OPTIMAL CODE GENERATION FOR CONTROL STRUCTURES

By

Munagala V. S. Ramanath

Computer Sciences Technical Report #461

December 1981

OPTIMAL CODE GENERATION FOR CONTROL STRUCTURES

by

MUNAGALA V. S. RAMANATH

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1981

TABLE OF CONTENTS

OPTIMAL CODE GENERATION FOR CONTROL STRUCTURES

MUNAGALA V. S. RAMANATH

Under the supervision of

Assistant Professor Marvin H. Solomon

ABSTRACT

We investigate the problem of generating code for programs so that the number of unconditional branch instructions, or ´jumps´, in the object code is minimized. We show that the problem is NP-complete in general and that polynomial algorithms exist provided the source programs are limited to using a restricted set of control structures. In particular, we show that if we restrict the set of control structures to IF-THEN-ELSE and LOOP-ENDLOOP with multilevel exits, there is an efficient algorithm for finding the minimal- jump translation. The time complexity of the algorithm is quadratic in the size of the program provided the maximum number of nested loops that may be exited by an exit statement is bounded by a constant. We show that more directed graphs can be expressed if this bound is (i+1) than if it were i. We also compare the class of flowgraphs that can be obtained with these control structures to the well-known class of reducible flowgraphs.

ACKNOWLEDGEMENTS

It is with unqualified pleasure that I express my gratitude to Dr. Marvin H. Solomon, my advisor; that his perceptive comments played a formative role in this thesis cannot be gainsaid. His belief that what can be said in two words ought not to be said in ten resulted in the compaction of my expansive prose. His preference for assuming impatient curiosity, rather than conscientious pertinacity, on the part of the typical reader led to the inclusion of more examples and exegetic comment, to the exclusion of meretricious and obscure notation which I unfailingly generated and, whenever possible, to the substitution of perspicuous prose for the less transparent, though perhaps more precise, symbolic derivations. It will brook no denial that, but for his suggestions, several of my proofs would be substantially more complex than they are. His assistance in various advisory, bureaucratic and academic matters was generous and ungrudging. Most of all I am indebted to him for suggesting the central problem of this thesis -- a problem that, it can be said in retrospect, had all the desirable attributes of formalizability, tractability and apparent utility.

I am grateful to the members of my committee for their helpful suggestions, especially to Dr. Raphael

Finkel for his careful reading of the manuscript and to Dr. Charles Fischer for his assistance in locating journal articles.

CHAPTER 1

INTRODUCTION AND OVERVIEW

1. INTRODUCTION

It is widely recognized that machine code produced by high-level language translators is not as compact and "efficient" as could be produced by a competent assembly-language programmer. The impact of various inefficiencies can be moderated by sundry "optimization" techniques. One kind of inefficiency that is not addressed by known techniques is the presence of an unnecessarily large number of unconditional branches or "jumps" in the object code. Figure 1 shows a source program and two translations; all six jumps in the standard translation can be eliminated by paying careful attention to the linear order in which the blocks of machine code are placed.

The linear order in which code is generated and ultimately loaded often tends to be precisely the order in which the corresponding source code statements are supplied to the compiler by the programmer; transfer of control between noncontiguous blocks of machine code is achieved by jumps, many of which may not be necessary.

FIGURE 1
A program and two translations

```
LOOP
    IF B1 THEN S1; EXIT 1 ENDIF;
    LOOP
        IF B2 THEN S2; EXIT 1 ENDIF;
        LOOP
            IF B3 THEN S3; EXIT 1 ENDIF;
            IF B4 THEN EXIT 1 ENDIF;
            IF B5 THEN EXIT 3 ENDIF;
        ENDLOOP;
        IF B6 THEN EXIT 1 ENDIF;
        IF B7 THEN EXIT 2 ENDIF
    ENDLOOP;
    IF B8 THEN EXIT 1 ENDIF
    S4
ENDLOOP;
S5
```

(a) The Source Program

```
START:if ¬B1 then L2
      S1
      jump M5
   L2:if ¬B2 then L3
      S2
      jump L8
   L3:if ¬B3 then L4
      S3
      jump L6
   L4:if B4 then L6
      if B5 then M5
      jump L3
   L6:if B6 then L8
      if B7 then M5
      jump L2
   L8:if B8 then M5
      S4
      jump START
   M5:S5
```

(b) The standard translation

```
   L4:if B4 then L6
      if B5 then M5
   L3:if ¬B3 then L4
      S3
   L6:if B6 then L8
      if B7 then M5
   L2:if ¬B2 then L3
      S2
   L8:if B8 then M5
      S4
START:if ¬B1 then L2
      S1
   M5:S5
```

(c) An optimal translation

B1-B8 are Boolean expressions, S1-S5 are simple statements, START is the entry point, ¬ is logical negation and "EXIT i" indicates that i levels of loops are to be exited.

Whereas the order of the source-code statements may be defensible on grounds of readability, aesthetics, tradition, or language syntax, there is no reason for this order to be preserved in the object code.

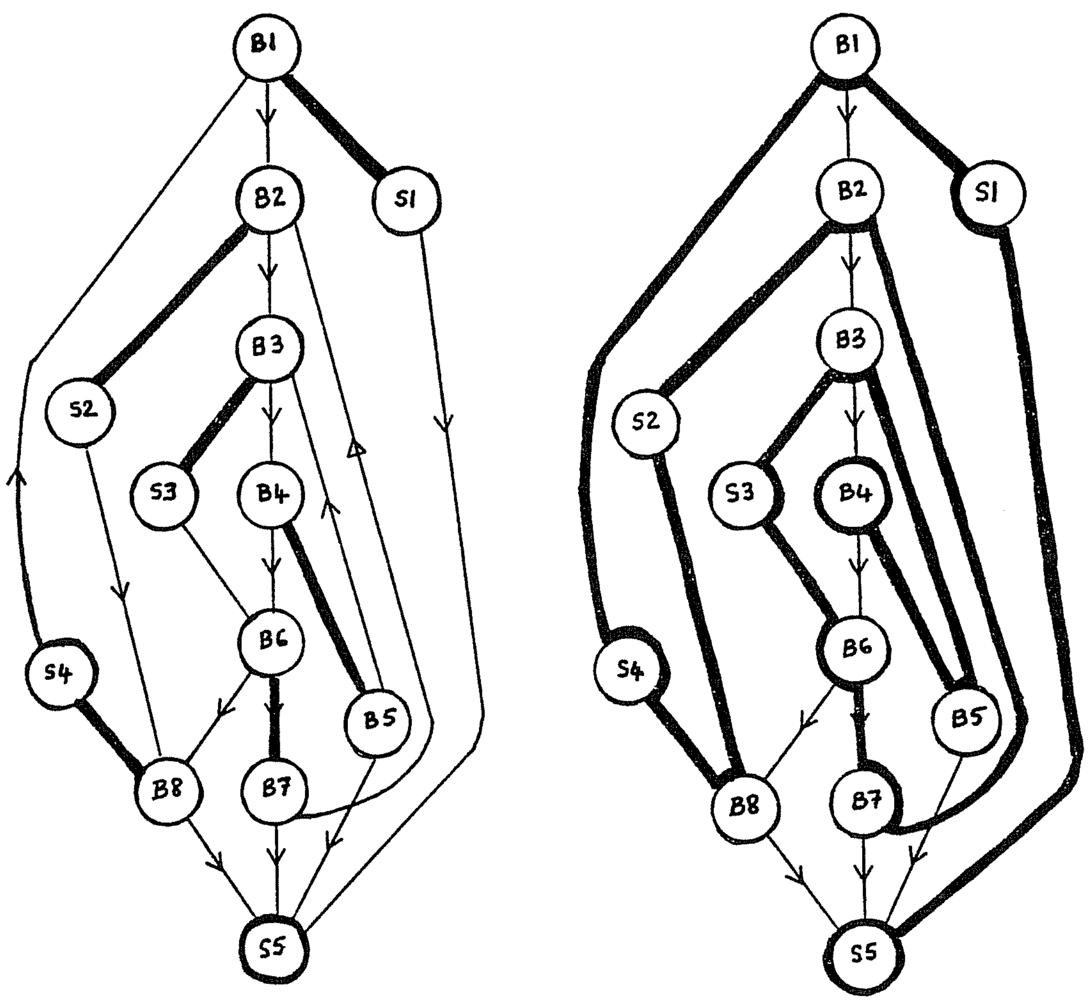
Given a program, we address the problem of finding a translation in which the number of jumps is minimal. Such a translation will result in a smaller object module in general; if the eliminated jumps are on frequently executed paths there may also be a substantial saving in execution time.

We confine ourselves to translations that preserve the nodes and arcs of the flowgraph and ignore the specific nature of the statements represented by the nodes; that is to say, techniques such as node splitting and code motion are beyond the scope of this dissertation since they modify the flowgraph. Any jump-free segment of code in the translation determines a simple path in the flowgraph. Any translation therefore determines a collection of pairwise disjoint simple paths that collectively cover all the nodes of the flowgraph. Such a collection is called a <u>dissection</u>. Conversely, given a dissection of a flowgraph, we can find a corresponding translation such that each path of the dissection yields a jump-free code segment; in each case, the number of jumps in the translation equals the number of paths in the dissection. Finding a minimal-jump translation of a program, therefore, corresponds to finding a minimal-cardinality dissection for the flowgraph of the program. Figure 2 shows the dissections corresponding to the translations of Figures 1(b) and 1(c).

FIGURE 2

The dissections corresponding to the two translations

of Figure 1



2. OVERVIEW

The problem, in its most general form, can be stated as:

> THE DISSECTION PROBLEM (DP) Given a digraph G and a positive integer K, does G have a dissection of cardinality at most K ?

Since a special case of DP (K=1) is the Hamiltonian Path problem (HP) which is known to be NP-Complete [1,2] even for the restricted class of planar digraphs with both the indegree and the outdegree of each node bounded by two, DP itself must be NP-Complete for that class. The only other known attempt to solve DP is [3] where a polynomial-time algorithm for dags and a heuristic for general digraphs are presented. We present an algorithm to solve this problem for the class of ´structured´ program graphs (those obtained by using only IF-THEN-ELSE, LOOP-ENDLOOP and multi-level exits).

By an SFG (Structured Flow Graph) we mean the flow-graph of a structured program. For each r in {0,1,2,...} let SFG(r) denote the class of SFGs that can be represented by a structured program using at most r-level exits. Our algorithm shows that DP is solvable in quadratic time for each class SFG(r); furthermore, we show that the class of all SFGs is the same as the well-known class of Reducible Flowgraphs (RFGs) and that the classes SFG(r) form a strict linear order under containment. The question remains open whether DP or even HP is NP-Complete for RFGs.

The rest of this thesis is organized as follows: Chapter 2 presents the formal model and the algorithm and states the main theorems regarding the correctness and the time complexity of the algorithm. Chapter 3 contains proofs of the theorems of previous chapter. Chapter 4 shows that the class of SFGs is identical to the class of RFGs and examines related issues. Chapter 5 contains discussion, open questions and conjectures.

CHAPTER 2

THE ALGORITHM

1. MOTIVATION

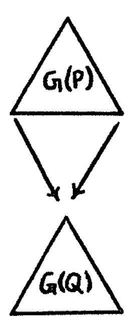
We consider programs that can be built up from atomic statements (assignment statements, input/output statements, procedure calls, but no "go to" statements) and Boolean expressions using the following operations:

(a) Concatenation

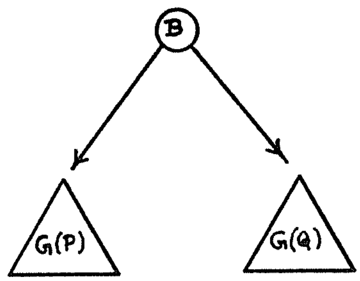
(b) IF-THEN-ELSE-ENDIF

(c) LOOP-ENDLOOP with multilevel exits of the form "EXIT i" where i≥1 specifies the number of nested levels of loops to be exited.

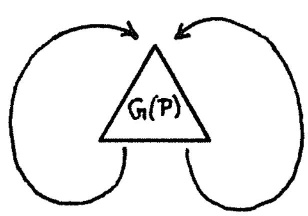
There are analogous operations that produce the flowgraph of a compound statement from the flowgraphs of its components. Suppose G(P) and G(Q) are flowgraphs for programs P and Q respectively. The flowgraph G(P;Q) of the compound statement "P;Q" is obtained by drawing arcs from the ´finish´ nodes of G(P) to the ´start´ node of G(Q):



The flowgraph G(IF B THEN P ELSE Q ENDIF) of the compound statement "IF B THEN P ELSE Q ENDIF" is obtained by creating a new node (corresponding to the Boolean expression B) and drawing arcs from it to the 'start' nodes of G(P) and G(Q):



The flowgraph model, therefore, must include specifications of a 'start' node and a set of 'finish' nodes. In the two cases above, the start node of the resulting graph is simply the start node of G(P) in the first case and the new node in the second; the finish nodes of the resulting graph are the finish nodes of G(Q) in the first case and the union of the finish nodes of G(P) and G(Q) in the second. The case of G(LOOP P ENDLOOP) is more involved. We draw arcs from the finish nodes of G(P) to its start node; the start node of the new graph is the start node of G(P):



The finish nodes of the new graph, however, ought to be all those nodes from which an exit of this particular loop is possible; in order to precisely identify these nodes we augment our model with the specification of a set of nodes called the level-i exit nodes for each $i \geq 1$. A node of G(P) is a <u>level-i exit node</u> iff an exit from i nested loops surrounding the program fragment P is possible immediately after performing the computation represented by that node. Thus, finish nodes can be considered to be level-0 exit nodes. Now, when the LOOP-ENDLOOP operation is applied to P, the levels of all the nodes of G(P) need to be decremented by one since one surrounding loop is now accounted for. We can now describe how to construct the flowgraph G(LOOP P ENDLOOP) from the flowgraph G(P):

-- draw arcs from the level-0 exit nodes of G(P) to its start node.

-- the start node of the resulting graph is the start node of G(P).

-- the level-i exit nodes of the resulting graph are the level-(i+1) exit nodes of G(P).

The level-i exit nodes may be specified by labelling each node with an integer specifying its exit level; a single integer may prove insufficient since a node may be both a level-i and a level-j exit node (eg. IF B THEN EXIT i

ELSE EXIT j ENDIF). So we label each node with a set of integers.

2. NOTATION

ϕ is the empty set ;

$\mathbb{I} = \{0,1,2,...\}$;

$\uparrow$ is a symbol used to label the start node of a flowgraph ;

P(Y) denotes the power set of Y ;

|Y| denotes the cardinality of Y ;

A x B denotes the Cartesian product of sets A and B.

For any set A we define

$A^+ = A \cup \{\uparrow\}, \quad A^- = A - \{\uparrow\}, \quad \hat{A} = A \cap \{\uparrow\}$

If $A \subseteq \mathbb{I}^+$ we define

$A+1 = \{a+1 \mid a \in A \cap \mathbb{I}\} \cup \hat{A}$, and,

$A-1 = \{a \mid a \in \mathbb{I} \text{ and } a+1 \in A\} \cup \hat{A}$

A unit set is a set whose cardinality is one. A relation R is any set of ordered pairs ($R \subseteq A \times B$ for some sets A, B). If R is a relation we define

$DOM(R) = \{x \mid (x,y) \in R \text{ for some } y\}$

$RAN(R) = \{y \mid (x,y) \in R \text{ for some } x\}$

If R and S are relations, $x \in DOM(R)$ and $A \subseteq DOM(R)$, we define

$R(x) = \{y \mid (x,y) \in R\}$

$R[A] = \{y \mid y \in R(x) \text{ for some } x \in A\}$

$R^{-1} = \{(y,x) \mid (x,y) \in R\}$

$R \circ S = \{(x,z) \mid (x,y) \in R \text{ and } (y,z) \in S \text{ for some } y\}$

A relation R is called a <u>function</u> iff $|R(x)| = 1$ for all x in DOM(R). Every relation $R \subseteq A \times B$ determines a function

$\langle R\rangle : P(DOM(R)) \longrightarrow P(RAN(R))$ defined by $\langle R\rangle(X) = R[X]$.

The following special relations will be useful later:

For any set Y, $id_Y = \{ (y,y) \mid y \in Y \}$

For any $i \in \mathbb{I}$, $z_i = ((id_{\mathbb{I}}+) - \{(0,0)\}) \cup \{(0,i)\}$

For any $k \in \mathbb{I}^+$, $del_k = (id_{\mathbb{I}}+) - \{(k,k)\}$

Finally, $decr = \{(\uparrow,\uparrow)\} \cup \{(i+1,i) \mid i \in \mathbb{I}\}$

We denote

$\langle z_i\rangle$ by Z_i,

$\langle decr\rangle$ by DECR, and

$\langle del_k\rangle$ by DEL_k.

3. THE FORMAL MODEL

A <u>program graph</u> G is a triple (N,A,L) where:

N is a non-void finite set of nodes.

$A \subseteq N \times N$ is a set of directed arcs.

$L \subseteq N \times \mathbb{I}^+$ is a relation that associates labels with nodes such that $|L^{-1}(\uparrow)|=1$ (that is, exactly one node is labelled by $\uparrow$).

The unique element of $L^{-1}(\uparrow)$ is denoted by s(G) and called the start node of G, the elements of $L^{-1}(i)$ are called the level-i exit nodes for each $i \geq 0$, and the set $L^{-1}(0)$ is denoted by F(G) and its members are also called finish nodes of G. The pair (N,A) is called the digraph of G. We will write N_G, A_G and L_G if we need to explicitly identify the program graph in question.

Implicit in the above is the assumption that nodes are associated with the atomic actions of a program (simple statements or Boolean expressions). Each node of a program graph is distinguishable from all other nodes of that program graph (by a unique name for instance). Distinct nodes of a program graph may however be associated with the same atomic action as would happen, for example, when the assignment "X:=0" occurs in two different places in a program. We emphasize that labels are not used to distinguish one node from another and so several distinct nodes may have the same set of labels.

A program expression (or simply expression) E, and its program graph G = (N,A,L) are defined recursively:

(a) For any $i \in \mathbb{I}$, and any node n, E=EXIT(i,n) is an expression and $N=\{n\}$, $A=\phi$, $L=\{n\} \times \{\uparrow,0,i\}$.

[If i=0 this corresponds to the simple statement associated with n; if $i \geq 1$, it corresponds to either:

IF B THEN EXIT i ENDIF or

IF B THEN ELSE EXIT i ENDIF

where B is the Boolean expression associated with n.]

(b) Suppose E1 and E2 are expressions, G1=(N1,A1,L1) and G2=(N2,A2,L2) are their respective program graphs, $N1 \cap N2=\phi$, t is a node not in ($N1 \cup N2$) and $i \in \mathbf{I}$, $i \geq 1$. Then, E and G=(N,A,L) as defined by the five operations below are respectively an expression and its program graph.

<u>BREAK</u>

E=BREAK(E1,i), N=N1, A=A1, $L=z_i \circ L1$

[If E1 is a program, this corresponds to the new program "E1;EXIT i". Since control can reach the "EXIT i" only from the finish nodes of G1 we simply replace all 0-labels with i-labels using the relation z_i. Note that we do not create a new node for the "EXIT i" statement.]

<u>LOOP</u> (See Figure 3)

E=LOOP(E1),

N=N1, $A=A1 \cup (F(G1) \times \{s(G1)\}$, $L=decr \circ L1$.

[If E1 is a program, this corresponds to the new program "LOOP E1 ENDLOOP". The 'decr' relation is used to specify that the new level-i exit nodes are the old level-(i+1) exit nodes; the new arcs are given by $F(G1) \times \{s(G1)\}$]

<u>CAT</u> (See Figure 3)

E=CAT(E1,E2),

N=N1 $\cup$ N2, A=A1 $\cup$ A2 $\cup$ (F(G1) x $\{$s(G2)$\}$),

L=(del_0 o L1) $\cup$ ($del_\uparrow$ o L2)

[This corresponds to concatenating two programs E1 and E2. The two parts of the labelling relation L specify respectively that the finish nodes of G1 are no longer finish nodes and that the start node of G2 is no longer the start node; all other labels are unaltered.]

<u>IF</u> (See Figure 3)

E=IF(t,E1), N=N1 $\cup$ $\{t\}$, A=A1 $\cup$ $\{$(t,s(G1)$\}$,

L=($\{t\}$ x $\{\uparrow,0\}$) $\cup$ ($del_\uparrow$ o L1)

[This corresponds to either of :

IF B THEN ELSE E1 ENDIF

or IF B THEN E1 ELSE ENDIF where B is the Boolean expression associated with t. The new node t is the start node and a finish node of the new graph: this is specified by $\{t\}$ x $\{\uparrow,0\}$ in the labelling relation.]

<u>ELSE</u> (See Figure 3)

E=ELSE(t,E1,E2), N=N1 $\cup$ N2,

A=A1 $\cup$ A2 $\cup$ ($\{t\}$ x $\{$s(G1),s(G2)$\}$),

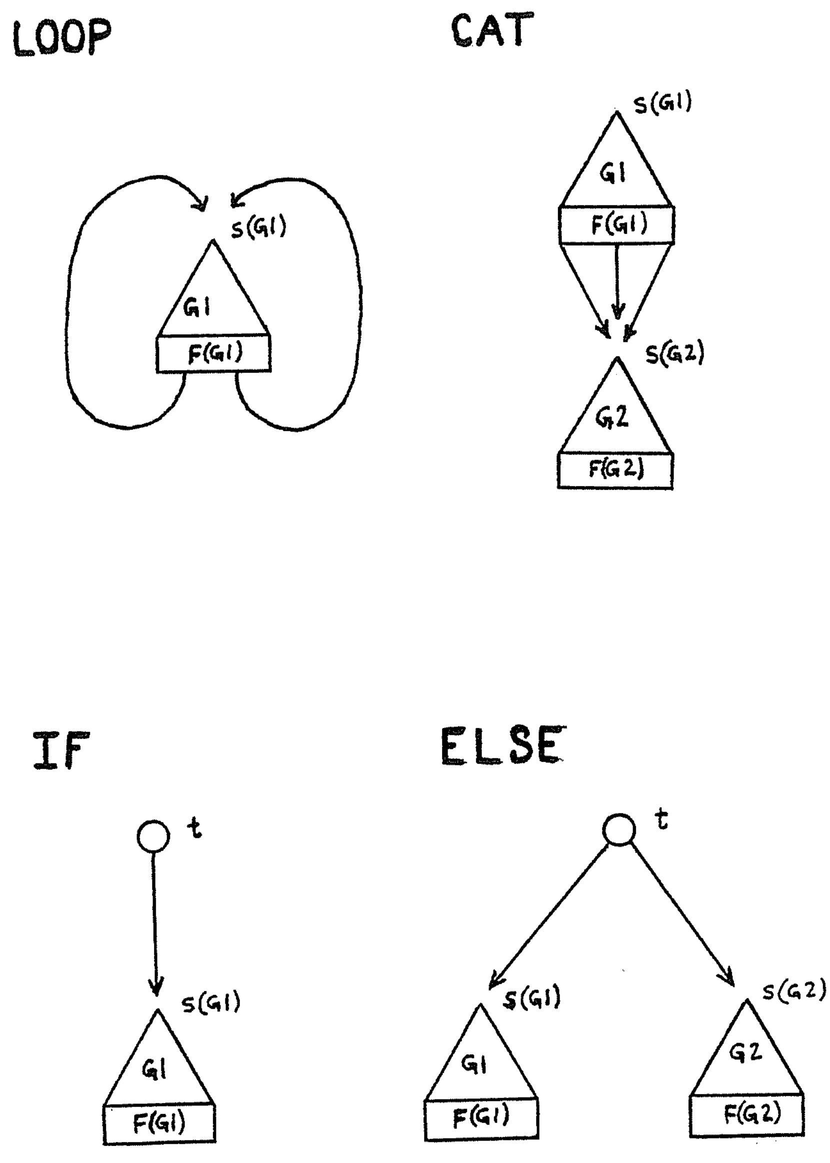
L=($del_\uparrow$ o(L1 $\cup$ L2)) $\cup$ $\{$(t,$\uparrow$)$\}$

[This corresponds to the statement

IF B THEN E1 ELSE E2 ENDIF]

FIGURE 3

SPG operations



A program graph is called a *Structured Program Graph* (*SPG*) iff it is the program graph of some expression; the six operations listed above are called *SPG-operations*. The *size* of an expression E is the number of SPG-operations used in E; the *rank* of E is max{ i | EXIT(i,n) or BREAK(−,i) occurs in E}; that is, the rank is the maximum depth of nested loops that are exited by an "EXIT i" statement in the program. If E is an expression, G its program graph and H the digraph of G we will say that

G is a program graph of H,

E is an expression for H, and

E is an expression for G.

We will sometimes use graph-theoretic terms in connection with an expression; it is to be understood in these cases that the terms apply to the program graph of the expression. Thus if E is an expression, the phrases: the start node of E, the nodes of E, the arcs of E, a path in E, all refer to the program graph of E.

REMARKS :

(1) L is always a finite relation.

(2) Each node of an SPG can have at most two numeric labels; that is, $|L(n)-\{\uparrow\}| \leq 2$ for all nodes n.

(3) An SPG can have any number of finish nodes but can

have at most one start node.

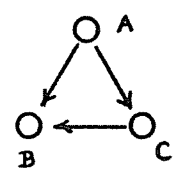
Several points are noteworthy about our model:

1. We do not explicitly label the ´TRUE´ and ´FALSE´ branches; this information, though easy to incorporate, is superfluous for our purposes, since randomly interchanging the ´TRUE´ and ´FALSE´ branches leaves the underlying digraph, and hence the minimal-cardinality dissection, unchanged.
2. We only deal with graphs where the outdegree of each node is bounded by two.
3. Other models in the literature [4,5,6] create individual nodes for the statements "EXIT i" or for the keywords "LOOP" and "ENDLOOP" (sometimes called "REPEAT" and "END" respectively). We diverge from this practice for several reasons: Firstly, these statements (or keywords) merely represent control information like the keywords "THEN" and "ELSE" in an IF statement and do not constitute atomic actions as we understand them. Secondly, creating nodes for them destroys the correspondence between jump-free segments in the object code and paths in the flowgraph. Thirdly, graph isomorphism in our model becomes the more obscure "very strong

equivalence" in the other models [6]. Figure 4 shows an example of an infinite class of programs, all of which have the same program graph in our model but have progressively larger non-isomorphic flowcharts in the other models.

4. Even though we have not formally defined the relationship between a program and the expression for it, it should be clear that given a program, a parse tree for it provides an expression.

5. If digraphs are to model programs accurately, it is essential that they include specifications of start and finish nodes, as our model does, since the same underlying digraph could represent different programs depending on the start/finish specifications. For instance, the digraph:



represents the following program if A is the start node and B is the only finish node:

IF A THEN C ELSE

ENDIF ;

B

However, if B and C are <u>both</u> required to be finish nodes, a more complex program is necessary:

```
LOOP
IF A THEN
  IF C THEN EXIT 1
  ELSE ENDIF
ELSE ENDIF;
B; EXIT 1
ENDLOOP
```

6. We make no assumptions about whether or not Boolean expressions may have side effects; that is, we permit the case where the evaluation of a predicate modifies the values of the other variables of the program via, for example, function calls. The models of [4] and [6] assume that predicate evaluation can have no side effects.

7. It is possible in our model to obtain graphs in which there are unreachable nodes (i.e. there is dead code in the corresponding programs). This will happen whenever the CAT(P,Q) operation is applied and the program graph of P has no finish nodes. We could prohibit the use of the CAT operation under such circumstances but we do not do so since unreachable nodes do not cause any difficulties in our algorithm.

Figure 4

An infinite class of expressions all of which have isomorphic program graphs in our model.

E_0=EXIT(0,n), E_i=LOOP(BREAK(E_{i-1},1)) for $i \geq 1$.

The corresponding class of programs $\{P_i \mid i \in \mathbb{I}\}$ is given by:

P_0 : <code for n> P_i : LOOP   (for $i \geq 1$)
<program P_{i-1}>
EXIT 1
ENDLOOP

All of these expressions have the same program graph $G = (N, A, L)$ in our model : $N=\{n\}$, $A=\phi$, $L=\{(n,\uparrow),(n,0)\}$.

In the other models of the literature, the flowcharts for these programs are "very strongly equivalent" but not isomorphic:

```
↓      ↓            ↓
n      LOOP         LOOP         etc.
↓      ↓            ↓
       n            LOOP
       ↓            ↓
       EXIT 1       n
       ↓            ↓
       ENDLOOP      EXIT 1
       ↓            ↓
                    ENDLOOP
                    ↓
                    EXIT 1
                    ↓
                    ENDLOOP
                    ↓
```

4. DEFINITIONS

Throughout this section, G=(N,A,L) is an arbitrary but fixed SPG. The definitions of this section are illustrated by an example at the end of the section. A labelled path $\gamma(G)$ in G is a pair (α, β) where:

$\alpha = (a_1, a_2, \ldots, a_k)$ is a sequence of distinct nodes $a_i \in N$, such that (a_i, a_i+1) is an arc in G for $1 \leq i < k$; α is called the node sequence of γ, a_1 is called the beginning of γ and denoted by $b(\gamma)$ and a_k is called the end of γ and denoted by $e(\gamma)$.

$\beta = (L(a_1), L(a_2), \ldots, L(a_k))$ is called the label sequence of γ (we will write γ for $\gamma(G)$ when there is no ambiguity).

A labelled path, as we have defined it here, is a simple path in the digraph of G (in the usual graph-theoretic sense) together with the sets of labels on those nodes. If E is an expression and E1 a subexpression of E then, a labelled path in E1 may have the same node sequence as a labelled path in E, but their label-sequences could be quite different owing to the difference in the labelling relations of E and E1. Suppose H is an arbitrary SPG and $\gamma(H) = (\alpha, \beta)$ is an arbitrary labelled path in H such that α is also a simple path in the digraph of G. We write $L_G(\gamma)$ for the labelled path obtained from α by using the labelling relation L of G:

$$L_G(\gamma) = (\alpha, (L(a_1), \ldots, L(a_k)))$$

We extend this notation to collections of labelled paths. If w is any collection of labelled paths in any SPG such that $L_G(\gamma)$ is well-defined for each γ in w, then, we define:

$$L_G(w) = \{ L_G(\gamma) \mid \gamma \in w \}$$

This is a slight abuse of notation, since L_G is a transformation on labelled paths (and labelled path collections) and also the labelling relation of G; it should, however, be clear from the context as to which is meant.

Two labelled paths are <u>disjoint</u> iff their node sequences have no common elements. If γ and γ' are disjoint labelled paths in G such that there is an arc in G from the end of γ to the beginning of γ', their <u>concatenation</u>, denoted by "$\gamma.\gamma'$", is the labelled path obtained by concatenating their corresponding node and label sequences. A <u>dissection</u> v of G is a collection of labelled paths in G such that every node of G occurs in <u>exactly</u> one member of v. A dissection v of G is called an <u>optimal dissection</u> iff no dissection of G has cardinality lower than $|v|$. The <u>cost of G</u>, denoted by d(G), is the cardinality of an optimal dissection of G. We will write v_G or v(G) if there is need to explicitly identify the

SPG for which v is a dissection. We will write v_E or v(E) to indicate that v is a dissection for the program graph of the expression E.

If $\gamma = (\alpha, (c_1, \ldots, c_k))$ is a labelled path in G, the <u>signature</u> of γ, denoted by sg(γ), is the subset of $\mathbb{I}^+$ defined by:

$$sg(\gamma) = \hat{c}_1 \cup c_k^-$$

In other words, sg(γ) consists of all the labels of the end of γ except $\uparrow$, together with $\uparrow$ if it is a label of the beginning of γ. A labelled path γ in G is called a <u>T-path</u> iff its signature is the set T (i.e. sg(γ) = T), a <u>start-path</u> iff it begins at the start-node of G (i.e. $\uparrow \in$ sg(γ)) and a <u>finish-path</u> iff it ends at a finish-node of G (i.e. $0 \in$ sg(γ)).

<u>REMARK</u> (4) A dissection can have any number of finish-paths but at most one start-path.

The following example illustrates the concepts of this section.

<u>EXAMPLE</u> (1) Two expressions R and E and their respective program graphs H and G are shown in Figure 5. The corresponding programs are shown below:

```
R:  IF n1 THEN
       IF n3 THEN
         IF n7 THEN EXIT 2
         ENDIF
       ELSE n6
       ENDIF
    ELSE IF n2 THEN
            IF n5 THEN EXIT 1
            ENDIF ;
          EXIT 3
          ENDIF ;
          IF n4 THEN EXIT 2
          ENDIF
    ENDIF
```

```
E:  LOOP
    <program R>
    ENDLOOP
```

The following table shows some node sequences and the labelled paths they determine respectively in H and G.

NODE SEQUENCE	LABELLED PATH IN H	PATH IN G
$\alpha_1=(n1,n3,n6)$	$\gamma_1=(\alpha_1,(\{\uparrow\},\phi,\{0\}))$	$\delta_1=(\alpha_1,(\{\uparrow\},\phi,\phi))$
$\alpha_2=(n2,n4)$	$\gamma_2=(\alpha_2,(\phi,\{0,2\}))$	$\delta_2=(\alpha_2,(\phi,\{1\}))$
$\alpha_3=(n5)$	$\gamma_3=(\alpha_3,(\{1,3\}))$	$\delta_3=(\alpha_3,(\{0,2\}))$
$\alpha_4=(n7)$	$\gamma_4=(\alpha_4,(\{0,2\}))$	$\delta_4=(\alpha_4,(\{1\}))$

For each i, $\delta_i = L_G(\gamma_i)$.

The concatenations

$\delta_2.\delta_1$ = ((n2,n4,n1,n3,n6),(ϕ,{1},{$\uparrow$},ϕ,ϕ))
and $\delta_4.\delta_1$ = ((n7,n1,n3,n6),({1},{$\uparrow$},ϕ,ϕ))
are well-defined labelled paths in G since (n4,n1) and (n7,n1) are arcs of G; the concatenation $\delta_1.\delta_2$ is not well-defined in G since (n6,n2) is not an arc of G.

The signatures of these labelled paths are:

PATH	SIGNATURE
γ_1	$\{\uparrow,0\}$
γ_2	$\{0,2\}$
γ_3	$\{1,3\}$
γ_4	$\{0,2\}$
δ_1	$\{\uparrow\}$
δ_2	$\{1\}$
δ_3	$\{0,2\}$
δ_4	$\{1\}$
$\delta_2.\delta_1$	ϕ
$\delta_4.\delta_1$	ϕ

The collection $w = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ is a dissection for H.
The collections

$v = \{\delta_1, \delta_2, \delta_3, \delta_4\}$,

$x = \{\delta_2.\delta_1, \delta_3, \delta_4\}$, and

$y = \{\delta_4.\delta_1, \delta_2, \delta_3\}$ are all dissections for G.

The start and finish paths of these dissections are given below:

DISSECTION	START-PATH	FINISH-PATHS
w	γ_1	$\gamma_1, \gamma_2, \gamma_4$
v	δ_1	δ_3
x	none	δ_3
y	none	δ_3

Figure 5

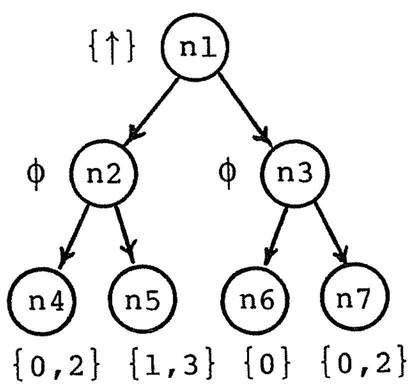
Two expressions and their program graphs.

R = ELSE(n1,ELSE(n3,EXIT(2,n7),n6),CAT(IF(n2,

BREAK(EXIT(1,n5),3)),EXIT(2,n4)))

E = LOOP(R)

<u>H</u> (<u>prog</u>. <u>graph</u> <u>of</u> <u>R</u>) <u>G</u> (<u>prog</u>. <u>graph</u> <u>of</u> <u>E</u>)



The labels of each node are shown next to it. Both R and E have rank 3; their sizes are respectively 8 and 9.

5. ALGORITHM PRELIMINARIES

Let D = { v | v is a dissection for some SPG }. We define functions # and f : $(P(\mathbb{I}^+) - \{\phi\}) \times D \longrightarrow \mathbb{I}$ by

$\#(T,v)$ = the number of T-paths in v

$= |\{\gamma \mid \gamma \in v \text{ and } sg(\gamma) = T\}|$

$f(T,v) = \min \{ |T|, \#(T,v) \}$

Every dissection v of an arbitrary SPG G determines a function f_v: $(P(\mathbb{I}^+)-\{\phi\}) \longrightarrow \mathbb{I}$, called the <u>character</u> of v, defined by $f_v(T) = f(T,v)$. We define an equivalence relation on dissections based on their character:

$v \sim w$ iff $f_v = f_w$ [i.e. $f(T,v) = f(T,w)$ for all T].

This equivalence relation has a crucial role to play in the algorithm. Example 2 below gives the characters of the dissections of Example 1.

For any collection X of dissections we define an <u>optimal choice-set from X</u> to be any collection of dissections Y obtained by choosing from each equivalence class C of X, <u>exactly</u> one dissection whose cardinality is minimal in C.

<u>EXAMPLE</u> (2) The characters of the dissections v, w, x, and y of Example 1 are given below.

T	$f_w(T)$	$f_v(T)$	$f_x(T)$	$f_y(T)$
$\{\uparrow\}$	0	1	0	0
$\{1\}$	0	1	1	1
$\{0,2\}$	2	1	1	1
$\{1,3\}$	1	0	0	0
$\{\uparrow,0\}$	1	0	0	0
all other T	0	0	0	0

Suppose E, E1, E2 are expressions and v1, v2 are dissections for E1, E2 respectively. We define several operations on dissections; each operation is the analogue of an SPG-operation and produces a set of dissections for a composite graph from dissections of the component subgraphs. For each operation we provide, in addition to a precise and formal definition using the notation of the previous section, an informal definition where only the node sequences of the dissections are specified, it being understood that the labels of E are used in all cases to obtain the corresponding label sequences. These operations are illustrated pictorially in Figure 6 (which follows these definitions) and by a specific example at the end of the section.

(a) If E = BREAK(E1,i) then

BREAK(v1,i) = { L_E(v1) }

= { The unique dissection of E whose node sequences are exactly those of v1 }

<u>REMARK</u> (5) L_E(v1) is a single dissection, so BREAK(v1,i) is a unit set.

(b) If E = LOOP(E1) then

$LOOP_1$(v1) = {L_E(v1)}

= {The unique dissection of E whose node sequences are exactly those of v1 }

$LOOP_2$(v1) = { L_E(v1-{p,q}) ∪ {L_E(p).L_E(q)} | q is a start path and p is a finish path in v1 }

= { v | v is a dissection of E obtained from v1 by concatenating a finish path and a start path. }

LOOP(v1) = $LOOP_1$(v1) ∪ $LOOP_2$(v1)

<u>REMARK</u> (6) $LOOP_1$(v1) is a unit set; $LOOP_2$(v1) may have zero, one or more elements.

(c) If E = CAT(E1,E2) then

CAT_1(v1,v2) = { L_E(v1) ∪ L_E(v2) }

= { The unique dissection of E whose node sequences are those of v1 together with those of v2 }

$CAT_2(v1,v2) = \{ L_E((v1-\{p\}) \cup (v2-\{q\})) \cup \{L_E(p).L_E(q)\} \mid$ p is a finish path p of v1 and q is a start path of v2 }

= { v | v is a dissection of E obtained from v1 $\cup$ v2 by concatenating a finish path of v1 to a start path of v2 }

$CAT(v1,v2) = CAT_1(v1,v2) \cup CAT_2(v1,v2)$

<u>REMARK</u> (7) $CAT_1(v1,v2)$ is a unit set; $CAT_2(v1,v2)$ may have zero, one or more elements.

(d) If E = IF(j, t, E1), let p´ be the single-node labelled path $(t,L_E(t))$ in E. Then,

$IF_1(t,v1) = \{ L_E(v1) \cup \{p'\} \}$

= { The unique dissection of E whose node sequences are those of v1 together with (t) }

$IF_2(t,v1) = \{ L_E(v1-\{q\}) \cup \{p'.L_E(q)\} \mid$ q is a start path of v1 }

= { v | v is a dissection of E obtained from v1 by concatenating (t) with a start path of v1 }

$$IF(t,v1) = IF_1(t,v1) \cup IF_2(t,v2)$$

REMARK (8) $IF_1(t,v1)$ is a unit set; $IF_2(t,v1)$ is either empty or a unit set.

(e) If $E = ELSE(t, E1, E2)$, let p' be the single-node labelled path $(t, L_E(t))$ in E. Then,

$$ELSE_1(t,v1,v2) = \{ L_E(v1) \cup L_E(v2) \cup \{p'\} \}$$

$= \{$ The unique dissection of E whose node sequences are those of v1 and v2 together with (t) $\}$

$ELSE_2(t,v1,v2)$

$= \{ L_E(v2 \cup (v1-\{q\})) \cup \{p'.L_E(q)\} \mid$ q is a start path of v1 $\} \cup$

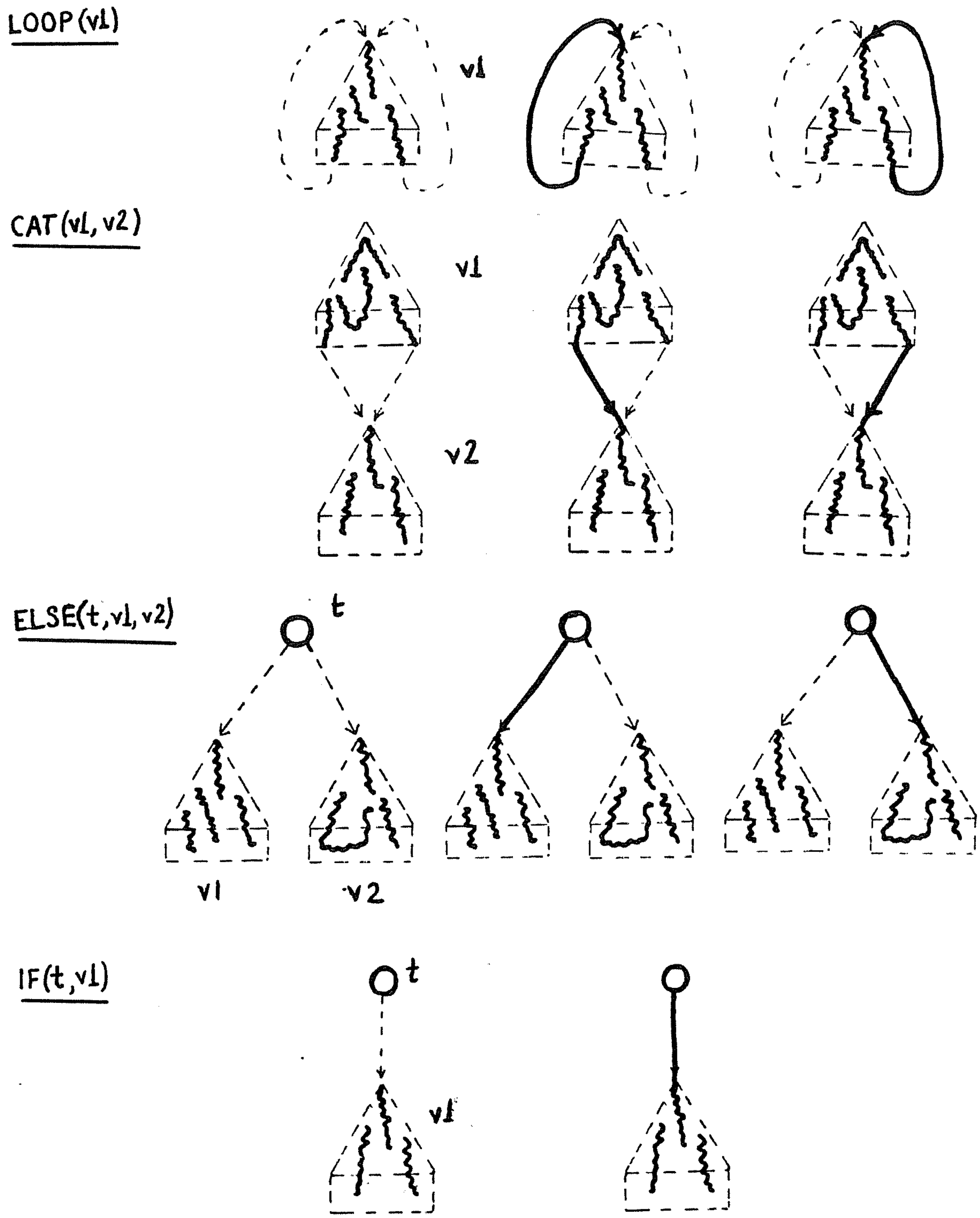
$\{ L_E(v1 \cup (v2-\{q\})) \cup \{p'.L_E(q)\} \mid$ q is a start path of v2 $\}$

$= \{ v \mid$ v is a dissection of E obtained from $v1 \cup v2$ by concatenating (t) with a start path of v1 or v2 $\}$

REMARK (9) $ELSE_1(t,v1,v2)$ is a unit set; $ELSE_2(t,v1,v2)$ has no more, though possibly fewer, than two elements.

FIGURE 6

SPG-Operations on dissections



We will use OP(i,t,v1,v2) to mean any one of: BREAK(v1,i), LOOP(v1), CAT(v1,v2), IF(t,v1) or ELSE(t,v1,v2). Likewise, OP(i,t,E1,E2) will denote any of the corresponding five SPG-operations.

<u>EXAMPLE</u> (3) We refer to Figure 5 and the dissections v, w, x and y of Example 2.

$LOOP_1(w) = \{v\}$; $LOOP_2(w) = \{x, y\}$ and

$LOOP(w) = \{v, x, y\}$

6. THE ALGORITHM

This section contains a few prefatory remarks, the main algorithm, and the two main theorems regarding its correctness and time-complexity; the entirety of the next chapter is occupied with the proofs of these theorems.

The algorithm keeps track of a family of dissections for each SPG and, as each SPG-operation is applied, constructs members of this family for a composite SPG from the (recursively computed) families for the component subgraphs. The family must be chosen with care; the total number of dissections of a program graph can grow exponentially in the size of the graph and so the family of all dissections will not do. The family that our algorithm uses is an optimal choice-set (Section 5) from the class of all dissections. The equivalence relation ~ provides sufficiently fine resolution to ensure that this choice-set has all the necessary information (to build the choice-set for the larger graph from the ones for the smaller graphs) and yet is sufficiently coarse to ensure that the number of equivalence classes does not grow exponentially in the size of the graph.

ALGORITHM Opt-Dissection

INPUT An expression E.

OUTPUT An optimal dissection v for the flowgraph of E.

METHOD Call Dissect(E,Y); v := MIN(Y).

(* MIN(Y) is a minimum-cardinality member from the family of sets Y; the procedure DISSECT is defined below. *)

```
PROCEDURE Dissect(E,Y);
(* E is an expression ; the procedure returns an optimal
choice set in Y. *)
BEGIN IF E is of the form EXIT(i,n) THEN
         Let X := {u} where u is the unique dissection
                              of the flowgraph of E.
      ELSE  (* E = OP(i,t,E1,E2) for some expressions E1
             and E2 where OP is some SPG-operation other
             than EXIT. *)
            Dissect(E1,Y1); Dissect(E2,Y2);
            Let X := { v | v ∈ OP(i,t,v1,v2) for
                              some v1 ∈ Y1 and v2 ∈ Y2 }
      ENDIF;
      (* The following loop creates Y from the set X
      created above *)
      Y := ∅ ;
      FOR EACH equivalence class C of X (under ~) DO
          Y := Y ∪ MIN(C)
 END; (* of PROCEDURE Dissect *)
```

The algorithm has been stated in very general terms to facilitate a quick overview of the various steps involved. We now present an expanded version in Pascal-like syntax; this version clarifies some implementation details and is useful in computing the time-complexity of the algorithm. The line numbers at left (in PROCEDURE Create_Y) are used in the proofs of the next chapter.

```
PROGRAM Opt-dissection ;
VAR  E : expression ;
     v,w : dissection ;
     Y : SET OF dissection ;
  PROCEDURE Dissect( E : expression ;
                     VAR Y : SET OF  dissection) ;
  VAR Y1,Y2,X : SET OF dissection (initially void) ;

    FUNCTION Compare_f(v,w : dissection) : BOOLEAN ;
    (* Returns TRUE if v and w have the same
    character, FALSE otherwise *)
    VAR L : INTEGER ; S : SET ;
      BEGIN (* function Compare_f *)
      L := MAX{ i ∈ I | i labels some node
                     in v or w } ;
      S := { i ∈ I | 0≤i≤L }+ ;
      (* The following loop compares the characters
         of v and w;  only a finite number of
```

```
     comparisons are necessary since f(T,v) and
     f(T,w) are non-zero for only finitely many T
     as we shall prove in the next chapter. *)
  FOR EACH T IN {T | |T⁻| ≤ 2, T ⊆ S } DO
    IF (f(T,v) ≠ f(T,w)) THEN
       Compare_f := FALSE ; RETURN
    ENDIF;
  Compare_f := TRUE ;
  END ; (* function Compare_f *)

PROCEDURE Create_Y (S : SET OF dissection ) ;
(* Creates the optimal choice-set of the set S
   in the variable Y which is a reference
   parameter to Dissect *)
VAR v,w : dissection ;
  BEGIN (* procedure Create_Y  *)
```

```
(1)     LL : FOR EACH v IN S DO
(2)            FOR EACH w IN Y DO
(3)              IF Compare_f(v,w) THEN
                   (* The equivalence class of v al-
                   ready has a representative in Y *)
(4)                IF (|v| < |w|) THEN
                     (* v is a cheaper representative
                        and so replaces w in Y *)
                     Y := (Y-{w}) U {v}
                   ENDIF ;
                   (* In either case, continue
                      with the next iteration
                      of the outer loop *)
                   CONTINUE LL
                 ENDIF ;
               ENDFOR (* Inner loop *)
               (* If control reaches here, the
                  equivalence class of v has no
                  representative in Y so far; we
                  therefore add it to Y *)
               Y := Y U {v}
             ENDFOR (* outer loop *)
        END ; (* procedure Create_Y *)
```

```
BEGIN (* procedure Dissect *)
IF E = EXIT(i,n) THEN
  Y := { the unique dissection of E };
  RETURN
ENDIF ;
(* Here E is not atomic *)
IF E = BREAK(E1,i) THEN
  Dissect(E1,Y1) ;
  FOR EACH v IN Y1 DO
    X := X U BREAK(v,i)
ELSE IF E = LOOP(E1) THEN
  Dissect(E1,Y1) ;
  FOR EACH (v IN Y1) DO
    X := X U LOOP(v)
ELSE IF E = CAT(E1,E2) THEN
  DISSECT(E1,Y1);  DISSECT(E2,Y2) ;
  FOR EACH (v IN Y1) AND EACH (w IN Y2) DO
    X := X U CAT(v,w)
ELSE IF E = IF(t,E1) THEN
  Dissect(E1,Y1) ;
  FOR EACH v IN Y1 DO
    X := X U IF(t,v)
ELSE IF E = ELSE(t,E1,E2) THEN
  Dissect(E1,Y1) ;  Dissect(E2,Y2) ;
  FOR EACH (v IN Y1) AND EACH (w IN Y2) DO
```

```
    X := X ∪ ELSE(t,v,w)
  ENDIF ;
  (* using the set X formed above, we now create an
     optimal choice-set from it *)
  Create_Y(X) ;
  END ; (* procedure Dissect *)


BEGIN (* main program *)
  READ(E) ;  Dissect(E,Y) ;
  (* Find an element of Y whose cardinality is
     minimal. *)
  v := an arbitrary member of Y ;
  FOR EACH w IN Y DO
    IF (|w| < |v|) THEN
      v := w
    ENDIF;
  WRITE("An optimal dissection is:" , v)
END. (* main program *)
```

The principal properties of the algorithm are summarized by the following theorems:

<u>THEOREM</u> <u>1</u> The output v of the algorithm is an optimal dissection for the flowgraph of E.

<u>THEOREM</u> <u>2</u> The running time of OPT-DISSECTION is $3^{O(r^2)}n^2$ where r is the rank of E and n is size of E.

CHAPTER 3

PROOFS OF THEOREMS

1. PRELIMINARY RESULTS

LEMMA (1) Suppose T and T′ are arbitrary subsets of $\mathbb{I}^+$ and v is an arbitrary disection of an SPG G.

(a) $f(T,v) = 0$ iff $\#(T,v) = 0$

(b) $f(T,v) \neq 0 \implies |T^-| \leq 2$

(c) $f(T,v) = 0$ for all but finitely many T

(d) $\uparrow \in T \implies f(T,v) = \#(T,v) \leq 1$

(e) $\left.\begin{array}{l}\uparrow \in T \cap T' \text{ and} \\ \#(T,v) = 1\end{array}\right\} \implies T=T'$ or $\#(T',v)=0$

(f) $|T| > f(T,v) \implies f(T,v) = \#(T,v)$

Proof

(a) and (f) follow from the definitions of # and f; (b) follows from Remark (2); (c) follows from Remark (1); (d) and (e) are simple consequences of Remark (4).

LEMMA (2) For any $T \subseteq \mathbb{I}^+$, $i \in \mathbb{I}$ and $k \in \mathbb{I}$

(a) $$Z_i^{-1}(T) = \begin{cases} \phi \text{ if } 0 \in T \\ \{T\} \text{ if } 0 \notin T \text{ and } i \notin T \\ \{T,\ T \cup \{0\},\ (T-\{i\}) \cup \{0\}\} \text{ if } 0 \notin T \text{ and } i \in T. \end{cases}$$

(b) $DECR^{-1}(T) = \{\ T+1,\ (T+1) \cup \{0\}\ \}$

(c) $DEL_k^{-1}(T) = \begin{cases} \phi \text{ if } k \in T \\ \{T, T \cup \{k\}\} \text{ otherwise.} \end{cases}$

Proof:

(a) If $0 \in T$, then $Z_i(T') = T$ is impossible for any T'.

If $0 \notin T$, $i \notin T$ then suppose $Z_i(T') = T$. Since Z_i affects only 0's and i's, we must have $T=T-\{0,i\}=T'-\{0,i\}$. If either 0 or i was in T', i would be in $Z_i(T') = T$. Hence we must have $T'=T$.

If $0 \notin T$, $i \in T$, suppose $Z_i(T')=T$. As before, $T-\{i\}=T-\{0,i\}=T'-\{0,i\}$. There are four possibilities:

$0,i \in T' \implies T' = T \cup \{0\}$

$0 \notin T', i \in T' \implies T' = T$

$0 \in T', i \notin T' \implies T' = (T-\{i\}) \cup \{0\}$

$0,i \notin T'$ is impossible since $i \in T$.

(b) Suppose $DECR(T')=T$. Then $T+1 \subseteq T'$. The only other element that could be in T' is 0 .

(c) If $k \in T$ then $DEL_k(T') = T$ is impossible for any T'. If $k \notin T$ and $DEL_k(T') = T$ then we must have $T \subseteq T'$. The only other element that could be in T' is k.

<u>LEMMA</u> (3) Let

c be any element of $\mathbb{I}$

K be any finite set

A be any non-void subset of $\mathbb{I}^+$

v be any element of D

B_k be any non-void subset of $\mathbb{I}^+$ for each $k \in K$

v_k be any element of D

$$[f] = \sum_{k \in K} f(B_k, v_k)$$

$$[\#] = \sum_{k \in K} \#(B_k, v_k)$$

Then

(a) $\min\{|A|, c+[f]\} \leq \min\{|A|, c+[\#]\}$

(b) Equality holds in (a) if $c \geq 0$ and $|B_k| \geq |A|$ for all $k \in K$.

(c) If $0, \uparrow \notin A$ and $f(A \cup \{0\}, v) \geq 1$ then

$$\min\{|A|, -1 + f(A \cup \{0\}, v) + [f]\}$$
$$= \min\{|A|, -1 + \#(A \cup \{0\}, v) + [\#]\}$$

Proof:

(a) By definition of f, $f(T,v) \leq \#(T,v)$ for all T and v.

(b) Let LHS and RHS denote respectively the left hand side and right hand side of the inequality (a). Suppose the assumptions of (b) hold. We need only show that LHS $\geq$ RHS; equality then follows from (a).

Case I $|A| \leq c+[f]$. Here LHS $= |A| \geq$ RHS.

Case II $|A| > c+[f]$. For all $k \in$ we have:

$|B_k| \geq |A| > c+[f] \geq f(B_k, v_k)$ and so $f(B_k, v_k) = \#(B_k, v_k)$ by Lemma 1(f); equality follows.

(c) Let LHS and RHS denote respectively the left and right hand sides of the desired equality; assume that the premises of (c) hold. Then $0 \leq LHS \leq RHS \leq |A|$. By assumption and from Lemma 1(b) we see that $|(A \cup \{0\})| = |(A \cup \{0\}) - \{\uparrow\}| \leq 2$ and so $|A| = 1$ and $|A \cup \{0\}| = 2$. So we have: $0 \leq LHS \leq RHS \leq 1$.

If LHS = 1 then LHS=RHS follows trivially.

If LHS = 0, then, since $|A| = 1$ we must have

$-1+f(A \cup \{0\},v)+[f] = 0$ and so from our assumptions we see that $f(A \cup \{0\},v) = 1$ and $f(B_k,v_k) = 0$ for all $k \in K$. Now $|A \cup\{0\}| = 2$ and $f(A \cup \{0\},v) = 1$ together imply that $\#(A \cup \{0\},v) = 1$ (Lemma 1(f)). $f(B_k,v_k) = 0$ implies that RHS = 0 by Lemma 1(a). Thus LHS = RHS = 0.

The following Lemma proves that any dissection for a composite program graph can be expressed in terms of dissections of the component program graphs and expresses the cardinality of the former in terms of the cardinalities of the latter.

<u>LEMMA</u> (4) Suppose E, E1 and E2 are arbitrary expressions and v is any dissection for E.

(a) If E = EXIT(i,n) then v is the only dissection for for E and $|v| = 1$.

(b) If E = BREAK(E1,i) then

(i) For any dissection w1 of E1,

$$v \in BREAK(w1,i) \implies |v| = |w1|$$

(ii) There is a dissection w1 of E1 such that $v \in BREAK(w1,i)$.

(c) If E = LOOP(E1) then

(i) For any dissection w1 of E1,

$$v \in LOOP_1(w1) \implies |v|=|w1|$$

$$v \in LOOP_2(w2) \implies |v|=|w1|-1$$

(ii) There is a dissection w1 of E1 such that $v \in LOOP(w1)$

(d) If E = CAT(E1,E2) then

(i) For any dissections w1, w2 of E1, E2 respectively,

$$v \in CAT_1(w1,w2) \implies |v| = |w1|+|w2|$$

$$v \in CAT_2(w1,w2) \implies |v| = |w1|+|w2|-1$$

(ii) There are dissections w1, w2 of E1, E2 respectively such that $v \in CAT(w1,w2)$.

(e) If E = IF(t,E1) then

(i) For any dissection w1 of E1,

$$v \in IF_1(t,w1) \implies |v|=|w1| + 1$$

$$v \in IF_2(t,w1) \implies |v|=|w1|$$

(ii) There is a dissection w1 of E1 such that $v \in IF(t,w1)$

(d) If E = ELSE(t,E1,E2) then

(i) For any dissections w1, w2 of E1, E2 respectively,

$v \in ELSE_1(t,w1,w2) \implies |v| = |w1|+|w2|+1$

$v \in ELSE_2(t,w1,w2) \implies |v| = |w1|+|w2|$

(ii) There are dissections w1, w2 of E1, E2 respectively such that $v \in ELSE(t,w1,w2)$.

Proof:

The proof is very simple in all cases. We illustrate the general idea by proving (d). Suppose E = CAT(E1,E2); the conclusions of (d)(i) follow directly from the definitions of CAT_1 and CAT_2 (Section 6). For (ii) there are two cases:

Case 1. Every node sequence of v is contained entirely either in E1 or in E2. Hence we can write $v = v1 \cup v2$ where the node-sequences of v1 are entirely from E1 and those of v2 entirely from E2. Setting $w1 = L_{E1}(v1)$ and $w2 = L_{E2}(v2)$ we see that (ii) follows.

Case 2. There is a labelled path p in v which is the concatenation of labelled paths q1 and q2 such that the node-sequence of q1 is entirely in E1 and that of q2 entirely in E2. We can therefore write $v = v1 \cup v2 \cup \{p\}$ where the node-sequences of v1 are entirely in E1 and

those of v2 entirely in E2.

We set $w1 = L_{E1}(v1 \cup \{q1\})$ and

$w2 = L_{E2}(v2 \cup \{q2\})$, and now (ii) follows.

The next six lemmas express the character of a dissection for a composite program graph in terms of the characters of its component dissections (whose existence is ensured by the previous lemma). We need some notation first.

If B is any predicate, let $\bar{B}$ denote the Boolean negation of B. For any $T \subseteq \mathbb{I}^+$, any dissection w and any $g \in \{Z_i^{-1}, DECR^{-1}, DEL_0^{-1}, DEL_\uparrow^{-1}\}$ define:

$$S(\#,T,w,g) = \sum_{X \in g(T)} \#(X,w) \quad \text{and}$$

$$S(f,T,w,g) = \sum_{X \in g(T)} f(X,w).$$

Suppose, for Lemmas (5) through (10), that E is any expression, v is a dissection for E and T is an arbitrary element of $DOM(f_v)$.

<u>LEMMA</u> (5) If E = EXIT(i,n) then

(a) $\#(T,v) = \begin{cases} 1 & \text{if } T = \{0,i,\uparrow\} \\ 0 & \text{otherwise} \end{cases}$

(b) $f_v(T) = \begin{cases} 1 & \text{if } T = \{0,i,\uparrow\} \\ 0 & \text{otherwise} \end{cases}$

<u>Proof</u>:

E has a unique labelled path and its signature is $\{0,i,\uparrow\}$.

<u>LEMMA</u> (6) If E = BREAK(E1,i) for some expression E1 and some i>0, let w be a dissection for E1 such that v ∈ BREAK(w,i) [such a w exists by Lemma (4)(b)]. Then,

(a) $\#(T,v) = S(\#,T,w,Z_i^{-1})$

(b) $f_v(T) = \min\{\ |T|, S(f,T,w,Z_i^{-1})\ \}$

<u>Proof</u>:

(a) The T-paths of v are precisely the X-paths of w such that $Z_i(X) = T$.

(b) Let $h(T) = Z_i^{-1}(T)$.

If $0 \in T$ then $\#(T,v) = 0$ and so $f_v(T) = 0$.

By Lemma 2(a), $h(T) = \phi$ and so $S(\#,T,w,h) = 0$ and hence the given equality holds.

If $0 \notin T$ then by Lemma 2(a), $X \in h(T) \implies |X| \geq |T|$.

So $f_v(T)$

$= f(T,v) = \min\{|T|, \#(T,v)\}$ (by definition)

$= \min\{|T|, S(\#,T,w,h)\}$ (using part (a))

$= \min \{|T|, S(f,T,w,h)\}$ (by Lemma 3(b))

<u>LEMMA</u> (7) If E = LOOP(E1) for some expression E1, let w be a dissection for E1 such that v ∈ LOOP(w) [such a w exists by Lemma (4)(c)]. Then,

(i) If $v \in LOOP_1(w)$ then

(a) $\#(T,v) = S(\#,T,w,DECR^{-1})$

(b) $f_v(T) = \min \{|T|, S(f,T,w,DECR^{-1})\}$

(ii) If $v \in LOOP_2(w)$ then (by def. of $LOOP_2$)

$v = L_E(w-\{p1,p2\}) \cup \{q\}$ where L_E is the labelling relation of the program graph of E, p1 and p2 are respectively finish and start paths of w and q is the labelled path "$L_E(p1).L_E(p2)$"; let

$h(T) = DECR^{-1}(T)$,

B1 be the predicate "sg(q) = T",

B2 be the predicate "sg(p1) ∈ h(T)",

B3 be the predicate "sg(p2) ∈ h(T)",

$$K2 = \begin{cases} 1 & \text{if } B1 \\ 0 & \text{otherwise} \end{cases}$$

$$K3 = \begin{cases} 1 & \text{if } B2 \text{ or } B3 \\ 0 & \text{otherwise} \end{cases}$$

(a) $\#(T,v) = S(\#,T,w,h) + K2 - K3$

$$\text{(b)}\quad f_v(T) = \begin{cases} 0 \quad \text{if } (\bar{B1} \wedge \bar{B2} \wedge B3) \\ \min\{|T|, S(f,T,w,h)\} \text{ if} \\ \quad (B1 \wedge B2 \wedge \bar{B3}) \\ \quad \text{or } (\bar{B1} \wedge \bar{B2} \wedge \bar{B3}) \\ \min\{|T|, S(f,T,w,h)+1\} \\ \quad \text{if } (B1 \wedge \bar{B2} \wedge \bar{B3}) \\ \min\{|T|, S(f,T,w,h)-1\} \\ \quad \text{if } (\bar{B1} \wedge B2 \wedge \bar{B3}) \end{cases}$$

<u>Proof</u>:

(i)(a) The T-paths of v are exactly the X-paths of w such that DECR(X) = T.

(b) By LEMMA 2(b) $|X| \geq |T|$ for all X in h(T).

$f_v(T) = f(T,v) = \min\{|T|, \#(T,v)\}$

$= \min\{|T|, S(\#,T,w,h)\}$ (by part(a))

$= \min\{|T|, S(f,T,w,h)\}$ (by Lemma (3)b)

(ii)(a) Ignoring the labelled paths $L_E(p1)$, $L_E(p2)$ and q for the moment, we can estimate the number of T-paths of v by counting the number of X-paths of w such that DECR(X) = T. Corrections may be necessary to this estimate for two reasons:

-- If q is a T-path of v then we need to add one to this estimate; K2 does this.

-- If either $L_E(p1)$ or $L_E(p2)$ was included in this estimate (both cannot be, since p2 is the start-path) we need to subtract one from this estimate; K3 does this.

(b) The three combinations:

$(B1 \wedge B2 \wedge B3)$, $(B1 \wedge \overline{B2} \wedge B3)$,

and $(\overline{B1} \wedge B2 \wedge B3)$ are not accounted for; but they are impossible as can be seen from:

$B3 \implies \uparrow \in T \quad (\uparrow \in sg(p2))$

$B2 \implies \uparrow \notin T \quad (\uparrow \notin sg(p1)$ by Remark (5))

$B1 \implies \uparrow \notin T \quad (\uparrow \in sg(q)$ iff $\uparrow \in sg(p1))$

Clearly v cannot have a start-path (since $v \in LOOP_2(w)$). Now, if $(\overline{B1} \wedge \overline{B2} \wedge B3)$ then $\uparrow \in T$ and so $\#(T,v) = 0$. Using this fact, the expression for $\#(T,v)$ of part (a) and the definitions of K2 and K3 we get:

$$\#(T,v) = \begin{cases} 0 & \text{if } (\overline{B1} \wedge \overline{B2} \wedge B3) \\ \min\{|T|, S(\#,T,w,h)\} & \text{if } (B1 \wedge B2 \wedge \overline{B3}) \text{ or } (\overline{B1} \wedge \overline{B2} \wedge \overline{B3}) \\ \min\{|T|, S(\#,T,w,h)+1\} & \text{if } (B1 \wedge \overline{B2} \wedge \overline{B3}) \\ \min\{|T|, S(\#,T,w,h)-1\} & \text{if } (B1 \wedge B2 \wedge \overline{B3}) \end{cases}$$

By Lemma 2(b), $X \in DECR^{-1}(T) \implies |X| \geq |T|$. So the given expressions for $f_v(T)$ follow from Lemma 3(b) and the expressions for $\#(T,v)$ derived above in all cases except the last, namely,

$(B1 \wedge B2 \wedge \overline{B3})$.

Now, B2 ==> $sg(p1) \in h(T)$

==> $sg(p1) = (T+1) \cup \{0\}$ (Lemma 2(b)).

Hence $f((T+1) \cup \{0\}, w) \geq 1$. ----------- (*)

So $f_v(T) = f(T,v)$

$= \min \{|T|, \#(T,v)\}$ (by definition)

$= \min \{|T+1|, S(\#,T,w,h)-1\}$ (derived above)

$= \min \{|T+1|, -1 + \#((T+1) \cup \{0\}, w)$

$+ \#(T+1,w)\}$ (Lemma 2(b))

$= \min \{|T+1|, -1 + f((T+1) \cup \{0\}, w)$

$+ f(T+1,w)\}$

(since (*) holds, Lemma 3(c) applies)

$= \min \{|T+1|, -1 + S(f,T,w,h)\}$ (Lemma 2(b))

<u>LEMMA</u> (8) If E = CAT(E1,E2) for some expressions E1 and E2, let w1 and w2 be dissections for E1 and E2 respectively such that $v \in$ CAT(w1,w2) [such w1 and w2 exist by Lemma 3(d)].

Let $h(T) = DEL_0^{-1}(T)$ and $h'(T) = DEL_\uparrow^{-1}(T)$.

(i) If $v \in CAT_1(w1,w2)$ then

(a) $\#(T,v) = S(\#,T,w1,h) + S(\#,T,w2,h')$

(b) $f_v(T) = \min \{ |T|, S(\#,T,w1,h) + S(\#,T,w2,h')$

(ii) If $v \in CAT_2(w1,w2)$ then, (by definition of CAT_2)

$v = L_E((w1-\{p1\}) \cup (w2-\{p2\})) \cup \{q\}$ where

$p1$ is a finish path of $w1$, $p2$ is the start path of $w2$ and q is the path "$L_E(p1).L_E(p2)$".

Let B1 be the predicate "$sg(q) = T$"

B2 be the predicate "$sg(p1) \in h(T)$"

B3 be the predicate "$sg(p2) \in h'(T)$"

$$K1 = \begin{cases} 1 & \text{if } B1 \\ 0 & \text{otherwise} \end{cases}$$

$$K2 = \begin{cases} 1 & \text{if } B2 \\ 0 & \text{otherwise} \end{cases}$$

$$K3 = \begin{cases} 1 & \text{if } B3 \\ 0 & \text{otherwise} \end{cases}$$

(a) $$\#(T,v) = S(\#,T,w1,h) + S(\#,T,w2,h') + K1 - K2 - K3$$

(b) $$f_v(T) = \begin{cases} \min\{|T|,\ S(f,T,w1,h) + f(T,w2)\} \text{ if} \\ \quad (B1 \wedge B2 \wedge B3) \text{ or } (\bar{B1} \wedge \bar{B2} \wedge B3) \\ \min\{|T|,\ S(f,T,w1,h) + S(f,T,w2,h')\} \text{ if} \\ \quad (B1 \wedge B2 \wedge \bar{B3}) \text{ or } (B1 \wedge \bar{B2} \wedge B3) \\ \quad \text{or } (\bar{B1} \wedge \bar{B2} \wedge \bar{B3}) \\ \min\{|T|, S(f,T,w1,h) + S(f,T,w2,h') + 1\} \\ \quad \text{if } (B1 \wedge \bar{B2} \wedge \bar{B3}) \\ \min\{|T|, S(f,T,w1,h) + S(f,T,w2,h') - 1\} \\ \quad \text{if } (\bar{B1} \wedge B2 \wedge \bar{B3}) \end{cases}$$

<u>Proof</u>:

(i)(a) The T-paths of v are exactly

$$\begin{cases} \text{the X-paths of w1 such that } DEL_0(X) = T \text{ and} \\ \text{the X-paths of w2 such that } DEL_{\uparrow}(X) = T. \end{cases}$$

~~(b) By LEMMA 2(c), $X \subseteq h(T) \cup h'(T) \implies |X| \geq |T|$.~~

$$f_v(T) = f(T,v) = \min\{|T|, \#(T,v)\} \quad \text{(def. of f)}$$

$$= \min\{ |T|, S(\#,T,w1,h)+S(\#,T,w2,h') \}$$

(by part (a))

$$= \min\{ |T|, S(f,T,w1,h)+S(f,T,w2,h') \}$$

(by Lemma 3(b))

(ii)(a) We can count the T-paths of v as in (i)(a) above. This count could be inaccurate for three reasons:

-- q has not been counted but should be; K1 corrects for this.

-- p1 has been counted but should not be; K2 corrects for this.

-- p2 has been counted but should not be; K3 corrects for this.

(b) The only combination of truth values that is not accounted for is $(\overline{B1} \wedge B2 \wedge B3)$ but this cannot occur, since,

$B2 \implies sg(p1) \in h(T) \implies sg(p1) - \{0\} = T$

$B3 \implies sg(p2) \in h'(T) \implies sg(p2) - \{\uparrow\} = T$

Now $(B2 \wedge B3) \implies \uparrow \notin T \implies \uparrow \notin sg(p1)$.

So, $$sg(q) = sg(L_E(p1) . L_E(p2)) = (\widehat{sg}(p1) \cup sg(p2)^{-})$$

$= \phi \cup T$

$= T$ and so B1 holds.

Using the expression for #(T,v) derived in (a), the definitions of K1, K2 and K3, and the fact that B2 ==> sg(p1) = $T \cup \{0\}$ ==> #($T \cup \{0\}$, w) = 1, we get the following expressions for #(T,v):

$$\#(T,v) = \begin{cases} S(\#,T,w1,h) + \#(T,w2) & \text{if } (B1 \wedge B2 \wedge B3) \text{ or } (\overline{B1} \wedge \overline{B2} \wedge B3) \\ S(\#,T,w1,h) + S(\#,T,w2,h') & \text{if } (B1 \wedge B2 \wedge \overline{B3}) \text{ or } (B1 \wedge \overline{B2} \wedge B3) \text{ or } (\overline{B1} \wedge \overline{B2} \wedge \overline{B3}) \\ S(\#,T,w1,h) + S(\#,T,w2,h') + 1 & \text{if } (B1 \wedge \overline{B2} \wedge \overline{B3}) \\ S(\#,T,w1,h) + S(\#,T,w2,h') - 1 & \text{if } (\overline{B1} \wedge B2 \wedge \overline{B3}) \end{cases}$$

Now from Lemma 2(c), $X \in h(T) \cup h'(T)$ ==> $|X| \geq |T|$ and so the given expressions for $f_v(T)$ follow from Lemma 3(b) in all cases except the last, viz. ($\overline{B1} \wedge B2 \wedge \overline{B3}$); in this case, B2 ==> sg(p1) − {0} = T and so $0 \notin T$ and so f($T \cup \{0\}$, w1) $\geq$ 1 (since sg(p1) = $T \cup \{0\}$). Furthermore, we can assume that $\uparrow \notin T$ since

$\uparrow \in T$

$\Longrightarrow S(\#,T,w1,h) = 1$ and $S(\#,T,w2,h') = 0$

since $(B2 \wedge \overline{B1})$ holds.

$\Longrightarrow S(f,T,w1,h) = 1$ and $S(f,T,w2,h') = 0$

$$\begin{aligned}\Longrightarrow f_v(T) &= \min\{|T|, \#(T,v)\} \\ &= \min\{|T|, S(\#,T,w1,h) \\ &\qquad + S(\#,T,w2,h') - 1\} \\ &= \min\{|T|, 0\} \\ &= \min\{|T|, S(f,T,w1,h) \\ &\qquad + S(f,T,w2,h') - 1\}\end{aligned}$$

We can now apply Lemma 3(c) to get the result since $0 \notin T$, $\uparrow \notin T$, and $f(T \cup \{0\}, w1) \geq 1$.

<u>LEMMA</u> (9) If $E = IF(t,E1)$ for some expression E1, let w be a dissection for E1 such that $v \in IF(t,w)$ [such a w exists by LEMMA 4(e)]. Let $h(T) = DEL_{\uparrow}^{-1}(T)$ and q1 denote the single-node labelled path $(\{t\}, \{0,\uparrow\})$ in E.

(i) If $v \in IF_1(t,w)$ then let $K1 = \begin{cases} 1 \text{ if } T = \{\uparrow, j\} \\ 0 \text{ otherwise.} \end{cases}$

(a) $\#(T,v) = S(\#,T,w,h) + K1$

(b) $f_v(T) = \min\{|T|, S(f,T,w,h) + K1\}$

(ii) If $v \in IF_2(t,w)$ then (by definition of IF_2)

$v = L_E(w-\{p1\}) \cup \{q1.L_E(p1)\}$ where

p1 is the start-path of w.

Let q denote the path "$q1.L_E(p1)$" and let

$h(T) = DEL_{\uparrow}^{-1}(T)$

B1 be the predicate "$sg(q) = T$"

B2 be the predicate "$sg(p1) \in h(T)$"

$$K2 = \begin{cases} 1 & \text{if } B1 \\ 0 & \text{otherwise} \end{cases}$$

$$K3 = \begin{cases} 1 & \text{if } B2 \\ 0 & \text{otherwise} \end{cases}$$

(a) $\#(T,v) = S(\#,T,w,h) + K2 - K3$

(b) $$f_v(T) = \begin{cases} \min\{|T|, S(\#,T,w,h)\} & \text{if } (\bar{B1} \wedge \bar{B2}) \\ \min\{|T|, S(\#,T,w,h) + 1\} & \text{if } (B1 \wedge \bar{B2}) \\ \min\{|T|, f(T,w)\} & \text{if } (\bar{B1} \wedge B2) \end{cases}$$

<u>Proof</u>:

(i)(a) We can estimate the number of T-paths of v by counting the number of X-paths of w such that $DEL_{\uparrow}(X)=T$. This estimate could be inaccurate if q1 is also a T-path of v; K1 corrects for this.

(b) By Lemma 2(c), $X \in h(T) \implies |X| \geq |T|$ and so Lemma 3(b) can be applied as in the proofs of the preceding Lemmas.

(ii)(a) We can estimate the number of T-paths of v by counting the X-paths of w such that $DEL_{\uparrow}(X) = T$. This estimate may be inaccurate for two reasons:

-- q has not been counted but should be; K2 corrects for this.

-- p1 has been counted but should not be; K3 corrects for this.

(b) ~~The combination of truth values for B1 and B2~~ that is not accounted for is (B1 $\wedge$ B2), but this case is impossible since,

$B1 \implies sg(q) = T \implies \uparrow \in T$

$B2 \implies sg(p1) \in h(T) \implies \uparrow \notin T$

We note also that $B2 \implies sg(p1) = T \cup \{\uparrow\}$

$\implies S(\#,T,w,h) = \#(T,w) + 1$ (by Lemma 2(c)).

Using this fact, the definitions of S(#,T,w,h), K2 and K3, the expression for #(T,v) derived in part (a), and Lemma 3(b) we can get the given expressions for $f_v(T)$.

<u>LEMMA</u> (10) If E = ELSE(t,E1,E2) for some expressions E1 and E2, let w1 and w2 be dissections for E1 and E2 respectively [such w1 and w2 exist by Lemma 4(f)]. Let q1 denote the single-node labelled path $(t,\{\uparrow\})$ in E and let $h(T) = DEL_{\uparrow}^{-1}(T)$.

(i) If $v \in ELSE_1(t,w1,w2)$ then let $K1 = \begin{cases} 1 & \text{if } T=\{\uparrow\} \\ 0 & \text{otherwise} \end{cases}$

(a) $\#(T,v) = S(\#,T,w1,h) + S(\#,T,w2,h) + K1$

(b) $f_v(T) = \min\{|T|, S(\#,T,w1,h) + S(\#,T,w2,h) + K1\}$

(ii) If $v \in ELSE_2(t,w1,w2)$ then by definition of $ELSE_2$,

<u>either</u>

$v = L_E((w1-\{p1\}) \cup w2) \cup \{q1.L_E(p1)\}$ where $p1$ is the start-path of $w1$.

<u>or</u>

$v = L_E(w1 \cup (w2-\{p2\})) \cup \{q1.L_E(p2)\}$ where $p2$ is the start-path of $w2$.

We will consider only the former alternative; the latter follows by symmetry. Let q denote the path "$q1.L_E(p1)$" and let

B1 be the predicate "sg(q) = T"

B2 be the predicate "sg(p1) $\in$ h(T)"

$$K2 = \begin{cases} 1 & \text{if } B1 \\ 0 & \text{otherwise} \end{cases}$$

$$K3 = \begin{cases} 1 & \text{if } B2 \\ 0 & \text{otherwise} \end{cases}$$

(a) $\#(T,v) = S(\#,T,w1,h) + S(\#,T,w2,h) + K2 - K3$

(b) $$f_v(T) = \begin{cases} \min\{|T|,\ S(f,T,w1,h) + S(f,T,w2,h)\} & \text{if } (\overline{B1} \wedge \overline{B2}) \\ \min\{|T|,\ S(f,T,w1,h)+S(f,T,w2,h)+1\} & \text{if } (B1 \wedge \overline{B2}) \\ \min\{|T|,\ S(f,T,w2,h) + f(T,w1)\} & \text{if } (\overline{B1} \wedge B2) \end{cases}$$

<u>Proof</u>:

(i)(a) We can estimate the number of T-paths of v by counting the X-paths of w1 and w2 such that

$DEL_{\uparrow}(X) = T$. This estimate could be inaccurate if q1 is a T-path; K1 corrects for this.

~~(b) By Lemma 2(c), $X \in h(T) \implies |X| \geq |T|$ and so~~ Lemma 3(b) can be applied as in the proofs of the previous Lemmas.

(ii) (a) We can estimate the number of T-paths of v by counting the X-paths of w1 and w2 such that $DEL_{\uparrow}(X) = T$. This estimate may be inaccurate for two reasons:

-- q has not been counted and should be; K2 corrects for this.

-- p1 has been counted but should not be; K3 corrects for this.

(b) The truth value combination that is not accounted for is (B1 $\wedge$ B2), but this is impossible since,

B1 ==> sg(q) = T ==> $\uparrow \in T$

B2 ==> sg(p1) $\in$ h(T) ==> $\uparrow \notin T$

We also note that

B2 ==> sg(p1) = T $\cup$ $\{\uparrow\}$

==> $\#(T^+, w1) = 1$

==> $S(\#, T, w1, h) = \#(T, w1) + 1$

(by Lemma 2(c)).

Using this fact, the result of part (a), the definitions of K2 and K3, and Lemma 3(b) we can get the given expressions for $f_v(T)$.

The following Lemma is central to the proof of Theorem 1 and is proved using the previous six Lemmas (5-10).

<u>LEMMA</u> (11) Suppose E, E1 and E2 are expressions, v1 and w1 are dissections for E1, and v2 and w2 are dissections for E2.

(a) If E = BREAK(E1,i) and v1 ~ w1 then

$w \in$ BREAK(w1,i) ==> v ~ w for some $v \in$ BREAK(v1,i)

(b) If E = LOOP(E1) and v1 ~ w1 then

$w \in LOOP_1(w1)$ ==> v ~ w for some $v \in LOOP_1(v1)$

$w \in LOOP_2(w1)$ ==> v ~ w for some $v \in LOOP_2(v1)$

(c) If E = CAT(E1,E2), v1 ~ w1 and v2 ~ w2, then

$w \in CAT_1(w1,w2)$ ==> v ~ w for some $v \in CAT_1(v1,v2)$

$w \in CAT_2(w1,w2)$ ==> v ~ w for some $v \in CAT_2(v1,v2)$

(d) If E = IF(t,E1) and v1 ~ w1, then

$w \in IF_1(t,w1)$ ==> v ~ w for some $v \in IF_1(t,v1)$

$w \in IF_2(t,w1)$ ==> v ~ w for some $v \in IF_2(t,v1)$

(e) If E = ELSE(E1,E2), v1 ~ w1 and v2 ~ w2, then

$w \in ELSE_1(t,w1,w2)$ ==>

v ~ w for some $v \in ELSE_1(t,v1,v2)$

$w \in ELSE_2(t,w1,w2)$ ==>

v ~ w for some $v \in ELSE_2(t,v1,v2)$

<u>Proof</u>:

We illustrate the idea of the proof by proving (b) and (c); the other cases are similar.

(b) Suppose E = LOOP(E1), v1 ~ w1 and $w \in$ LOOP(w1).

Let h = $DECR^{-1}$.

Case 1 w ∈ $LOOP_1$(w1). In this case, let v be the unique element of $LOOP_1$(v1).

For any non-void T ⊆ $\mathbb{I}^+$, we have

$$f(T,v) = \min\{|T|, S(f,T,v1,h)\} \text{ by Lemma 7(i)(b)}$$
$$= \min\{|T|, S(f,T,w1,h)\} \text{ since } v1 \sim w1$$
$$= f(T,w) \text{ by Lemma 7(i)(b).}$$

Case 2 w ∈ $LOOP_2$(w1). In this case, w must be of the form w = L_E(w1−{p1,p2}) ∪ {q} where p1 is a finish path of w1, p2 is the start path of w1 and q = L_E(p1).L_E(p2).

Let v = L_E(v1−{p1´, p2´}) ∪ {q´} where p1´ is a finish path of v1 such that sg(p1´) = sg(p1), p2´ is the start path of v1 (and sg(p2´) = sg(p2)) and q´ = L_E(p1´).L_E(p2´). It is possible to choose such p1´ and p2´ since v1 ~ w1. It is easy to see that v ∈ $LOOP_2$(v1), sg(q) = sg(q´), and that S(f,T,v1,h) = S(f,T,w1,h). It now follows directly from Lemma 7(ii)(b) that f(T,v) = f(T,w) and so v ~ w.

(c) Suppose E = CAT(E1,E2), v1 ~ w1, and v2 ~ w2.

Let h = DEL_0^{-1} and h´ = $DEL_{\uparrow}^{-1}$.

Case 1 w ∈ CAT_1(w1,w2). In this case, let v be the unique element of CAT_1(v1,v2). Now, for any non-void T ⊆ $\mathbb{I}^+$, we have,

$f(T,v) = \min\{\ |T|,\ S(f,T,v1,h) + S(f,T,v2,h')\}$

by Lemma 8(i)(b)

$= \min\{\ |T|,\ S(f,T,w1,h) + S(f,t,w2,h')\}$

since $v1 \sim w1$ and $v2 \sim w2$

$= f(T,w)$ by Lemma 8(i)(b).

Case 2 $w \in CAT_2(w1,w2)$. In this case, w must be of the form $w = L_E((w1 - \{p1\}) \cup (w2 - \{p2\})) \cup \{q\}$ where p1 is a finish path of w1, p2 is the start path of w2, and $q = L_E(p1).L_E(p2)$.

Let $v = L_E((v1 - \{p1'\}) \cup (v2 - \{p2'\})) \cup \{q'\}$ where p1′ is a finish path of v1 such that sg(p1) = sg(p1′), p2′ is the start path of v2 (and sg(p2′) = sg(p2)) and $q' = L_E(p1').L_E(p2')$. It is easy to see that $v \in CAT_2(v1,v2)$, sg(q) = sg(q′), S(f,T,v1,h) = S(f,T,w1,h) and that S(f,T,v2,h′) = S(f,T,w2,h′) since $v1 \sim w1$ and $v2 \sim w2$. It now follows directly from Lemma 8(ii)(b) that f(T,v) = f(T,w) and so $v \sim w$.

We now develop some results necessary for the proof of Theorem 2. For any non-void $T \in \mathbb{I}^+$, let

$$\max(T) = \begin{cases} 0 \text{ if } T = \{\uparrow\} \\ \text{the largest integer in } T \text{ otherwise.} \end{cases}$$

For any $r \in \mathbb{I}$, let

$$Q(r) = \{T \subseteq \mathbb{I}^+ \mid \max(T) \le r \text{ and } |T^-| \le 2\}.$$

LEMMA (12) For any $r \in \mathbb{I}$, $|Q(r)| = r^2+3r+3$

Proof:

A simple combinatorial argument shows that

$$|Q(r)| = 2[\binom{r+1}{1} + \binom{r+1}{2}] + 1$$

LEMMA (13) Let E be any expression of rank r. Then

$$|\{f_w \mid w \text{ is a dissection of } E\}| = 3^{O(r^2)}.$$

Proof:

From Lemma 1(b) it follows that

$$f_w(T) \neq 0 \implies \max(T) \leq r \text{ and } |T^-| \leq 2.$$

Since $0 \leq f_w(T) \leq 2$ for all T and w, it follows that the maximum number of character functions possible is $3^{|Q(r)|}$ and now Lemma 12 yields the desired result.

2. PROOFS OF THEOREMS 1 AND 2

The next two Lemmas refer to the algorithm of Chapter 2.

LEMMA (14) After the set Y has been created from X at the end of DISSECT, the following property holds:

$$v \in X \implies w \sim v \text{ and } |w| \leq |v| \text{ for some } w \in Y.$$

Proof:

Let C be the equivalence class of $v \in X$. Then some $w = \text{MIN}(C)$ must have been chosen for inclusion in Y; this w has the required properties.

LEMMA (15) If Y is the output of DISSECT and r is the rank of the input expression, then

(a) Distinct elements of Y have distinct characters.

(b) $|Y| = 3^{O(r^2)}$.

Proof:

(a) To form Y we choose exactly one element from each equivalence class of X.

(b) This follows from (a) and Lemma 13.

Proof of Theorem 1:

The statement of the Theorem is a simple consequence of the following assertion:

"If Y is the output of DISSECT(E,Y) and w is a dissection for E, then $v \sim w$ and $|v| \leq |w|$ for some $v \in Y$."

We now prove this assertion. Suppose Y is the output of DISSECT(E,Y) and w is a dissection for E. Then,

either

E = EXIT(i,n) in which case $w \in X$ and the result follows from Lemma 14.

or

E = OP(i,t,E1,E2) for some expressions E1, E2 and some OP $\in$ {BREAK, LOOP, IF, ELSE, CAT}.

In this case, by LEMMA 4, there are dissections w1 and w2 for E1 and E2 respectively such that $w \in$

OP(i,t,w1,w2). Assuming inductively that the assertion holds for the smaller expressions E1 and E2 we see that there are dissections v1 and v2 for E1 and E2 respectively such that:

$v1 \sim w1,\ v1 \in Y1,\ |v1| \leq |w1|,$

$v2 \sim w2,\ v2 \in Y2,\ |v2| \leq |w2|.$

Using Lemmas 4 and 11 we see that

$|v'| \leq |w|$ and $v' \sim w$ for some $v' \in OP(i,t,v1,v2)$.

From the way X was created we see that $v' \in X$. The conclusion of the assertion now follows from Lemma 14.

<u>Proof of Theorem 2</u>.

The cost of a single call to DISSECT is (exclusive of recursion):

Cost to compute X + Cost to compute Y from X.

Since $|Y1| = |Y2| = 3^{O(r^2)}$ from Lemma 15, it follows that the cost to compute X (from Y1 and Y2) is:

$$3^{O(r^2)} \cdot 3^{O(r^2)} \cdot O(n) = 3^{O(r^2)}\ n$$

Therefore, the outer loop (line (1) of Create_Y) in the procedure Create_Y is executed $3^{O(r^2)}$ n times; the inner loop (line (2)) is executed $3^{O(r^2)}$ times (Lemma 15(b), once for each element of Y) and each iteration involves one call to the function Compare_f (line (3)). Each such call costs $O(r^2)$ by Lemma 12. Hence the cost of computing Y from X is the product of these three quantities which is still $3^{O(r^2)}$ n. Since there can be at most n calls to DISSECT, the total running-time of the algorithm is as asserted by the theorem.

CHAPTER 4

THE POWER OF MULTILEVEL EXITS

1. OVERVIEW

In this chapter we show that the class of flowgraphs that can be produced using the SPG-operations is identical to the class of reducible flowgraphs. We show further, that (i+1)-level exits are strictly more powerful than i-level exits in the following sense: There are digraphs for which no program representation using at most i-level exits is possible but a program representation exists if (i+1)-level exits are permitted.

2. THE CLASS OF STRUCTURED FLOW GRAPHS

A *flowgraph* has often been defined in the literature to be a triple (N,A,s) where (N,A) is a digraph and $s \in N$ is a distinguished node called the *start node* with the property that every node in N is reachable from s. An SPG, as we have defined it, may contain nodes that are not reachable from the start node; consider, for example, the SPG of the expression

CAT(LOOP(CAT(BREAK(EXIT(0,A),1),EXIT(0,B))),EXIT(0,C))

The corresponding program is:

```
LOOP
  A;
  EXIT 1;
  B
ENDLOOP;
C
```

The node B is not reachable from the start node A. We can, however, obtain from an SPG G = (N,A,L) a flowgraph $G' = (N',A',s)$ as follows:

(i) $s = s(G)$

(ii) $N' = N - \{\text{all nodes not reachable from s}\}$

(iii) $A' = A \cap (N' \times N')$

G' is called a <u>Structured Flow Graph</u> (<u>SFG</u>).

We now review a few characterizations of Flowgraph Reducibility. If h is a node of a flowgraph F = (N,A,s), we define <u>the interval I(h) with header h</u> [7] as the set of nodes constructed as follows:

(1) h is in I(h).

(2) If $n \in N - \{s\}$ is not in I(h) but all the predecessors of n are in I(h), add n to I(h).

(3) Repeat (2) until I(h) is stable.

Every flowgraph F can be uniquely partitioned into disjoint intervals [7,8]. From such a partition we can create another flowgraph I(F) called the <u>derived flowgraph of F</u> by collapsing each interval into a single node. The sequence: $F=I^0(F)$, $I^1(F)$, ..., $I^m(F)$ where

$I^{m+1}(F) = I^m(F)$ is called the derived sequence of F and $I^m(F)$ is called the limit graph of F. The flowgraph F is said to be reducible iff its limit graph has exactly one node. Suppose G = (N,A) is a digraph and C = (a_0, a_1, ..., a_{n-1}) is a sequence of nodes of G. Denote addition mod n by $\oplus$. We say that C is a

path iff $0 \leq i < n-1 \implies (a_i, a_{i+1})$ is an arc of G.

cycle iff $0 \leq i < n \implies (a_i, a_{i\oplus 1})$ is an arc of G.

A node ´a´ of a flowgraph F = (N,A,s) is said to dominate node ´b´ iff every path from s to ´b´ contains ´a´. A simple cycle [simple path] is a cycle [path] all of whose nodes are distinct. A node n in a subgraph C of a flowgraph F = (N,A,s) is called an entry node of C iff there is a path p in F from s to n such that $p \cap C = \{n\}$. We call n an open node of C iff (n=s) or ($(x,n) \in A$ for some node x not in C). Every entry node of C must be an open node of C but not necessarily conversely. C is called multi-entry iff it has more than one entry node. A subgraph H of a digraph G is called strongly connected iff for every pair of nodes $a,b \in H$, there is a path in H from a to b. H is called a strongly connected component (SCC) of G iff it is strongly connected and is not properly contained in any other strongly connected subgraph of G. Any digraph G can be uniquely partitioned into strongly connected components. An SCC is called trivial iff it has no arcs and has exactly one node. A digraph G is said to have the unique-open-node property iff every SCC of G has a unique open node. We now define operations T1-T4 on any digraph G; T2 is applicable only if G is a flowgraph.

T1:

For each node n of G, remove the arc (n,n) if it exists in G [9].

T2:

For each node n of G, if n is not the start node of G and (m,n) is the unique incoming arc of n, then create new arcs out of m such that every successor of n is a successor of m. Then remove n [9].

T3:

For each SCC H of G, if H has a unique open node n then delete all in-arcs of n [10].

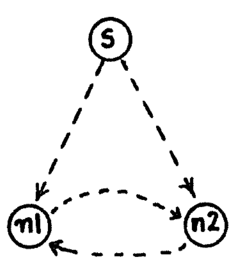
T4:

Same as T3, but only those in-arcs of n that originate within H are deleted.

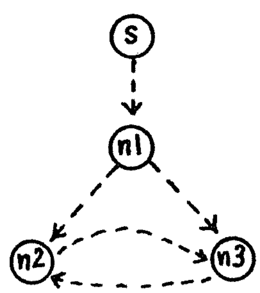
If G is a digraph, let $T3^{i}(G)$ denote the digraph obtained from G by i applications of T3; $T4^{i}(G)$ is defined similarly. Let $k = \min\{i \mid T4^{i}(G) = T4^{i+1}(G)\}$. The digraph $T4^{k}(G)$ is called the *core* of G. The notion of *Depth First Search* (*DFS*) may be found in [11,12]. Every DFS of a digraph determines a set of *back arcs*, whose deletion renders the digraph acyclic.

A flowgraph (N,A,s) is said to *contain a forbidden subgraph* iff it has a subgraph of the form (a) or (b) below where s, n1, n2, and n3 are all distinct nodes and the dotted lines are disjoint paths (i.e. any pair of paths have no nodes in common except possibly the endpoints).

(a)



(b)



<u>THEOREM</u> <u>3</u> For a flowgraph F = (N,A,s), the following are equivalent:

(a) (REDUCIBLE) [7] F is reducible.

(b) (COLLAPSIBLE) [9] Repeated applications of T1 and T2 eventually yield a single node.

(c) (ARRANGEABLE) [13] There is a total order '<' of N such that $s=a_0, a_1, \ldots, a_n$ is a simple path in F ==> $a_i < a_{i+1}$ for $0 \leq i \leq n-1$.

(d) (SINGLE-ENTRY) [13] Every strongly connected subgraph of F has a unique entry node.

(e) Every simple cycle of F has a unique entry node.

(f) (WELL-FORMED) [10] For all $i \geq 0$, $T3^i(F)$ has the unique-open-node property.

(g) For all $i \geq 0$, $T4^i(F)$ has the unique-open-node property.

(h) F has an acyclic core.

(i) F has a DFS starting at s such that the terminal node of every back arc dominates the initial node.

(j) (UNIQUE DAG) [9] Any two DFS's starting at s yield the same set of back arcs.

(k) F does not contain a forbidden subgraph.

Some preliminary results are necessary for the proof of this theorem.

REMARK (10) For an arbitrary digraph G, and $i \geq 0$

(a) If H is an SCC of $T4^i(G)$ [respectively $T3^i(G)$] and x is a unique open node of H, then $\{x\}$ is a trivial SCC of $T4^{i+1}(G)$ [$T3^i(G)$].

(b) Every SCC of $T4^{i+1}(G)$ [$T3^{i+1}(G)$] is contained in some SCC of $T4^i(G)$ [$T3^i(G)$].

(c) If H is an SCC of $T4^i(G)$ [$T3^i(G)$] with no open nodes then, for all $j \geq i$, H is an SCC of $T4^j(G)$ [$T3^j(G)$] with no open nodes.

(d) If H is an SCC of $T4^i(G)$ [$T3^i(G)$] with two or more open nodes then, for all $j \geq i$, H is an SCC of $T4^j(G)$ [$T3^j(G)$] with two or more open nodes.

(e) H is an SCC of $T4^i(G)$ [$T3^i(G)$] and
H is not an SCC of $T4^{i+1}(G)$ [$T3^{i+1}(G)$] iff
H is a non-trivial SCC of $T4^i(G)$ [$T3^i(G)$] and
H has a unique open node.

LEMMA (16) For any flowgraph F and any $i \geq 0$, every non-trivial SCC of $T4^i(F)$ [$T3^i(F)$] has at least one open node.

Proof:

Since $T4^0(F) = T3^0(F) = F$ the statement is true for $i=0$. Assume it holds for $i \leq k$ for some k and let H be a non-trivial SCC of $T4^{k+1}(F)$ [$T3^{k+1}(F)$].

<u>Case 1</u>

If H is a SCC of $T4^k(F)$ [$T3^k(F)$], by the induction hypothesis it has at least one open node in $T4^k(F)$ [$T3^k(F)$]. Now, H cannot have a unique open node in $T4^k(F)$ [$T3^k(F)$] by Remark (10e). So H has two or more open nodes in $T4^k(F)$ [$T3^k(F)$]. By Remark (10d) H has two or more open nodes in $T4^{k+1}(F)$ [$T3^{k+1}(F)$].

<u>Case 2</u>

If H is not an SCC of $T4^k(F)$ [$T3^k(F)$] then, by Remark (10b) $H \subseteq J$, $H \neq J$ for some SCC J of $T4^k(F)$ [$T3^k(F)$]. Also J must be non-trivial since H is so. Now,

-- J must have a unique open node, say x, in $T4^k(F)$ [$T3^k(F)$] by Remark (10e).

-- H cannot contain x, by Remark (10a).

-- Since J is an SCC, every element of H is reachable in J from x.

-- Hence H must have at least one open node in $T4^{k+1}(F)$ [$T3^{k+1}(F)$], since T4 [T3] affects only in-arcs of x.

<u>LEMMA</u> (17) Suppose F is a flowgraph and $i \geq 0$. If H is a non-trivial SCC of $T4^k(F)$ [$T3^k(F)$] and x is a unique open node of H then, H is a non-trivial SCC of $T3^k(F)$ [$T4^k(F)$] and x is a unique open node of H.

Proof:

We proceed by induction. The result is trivial for $i = 0$ since $T4^0(F) = T3^0(F) = F$. Assume the result holds for all $i \leq k$. Let H be a non-trivial SCC of $T4^{k+1}(F)$ [$T3^{k+1}(F)$] and let x be a unique open node of H.

-- H cannot be an SCC of $T4^k(F)$ [$T3^k(F)$] (Remarks 10(c), 10(d) and 10(e)).

-- So, $H \subseteq J$, $H \neq J$ for some non-trivial SCC J of $T4^k(F)$ [$T3^k(F)$] (Remark 10(b)).

-- J has a unique open node, say y, in $T4^k(F)$ [$T3^k(F)$] (Remark 10(e)).

-- by the induction hypothesis J is a non-trivial SCC of $T3^k(F)$ [$T4^k(F)$] and y is a unique open node of J in $T3^k(F)$ [$T4^k(F)$].

-- H cannot contain y (Remark 10(a)).

-- x is a unique open node of H in J since it is a unique open node of H in $T4^{k+1}(F)$ [$T3^{k+1}(F)$] and T4 does not affect arcs to H from outside.

-- Hence x is a unique open node of H in $T3^{k+1}(F)$ [$T4^{k+1}(F)$].

-- H is a non-trivial SCC of $T3^{k+1}(F)$ [$T4^{k+1}(F)$] since T3 does not affect arcs of H and H is a maximal strongly connected subgraph of J.

LEMMA (18) If F is a flowgraph and $i \geq 0$ then $T4^i(F)$ is a flowgraph with the same start node as F.

Proof:

Since T4 only removes the in-arcs of the unique open node of an SCC that originate inside the SCC, reachability from the start node of F is unaffected.

Proof of Theorem 3

(a) <==> (b) <==> (i) <==> (j) <==> (k) is shown in [9].

(a) <==> (c) <==> (d) is shown in [13].

(e) <==> (k) is easy to see.

(f) is mentioned in [10] without any mention of the notion of reducibility.

We will show that (f), (h) and (k) are all equivalent to (g).

(g) ==> (h)

Suppose the core of F has a cycle C. Let H be the SCC of the core of F which contains C. H must be non-trivial since there is at least one arc in C. By Lemma (16), H has at least one open node. By definition of the core, it cannnot have a unique open node. Hence it has two or more open nodes and so (g) fails.

(h) ==> (g)

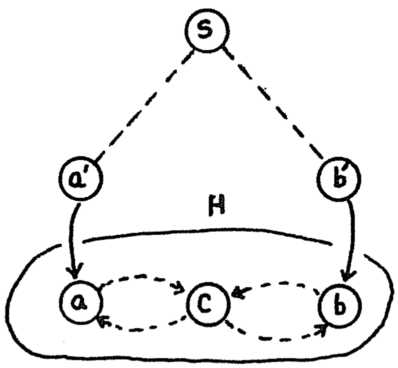
Suppose $T4^i(F)$ fails to have the unique-open-node property. Let H be an SCC of $T4^i(F)$ with two open nodes. By Remark 10(d), H is a non-trivial SCC of the core of F and so (h) fails.

(g) <==> (f)

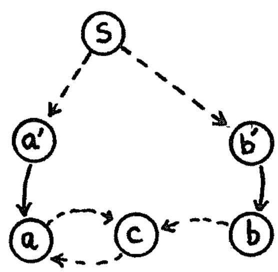
~~This follows directly from Lemma (17).~~

(k) ==> (g)

Suppose for some $i \geq 0$, $T4^i(F)$ fails to have the unique-open-node property. Let H be an SCC of $T4^i(F)$ with two open nodes a,b, $a \neq b$. Let (a´,a) and (b´,b) be arcs of $T4^i(F)$ with a´, b´ $\notin$ H.

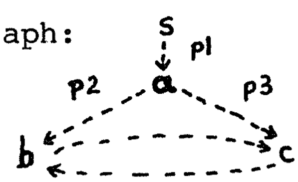


By Lemma (18), there are paths p1 and p2 from s to a´ and b´ respectively. No element of H can be in p1 or p2 (otherwise we would have either a´ $\in$ H or b´ $\in$ H). Since H is strongly connected, there are paths q1 and q2 from a to b and b to a respectively. Let c be the first node of q1 other than a that occurs in q2. Now F has a forbidden subgraph:



(g) ==> (k)

Suppose F has a forbidden digraph:



Assume that (g) holds. For all $i \geq 0$, $T4^i(F)$ must contain all the arcs of p1, p2 and p3; this can be seen as follows:

> Suppose (x,y) is the first arc of "p1.p2" that is deleted and this happens when T4 is applied to $T4^j(F)$. This means that y is the unique open node of some SCC of H of $T4^j(F)$ which also contains x. Since there is a path from s to x in $T4^j(F)$ which avoids y (viz. an initial segment of p1.p2), it follows that H has an open node other than y. This is a contradiction of the assumption that (g) holds. A similar argument may be used for tha path "p1.p3" .

Consider the first time an element of p_{bc} or p_{cb} becomes an open node of an SCC H of some $T4^i(F)$ (if no element ever does, the cycle "$p_{bc}.p_{cb}$" is in the core of F and so (h), and hence (g), are violated).

Now H cannot contain any element of p1 (otherwise an element of p1 would be another open node of H). Let a´ and b´ be the first elements respectively in paths p1 and p3 which are in H. They must both be open nodes of H and so (g) is violated. This completes the proof of Theorem 3.

We will now show that every structured flowgraph is reducible.

THEOREM 4 Every SFG is an RFG .

Proof:

Let MESC stand for "Multi-entry Simple Cycle." We will show that no SPG has a MESC. The result then follows from Theorem 3(e). We proceed by induction on the size of the expression. Clearly, if we have an expression of size 1, it cannot have a MESC since only one node exists in the flowgraph. Assume that the program graphs of all expressions of size at most n have no MESCs. Let E be an expression of size n+1. We will show that the program graph of E has no MESCs. There are five cases to consider, one per SPG-operation.

Case 1

Suppose E = BREAK(E1,i) for some expression E1 and some i $\geq$ 1. By the induction hypothesis, E1 has no MESCs since it is a smaller expression than E. Since E has the same nodes, arcs and start node as E1, it follows that E has no MESCs either.

Case 2

Suppose E = LOOP(E1) for some expression E1. As before, E1 has no MESCs. Hence, any MESC of E must use a newly created arc of the form (x, s(E)). But by the definition of an entry node, any subgraph containing the start node has a unique entry node.

Case 3

Suppose E = CAT(E1,E2) for some expressions E1 and E2. As before, neither E1 nor E2 can have any MESCs. Any simple cycle of E would have to be contained entirely in E1 or entirely in E2. If E had a MESC, it is easy to see that it must either be a MESC of E1 or a MESC of E2 since all paths from the start node of E to any node of E2 must pass through s(E2).

Case 4

Suppose E = IF(t,E1) for some expression E1 and some node t. As before E1 cannot have any MESCs. Since s(E1) dominates all nodes of E1 in E, we see that any MESC of E must also be a MESC of E1.

Case 5

Suppose E = ELSE(T,E1,E2) for some expressions E1 and E2 and some node t. As in the previous cases it is easy to show that any MESC of E must be either a MESC of E1 or a MESC of E2.

This completes the proof of Theorem 4.

We will now show that every RFG is an SFG. An algorithm for producing a structured program (using infinite loops with multilevel exits and IF-THEN-ELSE statements) from a well-formed flowchart is presented by Kasami et.

al. in [10]. Their model of a flowchart, however, is different from ours since they associate primitive actions with *arcs* rather than *nodes*. So, rather than modify their algorithm to accomodate our model, we present an alternative algorithm based on the concepts of [14]. Our algorithm takes any RFG and produces a program for it using loops with multilevel conditional and unconditional exits. We first review a few definitions from [14].

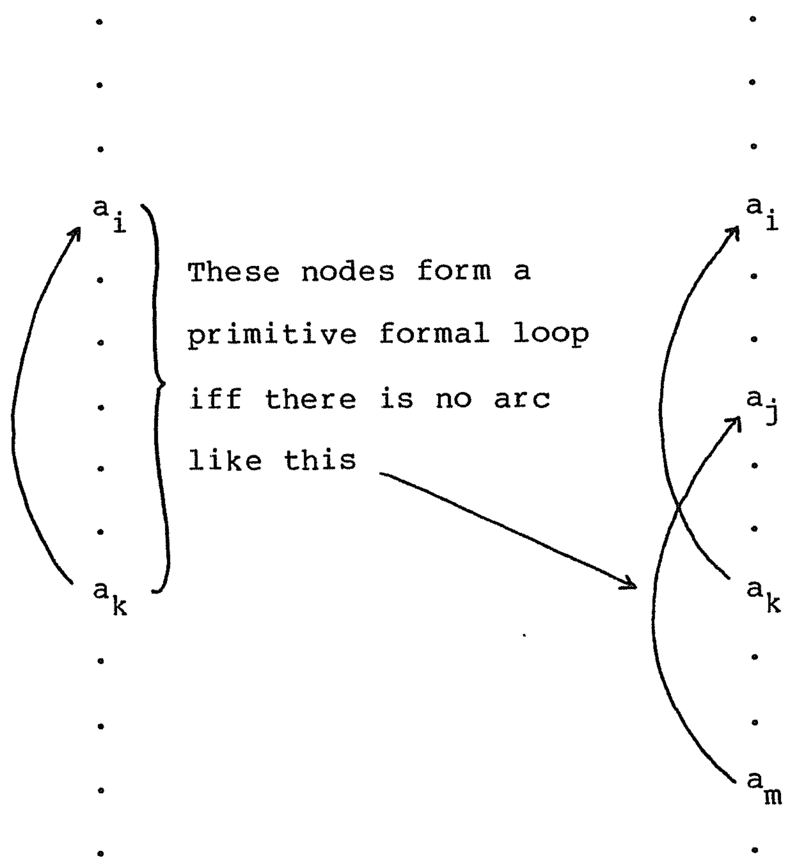
Suppose F = (N,A,s) is a flowgraph and the nodes of F are arranged in an arbitrary but fixed linear order: $a_1, a_2, \ldots a_n$. For $i \leq j$, let $[a_i, a_j] = \{ a_k \mid i \leq k \leq j \}$. Any arc (a_i, a_j) where $i < j$ is called a *forward arc*; any path that uses only forward arcs is called a *forward path*. An arc (a_i, a_j) where $i \geq j$ is called a *reverse arc*. For $i \leq k$, $[a_i, a_k]$ is called a *primitive formal loop* iff

(i) (a_k, a_i) is an arc of F and

(ii) there is no arc (a_m, a_j) such that

$i < j \leq k < m$.

The following figure illustrates this definition.



The set of nodes $[a_h, a_k]$ is called a <u>formal</u> <u>loop</u> iff

(a) it is a primitive formal loop or

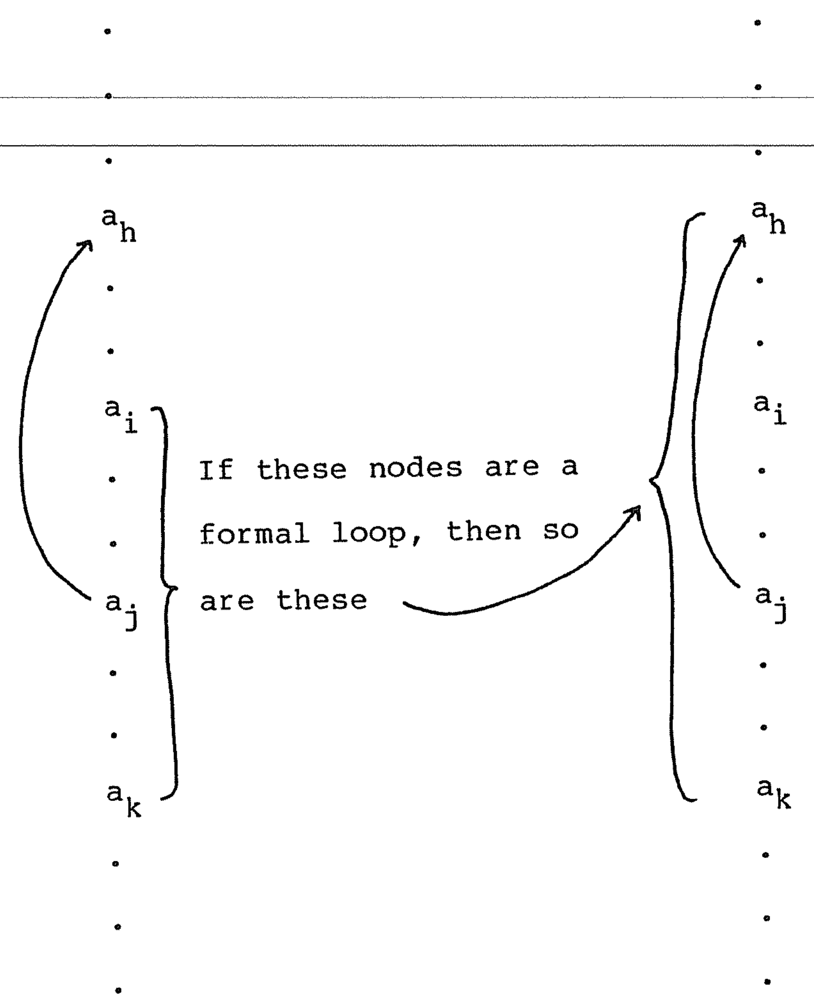
(b) there are indices i and j such that

(i) $[a_i, a_k]$ is a formal loop and

(ii) (a_j, a_h) is an arc of F and

(iii) $h < i \leq j < k$

The following figure illustrates this definition.



A <u>maximal formal loop</u> is a formal loop that is not properly contained in any other formal loop. It is easy to see that if (a_i, a_j) is a reverse arc then $[a_j, a_i]$ is contained in some maximal formal loop and that any two maximal formal loops are disjoint.

A linear order is called a <u>straight order</u> iff

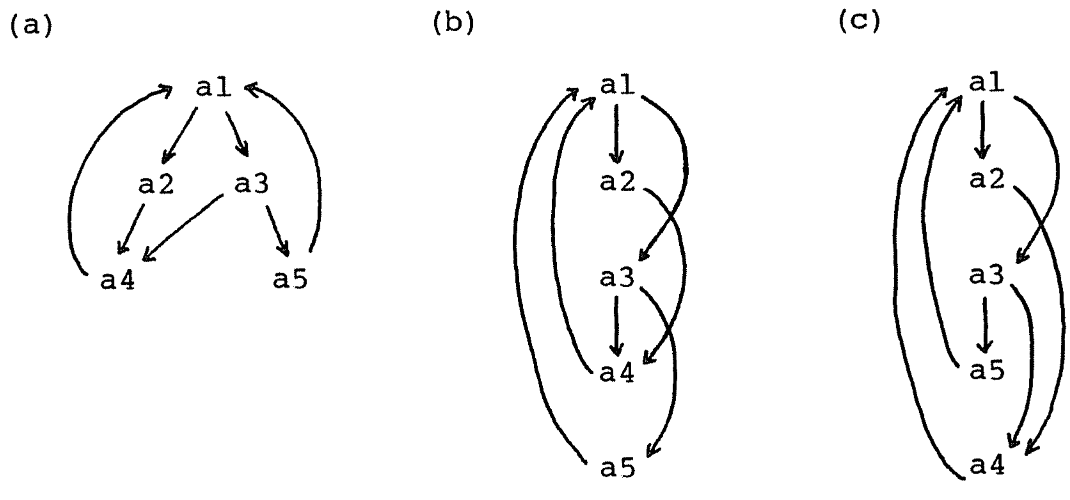
(i) Every formal loop is strongly connected and

(ii) There is a forward path from the start node of F to every other node, and

(iii) For every reverse arc (a_i, a_j), there is a forward path from a_j to every element of $[a_j, a_i]$.

It is possible to show, though we do not do so here, that (iii) is redundant for RFGs. The following example illustrates the concept of a straight order.

<u>EXAMPLE</u> (4) For the flowgraph (a) below, (b) is a straight order but (c) is not since there is no path from a2 to a1 in [a1, a5].



Given a straight order for an RFG we see that if $[a_i, a_j]$ is a formal loop then, by Theorem 3(d) and properties (i) and (ii) of a straight order, a_i must be the unique entry node of that formal loop. A two-phase algorithm for producing a straight order for <u>any</u> flowgraph appears in [14]. We are now ready to present our algorithm.

ALGORITHM STRUCTURE

INPUT An RFG G.

OUTPUT A program using LOOP-ENDLOOP with conditional and unconditional (labelled) multilevel exits.

METHOD

Step 1 Using the algorithm of [14] , output the nodes of G in straight order:

$$a_1$$
$$a_2$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$a_n$$

Step 2 Replace each node a_i by the appropriate statements given below: ($<a_i>$ denotes the simple statement or predicate that corresponds to the node a_i; Fj and ILj are used to label LOOPs.)

If a_i has no successors, replace it by $<a_i>$.

If a_i has exactly one successor, a_j, then

If $j \leq i$ (reverse arc) then replace it by: $<a_i>$;
EXIT ILj

If $j > i$ (forward arc) then replace it by: $<a_i>$;
EXIT Fj

If a_i has exactly two successors a_j, and a_k, then

If (j>i) and (k>i) (both forward arcs), replace a_i by:

```
{ IF <ai> THEN EXIT Fk ENDIF;
{ EXIT Fj
```

If (j≤i) and (k≤i) (both reverse arcs), replace a_i by:

```
{ IF <ai> THEN EXIT ILk ENDIF;
{ EXIT ILj
```

If (j≤i) and (k>i) (one forward and one reverse arc), replace a_i by:

```
{ IF <ai> THEN EXIT Fk ENDIF;
{ EXIT ILj
```

Denote by CODE(a_i) the statements that replaced a_i in this step.

<u>Step 3</u> We now create the reverse arcs.

For each maximal formal loop [a_i, a_j] do

For m = i to j do

If a_m has an incoming reverse arc then

```
Write { LOOP        } just before CODE(am) and
      { ILm: LOOP   }

      { ENDLOOP ILm } just after CODE(aj).
      { ENDLOOP     }
```

The following figure illustrates this step.

```
.                                   .
.                                   .
.                                   .
CODE(a_m)                           LOOP
.                                   ILm: LOOP
.                 ===>                   CODE(a_m)
.                                        .
CODE(a_j)                                .
.                                        .
.                                        CODE(a_j)
.                                   ENDLOOP ILm
                                    ENDLOOP
                                    .
                                    .
                                    .
```

Step 4 We now create the forward arcs.

For $m = 2$ to n do

Let j be the smallest integer such that (a_j, a_m) is an incoming forward arc of a_m.

Write "ENDLOOP Fm" just before the sequence of LOOP-keywords that immediately precedes CODE(a_m). {If no such sequence exists then write it just before CODE(a_m).}

Write "Fm: LOOP" just before the outermost statement that contains CODE(a_j) but not CODE(a_m). {If no such statement exists, write it just before CODE(a_j). }

The following figure illustrates this step.

```
.
.
.
LOOP          <---- The outermost statement
.                   X containing CODE(a_j)
.                   but not CODE(a_m).
CODE(a_j)
.
.
ENDLOOP
.  }  <------------ The set of statements Y
.  }
LOOP      }  <----- The sequence of
  LOOP    }         LOOP-keywords
  CODE(a_m)         immediately preceding
  .                 CODE(a_m).
  .
  ENDLOOP
.
.
ENDLOOP
.
.
.
```

$\Longrightarrow$

```
.
.
.
Fm: LOOP
  X
  Y
ENDLOOP Fm
LOOP
  LOOP
  CODE(a_m)
  .
  .
  ENDLOOP
.
.
ENDLOOP
.
.
.
```

<u>EXAMPLE</u> (5) We illustrate this algorithm with the straight order given in Example 4(b):

```
a1                       IF <a1> THEN EXIT F2 ENDIF;
                         EXIT F3;
a2                       <a2>;
                         EXIT F4
a3        Step 2         IF <a3> THEN EXIT F4 ENDIF;
          ======>        EXIT F5
a4                       <a4>;
                         EXIT IL1
a5                       <a5>;
                         EXIT IL1
```

```
                 LOOP
                 IL1:  LOOP
                       IF <a1> THEN EXIT F2 ENDIF;
                       EXIT F3
                       <a2>;
      Step 3           EXIT F4
      ======>          .
                       .
                       .
                       ENDLOOP IL1
                 ENDLOOP
```

The following program results after one iteration of the "For" loop of Step 4 (with m = 2) to create the incoming forward arcs of a_2.

```
LOOP
IL1: LOOP
     F2: LOOP
         IF <a1> THEN EXIT F2 ENDIF;
         EXIT F3
         ENDLOOP F2;
     <a2>;
     EXIT F4
     .
     .
     .
     ENDLOOP IL1
ENDLOOP
```

The final program output by the algorithm is

```
        LOOP
        IL1: LOOP
             F5: LOOP
                 F4: LOOP
                     F3: LOOP
                         F2: LOOP
                             IF <a1> THEN EXIT F2
                             ENDIF;
                             EXIT F3
                             ENDLOOP F2;
Step 4                       <a2>;
======>                      EXIT F4
                         ENDLOOP F3;
                       IF <a3> THEN EXIT F4;
                       EXIT F5
                       ENDLOOP F4;
                    <a4>;
                    EXIT IL1
                    ENDLOOP F5;
                <a5>;
                EXIT IL1
                ENDLOOP IL1
        ENDLOOP
```

<u>LEMMA</u> (19) Suppose G = (N,A,s) is an RFG that is input to Algorithm Structure and suppose P is the output. If G´ = (N´,A´,s´) is the SFG of an expression for P, then G´ = G.

<u>Proof</u>:

We must prove that the loops created by the algorithm are properly nested. This is clear for the loops created in Step 3. Call the loops created by Step 4 <u>fake loops</u>. We must show that when we create a fake loop "Fm", there is no interlocking loop as shown below:

```
              .
              .
     Fm:  LOOP
          CODE(a_j)
              .
              .
              .
          LOOP
          CODE(a_i)
              .
              .
          ENDLOOP Fm
          CODE(a_m)
              .
              .
          ENDLOOP
              .
              .
```

We will show that if such an interlocking loop exists, the flowgraph cannot be reducible. Let a_i be the first node in the interlocking loop. Clearly the interlocking loop cannot be a fake loop since we create the forward incoming arcs to the nodes in the order $a_1, \ldots, a_n$. Hence, a_i has an incoming reverse arc and so there is a formal loop of the form $[a_i, a_k]$ for some $k > i$. This formal loop is strongly connected (property (i) of a straight order) and has two entry nodes a_i and a_m violating Theorem 3(d). Hence P is a legal program and so G′ exists.

We now need to show that s=s′, N=N′, and A=A′. The first two equalities are obvious. For the last, suppose (a_i, a_j) is an arc of A. If $j \leq i$, then it is a reverse arc and so is created in Step 3 and so is in A′. If $j>i$ it is a forward arc and so is created in A′ in Step 4. Hence $A \subseteq A'$. Now, if a node n has outdegree 2 in G′ then CODE(n) must be of the form:

```
IF <n> THEN EXIT __ ENDIF;
EXIT __
```

But this can only happen in Step 2 of the algorithm if n has 2 successors in G. So n has outdegree 2 in G and so $|A'| \leq |A|$ and so $A' = A$.

<u>THEOREM</u> <u>5</u> Every RFG is an SFG.

<u>Proof</u>:

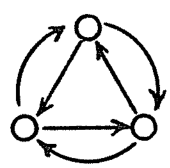
Follows directly from Algorithm Structure and Lemma 19.

3. THE HIERARCHY OF FLOWGRAPHS

We now compare the expressive power of level-(i+1) exits with that of level-i exits. If G is an arbitrary digraph or flowgraph, we define the rank of G to be

$$\min \{r(E) \mid E \text{ is an expression for } G\}.$$

If G is not the digraph of any expression, its rank is undefined; such a digraph is shown below:



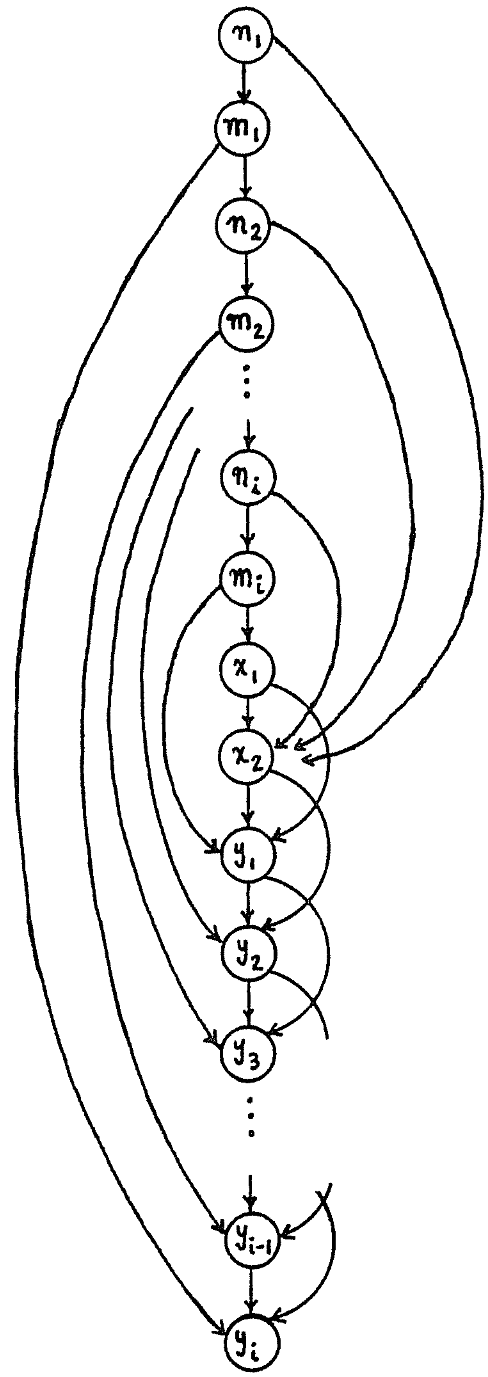
That level-(i+1) exits are more powerful than level-i exits can be demonstrated by showing that, for each r ∈ {0,1,2,...}, there is a digraph of rank r. A similar result has been derived in [4]. Our next theorem shows that this greater power can be demonstrated even within the domain of directed acyclic graphs. By a Hamiltonian dag we mean a dag which has a Hamiltonian path.

THEOREM 6 For each i ∈ {0,1,2,...} there is a Hamiltonian dag of rank i.

The dag H_i of Figure 7 has rank i. The rest of this chapter contains the proof of this assertion.

Figure 7

The dag H_i:



Several Lemmas are necessary for the proof of Theorem 6; we now proceed to develop them.

Assume that E is an expression and E1 is a subexpression of E; this assumption will remain in effect up to the end of Lemma 28.

A node of E1 is called incomplete iff its indegree in E is strictly greater than its indegree in E1. A node y of E is called a successor of E1 iff for some node x in E1, (x,y) is an arc of E but not of E1. Two expressions will be called equivalent if they have tha same rank and the same SFG. We write $E' \subseteq E$ if E´ is a subexpression of E. If E1, E2, ... are expressions, we will denote the set of nodes of their respective program graphs by N1, N2, ...

LEMMA (20)

x is incomplete in E1 ==> x = s(E1).

Proof:

The only node to which incoming arcs may later be added is s(E1).

COROLLARY Any subexpression of E can have at most one incomplete node.

We say that E is a

BREAK-expression if E = BREAK(E´,i) for some i, E´,

LOOP-expression if E = LOOP(E´) for some E´, CAT-expression if E = CAT(E´,E´´) for some E´, E´´,

IF-expression if E = IF(t,E´) for some t, E´,

ELSE-expression if E = ELSE(t,E´,E´´) for

some t,E´,E´´.

If E = OP(E1,E2,t,i) for some SPG operation OP, we say that E1 and E2 are <u>immediate subexpressions</u> of E and that E is the <u>immediate superexpression</u> of E1 and of E2.

<u>LEMMA</u> (21)

E1 has i successors ==> r(E1) $\geq$ i-1.

<u>Proof</u>:

Let L1 be the labelling relation of the program graph of E1 and let SUCC(E1) denote the set of all successors of E1. Assume E1 has i successors. Suppose there is a function g : A ---> SUCC(E1) such that

(i) A $\subseteq$ RAN(L1) and

(ii) g is onto.

Then, |SUCC(E1)| = |RAN(g)| (since g is onto)

$\leq$ |DOM(g)| (true for any function)

$\leq$ |RAN(L1)| (since A $\subseteq$ RAN(L1))

$\leq$ r(E1) + 1 (by definition or rank)

and so we have r(E1) $\geq$ SUCC(E1) - 1 = i - 1.

We will now define a function g with the required properties (i) and (ii).

Let $H = \{ (E',n,i) \mid E'$ is a subexpression of E and "i" is one of the labels of n in $E' \}$

We define a partial function $h\colon H \dashrightarrow H \cup \{\text{nodes of } E\}$ by

$$h(E',n,0) = x \quad \text{iff} \begin{cases} (x = s(E') \text{ and } LOOP(E') \subseteq E) \text{ or} \\ (x = s(E'') \text{ for some } E'' \text{ such that} \\ \quad CAT(E',E'') \subseteq E\ ) \end{cases}$$

$h(E',n,i) = (E'',n,j)$ iff one of the following holds:

(a) $E''=BREAK(E',k)$ and

$$((i > 0 \text{ and } i = j) \text{ or } (i = 0 \text{ and } j = k))$$

(b) $E''=LOOP(E')$ and $j = i - 1$

(c) $E'' = CAT(E',E1)$ for some E1 and

$$(i = j \text{ and } i > 0)$$

(d) $i = j$ and

$$\begin{cases} E'' = CAT(E1,E') \text{ for some E1 or} \\ E'' = IF(t,E') \text{ or} \\ \left.\begin{array}{l} E'' = ELSE(t,E',E1) \text{ or} \\ E'' = ELSE(t,E1,E') \end{array}\right\} \text{ for some E1} \end{cases}$$

We note that the value of h depends only on i and E'. (*) Let R be the transitive closure of the relation h. Suppose E2 is an immediate subexpression of E' and $(E',n,j) \in H$ and n is a node of E2. By examining the various cases that arise depending on the SPG operation that was applied to E2 to get E', it is easy to see that there is an element (E2,n,i) in H such that (E2,n,i) R (E´,n,j); that is, n must have had some label i in E2 which resulted in its having the label j in E´. By using this argument inductively we can prove that

$$\left.\begin{array}{l} E2 \subseteq E' \\ (E',n,j) \in H \\ n \in N2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \text{For some } (E2,n,i) \in H, \\ (E2,n,i)\ R\ (E',n,j) \end{array}\right. \qquad ---(**)$$

Every element p ∈ H determines a unique sequence called the <u>h-sequence</u> of p: $p=p_0, p_1, p_2, \ldots, p_k$ where

(a) $p_i \in H$ for $0 \le i \le k$

(b) $h(p_i) = p_{i+1}$ for $0 \le i < k$ and

(c) Either $h(p_k)$ is undefined or it is a node of E.

[If p = (E´,n,i), the node n may, at some future stage, acquire a successor as a consequence of its having the label i in E´; if it does, $h(p_k)$ identifies this successor node.]

Suppose p_0 = (X,n,i) and q_0 = (X,m,i) are two elements of H. As noted earlier (*), the value of h depends only on the first and third components of the argument. It follows that the h-sequences of p_0 and q_0 are of equal length and $h(p_k)$ and $h(q_k)$ (the last elements of the respective h-sequences) are either both undefined or are equal.

We are now ready to define the required partial function g: RAN(L1) ---> SUCC(E1)

$$g(i) = m \quad \text{iff} \begin{cases} \text{for some triples } u1 = (E1,n,i) \text{ and} \\ u2 = (E',n,0) \text{ in H we have } h(u2) = m \\ \text{and } (u1,u2) \in R. \end{cases}$$

By the conclusion immediately preceding this definition, we see that g is well-defined. We now need to show that g is onto. Suppose m is a successor of E1; we will show that g(i) = m for some i. By definition of successor, there is some node n in E1 such that (n,m) is an arc of E but not of E1. Hence there is a triple (E',n,0) = u in H such that h(u) = m and E1 $\subseteq$ E'. By (**), there is a triple (E1,n,i) = v in H such that (v,u) is in R. By definition of g, g(i) = m and so g is onto.

LEMMA (22)

$$\left.\begin{array}{l} \text{E1 is a LOOP-expression} \\ \text{and E1 has i successors} \end{array}\right\} \Longrightarrow \; r(E) \geq i$$

Proof:

Assume that E1 = LOOP(E2) and |SUCC(E1)| = i. Let L1 and L2 be the labelling relations of E1 and E2 respectively. Then L1 = decr $\circ$ L2 by definition of the LOOP operation and so |RAN(L1)| = |RAN(L2)| - 1 $\leq$ (r(E) + 1) - 1 = r(E) and the result now follows as in the proof of the previous Lemma.

LEMMA (23)

No expression for a Hamiltonian dag can have an ELSE-subexpression.

Proof:

Suppose G is a Hamiltonian dag and E is an expression for it. Suppose E1 = ELSE(t,E2,E3) is a subexpression of E. It is easy to show inductively that if E1 $\subseteq$ E′ $\subseteq$ E then, any path from s(E′) to s(E2) or s(E3) must pass through the node t.

Since g is Hamiltonian, there is a path from s(E2) to s(E3) or one from s(E3) to s(E2). Assume without loss of generality that the former is the case. The path, say p, from s(E2) to s(E3) does not exist in E1 and does exist in E; let E4 be the smallest subexpression of E in which it exists. E4 must be a LOOP-expression: E4 = LOOP(E5). Hence p must be of the form "s(E2).p1.s(E5).p2.s(E3)" where p1 and p2 are some paths. By the assertion made at the beginning of this proof, the path "s(E5).p2.s(E3)" must contain the node t and so G has a cycle (which is a subsequence of "t.s(E2).p1.s(E5).p2"). This contradicts the assumption that G is a dag.

<u>LEMMA</u> (24)

If n is a finish node of E and is not a finish node of E1 then, E1 $\subseteq$ E2 $\subseteq$ E for some LOOP-expression E2.

<u>Proof</u>:

If 0 is not a label of n in E1, no operation other than LOOP can create it.

<u>LEMMA</u> (25)

Suppose $(a_1, a_2, \ldots, a_n)$ is a simple path in E. Then,

$$\left.\begin{array}{l} 1 \le i < j \le n \text{ and} \\ a_i, a_j \in N1 \text{ and} \\ a_j \text{ is incomplete in E1} \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} a_k \in N1 \text{ for all } k \\ \text{such that } 1 \le k \le i. \end{array}\right.$$

<u>Proof</u>:

By the Corollary to Lemma 20, a_i is complete in E1 and so $a_{i-1} \in$ N1. This argument may be repeated on a_{i-1}.

<u>LEMMA</u> (26)

For arbitrary expressions A, B and C,

(a) CAT(A,CAT(B,C)) and CAT(CAT(A,B),C) are equivalent.

(b) BREAK(CAT(A,B),i) and CAT(A,BREAK(B,i)) are equivalent.

<u>Proof</u>:

It is easily seen from the definitions that

(i) $del_k \circ del_k = del_k$ for all $k \in \mathbb{I}^+$.

(ii) $del_k \circ del_j = del_j \circ del_k$ for all $k,j \in \mathbb{I}^+$.

(iii) $z_i \circ del_{\uparrow} = del_{\uparrow} \circ z_i$ for all $i \in \mathbf{I}$.

(iv) $z_i \circ del_0 = del_0$ for all $i \in \mathbf{I}$.

(v) Composition is distributive over union.

Obviously the pairs in (a) and (b) have the same rank, the same nodes and the same arcs. We will show that they have the same labelling relations.

(a) By definition of the CAT operation, the labelling relation of CAT(A,CAT(B,C)) is:

$$(del_0 \circ L_A) \cup (del_{\uparrow} \circ ((del_0 \circ L_B) \cup (del_{\uparrow} \circ L_C)))$$

$$= (del_0 \circ del_0 \circ L_A) \cup (del_0 \circ del_{\uparrow} \circ L_B)$$

$$\cup (del_{\uparrow} \circ del_{\uparrow} \circ L_C) \quad \text{(by (i),(ii) and (v) above)}$$

$$= (del_0 \circ ((del_0 \circ L_A) \cup (del_{\uparrow} \circ L_B))) \cup (del_{\uparrow} \circ L_C)$$

(by (i) and (v) above)

= the labelling relation of CAT(CAT(A,B),C).

(b) may be similarly proved using (iii), (iv) and (v).

<u>LEMMA</u> (27)

Suppose E1 and E1´ are equivalent. If E´ is the expression obtained from E by replacing E1 with E1´, E and E´ are equivalent.

<u>Proof</u>:

It is obvious that E and E´ have the same rank and the same set of nodes. Let E2 be the immediate superexpression of E1 and let E2´ be obtained from E2 by replacing E1 with E1´. From the definitions of the various SPG operations we see that E2 and E2´ are equivalent. The argument may be repeated with E2 in place of E1. The argument may be repeated with E2 in place of E1.

<u>LEMMA</u> (28)

Suppose E1 = CAT(A,B) and only CAT or BREAK operations are used between E1 and E and s(E) = s(A). Then there is an expression E´ = CAT(A,E2) for some E2 such that E and E´ are equivalent.

<u>Proof</u>:

Consider the immediate superexpression E2 of E1 in E. We must have E2 = CAT(CAT(A,B),C) or E2 = BREAK(CAT(A,B),i). By Lemma 26, there is an expression E2´ of the form CAT(A,E3) which is equivalent to E2. By Lemma 27, the expression obtained by replacing E2 with E2´ in E is equivalent to E. Repeating the process with E2´ in place of E1 we eventually arrive at the required expression E´.

All the Lemmas hereafter refer to the dag H_i (Figure 7). Assume that E is an expression for H_i. We define a few aliases for some of the nodes for notational convenience:

Let

$a_{2i+1} = x_1$,

$a_{2i+2} = x_2$,

$a_{2j} = m_j$ for $1 \le j \le i$ and

$a_{2j-1} = n_j$ for $1 \le j \le i$,

$W = \{ n_j \mid 1 \le j \le i \} = \{ a_{2j-1} \mid 1 \le j \le i \}$,

$V = \{ m_j \mid 1 \le j \le i \} = \{ a_{2j} \mid 1 \le j \le i \}$,

$X = \{ a_j \mid 1 \le j \le 2i+1 \} = W \cup V \cup \{x_1\}$,

$Y = \{ y_j \mid 1 \le j \le i \}$.

<u>LEMMA</u> (29)

If $E1 \in E$ and x_2 is not a node of E1 then, $|N1 \cap Y| \le 1$

<u>Proof</u>:

Suppose otherwise. Let $j < k$ be the two smallest indices such that $y_j, y_k \in$ N1 Y. Since y_j has an in-arc from

the preceding node t (which is either another node of Y or x_2), it follows that y_j must be incomplete in E1. Now y_k has an in-arc from outside E1 (either from another element of Y between y_j and y_k or from t) and so must also be incomplete in E1. This contradicts the Corollary to Lemma 20.

<u>LEMMA</u> (30)

$$\left.\begin{array}{l} E1 \in E \text{ and} \\ y_j \text{ is complete in } E1 \\ \quad \text{for some } j \geq 1. \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} (x_1,\ x_2 \in N1 \text{ and } j=1) \text{ or} \\ |N1 \cap Y| \geq 2 \end{array}\right.$$

<u>Proof</u>:

We show that under the assumptions of the Lemma, $|N1 \cap Y| = 1$ ===> $j = 1$. It then follows that x_1, $x_2 \in N1$ since (x_1, y_1) and (x_2, y_1) are arcs of H_i. Suppose $|N1 \cap Y| = 1$ and $j > 1$; we must have $y_{j-1} \in N1\ \ Y$ since (y_{j-1}, y_j) is an arc of H_i. This contradicts our assumption that $|N1 \cap Y| = 1$.

<u>LEMMA</u> (31)

$$\left.\begin{array}{l} E1 \in E \text{ and} \\ Y\ \ N1 = \phi \text{ and} \\ x_2 \text{ is complete in } E1 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \text{For some } k,\ 1 \leq k \leq i, \\ X - \{m_k\} \subseteq N1. \end{array}\right.$$

<u>Proof</u>: Let E2 be the smallest subexpression of E1 in which x_2 is complete. Since x_2 is complete in E2 we must have $W \cup \{x_1\} \subseteq N2$. If all the elements of $W \cup \{x_1\}$ are complete in E2 then $X \subseteq N2 \subseteq N1$ and the Lemma holds. Otherwise, let

$$k = \left\{\begin{array}{ll} i \text{ if } x_1 \text{ is incomplete in E2} & \text{-----------------}(*) \\ j\text{-}1 \text{ if } n_j \text{ is incomplete in E2} & \text{---------------}(**) \\ \quad \text{for some } j,\ 2 \leq j \leq i. & \\ \quad (\text{note: } n_1 \text{ is always complete}) & \end{array}\right.$$

We claim that $m_r \in N2$ for all r such that $1 \leq r \leq i$ and $r \neq k$. If this were not the case, let r be such that $1 \leq r \leq i$, $r \neq k$ and $m_r \notin N2$.

If $r = i$ then x_1 must be incomplete in E2 and by (*), we have $k=i=r$ contradicting our assumption that $r \neq k$.

If $r < i$ then n_{r+1} must be incomplete in E2 and by (**), we have $k = (r+1) - 1 = r$ and this is a contradiction as before.

Hence, for all r with $1 \leq r \leq i$, $r \neq k$, we have $m_r \in N2$. So $X - \{m_k\} \subseteq N2 \subseteq N1$ and we are done.

<u>LEMMA</u> (32)

If E1 is as assumed in the previous Lemma and is a LOOP expression, $r(E) \geq i$.

<u>Proof</u>:

By the previous Lemma, there is some k such that $1 \leq k \leq i$, and $X - \{m_k\} \subseteq N1$. Therefore E1 has i successors:

the elements of Y if $m_k \in N1$

the elements of $(Y - \{y_{(i-k+1)}\}) \cup \{m_k\}$ otherwise.

The result now follows from Lemma 22.

<u>LEMMA</u> (33)

If E1 is as assumed in Lemma 31, $r(E) \geq i$.

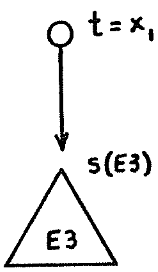
<u>Proof</u>:

Let E2 be the smallest subexpression of E1 in which x_2 is complete. Using Lemma 23 and the fact that BREAK creates no new arcs, we see that E2 must be an IF, CAT, or LOOP expression. We deal with these cases separately.

<u>Case 1</u> E2 = LOOP(E3).

The result follows from Lemma 32.

<u>Case 2</u> E2 = IF(t,E3).



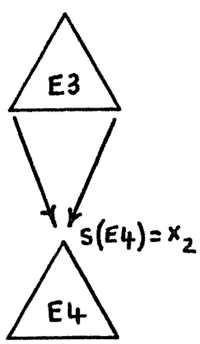
By Lemma 20, $x_2 = s(E3)$. If $x_1 \in N3$ then $X \subseteq N3$ by Lemma 25. Hence, $t \in Y$. This contradicts our assumption that N1 Y = ϕ. So $x_1 \notin N3$. But $x_1 \in N2$ since (x_1, x_2) is an arc of H_i and x_2 is complete in E2. So $t = x_1$. Now $W \subseteq N3$ since x_2 is complete in E2.

Since all the elements of W are complete in E3, we have $(W \cup V - \{m_i\}) \subseteq N3$. Now E3 has (i+1) successors:

$$\begin{cases} \text{the elements of } Y \cup \{m_i\} \text{ if } m_i \notin N3 \\ \text{the elements of } Y \cup \{x_1\} \text{ if } m_i \in N3. \end{cases}$$

The result now follows from Lemma 21.

<u>Case 3</u> E2 = CAT(E3,E4).



By Lemma 20, $x_2 = s(E4)$. By Lemma 31, $X - \{m_k\} \subseteq N2$ for some k such that $1 \leq k \leq i$. If x_1 is a node of E4, we would have $X \subseteq N4$ (by Lemma 25) and so E3 would be void; hence x_1 must be a node of E3. Two cases arise according to whether or not $m_k \in N2$.

<u>Case 3.1</u> $m_k \in N2$. (Hence $X \subseteq N2$)

There are two cases depending on whether N4 has any nodes of X.

Case 3.1.1 X N4 = ϕ

Here, we must have X $\subseteq$ N3 and so E3 has i+1 successors: Y $\cup$ $\{x_2\}$. The result follows from Lemma 21.

Case 3.1.2 X N4 $\neq \phi$.

Let p = max $\{ j \mid 1 \leq j \leq 2i,\ a_j \in N4 \}$. By Lemma 25, $a_j \in$ N4 for all j with $1 \leq j \leq p$.

Now $a_{p+1} \in X \subseteq$ N2 and of p, $a_{p+1} \in$ N3. Also, a_{p+1} must be incomplete in E2 since $a_p \in$ N4 and there are no arcs in E2 from N4 to N3. Now E2 has (i+1) successors, namely, the elements of Y $\cup$ $\{a_p\}$. The result now follows from Lemma 21.

Case 3.2 $m_k \notin$ N2

There are two cases, depending on whether E4 contains some node of X - $\{m_k\}$.

Case 3.2.1 E4 has no nodes of X - $\{m_k\}$.

We must have X - $\{m_k\} \subseteq$ N3 and so E3 has i+1 successors, namely, the elements of (Y - $\{y_{i-k+1}\}$) $\cup$ $\{m_k, x_2\}$. The result again follows from Lemma 21.

Case 3.2.2 E4 has some node of X - $\{m_k\}$.

Since $a_{2k} = m_k$ is not in N2 and $a_{2k+1} \in X - \{m_k\} \subseteq$ N2, a_{2k+1} must be incomplete in E2. Since x_2 is incomplete in E4, a_{2k+1} is in E3 and incomplete there. Let q = max$\{ j \mid 1 \leq j \leq 2i,\ a_j \in N4\}$. Now

$q < 2k-1$ is impossible since it implies that a_{q+1} is another incomplete node of E3 other than a_{2k+1} in violation of Lemma 20. Likewise, $q > 2k-1$ is also impossible since it implies, by Lemma 25, that $a_{2k} = m_k \in N4 \subseteq N2$. Hence $q = 2k-1$ and so by Lemma 25 we have $\{a_j \mid 1 \leq j \leq 2k-1\} \subseteq N4$. So $\{a_{2k+1} = x_1, a_{2k+2}, \ldots, a_{2i+1}\} \subseteq N3$. Let E5 be the smallest superexpression of E2 that has more nodes than E2. E5 exists since E has more nodes than E2. By Lemma 23, E5 must be an IF or a CAT expression. We consider these cases separately.

<u>Case</u> <u>3</u>.<u>2</u>.<u>2</u>.<u>1</u> E5 = IF(t,E6).

Since E6 has the same set of nodes as E2, a_{2k+1} must be incomplete in E6 (since $a_{2k} \notin N2$). But a_{2k+1} is not the start node of E5 and hence is complete in E5 and so $t = a_{2k} = m_k$. Now E5 has (i+1) successors, namely the elements of $Y \cup \{a_{2k}\}$ and the result follows from Lemma 21.

<u>Case</u> <u>3</u>.<u>2</u>.<u>2</u>.<u>2</u> E5 = CAT(E6,E7).

Here E2 can be a subexpression of either E6 or of E7 and so we have two further subcases.

<u>Case</u> <u>3</u>.<u>2</u>.<u>2</u>.<u>2</u>.<u>1</u> E2 is a subexpression of E7.

Since E2 has the same nodes as E7, a_{2k+1} must be incomplete in E7. Hence $s(E7) = a_{2k+1}$ and it is complete in E5; so $a_{2k} \in N6$ and must be incomplete in E6 since $a_{2k-1} \in N2 = N7$. Hence any member of $Y \cap N6$ must be complete in E6. Now $|N6 \cap Y| \leq 1$ by Lemma 29 since $x_2 \in N4 \implies x_2 \notin N6$. Since a_{2k} is incomplete in E6, any element of $N6 \cap Y$ must be complete in E6. This means that $N6 \cap Y$ is void (otherwise Lemma 30 is violated). Therefore, E6 has only one node: $m_k = a_{2k}$ (all other nodes are either in Y or in E7. Now E5 has i+1 successors, namely, the elements of $Y \cup \{a_{2k}\}$. The result now follows from Lemma 21.

<u>Case 3.2.2.2.2</u> E2 is a subexpression of E6.

Since $N6 = N2$, a_{2k+1} must be incomplete in E6. By assumption, E2 and Y have no common nodes and so E6 and Y are likewise. By Lemma 29, N7 and Y can have at most one common element; if they have no common elements at all, E7 has exactly one node: $a_{2k} = m_k$ and the result follows as usual from Lemma 21 since E5 has i+1 successors, namely, the elements of $Y \cup \{a_{2k+1}\}$. We now consider the case when they have exactly one element in common ($|N7 \cap Y| = 1$). That one element must be y_1, by Lemma 25. Now (x_1, y_1) is an arc of H_i and so x_1 must be a finish node of E6. But $x_1 \in N3$ and so cannot be a finish node of E2. By Lemma 24, there is a LOOP expression E8 with $E2 \subsetneq E8 \subseteq E6$. Now E8 has i successors, namely, the elements of $(Y - \{y_{i-k+1}\}) \cup \{m_k\}$ and so the result follows from Lemma 22. This completes the proof of Lemma 33.

LEMMA (34)

$$\left.\begin{array}{l} E1 \subseteq E \text{ and} \\ N1 = X \cup \{x_2\} \end{array}\right\} \Longrightarrow r(E) \geq i.$$

Proof:

If x_2 is complete in E1, the result follows from Lemma 33. Otherwise, E1 has i+1 successors, namely, the elements of $Y \cup \{x_2\}$ and the result follows from Lemma 21.

We now prove that the rank of H_i is at least i.

LEMMA (35)

If E is any expression for H_i, $r(E) \geq i$.

Proof:

Let E1 be the smallest subexpression of E such that $Y \cap N1 \neq \phi$ and $x_2 \in N1$. E1 must be a CAT or an IF expression by Lemma 23. We deal with these cases separately.

Case 1

E1 = CAT(E2,E3). There are two possibilities for x_2.

Case 1.1 $x_2 \in N2$ and $N3 \cap Y \neq \phi$.

By assumption about E1, N2 and Y have no common elements. By Lemma 29 $|N3 \cap Y| = 1$. THe unique node in $N3 \cap Y$ must be complete in E1 and so by Lemma 30, $N3 \cap Y = \{y_1\}$. Since (x_2, y_1) is an arc of H_i, we must have $s(E3) = y_1$.

Now we have two possibilities for x_1.

<u>Case</u> <u>1</u>.<u>1</u>.<u>1</u> $x_1 \in N3$.

Here x_1 is complete in E3 and by Lemma 25, $X \subseteq N3$. Now E3 has i+1 successors, namely, the elements of $Y \cup \{x_2\}$ and Lemma 21 yields the result.

<u>Case</u> <u>1</u>.<u>1</u>.<u>2</u> $x_1 \in N2$.

If x_2 is complete in E2 the result follows from Lemma 33. Otherwise $X \subseteq N2$ by Lemma 25 and now the result follows from Lemma 34.

<u>Case</u> <u>1</u>.<u>2</u> $x_2 \in N3$ and $N2 \cap Y \neq \phi$.

By assumption about E1, $N3 \cap Y$ is void. By Lemma 29, $|N2 \cap Y| = 1$. Also $x_2 \in N3$ implies that x_2 must be complete in E1. We now have two subcases depending on whether x_2 is complete in E3.

<u>Case</u> <u>1</u>.<u>2</u>.<u>1</u> x_2 is complete in E3.

The result follows from Lemma 33.

<u>Case</u> <u>1</u>.<u>2</u>.<u>2</u> x_2 is incomplete in E3.

By Lemma 20, we must have $s(E3) = x_2$. There are two possibilities for x_1.

<u>Case</u> <u>1</u>.<u>2</u>.<u>2</u>.<u>1</u> x_1 is in E3.

Since x_2 is incomplete in E3, x_1 must be complete in E3. By Lemma 25, $X \subseteq N3$ and the result follows from Lemma 34.

<u>Case 1.2.2.2</u> x_1 is in E2.

By Lemma 30 the unique element y of N2 $\cap$ Y must be incomplete in E2 and so s(E1) = s(E2) = y. Hence x_1 is complete in E2 and by Lemma 25 we have X $\subseteq$ N2. Let L (which may not exist) be the largest LOOP subexpression of E2 containing y. There are two possibilities for L.

<u>Case 1.2.2.2.1</u> L does not exist or y is its only node.

Let J be the largest subexpression of E2 that contains y and no other nodes. J exists since y must be created by EXIT(j,y) for some j (if y were created by an IF(y,-), there would be an arc from y to X which is clearly impossible; y cannot be created by an ELSE(y,-,-) by Lemma 23). If L exists L $\subseteq$ J. Since J has only one node node, there must be at least one operation between J and E2. Now, the first operation on J cannot be an IF(-,J) or a CAT(-,J) since s(E2) = y; it cannot be a LOOP or a BREAK due to our assumptions about J. Hence we must have CAT(J,E4) $\subseteq$ E2 for some expression E4. By assumptions about J and L, and by Lemma 23, there can only be CAT or BREAK operations between J and E2. Using Lemma 28, we get an expression

E2′ = CAT(J,E5) which is equivalent to E2. By Lemma 27 we can replace E2 with E2′ in E to get an expression E′ which is equivalent to E. Now E5 is a subexpression of E′ which has all the nodes of X (this is shown at the beginning of Case 1.2.2.2). Hence E5 has i+1 successors in E′, namely, the elements of $Y \cup \{x_2\}$ and by Lemma 21 we have $r(E') \geq i$. Since E is equivalent to E′, $r(E) \geq i$.

Case 1.2.2.2.2 L has at least one node other than y.

Since E2 has exactly one element of Y, we see that $L \cap X \neq \phi$. Let $k = \max\{ j \mid 1 \leq j \leq 2i+1, a_j \in L\}$. By Lemma 25, for all j with $1 \leq j \leq k$, $a_j \in L$ since y = s(E2) = s(L). We will show that L cannot have fewer nodes than E2. Assume otherwise. Let J be the largest subexpression of E2 that contains L and has no more nodes than L. By assumption E2 ≠ J and so there is at least one SPG operation between J and E2. The first such operation must be of the form CAT(J,E4) (IF(-,J) and CAT(-,J) are ruled out since s(E2) = s(L) = s(J) = y; BREAK and LOOP are ruled by the way J was defined; ELSE is ruled out by Lemma 23). Since a_1 is in J, it cannot be a finish node of CAT(J,E4). But a_1 must be a finish node of E2 since (a_1, x_2) is an arc of H_i and $x_2 = s(E3)$ is complete in E3. By Lemma 24 there is a LOOP expression that contains CAT(J,E4) and is contained in E2. This violates our assumptions about L and J. Hence L has all the nodes of E2. The result now follows by Lemma 22 since L has i successors, namely, the elements of $(Y - \{y\}) \cup \{x_2\}$.

<u>Case</u> <u>2</u> E1 = IF(t,E2).

It is impossible that $t \in Y$ and $x_2 \in E2$ since there are no arcs from Y to X. By assumption about E1, E2 has no nodes other than those of X. Hence it must be the case that $t = x_2$ and $N2 \cap Y \neq \phi$. By Lemma 29, $|N2 \cap Y| = 1$. By Lemma 20, the unique element of N2 Y is complete in E1 and by Lemma 30, that element must be y1. Let E3 be the subexpression of E2 which created the arc (x_1, y_1). Now $E3 = IF(x_1, E4)$ or E3 = CAT(E4,E5) are both impossible since they require y_1 to be complete in E3 and hence in E2. Hence E3 = LOOP(E4) where x_1 is a finish node of E4 and $y_1 = s(E4)$. By Lemma 25, $X \subseteq N4$ and so E3 has i successors, namely, the elements of $(Y - \{y_1\}) \cup \{x_2\}$ and now the result follows from Lemma 22.

Proof of Theorem 6.

Figure 8 shows a program of rank i for H_i. The Theorem now follows from the previous Lemma.

Figure 8

A program of rank i for H_i:

```
LOOP
 .
  .
   .
    LOOP
      LOOP
        IF <n1> THEN
           IF <m1> THEN
              IF <n2> THEN
                 IF <m2> THEN
                   .
                    .
                     .
                     IF <ni> THEN
                        IF <mi> THEN
                           IF <x1> THEN EXIT 1
                           ENDIF
                        ELSE EXIT 1 ENDIF
                     ENDIF
                     .
                    .
                   .
                 ELSE EXIT (i-1) ENDIF
              ENDIF
           ELSE EXIT i ENDIF
        ENDIF
        IF <x2> THEN EXIT 1
        ENDIF;
        EXIT 2
        ENDLOOP;
      IF <y1> THEN EXIT 1
      ENDIF;
      EXIT 2
      ENDLOOP
     .
    .
   .
  <yi-1> ; EXIT 1
ENDLOOP;
<yi>
```

CHAPTER 5

DISCUSSION AND CONCLUDING REMARKS

We have provided an algorithm that finds a minimum-jump translation for structured programs obtained by using loops with multilevel ´counting´ exits. Labelled exits are easily handled since they can be converted to ´counting´ exits. In addition, a wide variety of control structures, such as the REPEAT-UNTIL and WHILE loops of Pascal and the loop of Modula, are subsumed as special cases by our model.

We have shown that the class of flowgraphs that can be generated by such structured programs is the same as the class of reducible flowgraphs. Our algorithm can therefore be used to perform jump minimization in any object module, regardless of the compiler that produced it, provided it represents a reducible flowgraph; one need only apply our program structuring algorithm to get a structured program from the flowgraph and use this as input to our dissection algorithm. Obviously the efficiency of this procedure depends on the efficiency of the program structuring algorithm and also on the rank of the program it produces. The problem of producing a minimum-rank program from a given flowgraph needs further investigation.

No time complexity analysis is provided by Earnest et. al. in [14] for their algorithm which produces a straight order for the nodes of an arbitrary flowgraph; inasmuch as their algorithm is used by our program structuring algorithm for reducible flowgraphs, it would be useful to derive the time complexity of their procedure and to investigate whether improvements in speed are possible if we restrict ourselves to reducible flowgraphs.

Our jump minimization algorithm can also provide a good, though not necessarily optimal, translation for irreducible flowgraphs. We identify a set of arcs that can be deleted to leave a reducible flowgraph which has all the nodes of the original irreducible flowgraph and then find an optimal dissection for this residual flowgraph. Obviously, the smaller the number of arcs deleted, the better the chance that the result of this procedure is close to optimality. Finding such a set of arcs to delete is not hard (we could, for instance, do a depth first search and delete all back arcs) though finding one with the smallest number of arcs could be. This gives rise to a question analogous to the problem of finding a minimum feedback-edge set for digraphs:

> Given a flowgraph G and a positive integer K, does G have a set of K or fewer arcs which, when deleted, leave a reducible flowgraph ?

Whether the Dissection Problem is NP-complete for reducible flowgraphs is an open question. We conjecture that the Hamiltonian Path problem is NP-complete even for the restricted class of flowgraphs which are acyclic except for a single incoming arc to the start node, that is, flowgraphs that become acyclic if a single incoming arc to the start node is deleted.

Performing jump mimization could affect other code improvement procedures. For instance, preliminary evidence suggests that generating code for machines that have long and short branch instructions [15] after performing jump minimization is likely to yield better code than doing it before. Another related issue that will bear investigation is the impact of jump minimization on the dynamic behaviour of the program in a multiprogramming environment: It appears likely that the number of page faults and cache misses will be favourably affected.

Of theoretical interest are the connections between the cardinality of an optimal dissection of a digraph and certain graph-theoretic parameters. Let d(G) denote the cardinality just adverted to. Two nodes x and y in a digraph G are called arc independent iff neither (x,y) nor (y,x) is an arc of G; they are called path independent iff there is no (directed) path from x to y and there is no (directed) path from y to x in G. The maximum number of arc-independent nodes in G is called the [16] stability number of G; denote this number by $\alpha(G)$. The maximum number of path-independent vertices in G is called the path independence number of G; denote this number by i(G). Then, we have:

$$i(G) \leq d(G) \leq \alpha(G).$$

The first inequality is easy to see; the second is proved in [16,17]. It is easy to show by means of simple examples that all three of these parameters could, in general, be distinct. Whether, and under what conditions, equality obtains is not known.

We have a linear algorithm for the case when only IF-THEN-ELSE and REPEAT-UNTIL constructs are used (without EXIT statements) [18]. When WHILE statements are included, a more complicated, but still linear, algorithm exists. This suggests that the complexity of algorithm for the general case can be significantly improved by refining the process whereby dissections for smaller graphs are combined to produce dissections for the larger graph.

REFERENCES

[1] M. R. Garey and D. S. Johnson, Computers and Intractability - A Guide to the Theory of NP-Completeness, W. H. Freeman, San Francisco (1979).

[2] J. Plesnik, "The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound 2," Information Proc. Letters 8, 4, pp. 199-201 (April 1979).

[3] F. T. Boesch and J. F. Gimpel, "Covering the points of a digraph with point-disjoint paths and its application to code optimization," Journal of the ACM 24, 2, pp. 192-198 (April 1977).

[4] S. R. Kosaraju, "Analysis of structured programs," Journal of Comp. and Syst. Sci. 9, 3, pp. 232-255 (1974).

[5] J. C. Cherniavsky, P. B. Henderson, and J. Keohane, On the Equivalence of URE Flowgraphs and Reducible Flow Graphs, Proc. 1976 Conf. on Inf. Sci. and Sys., Johns Hopkins University ().

[6] H. F. Ledgard and M. Marcotty, "A Genealogy of Control Structures," Comm. of the ACM 18 , pp. 629-639 (1975).

[7] A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling Volume 2: Compiling, Prentice-Hall (1972).

[8] F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," Comm. of the ACM 19, 3 , pp. 137-147 (1976).

[9] M. S. Hecht, Flow Analysis of Computer Programs, American Elsevier, New York (1977).

[10] T. Kasami, W. W. Peterson, and N. Tokura, "On the capabilities of while, repeat, and exit statements," CACM 16, 8, pp. 503-512 (August 1973).

[11] R. E. Tarjan, "Depth First Search and Linear Graph Algorithms," SIAM J. Computing 1, 2, pp. 146-160 (1972).

[12] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley (1976).

[13] V. N. Kasyanov, "Some Properties of Fully Reducible Graphs," Information Processing Letters 2, 4 , pp. 113-117 (1973).

[14] C. P. Earnest, K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," Journal of the ACM 19, 1, pp. 23-42 (Jan 1972).

[15] T. G. Szymanski, "Assembling code for machines with span-dependent instructions," CACM 21, 5, pp. 300-308 (April 1978).

[16] C. Berge, Graphs and Hypergraphs, North Holland, Amsterdam (1973).

[17] T. von Gallai and A. N. Milgram, "Verallgemeinerung eines Graphentheoretischen Satzes von Redei," Acta. Sc. Math. 21 , pp. 181-186 (1960).

[18] M. V. S. Ramanath and M. H. Solomon, "Generating optimal code from flowgraphs," Computer Languages, (1982) To appear.