEFT ≠ NIL
      then begin benefit(TREE[v].LEFT);
                 b ← TREE[v].BENEFIT end
      else b ← 0;
   if TREE[v].RIGHT ≠ NIL
      then begin benefit(TREE[v].RIGHT);
                 b ← b + TREE[v].BENEFIT end;
   let μ(v) = u_1,...,u_k;
   for i ← 1 step 1 until k do
```

$$b \leftarrow \max(b, \pi(v,u_i) + \sum_{w \in L_{u_i}(v)} \text{TREE[w].BENEFIT});$$

```
   TREE[v].BENEFIT ← b
end
```

Figure 9: Benefit Algorithm

5.3 *Combined algorithm and refinements*

We are now able to present the complete solution by combining the match algorithm (Section 4.2) and the benefit algorithm (Section 5.2) and then augmenting the result to *save* the true selection. The true selection is saved using the SELECTCHAIN field of the TREE nodes. Each node holds a descriptor of the form

$$(i, v_1, \ldots, v_p)$$

where i is the idiom matching at that node (0 = trivial idiom) and $v_1,\ldots,v_p$ are the roots of the distinct operands (direct descendants for trivial idiom). (Only one instance of each operand must be kept since the others are redundant.)

The BENEFIT field of the TREE nodes is used to hold the benefit values as before. To compute

$$\sum_{w \in L_{u_i}} \text{TREE[w].BENEFIT}$$

which was used in the benefit algorithm, we add an additional field (bene) in the match descriptor to hold the accumulated benefit of the operands of the idiom. This field is updated when the SUSPENDed descriptors are returned to the active list (lines 29-30). The actual text of the algorithm is shown in Figure 10.

```
 1.  procedure match(v):
 2.    begin
 3.    local b;
 4.    local list SUSPEND, SELTEMP;
 5.    A ← SUSPEND ← NIL;
 6.    for i ← 1 step 1 until m do if T[i,1] = TREE[v].OP
          then A ← A, (v,i,1,0);
 7.    for each (u,i,j,bene,v1,...,vp) in ACTIVE do
 8.       if T[i,j+1]∈Φ
 9.          then if T[i,j+1] = TREE[v].OP
                then A ← A, (u,i,j+1,bene,v1,...,vp)
10.          else if T[i,j+1] = 'V̄k'
11.             then if TREE[v].NUMBER = TREE[vk].NUMBER
12.                then SUSPEND ← SUSPEND, (u,i,j+1,bene, v1,...,vp)
13.             else SUSPEND ← SUSPEND, (u,i,j+1,bene,v1,...,vp,v);
14.    ACTIVE ← A; b ← 0; SELTEMP ← (0);
15.    if TREE[v].LEFT ≠ NIL then begin match(TREE[v].LEFT);
16.                   b ← TREE[TREE[v].LEFT].BENEFIT;
17.                   SELTEMP ← SELTEMP, TREE[v].LEFT end;
18.    if TREE[v].RIGHT ≠ NIL then begin match(TREE[v].RIGHT);
19.                   b ← TREE[TREE[v].RIGHT].BENEFIT + b;
20.                   SELTEMP ← SELTEMP, TREE[v].RIGHT end;
21.    A ← NIL;
22.    for each (u,i,j,bene,v1,...,vp) in ACTIVE do
23.       if u = v then begin if b < π(v,i) + bene
24.             then SELTEMP ← (i,v1,...,vp);
```

```
                b ← max(b,π(v,i) + bene) end
                else A ← A, (u,i,j,bene,v_1,...,v_p);
        TREE[v].BENEFIT ← b; TREE[v].SELECTCHAIN ← SELTEMP;
        ACTIVE ← A;
        for each (u,i,j,bene,v_1,...,v_p) in SUSPEND do
           ACTIVE ← ACTIVE, (u,i,j,bene + b,v_1,...,v_p)
    end
```

Figure 10: Combine recognition and selection.

There are several possible improvements to the algorithm. First, there is no reason to apply the match algorithm to the repeated subtrees. Thus, if the repeated instances are coalesced during the numbering phase of the numbering algorithm to form a directed acyclic graph, then the match algorithm can be easily changed to operate on the dag. An additional advantage to using dag's is that many of the operands will be repeated by internal assignments, rather than explicit text, which are naturally represented this way.

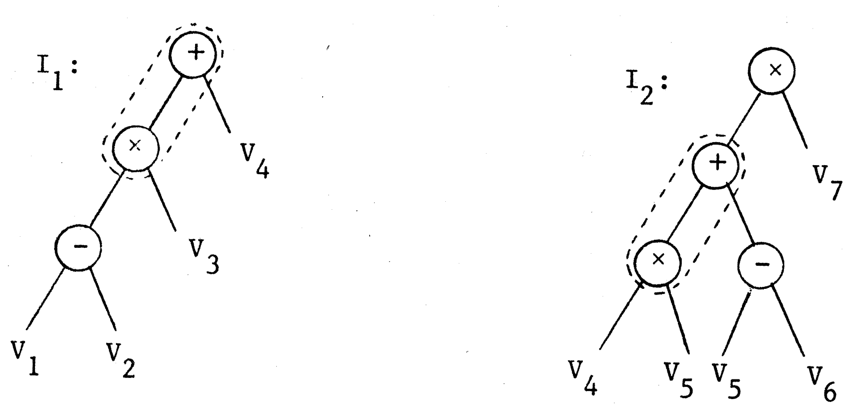
Another modification to the algorithm would be to make the comparison operation more exotic. For example, if the number of recognized idioms could be materially increased by recognizing ~A≠B when A=B is written, then the comparison operation could be made to test this by building "escape" indicators in the idiom representation. Then, specialized routines could be written to implement those more sophisticated tests. Commutativity and other identities could be treated in this manner, though empirical studies of language usage might be indicated before substantial effort is diverted to these refinements.

6. *Summary and directions for further research*

Idioms have been motivated and the identification, recognition and selection problems have been defined. For the recognition problem we have a worst-case algorithm operating in O(n log n) time and an average-case algorithm that is O(n). Ambiguity has been shown to be solvable in $O(n^2)$, and certain characteristics of the recognition and selection problems have been exposed in the ancillary lemmas. Two main lines of further research are obvious.

First, there is the language dependent question of idiom identification. We know a lot of APL idioms, but others will certainly be found. Little has been done for other languages and identification of idioms is a worthy task for the programmers expert in other languages. If other languages tend to have many idiomatic expressions it would suggest the utility of macro facilities for higher level languages.

The second line of research is to extend certain of the investigations begun here. One problem related to the ambiguity test is to find the potential overlaps in a set of idioms in an efficient way. The problem is that certain apparent overlaps cannot obtain due to conditions placed on the operands. For example,



cannot overlap since a V_3 internal match is incompatible with a V_5 internal match. A naive scheme analogous to the ambiguity test appears to have potential for improvement. (The ambiguity test itself can probably be improved!) There is also the question of actually computing the coefficients, c and c', of Theorem 3.1.

Finally, our complexity bounds for the algorithms are derived on the assumption that the input expression E grows without bound, and thus the size of the idiom set can be ignored. In a sense we are "matching the idioms to the expression." But what if the idiom set is large compared to a typical expression? Then we might wish to "match the expression to the idioms." This suggests that better algorithms are possible and that preprocessing of the idioms might be useful.

Acknowledgement

I wish to thank Alan Perlis for many useful and enjoyable discussions concerning idioms and Richard Ladner and Mark Brown for suggesting the numbering scheme.

References

[1] Websters 3rd International Dictionary.

[2] Private communication.

[3] A. J. Perlis and J. S. Rugaber.
The APL Idiom List.
Yale Computer Science Technical Report #87, (1977).

[4] D. Gries
Compiler Construction for Digital Computers.
Wiley, (1971).

[5] T. C. Miller.
Tentative Compilation: A Design for an APL Compiler.
Department of Applied Physics and Information Sciences.
Technical Report 78-CS-013.
University of California at San Diego (1978).

[6] T. P. Holls.
APL Programming Guide: Vector Optimizations.
IBM DPD Scientific Marketing Report, While Plains, NY (1978).

[7] D. E. Knuth.
The Art of Computer Programming, Fundamental Algorithms.
Addison-Wesley, rev. ed. (1973).

[8] J. A. Robinson.
Computational logic: the unification computation.

Machine Intelligence 6, pp. 63-72.

[9] M. S. Paterson and M. N. Wegman.

A linear time unification algorithm.

Proceedings of STOC 8 (1976).

[10] D. E. Knuth.

The Art of Computer Programming. Sorting and Searching.

Addison-Wesley, (1973).