INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeib Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR
77-19,286

RAMANATHAN, Jayashree, 1950-
GLORAL DATA FLOW ALGORITHMS AND THEIR
IMPLEMENTATIONS.

Rice University, Ph.D., 1977
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
RICE UNIVERSITY

GLOBAL DATA FLOW ALGORITHMS
AND THEIR IMPLEMENTATIONS

by

Jayashree Ramanathan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

Kenneth W. Kennedy

Houston, Texas
May, 1977
Dedicated to Cheena and my family.
Acknowledgements

The author wishes to thank her thesis advisor, Professor Ken Kennedy, for the support and encouragement he provided during the course of the research. Thanks are also due to Professors Blattner and Rao for serving as committee members. The author also wishes to thank Mrs. Velva Power for her helpfulness and competence in typing this thesis.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Introduction</td>
<td>11</td>
</tr>
<tr>
<td>1. A Deterministic Attribute Grammar Evaluator</td>
<td>1</td>
</tr>
<tr>
<td>2. Global Data Flow Analysis of Structured Programs</td>
<td>34</td>
</tr>
<tr>
<td>3. Analysis of Flow Graphs of Unstructured Programs</td>
<td>57</td>
</tr>
<tr>
<td>4. Global Data Flow Analysis of Reducible Flow Graphs</td>
<td>76</td>
</tr>
<tr>
<td>5. Conclusions and Recommendations</td>
<td>148</td>
</tr>
</tbody>
</table>
Introduction.

There are two main objectives underlying this thesis. The first is to develop a formal tool for the automatic implementation of translators. The second is to better understand the semantic aspects of programming languages with a view toward using the above mentioned implementation tool. This involves examining two general types of global optimization problems—the backward and forward data flow problems—defined on reducible flow graphs which model programs.

In Chapter 1 we introduce the concept of attribute grammars and show how these grammars can be used to specify a translation process. A source string must be 'evaluated' as specified by the attribute grammar in order to determine the object string. Here we present another method for evaluating the source strings in a given language. The treatment of 'dynamic sequence evaluators' given here draws heavily from the work of [KW].

We examine the semantic aspects of programming languages in stages. In Chapter 2 we consider only 'structured' programs. It is generally true that the most rewarding optimization efforts have been carried out on such programs. Here we show how to use attribute grammars to perform efficient global data flow analysis on the object code generated for programs containing very general control structures.

The parse tree and the control structure of unstructured programs are not related in any systematic way. This necessitates the pre-processing of control flow graphs described in Chapter 3. Here we describe an attribute grammar which determines the dominator tree underlying a flow graph, more efficiently than existing methods.
Chapter 4 uses dominator tree based attribute grammars to specify various optimization algorithms for any given reducible flow graph model of a program. The forward flow algorithm thus specified is of \( O(e, n)\alpha(e, n) \) where \( n \) is the number of nodes in the flow graph and \( e \) the number of edges. This bound is better than the existing bounds for doing forward flow analysis. The backward flow algorithm specified is of \( O(n + e \log(n)) \) and is relatively inefficient. We present a formal discussion of the distinction between the two types of data flow analysis problems. This discussion implies that the inefficiency of the backward flow algorithm may be due to the fact that backward flow analysis is an inherently more complex problem than forward flow analysis for reducible flow graphs.

In the concluding chapter (Chapter 5) recommendations are made for further research.

Much of the introductory material in sections 1.2, 1.3 and 1.4 is adapted from the original work in [KW]. In addition, basic material in 2.1 and 2.2 is adapted from [Ke2]. The author wishes to thank Kennedy and Warren for their permission to use this material.

\( \alpha \) is related to a functional inverse of the Ackermann's function. See Chapter 4 of [AHU] and [T3]
1. A Deterministic Attribute Grammar Evaluator.

The problem of formal definition of the semantics of programming languages has been the subject of investigation for more than ten years and many different approaches have been taken [deH]. Three general techniques have emerged: the denotational method in which each program is assigned a mathematical formula expressing its meaning; the operational method in which a formal description is given of the steps in executing a program; and the translational method in which a mapping is given which takes each program into an equivalent one in a language with "known" semantics. The last of these relates most naturally to the automatic implementation of programming languages since it provides a model for the compilation process.

A variety of techniques have been used to introduce the idea of semantics into the well-understood context-free grammars [A, SL, R, V, GH, I, LS, AU1, AU2]. The current work is based on the "attribute grammars" of Knuth [Kn2] which allows attributes depending on context to be passed down the parse tree as "inherited" attributes and attributes based on the meaning of terminal symbols within said context to be passed up the tree as "synthesized" attributes. These grammars have attracted widespread interest [B1, B2, Cu, D, J1, J2, NA, RS, W1, W12]. Warren and Kennedy have presented a general scheme for generating attribute grammar evaluators, using finite-state "treewalk evaluators" [Wa,KW]. In this work, we follow a somewhat different approach which requires less analysis at generation time but more at evaluation time. The method is based on techniques of global data flow analysis taken from work on compiler
optimization [Kel, Au3]. Because it does more of the analysis at compile time, this method may be less efficient than that of Warren and Kennedy but it is conceptually simpler and, consequently, easier to implement.

The plan of this chapter is as follows: Section 1.1 introduces attribute grammars and the associated terminology. In section 1.2 the dependencies of attributes upon one another is discussed and the "dependency graph" is introduced. Section 1.3 gives an overview of the evaluation concept and discusses previous work. These three introductory sections are directly adapted from the treatment by Warren [Wa], with permission. Section 1.4 begins the treatment of the new evaluation method with a discussion of "dynamic sequence evaluators" and how they work; section 1.5 continues the treatment by describing a general constructor for such evaluators. Finally, in section 1.6 the computational complexity of dynamic sequence evaluation is derived.

1.1 **Attribute Grammars.** In this section we define and motivate the concept of an "attribute grammar" and illustrate its usefulness as a definitional tool.

1.1.1. **Definition.** An attribute grammar is an ordinary context-free grammar augmented with attributes and semantic functions as described below.

**Grammar:** A reduced context-free grammar $CFG = (V_N, V_T, P, S)$. We write $V$ for $V_N \cup V_T$. A production $p \in P$ is written as

$$p : X_0 \rightarrow X_1 X_2 \ldots X_n$$
where $n_p \geq 1$, $X_0 \in V_N$, and $X_k \in V$ for $k = 1, 2, \ldots, n_p$. We write $p[k]$ to mean $X_k$ for $k = 0, 1, \ldots, n_p$. We assume the grammar is standardized with a 0-th production $0:S \rightarrow S'$ so that the start symbol $S$ occurs in no other production. A parse tree of $G$ is a finite ordered tree whose nodes are labelled with symbols from $V$, such that

(a) Each interior node is labelled with an element of $V_N$ and each leaf node is labelled with an element of $V_T$;

(b) For each interior node $t$, there is a production $p \in P$ such that $t$ is labelled with the symbol $p[0]$, $t$ has $n_p$ sons, and the $k$-th son of $t$ is labelled with the symbol $p[k]$. We say that $p$ applies at $t$, or equivalently that $t$ is a type-$p$ node.

**Attributes:** For each $X \in V$, there are disjoint finite sets $I(X)$ and $S(X)$ of inherited and synthesized attributes respectively. For $X = S$, the start symbol, and for $X \in V_T$, we require that $I(X) = \emptyset$. We write $A(X)$ for $I(X) \cup S(X)$. The attributes of each symbol $X$ are uniquely identified and determine the various components of the symbols "meaning." A production $p : X_0 \rightarrow X_1 X_2 \cdots X_n$ has the attribute occurrence $(a, k)$ if $a \in A(X_k)$ for $k = 0, 1, \ldots, n$. The entire set of attributes for the grammar is denoted as $A = \bigcup_{X \in V} A(X)$. Attribute occurrences are to be understood as variables which are used in writing the semantics for a production. Inherited attributes transmit information down the parse tree toward the leaves, while synthesized attributes transmit information up the tree toward the root. The start symbol may not have inherited attributes since it can have no ancestor. Terminal symbols may have no inherited attri-
butes because they have no associated semantics. The values of a terminal symbol's synthesized attributes are given initially; in a compiler, this is the job of the lexical scanner. The term "attribute" is often used ambiguously to mean some \( a \in A(X) \), as in "an attribute of a nonterminal"; to mean some occurrence \((a,k)\), as in "an attribute of a production"; or to mean a value attached to the parse tree, as in "an attribute of a node". It should always be clear from the context which sense is intended.

**Semantic functions**: For each production \( p \in P \), there is a set \( F(p) \) of semantic functions as follows: for every synthesized occurrence \((a,k)\) with \( k = 0 \), and for every inherited occurrence \((a,k)\) with \( k = 1,2,\ldots,n_p \), there is a semantic function \( f^p(a,k) \in F(p) \) mapping certain other attribute occurrences of \( p \) into a value for \((a,k)\). The dependency set of \( f^p(a,k) \) is denoted \( D^p(a,k) \) and contains those attribute occurrences of \( p \) used in the definition of the semantic function. The semantic functions specify the meanings of parse trees locally, only in terms of the attributes of a node and its immediate descendents. We do not consider the language in which the functions are written; we assume only that we can identify the attribute occurrences referenced in a function and that the function can be translated into machine code to do the evaluation. The entire set of semantic functions for the grammar denoted as \( F = \bigcup_{p \in P} F(p) \). An attribute grammar, \( AG \), can therefore be defined as \( AG = (CFG,A,F) \).

An attribute grammar is an ordinary context-free grammar extended to specify the "meaning" of each string in the language. Each grammar symbol has an associated set of "attributes," and each
production rule is provided with corresponding semantic rules expressing the relationships between the attributes of the symbols in the production. To find the meaning of a string, first we find its parse tree and then we determine the values of all the attributes of symbols in the tree.

As an example, in Figure 1.1.2 we give an attribute grammar, AG, to recognize strings belonging to the context sensitive language \( L = \{ww / w \in \{a,b\}^*\} \). A convenient strategy, used here to design the attribute grammar, is to first determine a context free grammar, CFG, such that \( L(CFG) \supset L \). Such a context free grammar is given in Figure 1.1.1a for the language \( L \). Semantics functions are then augmented to the productions of CFG to perform the task of examining attributes of symbols in the parse tree of any string \( \alpha \in L(CFG) \) and setting a predetermined flag attribute if \( \alpha \) also belongs to the language \( L \). See Figures 1.1.1b and 1.1.1c.

\[
\text{CFG} = (V_N, V_T, P, N) \quad \quad \quad \quad \quad \quad p = 0: S \rightarrow AB \\
V_N = (S, A, B) \\
V_T = (a, b) \\
1: S_1 \rightarrow AS_2 B \\
2: A \rightarrow a \\
3: A \rightarrow b \\
4: B \rightarrow a \\
5: B \rightarrow b .
\]

Figure 1.1.1a \( L(CFG) = \{w / w \in \{a,b\}^* \text{ and w contains an even number of symbols}\} \).
Flag attribute

Figure 1.1.1b. Parse tree for $aababa \in L(CFG)$

Figure 1.1.1c. Parse tree for $aabaab \in L \cap L(CFG)$
\[ V_N = \{s_0, s_1, a, b\} \]
\[ V_T = \{a, b\} \]
\[ I(s_0) = \emptyset \]
\[ I(s_1) = \{s_1.in\} \]
\[ I(a) = \{a.in\} \]
\[ I(b) = \{b.in\} \]

Semantic Functions (F)  

\[ 0: s_0 \rightarrow s_1 \quad S_1.in \leftarrow \emptyset; S_0.ok \rightarrow \begin{cases} \text{true} & \text{if } S_1.out = \emptyset \\ \text{false} & \text{else} \end{cases} \]

\[ 1: s_1 \rightarrow as_1'b \]
\[ A.in \leftarrow s_1.in; \]
\[ s_1'.in \leftarrow a.out; \]
\[ s_1.out \leftarrow b.out; \]
\[ b.in \leftarrow s_1'.out \]

\[ 2: s_1 \rightarrow ab \]
\[ A.in \leftarrow s_1.in; \]
\[ B.in \leftarrow a.out; \]
\[ s_1.out \leftarrow b.out; \]

\[ 3: a \rightarrow a \]
\[ A.out \leftarrow \land (\text{a.in}) \]

\[ 4: a \rightarrow b \]
\[ A.out \leftarrow \land (\text{b.in}) \]

\[ 5: b \rightarrow a \]
\[ \text{IF FIRST(b.in) = a \ THEN B.out = REST(b.in)} \]
\[ \text{ELSE B.out = b.in} \]

\[ 6: b \rightarrow b \]
\[ \text{IF FIRST(b.in) = b \ THEN B.out = REST(b.in)} \]
\[ \text{ELSE B.out = b.in} \]

FIRST(b.in) \rightarrow \text{Leftmost letter of the substring in b.in} 

REST(b.in) \rightarrow \text{Substring B.in with the leftmost letter deleted}

\[ AG = ([V_N, V_T, S_0.F], \{s_1.in, a.in, b.in, s_0.ok, s_1.out, a.out, b.out\}, F) \]

Figure 1.1.2. Attribute Grammar for Recognizing \( L = \{ww / w \in (a,b)^*\} \)
In figure 1.1.2 we introduce the rotation 'x.a' to stand for the 'a' attribute of the non-terminal 'x'. The production \( S_0 \rightarrow S_1 \) is introduced as a mechanism for initializing a unique occurrence of \( S_1.in \) in any parse tree—the occurrence of \( S_1.in \) nearest the root of the parse tree. The attribute grammar in figure 1.1.2 assigns the semantic value \text{TRUE} \) to \( S_0.ok \) occurring in the parse tree for string \( \alpha \) if \( \alpha \) is contained in the language \( L \). Similarly, it assigns the value \text{FALSE} \) to \( S_0.ok \) in the parse tree for string \( \alpha \) if \( \alpha \) is contained in \( L(CFG) \) but not in \( L \).

We can think of each node in the parse tree as a structured variable where fields are its attributes. Figure 1.1.3 shows the parse tree for aababa redrawn to show this. The fields have been filled as prescribed by the semantic rules. The field for \( S_0.ok \) contains false indicating the semantic value of aababa.

The semantic functions cause each half of the input string to be considered separately. These functions effectively scan the letters in the first half of the string in a left to right order. As each new letter is examined, it is concatenated to the scanned substring contained in an inherited attribute \( A.in \). The new substring is assigned as the value for the appropriate synthesized attribute \( A.out \). This synthesized information is passed along to the next letter as the inherited attribute \( A.in \). At the midpoint of the string, semantic functions effectively begin a scan of the second half of the input string. Each letter scanned is compared with the corresponding letter in the first half of the string—this information is available as the inherited attribute \( B.in \). If a match occurs the first letter of the sequence contained in \( B.in \) is deleted. The sequence, altered only if a match has
Figure 1.1.3. Evaluated Parse Tree for aaababa.

occurred, is assigned as the value of the appropriate synthesized attribute B.out and passed to the next letter as an inherited attribute B.in.

1.2. Dependency Constraints

Recall that the dependency set $D^P_{(a,k)}$ of a semantic function $f^P_{(a,k)}$ is the set of attribute occurrences that are used in the computation of the result attribute $(a,k)$. Naturally, a semantic
function cannot be evaluated until all the occurrences in its dependency set have been computed. This is the only constraint on the order of evaluation imposed by an attribute grammar. Our aim is to find a specific evaluation order which obeys this constraint. We therefore need some concepts and notation for the study of dependencies.

1.2.1. Definition. Following standard terminology from graph theory, a directed graph, \( G \), is a pair \( G = (V,E) \) where \( V \) is a finite set of vertices and \( E \) is a finite set of ordered pairs of vertices called edges or directed arcs. We say that there is a directed arc from \( v_1 \) to \( v_2 \) in \( G \) of \( v_1, v_2 \in E \). Dependency graph of a production \( p \in P \) is the graph

\[
DG_p = (DV_p, DE_p)
\]

where \( DV_p \) is the set of attribute occurrences for the production \( p \) and \( DE_p \) is the set of dependency arcs for that production. Specifically, if \( p \) is the production \( X_0 \rightarrow X_1 X_2 \ldots X_n \) then

\[
DV_p = \{(a, k) | a \in A(X_k) \forall k, \quad 0 \leq k \leq n\}
\]

and

\[
DE_p = \{(a_1, k_1), (a_2, k_2) | (a_1, k_1) \in DP (a_2, k_2)\}
\]

That is, there is a directed arc from attribute occurrence \( 0_1 \) to occurrence \( 0_2 \) if \( 0_1 \) is in the dependency set for \( 0_2 \). The arcs of such a dependency graph may be thought of as data flow paths. Figure 1.2.1 shows two ways of drawing the dependency graph for a rule from our example grammar; the second form is intended to suggest
a portion of the parse tree and will be used in the remainder of the paper to depict flow up and down the tree.

\textbf{Semantics}

\begin{align*}
S_1'.\text{in} & \rightarrow A.\text{out} \\
S_1'.\text{out} & \rightarrow B.\text{out} \\
A.\text{in} & \rightarrow S_1'.\text{in} \\
B.\text{in} & \rightarrow S_1'.\text{out}
\end{align*}

\[DG_1: \]

\[\begin{array}{c}
\text{(in,0)} \\
\text{(out,0)} \\
\text{(in,1)} \\
\text{(out,1)} \\
\text{(in,2)} \\
\text{(out,2)} \\
\text{(in,3)} \\
\text{(out,3)}
\end{array}\]

\[\begin{array}{c}
\text{(in,1)} \\
\text{(out,1)} \\
\text{(in,2)} \\
\text{(out,2)} \\
\text{(in,3)} \\
\text{(out,3)}
\end{array}\]

\[\begin{array}{c}
\text{(in,0)} \\
\text{(out,0)} \\
\text{(in,1)} \\
\text{(out,1)} \\
\text{(in,2)} \\
\text{(out,2)} \\
\text{(in,3)} \\
\text{(out,3)}
\end{array}\]

\textbf{Figure 1.2.1.} The Dependency Graph of a Production
If we have a parse tree of an attribute grammar, we can construct a dependency graph which represents all data flow paths in the tree. This graph is the result of "pasting together" copies of the $D_{p}$'s for productions occurring in the tree, as illustrated in Figure 1.2.2. If a subtree is detached from the rest of the parse tree, some data flow paths through the root of the subtree will be interrupted. Data flows into the subtree through inherited attributes of its root, and out of the subtree through synthesized attributes of the root. For this reason these attributes are sometimes referred to as input attributes and output attributes of the subtree. In a production, the input occurrences are those of the form $(i,0)$ where $i \in I(X_{0})$, and the output occurrences are those of the form $(s,0)$ where $s \in S(X_{0})$.

If the dependency graph of a parse tree contains a directed cycle, some of its attributes cannot be determined. In this case the tree is said to be circular semantically. A circular attribute grammar is one that can generate a circular parse tree. A grammar with circular semantics is obviously undesirable, in that we would like an attribute grammar to specify the meaning of every string in the language. Knuth [Kn2, Kn3] has given an algorithm to test an attribute grammar for circularity, but the worst-case complexity of that test is at least exponential [J1].

Kennedy and Warren [KW] have defined and used a restricted class of attribute grammars, the absolutely non-circular grammars, based on Knuth's original, erroneous circularity test [Kn2, Kn3]. However that restriction will not be required in this paper because the method to be presented works for all non-circular grammars.
Figure 1.2.2 The Dependency Graph of Parse Tree for ababa
1.3 Attribute Grammar Evaluation.

To put the role of evaluation in its proper perspective, let us consider our aims in processing an attribute grammar. Primarily we wish to automatically produce a programming language compiler from the attribute grammar specification for the desired translation. This situation is depicted in Figure 1.3.1, below.

![Diagram](image)

**Figure 1.3.1.** Processing Aim for Attribute Grammars.
In other words, we wish to process the given attribute grammar to produce a program which will semantically evaluate any correct input string of the language described. The generated compiler will read the input program and construct a parse tree for it; then it will compute the semantic attributes for each node in the parse tree. Specifically, the semantic evaluation phase should compute for each node of nonterminal type \( X \) the values of attributes in \( A(X) \). One possible attribute for a node is the stream of machine code which will execute the subtree tending from that node.

Thus, our goal in processing an attribute grammar is to generate a parser and a semantic evaluator for inclusion in the compiler. Techniques of parser generation are well-known (see, for example, [AU2]) and will not be discussed. To generate the semantic evaluator one should remember that it must be able to correctly evaluate any tree that the parser might generate from a correct source program.

Formally, a semantic tree of an attribute grammar is a parse tree in which each node labelled with \( X \in V \) (the node is of "Type \( X \)" with respect to the context-free grammar) has associated with it a structured variable whose field selectors are the elements of \( A(X) \). The parser generated from an attribute grammar builds a semantic tree for the given input program with undefined attribute fields. An evaluator for an attribute grammar accepts such a semantic tree and fills in the attribute fields according to the semantic rules from the original grammar.

In this paper, we will refer to the time when an evaluator runs as run-time even though the evaluator is itself part of a language compiler. Thus we think of the attribute grammar processor
as a compiler which produces programming language translators as object programs.

The simplest possible form of evaluator we might choose is the *definitional evaluator*, which will provide a conceptual model for later discussions.

Evaluation begins with all fields of the semantic tree undefined. At each step, some attribute of a node is chosen whose semantic function is "ready to be evaluated"; that is, all of whose argument occurrences are already defined. The chosen semantic function is executed and the corresponding field of the tree node is defined by setting it to the computed value. The process continues until all attributes in the tree have been defined. The definitional evaluator is nondeterministic, since at each step any attribute which is ready can be chosen.

Fang's system [F] which we described in the introduction, is based on this approach. As we pointed out, the resulting evaluators are hopelessly inefficient when implemented on currently available machines. The main causes of this inefficiency are the reliance on a great many parallel processes and the analysis of dependency constraints at run-time.

A definitional evaluator must have many parallel processes; one for each node in the semantic tree. These processes must be examined by a scheduler to determine which ones are currently "ready" for execution, which in turn requires a substantial amount of overhead processing by the scheduler.

In the current work, we take a slightly different approach by separating out the scheduling function into a pre-pass through the parse tree. The total evaluation is merely a sequence of individual
function evaluations; the problem is to choose the correct sequence, an approach that resembles the "node listing" technique of global flow analysis [Ke, AU3]. The result is an efficient deterministic evaluator.

1.4. The Dynamic Sequence Evaluator.

The evaluator produced by the system we propose will have three phases of execution:

1. Dependency Graph Construction,
2. Evaluation Sequence Generation,
3. Evaluation Sequence Execution.

All actual evaluation of attributes takes place in the third of these phases; the first two are devoted to determining the "best" way to go about evaluation.

Dependency Graph Construction.

In the first processing phase, an augmented dependency graph is constructed during a walk of the parse tree. Recall that the parser produces a "semantic" parse tree in which each node as an associated structured variable whose fields contain the attributes for that node; the specific nature of the attached attributes depend on the 'non-terminal type represented by the node. Assume that the nodes are assigned indices in some way and that they can be addressed by these indices. Let $\text{NT}(i)$ and $p(i)$ denote the associated non-terminal type and production for node $i$, where

$$p(i):X_0 \rightarrow X_1 \ldots X_n$$

and $X_0 = \text{N}(i)$. Note that $p(i)$ is the production used to generate the subtree tending from node $i$. 
At this point we introduce some new terminology to distinguish attribute occurrences in the grammar from attribute occurrences in the semantic parse tree.

1.4.1. **Definition:** Given a node indexed by \( i_0 \) in the parse tree with associated production

\[
p(i_0): X_0 \rightarrow X_1 \cdots X_n p(i_0),
\]

the attribute realization corresponding to attribute occurrence \((a,k)\) is denoted \((a,i_k)\) where \( i_k \) is the index of the parse tree node corresponding to \( X_k \).

For example, consider the parse tree fragment depicted in Figure 1.4.1 below.

![Parse Tree Fragment](image)

**Figure 1.4.1.** A Parse Tree Fragment.

The production for node 5 is

\[
p(5): X_0 \rightarrow X_1 X_2
\]
Therefore, attribute occurrence \((a,1)\) has a realization \((a,6)\)
because node 6 corresponds to \(X_1\) in the production \(p(i)\). The
advantage of using realizations is that they correspond to
"addresses" for the attributes being evaluated. To put it another
way, attribute occurrences are the formal parameters to semantic
functions, while attribute realizations are the actual parameters,
bound at evaluation time.

1.4.2. **Definition:** A **semantic instruction** is a call to a routine
to evaluate a particular semantic function \(r^P_{(a,k)}\) with dependency
set \(D^P_{(a,k)}\). The parameters to the semantic instruction are

- a. The **opcode** specifying which semantic function is to
  be evaluated.
- b. The **output**, the realization \((a,i_k)\) of the resultant
  attribute occurrence \((a,k)\).
- c. The **inputs**, realizations corresponding to each attri-
  bute occurrence in \(D^P_{(a,k)}\).

For example, consider production 2 from the example grammar dis-
cussed earlier (Figure 1.1.2.)

\[
S_1 \rightarrow AS_1^1B
\]

The third semantic function associated with that production is

\[
S_1.out \leftarrow B.out
\]

Suppose that the subtree fragment in Figure 1.4.2 has 2 as its
associated production:
The semantic instruction corresponding to the third semantic function would have

opcode:  (2,3)  - third function for production 2
output:  (out,4)  - corresponds to $S_1$.out
inputs:  (out,8)  - corresponds to B.out

The entire semantic instruction would be the triple: (opcode, output, inputs).

Semantic instructions will be executed on a semantic function machine (SFM) simulator generated by the constructor. The SFM will
be discussed in more detail later.

Each vertex in the augmented dependency graph will represent a unique attribute realization. Henceforth we will denote the vertex for realization \((a,i)\) by \(V(a,i)\). Each vertex \(v\) will have an associated semantic instruction \(SI(v)\) which computes the attribute realization for that vertex; e.g., \(SI(V(a,i))\) computes attribute \(a\) for tree node \(i\). Figure 1.4.3 depicts the modified dependency graph for the parse tree in Figure 1.2.2. The semantic instructions are not shown.

\[
\begin{align*}
&(in,2) \rightarrow (in,3) \rightarrow (out,3) \rightarrow (in,4) \rightarrow (in,6) \rightarrow (out,6) \\
&(in,4) \rightarrow (in,9) \rightarrow (out,9) \rightarrow (in,10) \rightarrow (out,10) \\
&(out,7) \rightarrow (in,8) \rightarrow (out,8) \rightarrow (out,4) \rightarrow (in,5) \rightarrow (out,5) \\
&(out,2) \rightarrow (ok,1)
\end{align*}
\]

Figure 1.4.3. Augmented Dependency Graph
(See Figure 1.2.2)

Actual construction of the dependency graph is done during a walk of the parse tree in which each node is visited exactly once; the order of visits is unimportant. When a parse tree node is visited, a plan provided by the constructor is executed. There is one plan for each production \(p\) in the attribute grammar, so the plan executed during a visit to node \(i\) is the plan the associated production \(p(i)\), denoted \(PLAN(p(i))\).

Execution of \(PLAN(p(i))\) performs these tasks:

1. For each semantic function \(f_{(a,k)}^{p(i)}\) associated with
production \( p(i) \), a vertex \( V(a,i_k) \) for the realization of the result attribute of that function is created and added to the graph.

(2) For each vertex created in step 1, the semantic instruction \( SI(V(a,i_k)) \), which computes the attribute realization for that vertex, is constructed and stored. This involves determining the realization of each attribute occurrence in the dependency set \( D_{p(i)}^{p(i)} \).

(3) For each semantic function \( f_{p(i)}^{p(i)} \) and for each edge \((b,j) \in D_{p(i)}^{p(i)}(a,k)\) an edge

\[(V(b,i_j),V(a,i_k))\]

is created and added to the dependency graph.

To illustrate this process, consider the parse tree fragment in Figure 1.4.2. The root is node 4 and its associated production is

\[2:S_1 \rightarrow AS_1'B\]

There are four semantic functions for this production (see Figure 1.1.2).

<table>
<thead>
<tr>
<th>opcode</th>
<th>semantic function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,1)</td>
<td>(A.in \rightarrow S_1.in)</td>
</tr>
<tr>
<td>(2,2)</td>
<td>(S_1'.in \rightarrow A.Sout)</td>
</tr>
<tr>
<td>(2,3)</td>
<td>(S_1'.out \rightarrow B.Sout)</td>
</tr>
<tr>
<td>(2,4)</td>
<td>(B.in \rightarrow S_1'.Sout)</td>
</tr>
</tbody>
</table>

When the plan is executed, it determines that the following correspondences exist:
non-terminal | node number
--- | ---
$S_1$ | 4
A | 6
$S'_1$ | 7
B | 8

Using these correspondences, the evaluator builds four nodes.

1. Attribute: (in.6)
   Instruction: $((2,1),(in,6),(in,4))$

2. Attribute: (in.7)
   Instruction: $((2,2),(in,7),(out,6))$

3. Attribute: (out,4)
   Instruction: $((2.3),(out,4),(out,8))$

4. Attribute: (in.8)
   Instruction: $((2,4),(in,8),(out,7))$

In addition, it adds edges representing the dependencies for each function.

$(V(in,4),V(in,6))$ for function 1
$(V(out,6),V(in,7))$ for function 2
$(V(out,8),V(out,4))$ for function 3
$(V(out,7),V(in,8))$ for function 4.

Formally, the execution of $\text{PLAN}(p(i))$ at node $i$ results in the construction of the dependency graph for $p(i)$.

$$\text{DG}_p(i) = (\text{DV}_p(i),\text{DE}_p(i)).$$

Since vertices and edges are added to a common set, the result after all tree nodes have been processed is
\[ \bigcup_{p \in \text{parse}} D\text{Gp}(i) = (\bigcup_{p \in \text{parse}} D\text{VP}(i), \bigcup_{p \in \text{parse}} D\text{EP}(i)) \]

But this is precisely the dependency graph for the whole tree.

Since plans will usually consist of high-level operations such as "find the index of the \( n \)th child of this tree node" or "create a vertex \( V(a,i_k) \) and insert the associated semantic function," plans should probably be passed from the constructor to the evaluator as instructions to a plan interpreter. When executed, these instructions would cause explicit calls to interpreter subroutines to perform the desired high-level operations. The specific form of the "plan language" is left to the implementor.

**Evaluation Sequence Generation**

Once the dependency graph is built, the evaluator produces a straight line program of semantic instructions to be executed during the final phase. This is done as follows. First, Knuth's "topological sort" [Kn1] is applied to produce a linear list

\[ (V_1, V_2, \ldots, V_m) \]

of the vertices in the dependency graph. The list produced is such that for any edge \((V_{j_1}, V_{j_2})\) in the dependency graph

\[ j_1 < j_2 \]

[Kn1].

Thus a vertex \( V_i \) representing a particular attribute realization \((a,i)\) can never precede the vertex representing an attribute \((b,j)\) on which \((a,i)\) depends; if the attributes are evaluated in the order of appearance of their vertices in the linear list, (left-to-right), then the required input attributes
will be available for each evaluation.

All that remains is to produce a straight-line program of semantic instructions; but this can be done by traversing the list of vertices from left to right, adding the semantic instruction for each vertex to the end of the program. The result is

\[(SI(v_1), SI(v_2), \ldots, SI(v_m))\]

which is the desired evaluation sequence.

**Evaluation Sequence Execution**

Phase 3 of the dynamic sequence evaluator simple executes the evaluation sequence generated in phase 2 on a semantic function machine (SFM) provided by the constructor. Since the address pairs for each attribute realization were coded into the semantic instructions during phase 1, the evaluation sequence provides all information needed to compute the attributes and store each one at its appropriate location in the parse tree. Given an instruction, the SFM would compute addresses in the tree of the inputs and outputs, load the addresses into standard index registers and branch to the code which executes the correct semantic function.

1.5 **The Constructor**

A constructor for dynamic sequence evaluators should perform three tasks:

1. Test for Circularity.
2. Generate a **Semantic function machine** (SFM).
3. Generate plans for constructing the dependency graph.

The outputs of phases 2 and 3 of the constructor can be combined with a skeleton evaluator, pre-programmed with common functions,
to produce the complete evaluator.

Figure 1.5.1. The Translator Writing System.

In practice, phases 2 and 3 of construction will probably be combined; however, it is convenient to separate them here for the purpose of exposition.

Circularity Testing

Any good constructor should analyze the input attribute grammar for circularity. There exists an algorithm, due to Knuth [Kn3], which performs this test; however, that algorithm has a time complexity which is potentially exponential in the size of the grammar. Thus we are faced with the possibility of very long running times in phase 1. Nevertheless, it would not be good practice to omit the test since that would allow the evaluator (which is part of a compiler) to discover circularities in parse trees presented to it. The average user would probably be a bit disturbed to read the compiler message:
ERROR -- ILL-DEFINED PROGRAMMING LANGUAGE

An acceptable compromise solution would be to use a stronger condition implying non-circularity but having a simpler test: for example, Bochmann's "left-to-right" condition \([B1,B2]\), or "absolute non-circularity" \([KW]\).

**SFM Generation**

Once the grammar has been verified, the constructor must produced a simulator for the semantic function machine. In practice, a skeleton SFM, containing code to bind attribute realizations to attribute occurrences and jump to the appropriate code sequence, will probably be included in the skeleton evaluator, All that is required of phase 2 is to generate the code sequence for each instruction.

Since the semantic functions will be "programmed" in some rudimentary language, the constructor simply compiles machine code into the SFM. For example, suppose the semantic function

\[
(2,3) \quad S_1.out \leftarrow B.out
\]

is being processed.

If we assume that the SFM binding code has stored the addresses of the starting locations of the variable length string arrays for the result and the input, respectively, in registers 1 and 2, then the following code is sufficient:

\[
(2,3): \text{MOVE string, with starting location pointed to by register 2, to array with starting location pointed to by register 1.}
\]
For sophisticated grammars with complex attributes such as "sets of code," more complicated code would be generated. We suggest machine code be directly produced since this will enhance the efficiency of the evaluator.

**Plan Generation**

In phase 3, the constructor builds a set of plans, one per production and stores them in a table indexed by production number. Since most of the operations in a plan are sophisticated (e.g., "add an edge to the dependency graph" or "find the address of the nth child of this tree node"), plans should be sequences of instructions to be executed by a "plan interpreter." The interpreter, complete with subroutines to perform the desired high-level operations, would be a part of the skeleton evaluator.

Assuming that the language in which semantic functions are programmed is sufficiently simple (say LR(k)), both phase 2 and phase 3 can be made to execute in time proportional to the length of the input string containing the attribute grammar [AU1]. Thus if a sufficiently simple test to guarantee non-circularity can be found, the constructor will be reasonably efficient, especially in view of the result it produces: a complete semantic evaluator.

1.6 **Computational Complexity**

The time and space complexities of the dynamic sequence evaluator are derived in this section. First, we adopt some uniform notation.
Terminology: Assume a fixed attribute grammar $G$, with numbered productions $p$, $0 \leq p \leq |P| - 1$; each production has $n_p + 1$ symbols (the extra symbol is $X_0$). The grammar $G$ has $|F|$ semantic functions

$$f_1, f_2, \ldots, f_{|F|}$$

Each semantic function $f_i$ has a time complexity, denoted

$$T(f_i),$$

a space complexity, denoted

$$S(f_i),$$

and a number of input arguments denoted

$$I(f_i).$$

Theorem 1:

If presented with a parse tree having $n$ nodes and $m$ attributes such that exactly $m_i$ attributes are computed by semantic function $f_i$, then the time complexity of the dynamic sequency evaluator is

$$O\left(\sum_{i=1}^{|F|} m_i (R(f_i) + I(f_i))\right)$$

Proof of Theorem 1:

We must analyze the complexity of each phase.

Phase 1: The dependency graph will have exactly $\sum_{i=1}^{|F|} m_i$ nodes, for each attribute in the tree. It will have an edge for each input attribute in a semantic function, or
\[
\sum_{i=1}^{\mid F \mid} m_i \times I(f_i)
\]

edges.

The time to construct the dependency graph is clearly linear in the number of vertices and edges so

\[
0 \left( \sum_{i=1}^{\mid F \mid} m_i \times (I(f_i) + 1) \right)
\]

steps are required.

Phase 2: The topological sort has been shown to be linear in the number of vertices and edges in the graph sorted, so once again

\[
0 \left( \sum_{i=1}^{\mid F \mid} m_i \times (I(f_i) + 1) \right)
\]

steps are required.

Phase 3: Each attribute is evaluated once, so

\[
0 \left( \sum_{i=1}^{\mid F \mid} m_i \times T(f_i) \right)
\]

steps are required in phase 3.

Summing these gives the desired result if we assume \( T(f_i) \geq 1 \) for all \( i \) so that the additive constant 1 disappears.

**Corollary to Theorem 1**

For a given attribute grammar, the time complexity of the
dynamic sequence evaluator is at worst

\[ 0(m(T_0 + I_0)) \]

where

\[ T_0 = \max_{1 \leq i \leq |F|} (T(f_i)) \]

and

\[ I_0 = \max_{1 \leq i \leq |F|} (I(f_i)) \]

The result is an obvious consequence of Theorem 1. It establishes that for a given attribute grammar, the evaluation time is linear in the size of the semantic parse tree. If all the semantic functions are simple, the evaluator should be quite fast.

**Theorem 2:**

Under the assumptions of theorem 1, the dynamic sequence evaluator has space complexity.

\[ 0 \left( \sum_{i=1}^{\left\lvert F \right\rvert} (S(f_i) + m_i I(f_i)) \right) \]

**Proof of Theorem 2:**

Phase 1: The space required for individual plans is proportional to the actual number of nodes and edges produced in the plan. The total for phase 1 is therefore proportional to the space for the dependency graph. The dependency graph, assuming semantic instructions are actually stored in the nodes, requires
\[ 0 \left( \sum_{i=1}^{F} m_i I(f_i) \right) \]

for node storage since an instruction for \( f_i \) must be long enough to hold \( I(f_i) \) inputs. A constant amount of space is required for each edge, but we have seen (in the proof of Theorem 1) that there are

\[ 0 \left( \sum_{i=1}^{F} m_i I(f_i) \right) \]

edges.

Thus phase 1 requires

\[ 0 \left( \sum_{i=1}^{F} m_i I(f_i) \right) \text{ steps}. \]

Phase 2: The required space is clearly proportional to the size of the dependency graph which was included in the computation for phase 1.

Phase 3: Space is required for each semantic function in the SFM or

\[ \sum_{i=1}^{F} S(f_i) \]

In addition, space for the evaluation sequence is needed in an amount equal to that for instruction storage in the dependency graph, which is no longer needed, can be used.
Thus the desired bound is established.

**Corollary to Theorem 2**

For a given attribute grammar the space complexity of the dynamic sequence evaluator is at worst

\[ O(|F| \times S_0 + mI_0) \]

where

\[ S_0 = \max_{1 \leq i \leq |F|} (S(f_i)) \]

and

\[ I_0 = \max_{1 \leq i \leq |F|} (I(f_i)) \]

Thus the space required is also linear in the size of the semantic parse tree.

1.6. **Conclusions**

We have described a method for constructing attribute grammar evaluators which, through dynamic generation of evaluation sequences, execute in time proportional to the size of the semantic parse tree. These evaluators are deterministic, so there is every reason to believe that they will have acceptable execution times when implemented.

An evaluator can be generated for any non-circular attribute grammar; however, the worst-case complexity of the test for non-circularity is so high that a stronger restriction on input grammars will probably need to be imposed. If the problem of circularity testing can be solved, the constructor- evaluator system should be suitable for inclusion in a general translator-writing system.
2. Global Data Flow Analysis of Structured Programs.

We introduce our treatment of global data flow algorithms by restricting ourselves to "structured" programs and leaving the consideration of "unstructured" programs for later chapters. Global data flow algorithms for structured programs have been considered by Rosen [R] and Jazayeri [J2] using an approach similar to ours. The nonterminals of a parse tree are used to provide an analysis of the flow graph which models the input string. However, the algorithms given here are specified using semantic functions which model very general control structures. Hence the formalism used here differs from the ones in [J2,R]. The advantage of using this formalism lies in the fact that it can be used to specify efficient algorithms for global data flow analysis. This is further discussed in the concluding section of this chapter. As is usual, the algorithms presented here will be based on the flow graph models of programs. Details regarding these models are given in Section 2.2. Section 2.2 gives a brief description of the interval analysis method for doing global data flow analysis. In Section 2.3 we introduce our method for modelling the control semantics of "structured" programs using attribute grammars. Using the appropriate model for a given program, that is using the control semantic tree, we present a linear attribute grammar algorithm to do global data flow analysis in Section 2.4.

2.1. Flow Graphs.

Flow graphs model program control transfer patterns which provide crucial information for the optimizing algorithms of a compiler. The program, expressed in some intermediate language, is scanned
and divided into basic blocks which are maximal sequences of instructions without branches. After the last statement in a block, control may transfer to any one of a number of blocks called successors of the block just executed. Each node of a flow graph represents a basic block and each edge represents a possible transfer of control between two basic blocks.

2.1.1. Definitions.

A flow graph is therefore a triple $G=(N, E, n_o)$ where

a) $N$ is a finite set of nodes

b) $E$ is a finite set of edges (a subset of $N \times N$), and

c) $n_o$ is the program entry node -- the unique node from which there is a "path" to every other node.

A path from node $x_1$ to node $x_k$ is a sequence of nodes $(x_1, x_2, \ldots, x_k)$ such that $(x_i, x_{i+1})$ is in $E$ for $1 \leq i < k$. The path length of $(x_1, \ldots, x_k)$ is $k - 1$. We say that the path above has a cycle if $x_i = x_j$ for $1 \leq i < j \leq k$. A simple path is a path which has no cycles.

For $(x, y) \in E$ we define $0(x, y)$, the origin of edge $(x, y)$, to be $x$ and $T(x, y)$, the target of edge $(x, y)$ to be $y$. This notion can be extended to paths. Let $p = (x_1, x_2, \ldots, x_k)$ be any path in $G$. The target of $p$, $T(p)$, is $x_k$ and the origin of $p$, $0(p)$ is $x_1$.

For each node $x$ in $N$ we define

$E_0(x) = \{(x, y) | (x, y) \in E\}$, the set of edges with origin $x$

$E_T(x) = \{(y, x) | (y, x) \in E\}$, the set of edges with target $x$

$p(x) = \{y | (x, y) \in E\}$, the set of predecessors of $x$
\[ s(x) = \{ y/(x, y) \in E \} \], the set of successors of \( x \).

Some of our analysis of flow graphs will be based on a partial ordering of the nodes in each graph. The relationship is defined below.

If \( x, y \) are in \( N \) and every path \( p \) from \( n_o \) to \( y \) contains \( x \) then we say that \( x \) dominates \( y \) in \( G \). The dominator tree of \( G \), denoted as \( DT \), is defined to be such that there is a path from \( x \) to node \( y \) in \( DT \) iff \( x \) dominates \( y \). The root of \( DT \) is \( n_o \).

\[ x \xrightarrow{\ast}{DT} y \] denotes a path of length greater than or equal to 0 from \( x \) to \( y \) in \( DT \).

\[ x \xrightarrow{1}{DT} y \] denotes a path of length greater than zero from \( x \) to \( y \) in \( DT \).

\[ x \xrightarrow{i}{DT} y \] denotes a path of length equal to \( i \) from \( x \) to \( y \) in \( DT \).

For any node \( x, x \neq n_o \), the immediate dominator of \( x \) in \( G \) is the node \( y \exists y \xrightarrow{1}{DT} x \). We shall denote this relationship as \( \text{id}(x) = y \). Every node in the dominator tree dominates itself.

The notation above can be generalized to any acyclic flow graph \( T \). A path from node \( x \) to node \( y \) in \( T \) will be denoted as \( x \xrightarrow{\ast}{T} y \). Finally, by convention \( \text{id}(n_o) = 0 \).

2.2. Live-Dead Analysis.

In this section we give a brief description of the solution to the live variable problem for a program based on the interval analysis of the program's flow graph. This topic is covered in detail by Kennedy [Ke2].

Let \( G \) be a flow graph and \( n \) a node of \( G \). The (maximum)
interval with header \( h \), denoted by \( I(h) \), is constructed by the following algorithm due to Cocke and Allen [CA1].

2.2.1. Algorithm A. Maximum Interval Construction

**Input:** Flow graph \( G \) and designated node \( h \).

**Output:** The set of nodes \( I(h) \).

**Notation:** If \( N_1 \) is a set of nodes \( \exists N_1 \subset N \), then
\[
S[N_1] = \bigcup_{x \in N_1} S(x) \quad \text{and} \quad P[N_1] = \bigcup_{x \in N_1} P(x).
\]

**Method:**

```
BEGIN

I(h) := \{h\};

\# Iteratively add nodes that have all their predecessors in \( I(h) \).

WHILE \( \exists x \in S[I(h)] - I(h) \)

such that \( P(x) \subseteq I(h) \)

DO

\( I(h) := I(h) \cup \{x\} \)

OD

END
```

We note here that the order in which nodes are added to \( I(h) \) will be important later and is called the interval order. Interval order is by no means unique (in fact, it depends on the order in which successors of \( I(h) \) are considered); however, it imposes a total order on the nodes of \( I(h) - \{h\} \) which preserves the partial order defined by the successor relation. The result is that if we process the nodes of \( I - \{h\} \) in interval order, we will process a node only after we have processed each of its predecessors in \( I \), and if we process in reverse interval order, we will process a node only after we have processed each of its successors in \( I - \{h\} \).
We now give algorithm B, due also to Allen and Cocke [AC1], which partitions the entire flow graph into a set of disjoint intervals.

2.2.2. Algorithm B. Interval Partition

**Input:** Flow Graph \( G = (N, E, n_o) \)

**Output:** The set of intervals \( \text{INTS}(G) \)

**Method:** We use a set \( H \) of interval headers.

BEGIN  
  #the program entry node is a header#
  \( H := \{ n_o \} \);
  \( \text{INTS}(G) := \emptyset \);
  \( H \neq \emptyset \) #there are more headers#
  DO
    \( x := \) an arbitrary node in \( H \) which has not been used as a header.
    \( H := H - \{ x \} \);
    Find \( I(x) \) using Algorithm A;
    \( \text{INTS}(G) := \text{INTS}(G) \cup \{ I(x) \} \)
    # Successors of \( I(x) \) which cannot be added are headers #
    \( H := H \cup (S[I(x)] - I(x)) \)
  OD

END

As an example of this process consider the flow graph in Figure 2.2.1.

The interval heads chosen by Algorithm B (2.2.2) will be 1, 2, 5 and the intervals are \( \{1\}, \{2,3,4\} \) and \( \{5,6,7\} \).
2.2.3. **Definition**

If $G$ is a flow graph, then the derived flow graph of $G$, denoted by $D(G)$, is defined as follows:

a) The nodes of $D(G)$ are the intervals of $INTS(G)$.

b) If $J, K$ are two intervals, there is an edge from $J$ to $K$ in $D(G)$ if and only if there exist nodes $n_j \in J$ and $n_k \in K$ such that $n_k$ is a successor of $n_j$ in $G$. Note that $n_k$ must be the header of $K$.

c) The initial node of $D(G)$ is $I(n_0)$.

We call $G$ the underlying flow graph of $D(G)$.

The sequence $(G_0, G_1, \ldots, G_m)$ is called the derived sequence for $G$ if $G = G_0$, $G_{i+1} = D(G_i)$ and $G_i \neq G_{i+1}$ for $0 \leq i < m$. Also $D(G_m) = G_m$. $G_i$ is called the derived graph of order $i$ and $G_m$ is the limit flow graph of $G$. A flow graph is said to be reducible if and only if its limit flow graph is the trivial flow graph, a single node with no edges; otherwise, the flow graph is irreducible. Throughout this presentation we shall deal only with
Reducible flow graphs.

**Live Analysis Problem** [Ke2]

A path in a flow graph $G$ is said to be **definition clear** with respect to variable $x$, or $x$-clear, if there is no assignment statement to $x$ in any node on that path. A variable $x$ is **live** at point $p$ in the flow graph if there exists an $x$-clear path from $p$ to a use of $x$. If there is no such path, $x$ is said to be **dead** at $p$. Thus, $x$ is live at $p$ if its value at $p$ may be used before it is redefined.

With efficiency considerations in mind, we will henceforth think of the flow graph nodes as representing extended* basic blocks. Our aim is to compute, for each node $b$ in the control flow graph, the set $\text{LIVE}(b)$ of all variables which are live on entry to $b$. Any global sets $\text{LIVE}(b)$ can be defined in terms of two sets which contain strictly local information. Given an extended basic block $b$ in the flow graph:

a) $\text{IN}(b)$ is the set of variables such that for $X \in \text{IN}(b)$ there is an $x$-clear path from the entry of $b$ to a use of $x$ within $G$; i.e., a use of $X$ appears in $G$ before any redefinition of $X$.

b) $T(b,c)$, defined for every successor $c$ of $G$, is the set of variables $X$ for which there is an $x$-clear path through $b$; i.e., the set of variables which are not defined in $b$ on the path from $b$ to $c$.

2.2.4. **Theorem.** Let $G=(N,E,n_0)$ be a flow graph. For any $b \in N$,

*Extended basic block is a tree of basic blocks. See [Ke2] for details regarding the usefulness of this definition.
\[
\text{LIVE}(b) = \text{IN}(b) \cup \bigcup_{y \in S(b)} (T(b,y) \cap \text{LIVE}(y))
\]

Proof. (As in Theorem 1 of [Ke1])

The interval method consists of two parts. First, the sets \( \text{IN} \) and \( T \) are computed for the intervals of the derived graph, then for intervals of the next derived graph and so on until these sets are available for the trivial limit graph. Then the equation of theorem 2.2.4 is applied to the highest order derived graph, then to the nodes of its underlying graph and so on until \( \text{LIVE}(b) \) has been computed for each node in the original graph. Thus there are two passes through the nodes and edges of the derived sequences.

We now present the input-output behavior of two algorithms that are called upon by the two part algorithm for doing live analysis using the interval method. The \( \text{IN}, T \) and \( \text{LIVE} \) sets are implemented using bit vectors.

2.2.5. **Algorithm C: \( \text{IN} \) and \( T \) for an Interval**

**Input**
1. The nodes of an interval \( I \) numbered from 1 to \( n_I \) in interval order
2. Successor and predecessor information for each node in the interval
3. The set of successor intervals \( J \) of \( I \) (successors in the derived graph) along with their headers \( h_J \).
4. \( \text{IN}(j), \ 1 \leq j \leq n_I \).
5. \( T(j,k), \ 1 \leq j \leq n_I, \ k \) ranging over all successors of \( j \).

**Output** \( \text{IN}(I) \) and \( T(I,J) \) for each successor \( J \) of \( I \) in the derived graph.

**Method** As in [Ke2].

2.2.6 Algorithm D  Live computation using Intervals.

**Input**  
(1) An interval $I$ with nodes numbered from 1 to $N_I$ in interval order.
(2) LIVE($I$)
(3) All successors $J$ of $I$ with their headers $h_J$
(4) LIVE($J$) for successors $J$ of $I$
(5) IN($j$), $1 \leq j \leq n_I$
(6) $T(j,k)$, $1 \leq j \leq n_I$, $k$ ranging over all the successors of $j$.

**Output**  
LIVE($j$), $1 \leq j \leq n_I$.

**Method**  
As in [Ke2]

We now present the complete algorithm to perform live analysis using the interval method.

2.2.7. Algorithm E  Live Analysis.

**Input**  
(1) A reducible flow graph $G = (N,E,n_o)$
(2) The derived sequence $(G_0,G_1,...,G_m)$ for $G$, where $G = G_0$ and $G_m$ is the trivial flow graph. $G_i = (N_i,E_i,n_i^o)$.
(3) IN($b$) for every $b \in N$
(4) $T(b,k)$ for every $(b,k) \in E$
(5) An interval order listing of the contents of each interval

**Output**  
LIVE($b$) for every $b \in N$.

**Method**

```
BEGIN  #Iterate through the derived sequence in inner-to-outer order#
FOR i = 1 to m
```

DO
   F1: FOR each node \( x \) in \( G_i \) \# \( x \) is an interval \#
   DO
      Apply Algorithm C to compute IN and T vectors for \( x \)
   OD
OD
   \# Compute LIVE for the whole program \#
   \[ \text{LIVE}(N^m_o) = \text{IN}(N^m_o) \]
   \# Iterate through the derived sequence in outer-to-inner order \#
   FOR \( i = m \) TO 1 by -1
   DO
      F2: FOR each node \( x \) in \( G_i \) \# \( x \) is an entered \#
      DO
         Apply Algorithm D to compute LIVE vectors for nodes contained in \( x \)
      OD
   OD
END

2.2.8 **Theorem** [Ke2] Algorithm E terminates and is correct.

In step F1, whenever Algorithm C is applied to an interval, the T and IN vectors for nodes of that interval have been computed, because F1 is applied to the derived graphs in increasing order. In step F2, whenever Algorithm D is applied to an interval LIVE vectors have been computed for entry to that interval and its successor intervals because F2 is applied to the derived graphs in decreasing order (i.e., the limit graph first, then its underlying graph, and so on).
2.3 Control Semantics

As mentioned earlier, the flow graph modelling a program depicts the control semantics of the program. We show here how to use attribute grammars as a means of specifying the control semantics associated with the syntax of structured programs. Only certain semantic aspects of these attribute grammars will be examined here; we shall isolate and study the semantics associated with the control transfers of a given program.

2.3.1 Definition. We augment control semantic functions to the productions of \( AG = (CFG, A, F) \) as follows. For each production \( p \in P \) (\( P \) is the set of productions of CFG) such that \( p: X_0 \rightarrow X_1, \ldots, X_n \)

the control semantic attribute \( X_i.cs \) is contained in \( I(X_i) \), for \( i = 1, 2, \ldots, n \).

The control semantics associated with production \( p \) is a flow graph \( CS(p) = (V, E, X_1.cs) \) where

1) \( V = \{ X_1.cs, X_2.cs, \ldots, X_n.cs \} \), the set of nodes

2) \( E = \{ (X_i.cs, X_j.cs)/X_1.cs, X_j.cs \in V \} \), the set of edges

3) \( X_1.cs \) is the unique start node.

\( CS(p) \) is required to be reducible in this exposition though this restriction is not essential to the approach presented here.

Each node \( X_i.cs \) can be thought of as representing some sequence of intermediate code generated by other semantic functions; however, details of these functions and the generated code are not dealt with here.
\[ V_N = \{Pr, \text{Blocklist}, \text{Block}\} \]
\[ V_T = \{\text{BEGIN}, \text{END}, \text{WHOLE}, \text{DO}, \text{OD}, \text{RELATION}, \text{STATEMENT LIST}\} \]

Productions of CFG

1. \( Pr \rightarrow \text{BEGIN} \text{ Blocklist } \text{END} \) \hspace{1cm} cs(1): \text{BEGIN.cs} \rightarrow \text{Blocklist.cs} \rightarrow \text{END.cs}
2. \( \text{Blocklist} \rightarrow \text{Blocklist} \text{ Block} \) \hspace{1cm} cs(2): \text{Blocklist.cs} \rightarrow \text{Block.cs}
3. \( \text{Blocklist} \rightarrow \text{Block} \) \hspace{1cm} cs(3): \text{Block.cs}
4. \( \text{Block} \rightarrow \text{WHILE RELATION DO} \) \hspace{1cm} cs(4): \text{WHILE.cs} \rightarrow \text{Relation.cs} \rightarrow \text{DO.cs} \rightarrow \text{Blocklist.cs OD.cs}
5. \( \text{Block} \rightarrow \text{STATEMENT LIST} \) \hspace{1cm} cs(5): \text{Statement list.cs}

Figure 2.3.1 Attribute Grammar with Control Semantics.

In Figure 2.3.1 we give an example of an attribute grammar, AG, with a control semantic flow graph associated with each production of its context free grammar. The control semantic flow graph of any production defines a relation on the cs attributes of the symbols on the right hand side of the production. This relation expresses possible transfers that may occur between blocks of generated code. For example, cs(4) in Figure 2.3.1 contains an edge from Blocklist.cs to Relation.cs. This edge depicts the possible transfer of control from the end of the code sequence generated for Blocklist to the beginning of the code sequence generated for Relation.

In order to retain the simplicity of our example we consider the symbols "Relation" and "Statement list" to be terminal symbols.

The control semantic tree illustrated in Figure 2.3.2 is formally defined below.

2.3.2 Definition As in Figure 2.3.2a, suppose the parse tree for a string \( \alpha \) is indexed in some way. cs(p(i)) denotes the control semantics associated with the production applied at node i of the
Figure 2.3.2a. Control Semantic Tree, $CST(\alpha)$.

Figure 2.3.2b. Control Flow Graph of String $\alpha$ in Figure 2.3.2a.
parse tree. That is, \( cs(p(i)) = (N(p(i)), E(p(i)), (cs,i_1)) \)

where \( N(p(i)) = \{ (cs,i_1), \ldots, (cs,i_p) \} \) (refer to 1.4.1 for notation)
\( E(p(i)) = \{ ((cs,i_x), (cs,i_m)) \mid (cs,i_x), (cs,i_m) \in E(p(i)) \} \) is an edge in
\( cs(p) \).

CST(\( \alpha \)), control semantic tree of \( \alpha \), is the parse tree with
each node \( i \) as a structured variable whose field selectors are
attributes associated with NT(i). In particular, each node \( i \)
contains a field for \( (cs,i) \) which is a sequence of pointers.

For the present, ignoring the details of the remaining fields, we
require that, for each node \( i \), the field \( (cs,i_x) \) contain a
pointer to \( (cs,i_m) \) if \( ((cs,i_x), (cs,i_m)) \in E(p(i)) \). That is,
the control semantic tree contains control transfer information in
the \( cs \) attributes of its non-terminal symbols.

The control flow graph modelling a string \( \alpha \) can be obtained
directly from CST(\( \alpha \)). Label the levels of CST(\( \alpha \)) in a root to
leaf order. Suppose there are \( m \) levels. We obtain a sequence
\( G_1, G_2, \ldots, G_m \) such that \( G_m \) is the control flow graph of \( \alpha \), and,
in some sense, \( G_{i+1} \) is the underlying flow graph of \( G_i \). Let
\( G_i = (N_i, E_i, n_i) \). \( G_{i+1} \) will be obtained by 'replacing' certain
nodes of \( G_i \) by the semantic control regions associated with pro-
ductions applied at these nodes.

Let \( j_1, j_2, \ldots, j_{k_i} \) be the indices of the nodes, at level \( i \) of
the parse tree, at which productions are applied to obtain the nodes
at level \( i+1 \). (See Figure 2.3.2a). Note \( \{ j_1, \ldots, j_{k_i} \} \subseteq N_i \).

\( G_{i+1} \) is the graph \( G_i \) transformed as follows. Replace the nodes
\( j_1, \ldots, j_{k_i} \) by the corresponding semantic flow graph regions
\( cs(p(j_1)), \ldots, cs(p(j_{k_i})) \). Then,
Figure 2.3.3a. Control Semantic Tree
Figure 2.3.3b. \( G'_6 \) is the Control Flow Graph of String Generated in Figure 2.3.3a.
\[ N_{i+1} = \{ N_i \setminus \{ j_1, j_2, \ldots, j_{k_i} \} \} \cup \bigcup_{\ell=1,2,\ldots,k_i} N(p(j_{\ell})). \]

The edges of \( E_{i+1} \) are the edges of \( E_i \) modified in the following ways. If there is an edge \((X, (cs, j_{\ell}))\) in \( E_i \) it is replaced by the edge \((X, (cs, j_{\ell}^{-1}))\), and if there is an edge \(((cs, j_{\ell}^{-1}), X)\) it is replaced by \(((cs, j_{\ell}^{-1}), X)\). \( E_{i+1} \) is the set obtained by replacing all possible edges in \( E_i \) and, in addition, each edge \(((cs, j_{m}^{-1}), (cs, j_{n})^{-1})\) in \( E(p(j_{\ell})) \), for \( 1 \leq \ell \leq k_i \), is placed in \( E_{i+1} \).

Figures 2.3.2b and 2.3.3b illustrate the above concepts, using the control semantic trees, in Figures 2.3.2a and 2.3.3a.

In other words, the graph \( G_{i+1} \) is obtained by replacing each node \( j_{\ell} \) in \( G_i \), \( 1 \leq \ell \leq k_i \), by a single entry, single exit, reducible region \( cs(p(j_{\ell})) \). We remark, without proof, that the graph \( G_{i+1} \) is a reducible flow graph, if \( G_i \) is reducible. Therefore, we can conclude inductively that \( G_m \) is reducible.

As discussed earlier, the nodes of \( G_m \) can be thought of as representing sequences of intermediate code generated by other semantic functions. A slight modification must be made to \( G_m \) before its nodes can be understood to represent maximal basic blocks. \( G_m \) might contain nodes \( i \) and \( j \), each node representing a code sequence with no branch instructions, such that the only successor of \( i \) is \( j \) and the terminal substring associated with \( i \) immediately precedes the terminal substring associated with \( j \) in \( \alpha \). Let \( G'_m \) denote the graph \( G_m \) with each pair of nodes exhibiting the above property 'collapsed' to a single node. We illustrate our informal discussion above with an example in Figures 2.3.3a and 2.3.3b.
Intuitively, we can view the 'collapsing' as a process of removing edges which convey trivial control information—the implicit transfer from any instruction in the code sequence to only the instruction which physically follows it. With a little further modification the nodes of $G'_m$ can be used to model extended basic blocks of $\alpha$.

The control semantic tree contains a detailed analysis of the flow graph for string $\alpha$. We shall indicate a technique, using this analysis, to do global data flow analysis in the next section.

2.4 Live Analysis Algorithm.

In this section we shall show how to augment semantic functions to an attribute grammar $AG$ for carrying out live variable analysis. The algorithm so specified is based on the control semantic trees of the strings generated by the context free grammar. Figure 2.3.2a illustrates how the control semantic tree for $\alpha$ provides information similar to that contained in an interval analysis of the flow graph modelling $\alpha$. The above remark can be made clearer with the following observation. If $p(i) : X_0 \rightarrow X_1, \ldots, X_n$ is the production applied at the parse tree node $i$, $(cs,i)$ is the composite node denoting the single entry, single exit region $cs(p(i))$. For example, $(cs,6)$ denotes the region $cs(p(6))$ in Figure 2.3.2a.

We shall assume that there are semantic functions augmented to the productions of $AG$ such that these functions simulate the traversals of the parse tree required by the algorithms given below. Refer to Section 3.2.1 for a detailed example which contains semantic functions to perform a similar task—traverse the parse tree in reverse depth first order.

The additional attributes, defined for the symbols $AG$, for the
purpose of doing live variable analysis is as follows:

For \( X \in V \), \( S(X) = S(X) \cup \{X.IN, X.T\} \).

For start symbol \( S \), \( S(S) = S(S) \cup \{S.LIVE\} \).

For \( X \in V - S \), \( I(X) = I(X) \cup \{X.LIVE\} \).

\( X.T \) can be thought of as an attribute with several components, each component representing thru information from \( X \) to some successor of \( X \) in the parse tree. The precise nature of these components depend on the occurrence of \( X \) in \( CST(\alpha) \). The attributes \( X.IN \), \( X.T \) and \( X.LIVE \) of the symbol \( X \) can be implemented as bit-vectors.

The two part algorithm given next is a generalization of the one given in [FRZ].

2.4.1 Algorithm F  Live analysis algorithm using an attribute grammar

Input  (1) \( CST(\alpha) \), for some \( \alpha \in L(CFG) \), with the levels indexed in a root to leaf order and the interior nodes indexed in a root-to-leaf, left-to-right order.

(2) For each level \( i \), \( LEFT(i) = \) the index of the leftmost interior node at level \( i \) and \( RIGHT(i) = \) the index of the rightmost interior node at level \( i \).

(3) \( IN \) and \( T \) attributes of each leaf node initialized by a preprocessor.

Output  (1) \( CST(\alpha) \) with attributes \( IN,T,LIVE \) computed for each symbol

#Bookkeeping  \( K = \) of levels in the semantic tree.#

#Part 1: Inside-out portion of the algorithm which computes the IN and \( T \) set of a composite node using the \( IN \) and \( T \) sets of its
component nodes. \( \text{CST}(\alpha) \) is traversed in a leaf to root, left to right order.

\[
\text{FOR } I = K - 1 \text{ STEP } -1 \text{ TILL } 1
\]

\[
\text{DO FOR } J = \text{LEFT}(I) \text{ STEP } 1 \text{ TILL } \text{RIGHT}(I)
\]

\[
\text{DO } \#\text{Select appropriate semantic function, if any. } J
\]

\[
is \text{ a composite node}\#
\]

\[
\text{IF } J \text{ is an interior node THEN}
\]

Use Algorithm \( G \) to compute \((IN,J),(T,J)\) using the values for \((IN,J_1),(T,J_1),(IN,J_2),(T,J_2),\ldots,\)

\[
(IN,J^n_{p(J)}),(T,J^n_{p(J)})
\]

\[
\text{OD}
\]

\[
\text{OD}
\]

\[
\#\text{Compute LIVE for the whole program}\#
\]

\[
(LIVE,1) = (IN,1)
\]

\[
\#\text{Part II: Outside-in portion of the algorithm which computes the}
\]

LIVE sets of the components of a composite node using the LIVE set for this node.

\[
\text{CST}(\alpha) \text{ is traversed in a root to leaf, right to left order}\#.
\]

\[
\text{FOR } I = 1 \text{ STEP1 } \text{TILL } K - 1
\]

\[
\text{DO FOR } J = \text{RIGHT}(I) \text{ STEP } -1 \text{ TILL } \text{LEFT}(I)
\]

\[
\text{DO } \#\text{Select appropriate semantic function, if any}\#
\]

\[
\text{IF } J \text{ is an interior node THEN}
\]

Use Algorithm \( H \) to compute

\[
(LIVE,J_1),(LIVE,J_2)\ldots(LIVE,J^n_{p(J)})\text{ using the value of}
\]

\[
(LIVE,J) \text{ and the values of } (LIVE,K),
\]

\[
K \text{ is a successor of } J \text{ at level } I .
\]

\[
\text{OD}
\]

\[
\text{DO}
\]
We now describe the input-output behavior of the algorithm G for the in-out phase, which implements the semantic function associated with production $p$, $p \in P$.

2.4.2 **Algorithm G.** Semantic function for $p$, $p \in P$.

**Input**
- (0) $CS(p(J))$
  - (1) Bit vector values for $(IN,J_1),(T,J_1)$, $(IN,J_2),(T,J_2)$, ..., $(IN,J_p),(T,J_p)$.

**Output**
- Bit vector values for $(IN,J),(T,J)$.

**Method**
Apply any standard live analysis algorithm to $CS(p(J))$ to obtain values for $(IN,J),(T,J)$.

The input-output behavior of algorithm H, for the out-in phase, for any $p \in P$ follows.

2.4.3 **Algorithm H.** Semantic function for $p$, $p \in P$.

**Input**
- (0) $CS(p(J))$
  - (1) $(LIVE,J)$
  - (2) $(LIVE,K)$ for each successor $K$ of $J$.

**Output**
- $(LIVE,J_1),(LIVE,J_2), ..., (LIVE,J_p)$.

**Method**
Apply any standard live analysis algorithm to $CS(p(J))$ to obtain the output.

It should be clear that the Algorithms G and H function for Algorithm F as Algorithms C and D (2.2.5 and 2.2.6) do for Algorithm E (2.2.7). In fact Algorithm F simulates Algorithm E. The proof of correctness for algorithm F follows directly from this observation and Theorem 2.2.8.

The Algorithms G and H can be written using the most efficient
method known for computing the LIVE, IN and T sets for nodes in
CS(p). The time required to analyze each CS(p) and determine values
for these sets is some constant. We also note that for LRK grammars
the size of the parse tree is linearly related to the size of the
input string. It follows that the time required for doing live
analysis is also linearly related to the size of the input string.

Though we have restricted ourselves to a specific problem--
live variable analysis--the methods described here should work for
other global data flow problems. We have dealt with only structured
programs in this chapter. That is, programs for which control
semantics can be defined in terms of locally occurring symbols in
the parse tree.

2.5 Conclusions.

The control semantic tree provides substantial information about
any input program. It models the control transfers of the program,
and, at the same time, it provides an analysis of the control trans-
fers. This control analysis is used by the semantic functions of
the attribute grammar, which do live variable analysis. More
specifically, during evaluation time, the input to the algorithms
G and H consists of

a) CS(p(J)) and

b) IN, T or LIVE attributes associated with the nodes of
CS(p(J)).

Algorithms G and H determine the output using a standard live
analysis technique [Ke1, Ke2, FKZ] on CS(p(J)). That is, the
implementation details of G and H are independent of the
syntactic symbols (especially keywords) in the production p(J).
That is, the generality introduced by control semantic graphs allows us to write efficient semantic algorithms for live analysis in a uniform way for any arbitrary region $CS(p)$. This is not possible using the formalism given in [J2, R].

Algorithms $G$ and $H$ can be made particularly efficient using the following trick. If $G$ or $H$ is called with argument $p$ for the first time, a parse of $CS(p)$, needed as a first step for its live analysis, is stored in a list pointed to by a non-zero entry in the $p^{th}$ element of a table. Subsequent calls on $G$ or $H$ with the same argument $p$ causes this parse to be retrieved thus avoiding the parsing pass of the live variable algorithm on $CS(p)$.

During the evaluation time (see Chapter 1), the evaluator causes algorithms $G$ and $H$ to be called, at certain nodes of the parse tree, with the actual control structure occurring at each of these nodes as the argument. Therefore, the complexity of doing global data flow analysis for a program depends only on the control structures occurring in the control semantic tree of the program.

The need for varied control structures arises quite routinely particularly when attempting to write efficient programs [Kn2]. The exact nature of these control structures, however, is not well understood. Some attempts have been made in this direction, notably in [KZ]. Here we have presented a method for the automatic implementation of optimizing, language translators which allows experimentation with varied control structures.
3. Analysis of Flow Graphs of Unstructured Programs.

In Chapter 2 we used an attribute grammar for specifying part of an optimizing compiler's translation process in terms of the properties of locally occurring symbols in the parse tree of any structured program. In this chapter we present a method for analyzing the flow graphs underlying unstructured programs using attribute grammars. This analysis is used by the optimizing algorithms given in Chapter 4. Unlike the situation for structured programs, the control semantics of an unstructured program is not related to its parse tree symbols in a 'straightforward manner'. There is a choice of approaches which may be taken by an optimizing translator for relating the syntactic pattern and the control semantic pattern of an unstructured program.

1) The syntactic structure of the program, that is the parse tree, can be retained. The synthesized, control semantic information can then be passed to various nodes of the parse tree via attributes and be made available for global data flow analysis.

2) Alternatively, the control flow graph can be synthesized by traversing the parse tree. The semantic structure underlying this flow graph, the dominator tree, can be used as a basis for the attribute grammar doing the global data flow analysis. The analysis can be carried out by passing local syntactic information via attributes to the various nodes of the dominator tree.

We shall follow the latter approach here.
In Section 3.1 we present some basic material on the characterizations of reducible flow graphs. Section 3.2 presents an algorithm for constructing the dominator tree of a flow graph and testing the flow graph for reducibility by examining certain attributes of the dominator tree nodes. This algorithm is somewhat more efficient than the one presented in [T2], as is indicated by the discussion in the concluding section 3.3.

3.1 Characterizations of Reducible Flow Graphs.

In this section we shall state some of the relevant characterizations of reducible flow graphs presented in [HU3]. First we introduce some basic material.

3.1.1 Definition. Let \( G = (N, E, n_0) \) be a flow graph. \( \text{DFST}(G) \) is \( G \) with its nodes numbered according to the depth first algorithm in [AHU]. Each edge is labelled as a tree edge, a back edge, a cross edge or a forward edge. This algorithm is linear in the size of the input graph \( G \). The depth first spanning tree underlying \( G \) is used as a basis for defining the following sets (refer to Ch. 5 of [AHU] for further details).

- \( B \): the set of back edges
- \( C \): the set of cross edges
- \( F \): the set of forward edges, and
- \( T \): the set of tree edges.

\( \text{ST}(G) \), the spanning tree of \( G \), is defined as the 2-tuple \( (N, T) \) where \( N \) is the set of nodes of \( G \) and \( T \) is the set of tree edges.

\( \text{ST}(v) \) is the subtree of \( \text{ST}(G) \) with root \( v \), \( v \in N \).
∀v∈N, BO(v) is the set of back edges with origin node u.
BT(v) is the set of back edges with target node u.
FO(v) is the set of forward edges with origin in node u.
FT(v) is the set of forward edges with target node u.
CO(v) is the set of cross edges with origin node u.
CT(v) is the set of cross edges with target node u.
STP(v) is the predecessor of v in the spanning tree ST(G).
STS(v) is the set of successors of v in the spanning tree ST(G).

∀v∈N, DAG(v), the directed acyclic graph with root v is the 2-tuple \((N_v,E_v)\) where \(N_v=\{u/u\text{ is a node of } ST(v)\}\)
\[E_v=\{(x,y)/(x,y)\in UF UT \text{ and } x,y\in N_v\} .\]

We now present two relevant characterizations of reducible flow graphs from [HU3].

3.1.2. **Lemma.** A flow graph G is reducible if and only if the dominance relations of \(\text{DAG}(1)^+\) and G are the same.

3.1.3. **Lemma.** A flow graph G is reducible if and only if its edge set can be partitioned into two sets \(E_1\) and \(E_2\) such that \(D=(N,E,n_0)\) is the DAG of G and each \((n,m)\) in \(E_2\) has \(n=m\) or \(m\) dominates \(n\) in G.

3.2. **Algorithm for the Reducibility Test.**

We shall assume here that the start symbol of a parse tree has as one of its synthesized attributes the control flow graph for the program. This attribute can be constructed in stages. First, using techniques in Chapter 2, a flow graph for the well structured control

\(+\text{DAG}\) will be used to denote \(\text{DAG}(1)\).
statements in the program is synthesized as an attribute of the parse tree start symbol. Information regarding unstructured control statements (i.e., GOTO's) is synthesized into tables and also made accessible at the parse tree start node. This information, concerning random transfers, is used to refine the flow graph, depicting well-behaved transfers, so that it now depicts the control pattern of the entire program.

In this section we give details of semantic functions which analyze the synthesized control flow graph. This control flow graph is first analyzed by a depth first numbering algorithm and re-organized as a depth first spanning tree augmented by certain other edges. The depth first spanning tree of the flow graph is the input to the semantic function, implemented as an attribute grammar, which determines its underlying dominator tree. This dominator tree then becomes the input for yet another semantic function, also implemented as an attribute grammar, which examines attributes of its nodes and determines the reducibility of the original flow graph.

In the next chapter we shall use dominator trees as the structures on which to base attribute grammars for doing data flow analysis. As mentioned before, we will be dealing only with reducible flow graphs. Hence the interest in testing a flow graph for reducibility in this chapter.

All implementations using attribute grammars can be greatly facilitated by using global attributes--attributes accessible from each node of a relevant structure.

We now give a brief description of the reducibility test which is implemented as a two part algorithm.
Part 1. Implements the 2-pass algorithm given in [AHU] for finding the dominator tree for $\text{DAG}(1)$ of the given flow graph $G=(N,E,n_0)$.

Part 2. Implements a multi-pass algorithm to traverse the dominator tree and examine the origin and target nodes of back edges and test for reducibility.

The advantage of the characterization of reducible flow graphs given in Lemma 3.1.2 lies in the fact that we can ignore the back edges when constructing the dominator tree for a reducible flow graph. The algorithm outlined above begins with the assumption that the input flow graph is reducible and constructs the dominator tree for $\text{DAG}(1)$ of $G$. It then traverses the dominator tree and uses the characterization, given in Lemma 3.1.3, to test $G$ for reducibility. Some of the attribute grammars, presented in this section, specify that the nodes of a tree be visited in reverse depth first order. These attribute grammars will be embellished versions of the following attribute grammar which specifies the reverse depth first traversal of some tree $\text{Tr}=(N,E,n_0)$.

Suppose the nodes of $\text{Tr}$ are indexed in some manner. We shall think of $\text{Tr}$ as a tree which can be obtained by repeatedly applying productions of the form

a) $B_{l, i} \rightarrow B_{j_1} \ldots B_{j_k}$. That is, if a node $v$, $v \neq 1$, in $\text{Tr}$ has $l$ successors then the production applied at node $v$ is $B_{l, j_1} \rightarrow B_{j_1} \ldots B_{j_k}$. The application of this production at node $v$ is denoted as $B_{v} \rightarrow B_{u_1} B_{u_2} \ldots B_{u_l}$ where $u_1, \ldots, u_l$ are the indices of the successors of $v$ in $\text{Tr}$. 
Two more production forms,

b) \( B^j \rightarrow B^j_{j_1} \ldots B^j_{j_k} \) and

c) \( B^i \rightarrow B^j \), are necessary for implementing the algorithm.

We shall think of \( B^i \) as the distinguished start symbol and \( B^j \) as a terminal symbol. If a node \( v \) in \( Tr \) is a leaf then we will assume a production of the form c) is implied at \( v \).

Let \( Tr' = [(N \cup \{ v/v \text{ is a leaf of } Tr \}], \)

\( (E \cup \{ (v,v)/v \text{ is a leaf} \}] \)

\[
\begin{array}{c|c|c}
\text{Tr:} & \text{Node} & \text{Production applied} \\
\hline
1 & B^l_1 & B^l_2 \ldots B^l_6 \\
2 & B^l_2 & B^l_3 \ldots B^l_4 B^l_5 \\
3 & B^l_3 & B^l_4 \\
4 & B^l_4 & B^l_5 \\
5 & B^l_5 & B^l_6 \\
6 & B^l_6 & B^l_7 \\
\end{array}
\]

Figure 3.2.1. Productions Applied at the Nodes of \( Tr \).

Attribute grammar for the reverse depth first traversal of \( Tr' \).

The attribute grammar given below provides the semantic functions for computing the attributes associated with the nodes of \( Tr' \) during a reverse depth first traversal. Let \( StTr' \) denote the semantic \( Tr' \)--that is, \( Tr' \) with each node structured to contain various attributes. Among these are

Inherited attribute of \( B^l_i \) (for \( i \neq 0 \)): (START, i)

Synthesized attribute of \( B^l_i \): (PROCESSED, i).
3.2.1 Productions

a) \[ B_{i_1}^0 \rightarrow B_{i_1}^1 \]  
   a.1) \( (\text{START}, 1) \leftarrow \text{True} \)
   a.2) \( \text{IF} (\text{PROCESSED}, 1) \text{ THEN} \)
       process final;
       \( (\text{PROCESSED}, 0) \leftarrow \text{True} \text{ FI} \)

b) \[ B_{i_1}^1 \rightarrow B_{i_1}^j \]  
   b.1) \( \text{IF} (\text{START}, i) \text{ THEN} \)
       process \( (B_{i_1}^j) \); \#compute
       the desired attributes
       of \( B_{i_1}^j \) \#
       \( (\text{PROCESSED}, i) \leftarrow \text{true} \text{ FI} \)

c) \[ B_{i_1}^j \rightarrow B_{j_1}^l \ldots B_{j_k}^l \]  
   c.1) \( (\text{START}, B_{j_l}^l) \leftarrow (\text{START}, B_{i_1}^j) \)
   c.2) \( \forall \ell, \ 1 \leq \ell < k \)
      \#a rule schema\#
      \( (\text{START}, j_{l+1}) \leftarrow (\text{PROCESSED}, j_{l+1}) \)
   c.3) \( \text{IF} (\text{PROCESSED}, j_1) \text{ THEN} \)
       \( (\text{PROCESS}, B_{i_1}^j); \)
       \( (\text{PROCESSED}, i) \leftarrow \text{true} \text{ FI} \).

Note: Semantic rule c.2 is really a schema.

The semantic functions given above formulate the following rules
which govern reverse depth first order traversal of \( T_{r'} \). The
attributes of node \( i \) are processed in some fashion if

a) all its descendants in \( T_{r'} \) are processed (see rule c.3) and,

b) all the subtrees of \( T_{r'} \) to the right of \( i \) -- subtrees
   whose leaves are to the right of \( i \) and whose roots are
   not ancestors of \( i \) -- are processed (see rule c.2) .
Figure 3.2.2a. Augmented Dependency Graph
for Example in Figure 3.2.1.

Before presenting the actual attribute grammars for the reducibility test, we outline the overall picture in Figure 3.2.3.
Initial production of CFG is $S_o \rightarrow x_1 \ldots x_n$.

Figure 3.2.3. Overall Picture
PLEASE NOTE:

This page not included in material received from the Graduate School. Filmed as received.

UNIVERSITY MICROFILMS
We shall say that a cross edge \((u,v) \in C\) is within \(\text{DAG}(\ell)\) if the nodes \(u,v \in N_\ell\). In order to find the dominator tree for 
\(G\), it is first essential to label the cross edges of \(G\). If \((u,v) \in C\) is within \(\text{DAG}(\ell)\) but not within \(\text{DAG}(\ell + 1) \ldots \text{DAG}(n)\) (recall that the nodes of \(G\) are indexed by the depth first numbering algorithm) then \(\text{LABEL}(u,v) = \ell\). The algorithm to be used for this purpose is

3.2.2. **Algorithm A.** Labelling of Cross Edges.

**Input** \(\text{DFST}(G)\) with nodes and edges labelled as in 3.1.1.

**Output** \(\text{LABEL}(u,v)\) for each \((u,v) \in C\).

\[
\begin{align*}
\text{BEGIN FOR} & \quad I = n \quad \text{STEP} -1 \quad \text{to} \quad 1 \quad \text{END} \\
\text{DO FOR} & \quad \text{each} \quad (J,I) \in \text{CT}(I) \quad \text{OD} \\
\text{DO} & \quad \text{LABEL}(J,I) = \text{FIND}(J) \quad \text{OD} \\
& \quad \text{UNION} \quad \text{STP}(I),\text{STP}(I),I) \\
\text{OD} & \quad \# \text{FIND, UNION as in Chapter 4 of [AHU].} \\
\end{align*}
\]

The example in Figure 3.2.4 illustrates Algorithm A. We shall use the spanning tree of \(G\) as the structural basis for the attribute grammar implementing Algorithm A. Let \(\text{SST}(G)\) denote the semantic spanning tree with each node as a structured variable. We shall assume that certain inherited attributes of the nodes of \(\text{SST}(G)\) have been initialized by the depth first algorithm.

We can specify the algorithm A using the attribute grammar 3.2.1 as a basis. In the interest of saving space, it is convenient to ascribe global attributes to \(B\&1\). These attributes are \FOREST and \CROSS EDGES LABELLED. The first is implemented using 2-3
trees and the second using a table with n-entries, each entry pointing to a list. UNION and FIND instructions in semantic rules involve the manipulation of FOREST as detailed in Chapter 4 of [AHU].

\[
\begin{align*}
I &= 8, & 1 &= \{1\} \cup \{8\} = \{1, 8\} \\
I &= 7, & 5 &= \{5\} \cup \{7\} = \{5, 7\} \\
I &= 6, & \text{LABEL}(7, 6) &= \text{FIND}(7) = 5 \\
& & 5 &= \{5, 7\} \cup \{6\} = \{5, 6, 7\} \\
I &= 5, & \text{LABEL}(8, 5) &= \text{FIND}(8) = 1 \\
& & 2 &= \{2, 5\} \cup \{5, 6, 7\} = \{2, 5, 6, 7\} \\
I &= 4, & \text{LABEL}(6, 4) &= \text{FIND}(6) = 2 \\
& & 2 &= \{2, 5, 6, 7\} \cup \{4\} \\
& & &= \{2, 5, 6, 7, 4\} \\
I &= 3, & \text{LABEL}(4, 3) &= \text{FIND}(4) = 2 \\
& & \text{FIND}(u) &= \text{name of the set containing } u. \\
C &= (8, 5), (7, 6), (6, 4), (4, 3)
\end{align*}
\]

Figure 3.2.4. Labelling of Cross Edges

The following block replaces 'process (B\_L)' in rules b.1) and c.3) of 3.2.1. We assume that an inherited attribute (CT, i), for each node i in SST(G), has been initialized by the depth first algorithm, as mentioned before.
3.2.3. # Steps taken to 'process' a node #

\begin{verbatim}
BEGIN FOR each \((x_j, j_2) \in (CT, j_2)\)

DO LINK\((x_j, j_2)\) to the list pointed to by

CROSS EDGES LABELLED(FIND(x)).

OD

UNION \((i, i, j_2)\)

END
\end{verbatim}

The above attribute grammar causes CROSS EDGES LABELLED\((k)\) to point to a list of cross edges each element of which is assigned the label \(k\). This global attribute of the initial node is computed during the execution of a reverse depth first order pass thru SST\((G)\). After CROSS EDGES LABELLED is computed, another reverse depth first order pass of SST\((G)\) is triggered. During this pass the global attribute, DT\((G)\) is computed. In the interest of simplicity, we shall only present the details of the second pass assuming CROSS EDGES LABELLED has been computed. That is, we shall ignore the details of the flag mechanism for triggering the second pass which computes the dominator tree attribute, DT.

Additional attributes of the nodes of SST\((G)\) are as described below--

Synthesized attributes of \(B_{i,j}^k\) (for \(i \neq 0\)):

\((F,i)\) - will contain 'composite' forward edges. This attribute can be implemented as a forest of 2-3 trees

\((C,i)\) - will contain 'composite' cross edges. This attribute can also be implemented as a forest of 2-3 trees.
Inherited attributes of $B_{t_1}^i$:

$(CO,i) = \text{cross edges originating at } i$.

$(FT',i) = (j,i)$ where $j = \max\{h/(h,i) \in FT(i)\}$.

Details of the attributes and semantic functions which specify the traversal of $SST(G)$ are given in 3.2.1. We now describe the steps that must replace 'process $(B_{t_1}^i)$' in rules b.1 and c.3. These steps are based directly on the algorithm for determining dominators in a directed acyclic graph in Section 5.11 of [AHU]. This algorithm gives the sequence of steps to be taken to synthesize attributes $(C,i)$ and $(F,i)$ when returning along the spanning tree edge $(i,j_g)$ during a reverse depth first traversal of $SST(G)$.

3.2.4. #Steps taken to 'process' a node #.

BEGIN Steps to be taken when returning along spanning tree edge $(i,j_g)$.

END.

Testing Flow Graphs for Reducibility

The traversals of $DT(G)$ described below provide a method of testing a flow graph $G$ for reducibility. We assume here that, each node $i$ of $DT(G)$ has an associated attribute $(BT,i)$ which is initialized by the depth first numbering algorithm. Refer to Definition 3.1.1. Additional synthesized attributes of node $i$ are $(NDF,i)$ and $(NR,i)$. $(NDF,i)$ will be assigned yet another index for node $i$ and $(NR,i)$ will be assigned the value $(NDF,k)$ where $k$ is the rightmost leaf of the subtree of $DT^+_i$ with root $i$. $IRRED$ is a global flag which is set if $G$ is irreducible. $^+DT$ will denote $DT(G)$. 
The attribute grammars will henceforth be presented using a somewhat different format as shown below.

3.2.5 Algorithm B. Testing for Reducibility.

Input: DT with attributes for each node i
(NDF,i): depth first number of node i in DT
(NR,i): NDF value of the rightmost leaf in subtree of DT with root i.
(BT,i): Set of back edges with target node i.

Output: ∀ node i, (NDF,i) and (NR,i).

IRRED: Global flag set if G is irreducible

Method:

# Pass 1#

j = 1

# Traverse DT in depth first order#

1. FOR i = 1 TO n # n = number of nodes in G #
   # Local Semantics #
   2 DO (NDF,i) ← j;
   3 j = j + 1;

OD

# Pass 2#

# Traverse DT in reverse depth first order#

4 FOR i = 1 TO n
   # Local Semantics#
   5 DO (NR,i) ← (NDF,r) where r is the rightmost successor of node i in DT.

OD
# Pass 3#

6 IRRED ← 0

# Traverse DT in depth first order#

7 FOR i = 1 TO n

# Local semantics#

DO # Test each back edge in (BT, i) as follows #

Let k be the index of node i such that (NDF, k) = i

Let (j, k) be the next edge in (BT, k);

# Note (NDF, k) = i#

IF ¬ ((NDF, k) < (NDF, j) ≤ (NR, k))

THEN IRRED ← 1; # IRRED is set if j is not a descendant of k in DT even though (j, k) ∈ (BT, k)#

It is straightforward to use the techniques of Section 3.1 to implement various types of traversals of the structure DT.

Each of the three passes in the above algorithm is described using:

a) the underlying structure, b) the attributes of each node in the structure, c) a description of the type of traversal of DT, and d) the local semantics, that is, the semantics associated with each production generating the structure. Therefore, an attribute grammar for specifying the above algorithm can be easily written.

Figure 3.2.5 illustrates the entire algorithm for the reducibility test.

3.2.6 Lemma. Algorithm 3.2.5 tests G for reducibility.

Proof. The test Algorithm 3.2.5, line 8, holds if and only if node hj is dominated by node k in DT(G). That is, E can be
G with each node labelled by its depth first number.

The DT attributed SST.

DT with each node labelled by the e-tuple \((i, \text{NDF}(i), \text{NR}(i))\). The back edges of ST are also indicated.

\((3, 2)\) is a back edge of \(G\).

\(i = 3, (3, 2) \in (BT, 3)\)

\((\text{NDF}, 3) = 2\)

\((\text{NDF}, 2) = 3\)

\(\text{NR}, 3) = \emptyset\)

\((3 < 2)\) is true. Therefore \(\text{IRRED} = 1\).

Figure 3.2.5. Testing for Reducibility
partitioned into two sets $E_1 = T \cup F \cup C$ and $E_2 = B$ such that $E_1$ is the edge set of DAG(1) and each $(j,k)$ in $E_2$ has $j = k$ or $k$ dominates $j$ in $G$, if the test in line 8 holds for each back edge $(j,k)$. That is, by lemma 3.1.3, the flow graph $G$ is reducible.

3.3 Conclusions.

The reducibility test presented in the earlier sections requires several passes of the parse tree, the spanning tree and the dominator tree. But, each pass is fairly efficient. Let $\lambda$ be the size of the input string ($\lambda$ may be the number of symbols in the string).

1. Complexity of the passes to determine the flow graph of $G$ is proportional to the size of the parse tree which is $O(\lambda)$ for LRK grammars.

2. Complexity of the passes to determine the spanning tree $ST(G)$ initialize certain inherited attributes is $O(\lambda)$.

3. Complexity of the pass to label cross edges is $O[\|N\| + (|C| + |F|) \alpha (|N|, |C| + |F|)]$. Here $\alpha(m,n)$ is related to a functional inverse of Ackerman's function. For details refer to [T5] and Chapter 4 of [AHU].

4. Complexity of the pass to determine the dominator tree, $DT(G)$ is $O[|N| + (|F| + |C|) \alpha (|N|, |F| + |C|)]$.

5. Complexity of the passes which traverse $DT$ and examine the back edge to test for reducibility is $O(|N| + |E|)$.

$|N|$ denotes the cardinality of set $N$.  

*
Therefore, the complexity of the reducibility test given here is $O((L + |E| + |N|)\alpha(|N|, |F| + |C|))$. This bound compares favorably with the bound $O(L + |N| + |E|)\alpha(|N|, |E|)$ indicated in [T2]. However considering the nature of the function $\alpha$, this improvement is small.

This chapter illustrated the use of attribute grammars as a definitional tool which, at the same time, provides an algorithm for translation. Several semantic functions given here were defined using attribute grammars. The results of Chapter 1 provide us with a method of evaluating the functions using the attribute grammar evaluator.

Two types of code improvement problems have emerged in the area of compiler optimization—the forward flow problem (FFP) and the backward flow problem (BFP). Variants of the interval analysis approach have been used in [GW], [HU4], [KU], [Ke3], [Ke4] and [U] to obtain solutions for both types of problems. These solutions are generally of \(O(n \times e)\) where \(n\) is the number of nodes and \(e\) the number of edges in the flow graph modelling a given program, provided the flow graph is reducible. (All results dealt with in this presentation consider only reducible flow graphs). The node listing technique, developed in [Ke1], can be used to solve backward and forward flow problems in \(O((n+e) \log(n+e))\) steps. We shall show here that the bound for the FFP can be improved to \(O(n+e \ \alpha(n,e))\) by using the node listing approach and the efficient data manipulation technique in [T3]. We also present a formal description of the crucial difference between the two types of problems which makes it impossible to achieve the same improvement for the BFP using similar techniques. A new \(O(n+e \ \log(n+e))\) algorithm using the node listing approach is given for the BFP. Our results seem to indicate that the backward and forward flow problems have distinct inherent complexities.

In section 4.1 we present a method for assigning convenient indices to the nodes of the dominator tree, for a reducible flow graph \(G\). Section 4.2 presents formal statements of FFP and BFP and precisely identifies some major differences between the two problems are precisely identified. Section 4.3 introduces \(+\) All logarithms are to the base 2.
The notion of path listing—a generalization of node listings in the algorithms of Section 4.4 to determine solutions for FFP and BFP. Section 4.5 contains the proofs of correctness for the algorithms. This is followed by a discussion of the complexity issues and other concluding remarks in Section 4.6.
4.1. **Basic Material**

We introduce this chapter with material on dominator trees from [T4]. Let $G$ be a reducible flow graph (r.f.g.) and $DT$ its underlying dominator tree. Lemma 4.1.1 characterizes a property of $DT$ which is used later for defining a convenient numbering of the nodes in $N$. Let $G = (N, E, n_0)$.

4.1.1. **Lemma [T4].** For each $e \in E$, $\text{id}(T(e)) \xrightarrow{DT} 0(e)$.

4.1.2. **Definition.** For any edge $e \in E$, let $x(e)$ be the node

$$\text{id}(T(e)) \text{ if } 0(e) = \text{id}(T(e)) \text{ otherwise let } x(e) \text{ be the node } v \exists! \text{id}(T(e)) \xrightarrow{1} DT. v \xrightarrow{*} 0(e).$$

We now define a way of assigning index numbers to the nodes in $N$. Let $N1$ be the indexing function. $N1$ will be used to specify a processing order for the nodes of $DT$.

4.1.3. a) $N1(x(e)) > N1(T(e))$, $e \in E$

b) If $x \xrightarrow{DT} y$ then $N1(x) > N1(y)$, $x, y \in N$.

We shall assume that each node in $N$ is mapped to a unique number between 1 and $|N|$.

Each node $i$ of $DT$ can be thought of having various attributes. Some of these attributes represent sets $BT(i), CT(i), FT(i), ET(i), BO(i), CO(i), FO(i)$ and $E0(i)$. If the nodes of $DT$ are processed according to their $N1$ values, from 1 to $|N|$, then for any edge $e$ in DAG of $G$, $T(e)$ is processed before $0(e)$ which is in turn processed before $\text{id}(T(e))$. In view of this comment, the proof of Lemma 4.1.4 should be obvious.

$^+$ We shall use $x(i)$ to denote the attribute $x$ of node $i$. 
Each node of $G$ contains its depth first number

1. The dominator tree underlying $G$ is denoted by dashed lines.
2. Each node $v$ contains the 2-tuple $(v, N(v))$.

<table>
<thead>
<tr>
<th>$e$</th>
<th>$x(e)$</th>
<th>$v$</th>
<th>$N1(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>(1,7)</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(1,6)</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(2,3)</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>(2,4)</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(2,5)</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(3,1)</td>
<td>0</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>(4,2)</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,3)</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,5)</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,1)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6,7)</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7,2)</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c) Let $p = 1, 7, 2, 4, 3$. Then $N1(1) > N1(7) > N1(4) > N1(3)$.

Figure 4.1.1. Indexing of DT
4.1.4 Lemma. Let $p = p_1 p_2 \ldots p_k$ be any path in DAG. If the nodes of DT are processed in increasing order as specified by their N1 values, then nodes in path $p$ are processed in the order $p_k, p_{k-1}, \ldots, p_1$. Conversely, if the nodes are processed in the decreasing order specified by their N1 values, then the nodes in path $p$ are processed in the order $p_1, p_2, \ldots, p_k$. (Figure 4.1.1c illustrates this lemma).

Henceforth, we will assume that dominator trees are oriented such that traversing the tree in a left to right, leaf to root order implies that the nodes are visited in increasing order as specified by their N1 values.

4.1.5. Definition. For each $j \in N$.

$$
\text{NDT}(j) = \{i/\text{id}(i) = j\}, \text{ the set of nodes dominated by } j.
$$

$$
\text{EDT}(j) = \{e/e \in E_{\text{DAG}}^+ \text{ and } T(e) \in \text{NDT}(j)\}, \text{ the set of edges dominated by } j.
$$

The algorithms given in later sections will require NDT and EDT to be topologically ordered. For the example in Figure 4.1.1, referring to each node by its N1 index, NDT(4) = \{3,2,1\} and EDT(4) = \{4,3)(3,2)(4,2)(3,1)(4,1)\}.


Formal Statements of the Problems.

In this section we shall first identify the general class of problems solved by the backward and forward flow algorithms given in Section 4.4. To do this we follow the unifying approach described in [CW], [KL],[Ku],[T4].

$E_{\text{DAG}} = \text{CUT}\text{UT}$, the edges in the DAG of $G$.

++ See [Kn].
Let $D$ be the problem domain which satisfies the axioms of a closed semiring with set union as the 'addition' operation and set intersection as the 'multiplication' operation.

4.2.1. **Backward Flow Problem (BFP).**

A BFP (Backward Flow Problem) is a triple $(G, C_1, I_1)$ where

a) $G$ is a reducible flow graph $(N, E, n_0)$,

b) $C_1$ is a mapping from $E$ to $D$,

c) $I_1$ is a mapping from $N$ to $D$.

A mapping $BF: N \rightarrow D$ is said to be a solution of the BFP $(G, C_1, I_1)$ if it satisfies the following conditions:

d) For each simple $^+$ path $p$

$$BF(0(p)) \supseteq C_1(p)^+ \cap BF(T(p)) \cup I_1(0(p))$$

e) For each $x \in N$

$$BF(x) \subseteq z(x)$$

for any mapping $z: N \rightarrow D$ such that

$$\forall e \in E \quad z(0(e)) \supseteq C_1(e) \cap z(T(e)) \cup I_1(0(e))$$

Note that condition d) gives the behavior of the solution while condition e) asserts that a solution must be minimal.

4.2.2. **Forward Flow Problem (FFP)**

A FFP (Forward Flow Problem) is a triple $(G, C_2, I_2)$ where

a) $G$ is a reducible flow graph $(N, E, n_0)$,

b) $C_2$ is a mapping from $E$ to $D$,

c) $I_2$ is a mapping from $N$ to $D$.

$^+$ See [Ke 1] for a discussion of why this restriction is valid.

$^+C$ is extended to map paths into sets. For any path $p = e_1 e_2 \ldots e_k$, $C(p) = C(e_1) \cap C(e_2) \cap \ldots \cap C(e_k)$. 

A mapping $\text{FF}: \mathbb{N} \rightarrow \mathcal{D}$ is said to be a solution of the
FFP($G, C_2, I_2$) if it satisfies the following conditions:

d) For each simple path $p$
$$\text{FF}(O(p)) \supseteq I_2(T(p)) \cup C_2(p) \cap \text{FF}(O(p)),$$
e) For each $x \in \mathbb{N}$
$$\text{FF}(x) \subseteq z(x) \text{ for any mapping } z: \mathbb{N} \rightarrow \mathcal{D} \text{ such that }$$
$$\forall e \in E \quad z(O(e)) \supseteq I_2(T(e)) \cup C_2(e) \cap z(O(e)).$$

Once again, node d) gives the behavior of the solution while condition e) asserts that a solution must be minimal.

Condition d) of 4.2.1 may be intuitively stated as follows: data associated with node $T(p)$, i.e., $BF(T(p))$, must be 'modified' by the cost associated with path $p$, i.e., $C_1(p)$, and propagated to node $O(p)$. That is, data must flow backward from $T(p)$ to $O(p)$. Similarly, condition d) of 4.2.2 may be viewed as follows: data associated with node $O(p)$, i.e., $FF(O(p))$, must be modified by the cost associated with path $p$, i.e., $C_2(p)$, and propagated to node $T(p)$. In other words, data must flow forward from $O(p)$ to $T(p)$.

Live flow analysis (Section 2.2) is an example of a BFP. For details the reader is referred to [Ke 2]. Common subexpression elimination, a FFP, is discussed in [GW], [KU], [U] and [Ke 5]. Since data flow along only the simple paths of $G$ need be considered in future we will deal only with simple paths. Therefore, we will not always specifically state that a path $p$ is simple.

In the remainder of this section we shall discuss the crucial differences between the data flow that must be considered for solving any BFP and any FFP for a reducible flow graph (rfg) $G$. As a
first step we define the cycles of $G$; and, for each cycle, the exit and entrance nodes through which data can 'flow' into and out of the cycle.

4.2.3. Definition. \( \forall e \in B \) (the set of back edges)

\[
\text{CYCLE}(e) = \{ x / x \in N \text{ and } x \text{ is on the path } T(e) \xrightarrow{+} 0(e) \}.
\]

\[
\text{EXIT}(e) = \{ x / x = 0(f) \text{ where } f \text{ is a path } x, y_1, y_2, \ldots, y_{k-1}, y_k \text{ such that}
\]

a) \( x = 0(f) \in \text{CYCLE}(e), \)

b) \( x \xrightarrow{\text{DAG}} y_{k-1}, \) and

c) \( y_1, y_2, \ldots, y_k \notin \text{CYCLE}(e) \)

d) \( (y_{k-1}, y_k) \in B \}. \) (See Figure 4.2.1a)

\[\text{Figure 4.2.1a. Exit Node of CYCLE}(e).\]
\( \text{ENTER}(e) = \{ x . x = T(f) \text{ where } f \text{ is a path} \}

y_1, y_2, \ldots, y_{k-1}, y_k, x \text{ such that }

a) \quad y_2 \xrightarrow{\text{DAG}} T(f) = x

b) \quad T(f) = x \in \text{CYCLE}(e)

c) \quad y_1, y_2, \ldots, y_k \in \text{CYCLE}(e)

d) \quad (y_1, y_2) \in B \}. \text{ (See Figure 4.2.1b)}

\[ \text{Figure 4.2.1b. Enter node of } \text{CYCLE}(e). \]
Each exit node $x$ of some cycle $e$, that is $x \in \text{EXIT}(e)$, can be thought of as a node channeling data between $\text{CYCLE}(e)$ and $\text{CYCLE}(y_{k-1}, y_k)$. Similarly, each entrance node $x$ of some cycle $e$, that is $x \in \text{ENTER}(e)$, can be thought of as a node channeling data between $\text{CYCLE}(e)$ and $\text{CYCLE}(y_1, y_2)$. There may be several exit and entrance nodes channeling data between two cycles.

Lemma 4.2.4 specifies a way of representing each simple path in $G$. This particular representation of each path is convenient for solving the FFP for $G$.

4.2.4. Lemma. Any simple path $p$ in rfg $G$ can be written as $t_1 b_1 t_2 b_2 \ldots t_{k-1} b_{k-1} t_k$ such that

a) $b_i$ is a back edge in $G$, $1 \leq i \leq k-1$.

b) $t_i$ is a path in DAG and can be represented as $x_i \xrightarrow{\text{DAG}} y_i \xrightarrow{\text{ST}} z_i$ where $x_i = T(b_{i-1})$, $y_i \in \text{ENTER}(b_i)$ and $z_i = O(b_i)$, $2 \leq i \leq k-1$ (See Figure 4.2.2)

c) $t_1, t_2$ are paths (possibly of length 0) in DAG.

![Diagram](image)

Figure 4.2.2. DAG path between $\text{CYCLE}(b_{i-1})$ and $\text{CYCLE}(b_i)$. 
Proof. By lemma 3.1.3, G is reducible implies there exists sets E₁,E₂ such that E₁ is the edge set of DAG and E₂ = B. That is, any path p can be written as \( t₁b₁t₂b₂\ldots tₖ₋₁bₖ₋₁tₖ \) by identifying each back edge in the path and choosing \( tᵢ \) to represent the appropriate path \( T(bᵢ) \xrightarrow{\text{DAG}} 0(bᵢ₊₁) \), \( 1 < i < k₋₁ \).
(The initial and the final segments, \( t₁ \) and \( t₂ \), may represent DAG paths of length 0). Furthermore, each \( tᵢ \), \( 1 < i < k \), can be written in the form required by b).

Let \( p \) be any simple path in rfg G. The direction of information flow in \( p \), which must necessarily be considered to solve the BFP, is from \( T(p) \) to \( 0(p) \). Therefore, before examining the representation of \( p \) convenient for BFP, we introduce the concept of path reversal.

4.2.5. Definition. Let \( p = p₁p₂\ldots p_{k₋₁}p_k \) be a simple path.\( \overline{p} = p_kp_{k₋₁}\ldots p₂p₁ \) is called a reverse path. The reverse of path \( u \xrightarrow{\text{DAG}} v \) will be denoted as \( u \xrightarrow{\text{DAG}} v \). Suppose \( p \) can be represented as \( u₁u₂\ldots u_k \) such that \( uᵢ \) is a path and \( T(uᵢ) = 0(uᵢ₊₁) \) for \( 1 < i < k \). Then, \( \overline{p} = u₁u₂\ldots u_k = u_kuₖ₋₁\ldots u₁ \)

4.2.6. Lemma. Every reverse path, \( \overline{p} \), can be represented as \( t₁b₁t₂b₂\ldots tₖ₋₁bₖ₋₁tₖ \) such that

a) \( bᵢ \in B, 1 ≤ i ≤ k₋₁ \)

b) \( tᵢ \) is a reverse path in DAG(1) and can be represented as \( xᵢ \xrightarrow{\text{DAG}} yᵢ \xrightarrow{\text{DAG}} zᵢ \) where \( zᵢ = 0(bᵢ₋₁), yᵢ \in \text{EXIT}(bᵢ), \)
\( xᵢ = T(bᵢ) \). (See Figure 4.2.3.)

c) \( t₁, t₂ \) are reverse paths (possibly of length 0) in DAG(1).
Figure 4.2.3. DAG path between $\text{CYCLE}(b_{i-1})$ and $\text{CYCLE}(b_i)$.

**Proof.** Analogous to proof of lemma 4.2.4.

We have shown in Lemmas 4.2.4 and 4.2.6 that the data flow along each simple path can be thought of as occurring in stages -- a stage consisting of flow within a cycle followed by flow along a DAG path between two cycles. The following lemmas show that, for a rfg $G$, cycles can be considered in a systematic order. FLIST and BLIST will be the orderings of back edges suitable for the FFP and the BFP, respectively. The order of the occurrences of cycles in path $p$ will also be the order in which these cycles occur in FLIST. Similarly, the order of the occurrences of cycles in any reverse path $\overline{p}$ will also be the order in which these cycles occur in BLIST. Therefore, if the cycles of $G$ are processed in the order of their occurrence in FLIST (BLIST) then the cycles of any path $p(\overline{p})$ are processed in the order of their occurrence in $p(\overline{p})$. 
4.2.7. Lemma. For any rfg $G$ there is an ordering $\text{FLIST} = e_1 e_2 \ldots e_{b-1} e_b$ of the back edges in $B$ such that if

a) there is a path $0(e_x) T(e_x) \overset{*}{\rightharpoonup} v \overset{*}{\rightharpoonup} u$ with $e_x \in B$ and $v \in \text{EXIT}(e_x)$, $u \in \text{CYCLE}(e_j)$

b) $v \overset{+}{\rightharpoonup} u$ implies $u \in \text{ENTER}(e_j)$

then

a) $1 \leq x < j \leq b$

b) $T(e_j)$ properly dominates $T(e_x)$ in $DT$.

Note: $b = |B|$.

Proof. Suppose $DT$ is traversed in reverse depth first order.

Let $\text{FLIST} = e_1 e_2 \ldots e_{b-1} e_b$ be the back edges ordered such that if $1 \leq i < j \leq b$ then the node $T(e_i)$ is traversed before node $T(e_j)$. It follows that either $T(e_i)$ belongs to a subtree to the right of $T(e_j)$ or else it is a descendant of $T(e_j)$ in $DT$. Let $0(e_x) T(e_x) \overset{*}{\rightharpoonup} v \overset{*}{\rightharpoonup} u$ be any path as in the hypothesis. We shall show $1 \leq x < j \leq b$ by considering the several cases of relative positions of $u$ and $v$ in $DT$.

Case 1. Suppose $v \overset{0}{\rightharpoonup}_{\text{DAG}} u$ that is node $v$ is node $u$. This case is illustrated in Figure 4.2.4a. If $T(e_x)$ properly dominates $T(e_j)$ then $v$ cannot be an exit node channeling data from $\text{CYCLE}(e_x)$ to $\text{CYCLE}(e_j)$ (see Definition 4.2.3). This contradicts the hypothesis. If $T(e_j)$ properly dominates $T(e_x)$, $v$ is a proper exit node and according to our discussion above $e_j$ occurs after $e_x$ in the

$+$ Node $x$ properly dominates node $y -- x \overset{+}{\rightharpoonup}_{\text{DT}} y$. 
Figure 4.2.4a. Case 1
in the ordering of back edges. That is, \( 1 \leq x < j \leq b \).

**Case 2.** Suppose \( v \) is a node in a subtree of \( DT \) to the right of \( u \). That is \( v \xrightarrow{DAG} u \). If \( T(e_x) \) properly dominates \( T(e_j) \) the node \( v \) cannot be an exit node between \( \text{CYCLE}(e_x) \) and \( \text{CYCLE}(e_j) \).

On the other hand, if \( T(e_x) \) and \( T(e_j) \) do not dominate each other then there is a path from the start node to \( T(e_x) \) to \( u \), \( u \in \text{ENTER}(e_j) \), which does not contain the node \( T(e_j) \). This implies \( G \) is irreducible—a contradiction. The last alternative, \( T(e_j) \) dominates \( T(e_x) \), if this case holds. And, if \( T(e_j) \) properly dominates \( T(e_x) \) then \( e_j \) appears after \( e_x \) in the above ordering i.e., \( 1 \leq x < j \leq b \). See Figure 4.2.4b.

1) \( v \notin \text{EXIT}(e_x) \)

2) \( G \) is irreducible

3) \( u \in \text{ENTER}(e_j) \), \( v \in \text{EXIT}(e_x) \)

Figure 4.2.4b. Case 2.
Figure 4.2.4c. Case 3.

Case 3. Suppose \( v \) dominates \( u \). Then either \( T(e_j) \) dominates \( T(e_x) \) or \( T(e_x) \) properly dominates \( T(e_j) \), and \( T(e_j), T(e_x) \) dominate \( u, v \). If \( T(e_j) \) dominates \( T(e_x) \) then \( T(e_x) \in \text{CYCLE}(e_j) \), \( v \in \text{EXIT}(e_x) \) then \( u \in \text{CYCLE}(e_j) \) with \( v \xrightarrow{\text{DAG}} u \). If \( T(e_x) \) dominates \( T(e_j) \) \( v \) cannot be an exit node between cycles \( e_x \) and \( e_j \). See Figure 4.2.4c.
Case 4. \( v \) cannot be a node in a subtree of \( DT \) to the left of \( u \). This is a direct consequence of the fact that all cross edges in \( DAG \) are from right to left.

The following algorithm gives acceptable ordering for BLIST.

4.2.8. Algorithm A. Ordering back edges for BFP.

**Input**: \( DT \) - dominator tree with nodes indexed by depth first numbers.

**Global attributes of nodes in DT**:

- \( LB \) - list of back edges in \( B \).
- \( LN \) - list of nodes in depth first order such that each node has more than one successor in \( DT \). This attribute can be determined by a depth first pass of \( DT \).
- \( NLN \) - Contains the next element of \( LN \)
  - \((LE, i)\) - will contain a list of back edges \( e \in 0(e) \) is a descendant of \( j \), \( T(e) \) is an ancestor of \( j \) and \( j \) is a successor of \( i \) in \( DT \). This attribute is initially empty.
  - \((LE1, i)\) - will contain list of back edges \( e \) such that \( 0(e) \) is a descendant of \( i \) and \( T(e) \) is an ancestor of \( i \).

**Output**: BLIST - will contain ordered edges, Initially empty.

**Inherited attributes of nodes in DT**:

- \((BT, i)\) - list of back edges with target node \( i \)
- \((BO, i)\) - list of back edges with origin node \( i \).
#Pass 1#

for i:=1 to n (in depth first order)
  do
    #1# if (BT,i) ≠ ø then
      for each e ∈ (BT,i)
        do if NLN ≠ ø then
          if 0(e) is a descendant of j and j ∈ (S,NLN)
            then (LE,j) ← (LE,j)∧e;
                (LE1,NLN) ← (LE1,NLN)∧e
          fi
        fi
      od
    fi;
  #2# if i=NLN then NLN ← next element of LN fi
  od

#pass 2#

for i:=1 to n (in depth-first order)
  do
    #3# if (BO,i) ≠ ø and e ∈ (BO,i) and e ∈ LB
      then BLIST ← BLIST∧e
        Delete e from LB
    fi
    #4# if ∃ j₁,j₂ ∈ (S,i) s.t. (LE,j₁) and
      (LE,j₂)
      are not empty
      then BLIST ← BLIST∧(LE1,i)ⁱ; #superscript the edges in (LE1,i)
    by i#
Pass 1.

LN = 2, 5, 10

NLN = 2

i = 1, \( (LE, 3) = (5, 1) \)

i = 2, \( (LE, 8) = (10, 2) \), NLN = 5

i = 3, \( (LE, 7) = (7, 3) \)

i = 4, \( (LE, 6) = (6, 4) \)

i = 5, NLN = 10

i = 6, i = , i = 8

i = 9, \( (LE, 11) = (11, 9) \)

i = 10, NLN = \( \emptyset \)

i = 11, i = 12 .

Pass 2.

i = 1

i = 2, BLIST = \( (5, 1)^2 (10, 2)^2 \)

i = 3, i = 4,

i = 5, BLIST = BLIST(7,3)^5 (6,4)^5

i = 8,

i = 9, BLIST = BLIST(9,8)

i = 10

i = 11, BLIST = BLIST(11,9)

BLIST = \( (5, 1)^2 (10, 2)^2 (7, 3)^5 (6, 4)^5 (9, 8) (11, 9) \)

Figure 4.2.6. Ordering of back edges, BLIST.
(MARK,i) ← 1;

Delete from LB each edge in (LE1,i)

fi

#Note: In step 1 of pass 1, the algorithm requires that 0(e) be
tested to see whether it is a descendant of the node j. Details
for the implementation of this test are given in Algorithm B (3.2.5)
in Section 3. #

Figure 4.2.5 illustrates Algorithm 4.2.8. The superscripts,
appeared to certain edges placed in BLIST, will be used in later
algorithms to identify edges to be processed in a single stage. The
ordering of back edges, for the same example, necessary for the FFP
is FLIST = (11,9)(9,8)(6,4)(7,3)(10,2)(5,1). The order of edges
in FLIST is the reverse of that in BLIST. In general this will
not be the case.

4.2.9. Lemma. For any rfg G there is an ordering—BLIST =
e_1,e_2,...,e_b—of back edges in B such that if

a) there is a reverse path $u \xrightarrow{\text{DAG}} v \xrightarrow{\text{DAG}} O(e_x)T(e_x)$
   with $u \in \text{CYCLE}(e_j)$, $v \in \text{ENTER}(e_x)$ and $e_x \in B$

b) $u \xrightarrow{\text{DAG}} v$ implies $u \in \text{EXIT}(e_j)$

then,

a) $1 \leq x < j \leq b$ .

b) $T(e_x)$ properly dominates $T(e_j)$ and $0(e_x),0(e_x)$ do
not dominate each other, or, $0(e_x)$ properly dominates
$0(e_j)$ in DT .

Proof. A desirable ordering of edges in B for Lemma 4.2.9 is
given in Algorithm 4.2.8.
Let \( e_1 e_2 \ldots e_b \) be the ordering specified by BLIST (Algorithm 4.2.8). Also let \( u \xrightarrow{\text{DAG}} v \xrightarrow{\text{DAG}} 0(e_x) T(e_x) \) be a reverse path as in the hypothesis. We shall once again consider the several cases of relative positions of \( u \) and \( v \) in DT.

First, however, we examine steps 1 and 2 of pass 1 (Algorithm 4.2.8). We find that if \( e_a, e_b \in B \) satisfy the condition that \( T(e_a) \) dominates \( T(e_b) \) and \( 0(e_a) \) and \( 0(e_b) \) do not dominate each other, then \( e_a \) and \( e_b \) appear on separate lists \( \text{(LE,} j_1) \) and \( \text{(LE,} j_2) \) -- attributes of node \( i \). In addition \( e_a \) appears before \( e_b \) in list \( \text{(LE1,} i) \), and therefore, in BLIST. Here the index \( i \) denotes the 'lowest common dominator' of \( 0(e_a) \) and \( 0(e_b) \) in DT, \( 0(e_a) \) is a descendant of \( j_1 \) and \( 0(e_b) \) is a descendant of \( j_2 \). Otherwise, in pass 2, \( e_a \) is placed before \( e_b \) in BLIST if \( 0(e_a) \) is traversed before \( 0(e_b) \) during a depth first traversal before \( 0(e_b) \) during a depth first traversal of DT. That is, \( e_a \) appears to the left of \( e_b \) if \( T(e_a) \) dominates \( T(e_b) \) and \( 0(e_a), 0(e_b) \) do not dominate each other or if \( 0(e_a) \) dominates \( 0(e_b) \).

**Case 1.** Suppose \( v \) occurs in a subtree to the left of \( u \), that is \( u \xrightarrow{\text{DAG}} v \). If \( T(e_x) \) properly dominates \( T(e_j) \) and \( 0(e_x) 0(e_j) \) do not dominate each other then \( v \) is an entrance node channeling data from \( \text{CYCLE}(e_x) \) to \( \text{CYCLE}(e_j) \). By our discussion above \( e_x \) appears before \( e_j \) in the ordering specified by BLIST, i.e., \( 1 \leq x < j < b \).

If \( T(e_j) \) dominates \( T(e_x) \) then \( v \) cannot be an entrance node.

This contradicts the hypothesis. If \( T(e_j), T(e_x) \) do not dominate each other then \( G \) is irreducible and again we have a contradiction of the hypothesis.
See Figure 4.2.5a for an illustration.

\[ v \notin \text{ENTER}(e_x) \]
\[ u \notin \text{EXIT}(e_j) \]

\[ v \notin \text{ENTER}(e_x) \]

\[ G \text{ is irreducible} \]

Figure 4.2.5a. Case 1.
Case 2. Suppose $u \xrightarrow{o} v$. In addition, if $T(e_x)$ properly dominates $T(e_j)$ and $0(e_x), 0(e_j)$ do not dominate each other then $v$ is an entrance node channeling data from $\text{CYCLE}(e_x)$ to $\text{CYCLE}(e_j)$ and $e_x$ appears before $e_j$ in BLIST, i.e., $1 < 1 < x < j \leq b$. If $T(e_j)$ dominates $T(e_x)$ and $0(e_j), 0(e_x)$ do not dominate each other then $v$ cannot be an entrance node. This contradicts the hypothesis. See Figure 4.2.5b. If $0(e_x)$ dominates $0(e_j)$ then $e_x$ appears before $e_j$ in BLIST, i.e., $1 \leq x < j \leq b$.

![Diagram](image)

Figure 4.2.5b. Case 2.

Case 3. Suppose $u$ dominates node $v$. Both $T(e_j)$ and $T(e_x)$ must dominate the nodes $u$ and $v$. If $T(e_x)$ dominates $T(e_j)$ then $T(e_j) \in \text{CYCLE}(e_x)$ and $T(e_j) \in \text{ENTER}(e_x)$. That is, $T(e_j) = v$ and $v \xrightarrow{o \text{ DAG}} u$. If $T(e_j)$ dominates $T(e_x)$ then $v$ cannot be an entrance node. See Figure 4.2.5c.
Case 4. $u$ is in a subtree of $DT$ to the left of $v$. This is not possible because of the nature of cross edges in DAG.

The cycles of $G$ will be processed online, in the order specified by Lemma 4.2.7, by the forward flow algorithm given in Section 4.4. Similarly, the cycles will be processed online, in the order specified by Lemma 4.2.9, by the backward flow algorithm presented in the same section.

Figure 4.2.7 illustrates a case where BLIST is not equal to the reverse of FLIST. The differences are motivated by efficiency considerations which will be pointed out in a later section.

$$BLIST = (4,2)(10,1)(8,3)(4,6)(9,5)$$
$$FLIST = (7,6)(9,5)(8,3)(4,2)(10,1)$$

Figure 4.2.7. Comparison of BLIST and FLIST.
We now reiterate the major difference between backward and forward flow using Figure 4.2.7. Even if $O(e_a)$ and $O(e_b)$, for $e_a, e_b \in B$, do not dominate each other, backward data flow can occur from $O(e_a)$ to nodes in $\text{CYCLE}(e_b)$ if $T(e_a)$ dominates $T(e_b)$. The example illustrates this for $e_a = (10,1)$ and $e_b = (8,3)$. That is, data must be propagated from node 10 to nodes in $\text{CYCLE}(8,3)$ before online processing of $\text{CYCLE}(9,5)$ can occur. In the case of forward flow, if data flows from $T(e_b)$ to nodes in $\text{CYCLE}(e_a)$ then $T(e_a)$ must dominate $T(e_b)$. Take, for example, $e_a = (8,3), e_b (9,5)$. In this case, before the online processing of $\text{CYCLE}(4,2)$, all the data flow from node 9, $9 \in \text{CYCLE}(9,5)$, to nodes in $\text{CYCLE}(8,3)$ must be known.

4.3. Pathlisting.

In this section we introduce the pathlisting concept which is a generalization of nodelistings introduced in [Kel]. A pathlisting for a forward flow problem, $(G,C2,I2)$, is a finite sequence of 2-tuples in which each path of $G$ can be 'completely represented'. Intuitively, each 2-tuple $S$ in this sequence 'controls' the flow of data from the origin node to the target node of $S$. Similarly, a pathlisting for a backward flow problem, $(G, C1, I1)$, is a finite sequence of 2-tuples in which each reverse path of $G$ can be 'completely represented'. Again, we can intuitively think of each 2-tuple as 'controlling' the flow of data from the origin node to the target node of $S$. In 4.3.1 we formally define the notion of 'complete representation' of a path in a pathlisting. Some aspects of this definition are similar to those of the propagation sequence concept in [T4]. The differences are motivated by our desire to
distinguish between the FFP and BFP.

4.3.1. **Definition.** We shall say that a sequence of 2-tuples

\[ R_1 R_2 \ldots R_t \]

represents a path \( p = x_1 x_2 \ldots x_k \) if \( p \) can be written

as \( (x_{i_1} x_{i_2} \ldots x_{i_j}) (x_{i_{j+1}} \ldots x_{i_{j+1}}) \ldots (x_{i_{k-1}} \ldots x_{i_k}) \) where

\[ a) \ T(R_j) = x_{i_j} \quad 1 \leq j \leq t. \]

\[ b) \ 0(R_j) = T(R_{j-1}), \quad 1 \leq j \leq t. \]

\[ c) \ 0(R_1) = x_1. \]

A path \( p \) can be completely represented in the finite sequence of

2-tuples \( FPL = S_1, S_2, \ldots, S_m \) if there is a sequence

\( p(1) \ldots p(2) p(1) \) of sets of representations such that

\[ a) \ p(1) \] contains \( S_1, i_1 \), \( S_1, i_2 \ldots S_1, i_k \) where \( S_1, i_j \) is the

\( i_j \)-th element of \( FPL \),

\( i_j < i_{j+1} \), for \( 1 \leq j < k \), and

\( p(1) \) represents path \( p \).

\[ b) \] If \( S_{q, i_1}, S_{q, i_2}, \ldots, S_{q, i_k} \) are 2-tuples which are elements of

representation sequences in \( p(q) \), then for each \( S_{q, i_r} \),

\[ 1 \leq r \leq k_q, \] which is not an edge in \( E \),

\( p(q+1) \) contains \( S_{q+1, i_x}, S_{q+1, i_{x+1}} \ldots S_{q+1, i_{x_r}} \) where

\( S_{q+1, i_j} \) is the 2-tuple \( S_{i_j} \) in \( FPL \), \( x \leq j \leq x_r \),

the sequence \( S_{q+1, i_x} S_{q+1, i_{x+1}} \ldots S_{q+1, i_{x_r}} \) represents

\( S_{q, i_r} \), the 2-tuple \( S_{q+1, i_{x_r}} \) occurs before the
the 2-tuple $S_{q,i_r}^x$ in FPL, and the 2-tuple $S_{q+1,i_j}^{x+1}$ appears before the 2-tuple $S_{q+1,i_{j+1}}^{x+1}$, $x \leq j < x_r$.

We shall refer to the complete representation of $p$ as $\text{CR}(p)$.

A pathlisting for the FFP $(G, C2, I2)$ is a finite sequence of 2-tuples in which every simple path of $G$ can be completely represented. FPL$(G)$ shall denote a pathlisting for the forward flow problem of $G$.

Pathlistings for the BFP $(G, C1, I1)$ can be similarly defined. Briefly, a pathlisting for the BFP, denoted as BPL$(G)$, is a finite sequence of 2-tuples in which every reverse path of $G$ can be completely represented. We shall denote the complete representation of a reverse path $\overrightarrow{p}$ as $\text{CR}(\overrightarrow{p})$.

We shall henceforth refer to an arbitrary member of BPL$(G)$ or a FPL$(G)$ as a segment. The term pathlisting for $G$, abbreviated as PL$(G)$ will be used to denote either a BPL$(G)$ or a FPL$(G)$.

The algorithms given here will use PL$(G)$ as a means of controlling the flow of data in $G$. Segments of PL$(G)$ will be processed in a left to right order. When a segment $S$ of PL$(G)$ is examined data at the left node of $S$ will be modified by the cost of segment $S$ and propagated to the right node of $S$. Therefore, we also associate cost sequences with FPL$(G)$ and BPL$(G)$.

4.3.2. Definition. The cost sequence of $\text{FPL}(G) = S_1S_2...S_m$ is the sequence $\text{C2}'(\text{FPL}(G)) = \text{C2}'(S_1)\text{C2}'(S_2)...\text{C2}'(S_m)^+$ where $^+$ In the rest of this chapter we will not distinguish between C2
C2'(S_i) = C2(S_i) if S_i is an edge in E. Otherwise, 
C2'(S_i) = \bigcup_{\text{all } \overline{P}_{S_i}} \text{Cl}(\overline{P}_{S_i}) \text{ where } \overline{P}_{S_i} \text{ is any reverse path from } O(S_i) \text{ to } T(S_i).

In Figure 4.3.1 we find

Let \text{FPL}_{\text{TREE}} = (1,2)(2,3)(2,12)(3,4)(4,5)(5,6)(6,7)(7,8)(8,9)(9,10)
(10,11)(12,13) \quad \text{FPL(G) = FPL}_{\text{TREE}}[(10,7)(7,8)(8,9)(9,10)]^2[(6,4)(4,5)(5,6)]^2
[(11,3)(3,4)(4,6)(6,7)(7,10)(10,11)]^2[(13,2)(2,12)(12,13)]^2

C2'. C2 will denote the more general cost function. C1 will denote the general cost function C1'.
\[ [(9,1)(1,2)(2,3)(3,7)(7,9)]^2 \]

\[ \ldots \]^2 denotes the contents of the brackets listed twice.

Figure 4.3.1. Example of FPL(G).

path \( p = 12\ 13\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 1 \) can be completely represented as follows--

\[
p(1) = (12,13)(13,2)(2,3)(3,7)(7,9)(9,1)
\]

\[
p(2) = (3,4)(4,6)(6,7),\ (7,8)(8,9)
\]

\[
p(3) = (4,5)(5,6)
\]

\[
\text{CR}(p) = p(3)\ p(2)\ p(1).
\]

Using Figure 4.3.1 we illustrate an important fact. Suppose \( \text{BPL}(G) = \overline{\text{FPL}(G)} \). That is, \( \text{FPL}(G) = \overline{\text{FPL}_{\text{TREE}}} \)

\[
[(9,7)(7,3)(3,2)(2,1)(1,9)]^2
\]

\[
[(13,12)(12,2)(2,13)]^2
\]

\[
[(11,10)(10,7)(7,6)(6,4)(4,3)(3,11)]^2
\]

\[
[(6,5)(5,4)(4,6)]^2
\]

\[
[(10,9)(9,8)(8,7)(7,10)]^2 \overline{\text{FPL}_{\text{TREE}}}
\]

\[
\overline{p}(1) = (1,9)(9,7)(7,3)(3,2)(2,13)(13,12)
\]

\[
\overline{p}(2) = (9,8)(8,7)(7,6)(6,5)(5,4)(4,3).
\]

Clearly \( \text{CR}(p) \neq \overline{\text{CR}(p)} \). The significance of this statement will now be discussed.

The data flow algorithms based on path listings must determine the cost sequence of path listings. The algorithms, given in the following section, determine cost sequences efficiently by making use of intermediate results as much as possible. For example, the cost of segment (3,7) in \( p(1) \) above is determined using values \( C_2(3,4), C_2(4,6) \) and \( C_2(6,7). \) (3,4),(4,6) and (6,7) are segments in \( p(2) \). The intermediate value for segment (4,6),
in turn, is determined using \( C2(4,5) \), \( C2(5,6) \). On the other hand
the cost of segment \((7,3)\) in \( P(1) \) is determined by directly
using \( C2(7,6) \), \( C2(6,5) \), \( C2(5,4) \) and \( C2(4,3) \). Since \((7,6)\),
\((6,5),(5,4)\) and \((4,3)\) are edges in \( P(2) \), no intermediate values
are used in computing \( C2(7,3) \). That is, the presence of the
segment \((4,6)\), which is not an edge in \( E \) and precedes \((3,7)\) in
FPL(\(G\)), asserts the availability of \( C2(4,6) \) as an intermediate
value useful in computing \( C2(3,7) \). It is desirable to use many
intermediate values when determining the cost of segments. This
would reduce the number of steps required to determine the cost of
each segment in a pathlisting and help shorten the length of the
pathlisting.

The algorithm in the following section determines FPL(\(G\))
such that, in general, many intermediate values are generated and
used. That is, in general the sequence \( CR(p) \) for any path \( p \)
is of length greater than two. In the same section we also give
an algorithm for determining a BPL(\(G\)). However, we find that in
contrast with the above algorithm, fewer reuseable intermediate
values can be generated. This makes BPL(\(G\)) longer than FPL(\(G\))
in general. We shall also discuss the property of the BFP which
makes this inefficiency unavoidable.

Once the sequence \( PL(G) \) and the cost of each segment \( S_i \)
in \( PL(G) \) is determined, we can solve for the FFP (or BFP)
using the FSOLVE (or BSOLVE) algorithm. These algorithms can
be thought of as propagating data 'through' each 'channel' \( S_i \).
4.3.3 Algorithm FSOLVE.

INPUT: FPL(G) with the value $C_2(S_i)$ for each $S_i$ in FPL(G).

$m = \text{length of } FPL(G)$

OUTPUT: $FF(i)$ for each $i \in N$.

#Initialize $FF'$ attribute of each node#

FOR $i = 1$ TO $n$
    DO $FF'(i) = I2(i)$ END

FOR $i = 1$ TO $m$ #the length of FPL(G)#
    DO $FF'(T(S_i)) = FF'(T(S_i)) \cup FF'(O(S_i)) \cap C2(S_i)$ END

FOR $i = 1$ TO $n$
    DO $FF(i) = FF'(i)$ END

4.3.4 Algorithm BSOLVE.

INPUT: BPL(G) with the value $C_1(S_i)$ for each $S_i$ in BPL(G).

$m' = \text{length of } BPL(G)$

OUTPUT: $BF(i)$ for each $i \in N$

#Initialize $BF'$ Attribute of each node#

FOR $i = 1$ TO $n$ DO $BF'(i) = I1(i)$ END

FOR $i = 1$ TO $m'$ #traverse the length of BPL(G)#
    DO $BF'(T(S_i)) = BF(T(S_i)) \cup BF'(O(S_i)) \cap C1(S_i)$ END

FOR $i = ITO n$
    DO $BF(i) = BF'(i)$ END
In the following lemma we show that FSOLVE applied to FPL(G) results in the value FF(i) for each node i ∈ N.

4.3.5. Lemma. FF' is the desired function FF.

Proof. Let p be any simple path u_1u_2...u_m and let CR(p) = p(4)p(3)...p(2)p(1). Since FPL(G) is processed in a left to right order, the representation α_i', α_i ∈ p(i+1), of any segment S_i in α_i, α_i ∈ p(i), is processed before the segment S_i. Let α_i+1 = S_i+1,x_1 S_i+1,x_2...S_i+1,x_i+1. FSOLVE applied to α_i+1 implies for each j, 1 ≤ j ≤ i+1, FF'(T(S_i)) ⊇ I2(0(S_i+1,x_j)) ∩ C2(S_i+1,j) ∩ C2(S_i+1,x_i+1).

Let α_i+1 ∈ p(i+1) be such that α_i+1 is the representation of some segment S_i,S_i in α_i, and the two-tuples in α_i+1 are all edges of G. Let a_1 a_2...a_k be the path S_i. FSOLVE applied to FPL(G) implies FF'(T(S_i)) ⊇ I2(a_k) ∪ C2(a_j,a_k) ∩ I2(a_k), 1 ≤ j ≤ k. That is, data from each node on path S_i is propagated to T(S_i).

Suppose α_i+1 ∈ p(i+1) is the representation of S_i in α_i, α_i ∈ p(i), and that α_i+1 can be written as S_i+1,x_1 S_i+1,x_2... S_i+1,x_k S_i+1,x_{k+1}. Let FF'(T(S_i+1,x_j) ⊇ I2(T(S_i+1,x_j)) ∪ C2(u,T(S_i+1,x_j)) ∩ I2(u), for each node u on the path S_{i+1},x_j, after the application of FSOLVE, 1 ≤ j ≤ k_{i+1}.

When FSOLVE is applied to α_i+1 we have FF'(T(S_i)) ⊇ FF'(T(S_i)) ∪ C2(T(S_i+1,x_j),T(S_i+1,x_k)) ∩ FF'(T(S_i+1,x_j)) for 1 ≤ j ≤ k_{i+1}. Substituting for FF'(T(S_i+1,x_j)) (induction
hypothesis) we have the following--

\[ \text{FF}'(T(S_i)) \supseteq \text{FF}'(T(S_i)) \cup \text{C2}(T(S_{i+1}, x_i), T(S_i)) \cap [\text{I2}(T(S_{i+1}, x_j)) \cup \text{C2}(u, T(S_{i+1}, x_j)) \cap \text{I2}(u)] \]

for each node \( u \) in segment \( S_{i+1}, x_j \) of path \( p \). (Recall that \( \text{FF}'(u) \) is initially \( \text{I2}(u) \)). That is,

\[ \text{FF}'(T(S_i)) \supseteq \text{FF}'(T(S_i)) \cup \text{I2}(T(S_{i+1}, x_j)) \cap \text{C2}(T(S_{i+1}, x_j), T(S_i)) \]

\[ \cup \text{C2}(u, T(S_{i+1}, x_j)) \cap \text{C2}(T(S_{i+1}, x_j), T(S_i)) \cap \text{I2}(u) . \]

That is, \( \text{FF}'(T(S_i)) \supseteq \text{FF}'(T(S_i)) \cup \text{C2}(u, T(S_{i+1}, x_j)) \cap \text{I2}(u) \) for each node \( u \) in each segment \( S_{i+1}, x_j \) of path \( p \), \( 1 \leq j \leq i+1 \).

Therefore we conclude that \( \text{FF}'(T(p)) \supseteq \text{I}(T(p)) \cup \text{C}(u_j, T(p)) \cap \text{I}(u_j) \) for each node \( u_j \) in path \( p \), \( 1 \leq j \leq m \). That is, \( \text{FF}' \) is the desired function \( \text{FF} \).

---

**Conclusion**

\[ S_i \]

**Induction Hypothesis**

\[ S_{i+1,1} \quad S_{i+1,2} \quad S_{i+1,3} \quad S_{i+1,4} \]

A dashed edge from node \( u \) to \( T(S_{i+1}, x_j) \) denotes the fact that \( \text{FF}'(T(S_{i+1}, x_j)) \) contains

\[ \text{C}(u, T(S_{i+1}, x_j)) \cap \text{FF}'(u) . \]

---

**Figure 4.3.2. Propagation of Data by Segments in FPL(G).**
4.3.6. **Lemma.** BF' is the desired function BF.

**Proof.** Similar to the proof of Lemma 4.3.4.

Examining the algorithms FSOLVE and BSOLVE, we find that the complexities of these algorithms are directly related to the length of FPL(G) and of BPL(G). The advantage of introducing the pathlisting concept lies in the fact that 'short' path listings can be determined for reducible flow graphs. We discuss how a pathlisting efficiently controls the flow of data using the FFP as an example. Suppose $CR(p) = p(i)p(i-1)\ldots p(1)$ completely represents path $p = x_1x_2\ldots x_k$ and at some stage data is propagated from node $x_i$ to $x_j = T(S_{m,u})$, where $i < j$, nodes $x_i, x_j$ belong to segment $S_{m,u}$ of $p$, and $S_{m,u} \in \alpha_m$ such that $\alpha_m$ represents $S_{m-1,v}$ which is a segment in $p(m-1)$. At a later stage this information can be retrieved as an attribute of node $T(S_{m,u})$ and propagated to $T(S_{m-1,v})$, when segments in $\alpha_m$ are processed by FSOLVE. That is, repogitation of data from $x_i$ to $x_j$ can be avoided when determining the data flow to node $T(S_{m-1,u})$.

In the following section we present algorithms to determine 'desirable' values for FPL(G) and BPL(G).

4.4. **Algorithms for Forward and Backward Flow Analysis.**

The algorithms given below use variations of the 'path compression' technique [T5] to obtain FPL(G) and BPL(G). We will discuss how to improve this method by using balancing techniques [T5], at a later time (Section 4.6). We first present the main algorithm for obtaining FPL(G).
4.4.1. Algorithm Main FFP.

Input: DT with attributes BT(i), BO(i), ET(i), EO(i), 
       CO(i), CT(i), FO(i), FT(i), NDT(i), EDT(i), L(i), 
       L^T(i), C1(e), C2(e), for each e ∈ EO(i), for 
       each node i.

FPL, FPL(G): Global attributes.

Output: FPL(G).

#The main program lists the three main traversals of DT#

BEGIN

0. Initialize FPL

1. TREE (FPL,DT) #The value of FPL at this point will be 
                  later referred to as FPL_TREE #

2. UNSTRUCTURED (FPL,DT).

3. TREE(FPL,DT)

4. FPL(G)=FPL.

END

4.4.2. Algorithm TREE

Input: DT as before.

       Global attribute FPL.

Output: FPL .

BEGIN

#Traverse DT in decreasing order as specified by the N1 value of 
nodes. For details on indexing refer to 4.1.3#

1. FOR i = 1 to n^+ 

+ The algorithms given in this section will assume the nodes of DT 
are indexed in various ways. A component preceding any traversal 
of DT will specify the particular indexing used.
DO #local semantics#

2. FOR each \((j,i) \in ET(i) - BT(i)\)
3. DO FPL \(\leftarrow FPL \land (j,i)\)
   OD #'\land' denotes concatenation#

OD

END

The following sub-algorithm uses 'path compression' to determine the additional segments to be added to FPL. Whenever a path \(p = p_1\ldots p_k\) is 'compressed', 2-tuples \((p_1p_2),(p_2p_3),\ldots (p_{k-1}p_k)\) are added to FPL. This ensures that the 'compressed' path, denoted by the 2-tuple \((p_1p_k)\), is represented.

4.4.3 Algorithm UNSTRUCTURED FFP

Input: DT as before.

TT: global attribute denoting DT, implemented as a list structure.

Output: FPL

0. Initialize TT as DT.

#Traverse DT in reverse depth first order#

1. FOR \(i = n \text{ STEP } - 1 \text{ TO } 0\)
   #Local semantics# .
2. WHILE \(u \in NDT(i)\) #NDT(i) contains nodes dominated by \(i\), in topological order#
3. DO Initialize list \(L(u)\);
4. \(\text{WHILE } e \in EDT(i) \text{ AND } T(e) = u \text{ OR } e \in BT(u)\)
5. DO IF \(e \in EDT(i) \text{ AND } T(e) = u \text{ AND } x(e) \neq \text{id}(T(e))\) #process edges dominated by \(i\) and with target node
u before processing back edges with target node u#

6. **THEN** f ← e;
   delete e from EDT(i);

7. **ELSE** f ← e;
   delete e from BT(u);

   #Compress path T(f) \uparrow_{TT} 0(f) if f is a back edge.
   Otherwise, compress path x(f) \uparrow_{TT} 0(f)#

8. FIXED_ANCESTOR ← \begin{cases} T(f) & \text{if } f \text{ is a back edge} \\ x(f) & \text{otherwise} \end{cases};

9. Initialize L^T(u); #Temporary list#

10. Let FIXED_ANCESTOR = a_1, a_2, ..., a_k = 0(f) be the nodes on the path FIXED_ANCESTOR \uparrow_{TT} 0(f);

11. **FOR** j = 2, ..., k

12. **DO** Make FIXED_ANCESTOR the new ancestor of a_j in TT:
    #Determine cost of compressed path (a_1, a_j)#

13. C2(FIXED_ANCESTOR, a_j) ← C2(FIXED_ANCESTOR, a_{j-1}) \cap 

    \cap C2(a_{j-1}, a_j);

14. L^T(u) ← L^T(u) \wedge (a_{j-1}, a_j);

**OD**

15. **IF** e ∈ BT(u)

16. **THEN** L(u) ← L(u) \wedge [e \wedge L^T(u)]^2;

17. **ELSE** L(u) ← L(u) \wedge L^T(u) \wedge e;

   #Compute and 'save' cost of path
   \begin{align*}
   &id(T(e)) \uparrow_{TT} x(e) \uparrow_{TT} 0(e) \uparrow_{TT} T(e) \\
   &C2(id(T(e)), T(e)) \cap C2(id(T(e)), T(e)) \\
   &U C2(id(T(e)), x(e)) \cap C2(x(e), 0(e)) \cap C2(e);
   \end{align*}

**OD** #Note line 17 determines the cost of a dominator tree edge#
16. Remove duplicates from the list \( L(u) \)

19. \( \text{FPL} \leftarrow \text{FPL} \land L(u) \)

\text{OD}

![Diagram with nodes and arrows]

Figure 4.4.1 \( G \) and its underlying structure--DT of \( G \).

In order to avoid the confusion of referring to the same node using different indices (specified for different traversals of DT), we will refer to each node only by its depth first number.
4.4.4. **Example to determine FPL(g)** (See Figure 4.4.1)

After step 1 of Algorithm MAIN FFP (4.4.2)

\[
FPL = FPL_{\text{tree}} = (1, 2) \\
(2, 3) \\
(3, 4) \\
(4, 9) \\
(9, 10) \\
(10, 11) \\
(11, 14) \\
(14, 15) \\
(15, 17) \\
(15, 16) (11, 16) \\
(11, 12) \\
(12, 13) (15, 13) \\
(4, 5) \\
(5, 6) \\
(6, 8) (13, 8) \\
(8, 7) .
\]

Steps taken by Algorithm UNSTRUCTURED FFP (4.4.3).

\[i = 17, \text{ NDT}(i) = \emptyset\]  
\[i = 15, \text{ NDT}(i) = 17 \quad \text{EDT}(i) = \{(15, 17)\}\]  
\[i = 14, \text{ NDT}(i) = 15 \quad \text{EDT}(i) = \{(14, 15)\}\]  
\[i = 16, \text{ NDT}(i) = \emptyset\]  
\[i = 12, \text{ NDT}(i) = \emptyset\]  
\[i = 13, \text{ NDT}(i) = \emptyset\]
Figure 4.4.2. Values assumed by TT (cont'd on following page).
Figure 4.4.2. Values assumed by TT (continuation).
\[ i = 11, \text{NDT}(i) = \{14, 16, 12, 13\} \]
\[ \text{EDT}(i) = \{(11,14),(15,16),(11,16),(11,12),(12,13),(15,13)\} \]
\[ u = 14, \]
\[ u = 16, L^T(u) = (14,15) \wedge (15,16) \]
\[ u = 12, \]
\[ u = 13, L^T(u) = (12,13) \]
\[ L^T(u) = (14,15) \wedge (15,13) \]
\[ i = 10, \text{NDT}(i) = \{11\} \]
\[ \text{EDT}(i) = \{(10,11)\} \]
\[ u = 11, L^T(u) = [(16,11)(11,16)]^2 \]
\[ i = 9, \text{NDT}(i) = \{10\} \]
\[ \text{EDT}(i) = \{(9,10)\} \]
\[ u = 10, L^T(u) = [(17,10),(10,11),(11,14),(14,15),(15,17)]^2 \]
See Figure 4.4.2b.
\[ i = 6, \text{NDT}(i) = \emptyset \]
\[ \text{EDT}(i) = \emptyset \]
\[ i = 5, \text{NDT}(i) = 6 \]
\[ \text{EDT}(i) = \{(5,6)\} \]
\[ i = 8, \text{NDT}(i) = \emptyset \]
\[ i = 7, \text{NDT}(i) = \emptyset \]
\[ i = 4, \text{NDT}(i) = \{9,5,8,7\} \]
\[ \text{EDT}(i) = \{(4,9),(4,5),(13,8),(6,8),(8,7)\} \]
\[ u = 9, L^T(u) = [(13,9),(9,10),(10,11),(11,13)]^2, \text{ (See Figure 4.4.2c)} \]
\[ u = 8, L^T(u) = (9,13),(13,8), \]
\[ L^T(u) = (5,6),(6,8) \]
\[ u = 7, \quad L^T(u) = (8, 7) \]

\[ i = 3, \quad NDT(i) = 4 \]
\[ EDT(1) = \{(3, 4)\} \]
\[ u = 4, \quad L^T(u) = [(7, 4), (4, 7)]^2 \]

\[ i = 2, \quad NDT(i) = 3 \]
\[ EDT(i) = \{(2, 3)\} \]
\[ u = 3, \quad L^T(u) = [(6, 3), (3, 4), (4, 5), (5, 6)]^2 \]
See Figure 4.4.2d.

\[ i = 1, \quad NDT(i) = 2 \]
\[ EDT(i) = \{(1, 2)\} \]

\[ i = 0, \quad NDT(i) = 1 \]
\[ u = 1, \quad L^T(u) = [(10, 1), (1, 2), (2, 3), (3, 4), (4, 9), (9, 10)]^2 \]
See Figure 4.4.2e.

\[ FPL_{\text{TREE}} = (1, 2), (2, 3), (3, 4), (4, 9), (9, 10), (10, 11), (11, 14), (14, 15) \]
\[ (15, 17), (11, 16), (15, 16), (11, 12), (12, 13), (15, 13) \]
\[ (4, 5), (5, 6), (6, 8), (13, 8), (8, 7) \]

\[ FPL = FPL_{\text{TREE}} \land (14, 15), (15, 16), (12, 13), (14, 15), (15, 13) \]
\[ [(16, 11), (11, 16)]^2 \]
\[ [(17, 10), (10, 11), (11, 14), (14, 15), (15, 17)]^2 \]
\[ [(13, 9), (9, 10), (10, 11), (11, 13)]^2 \]
\[ (9, 13), (13, 8), (5, 6), (6, 8), (8, 7) \]
\[ [(7, 4), (4, 7)]^2 \]
\[ [(6, 3), (3, 4), (4, 5), (5, 6)]^2 \]
\[ [(10, 1), (1, 2), (2, 3), (3, 4), (4, 9), (9, 10)]^2 \land FPL_{\text{TREE}} \]

Path \[ p = 16, 11, 12, 13, 8, 7, 4, 5, 6, 3 \] can be completely represented
at \[ CR(p) = p(2)p(1) \]
\[ p(1) = (16,11),(11,13),(13,8),(8,7),(7,4),(4,5),(5,6),(6,3) \]
\[ p(2) = (11,12),(12,13) \]

For path \[ p = 9,10,11,12,13,8,7,4 \]
\[ p(1) = (9,13),(13,8),(7,4) \]
\[ p(2) = (9,10),(10,11),(11,13) \]
\[ p(3) = (11,12),(12,13) \]
\[ CR(p) = p(3)p(2)p(1) \]

Main algorithm for the BFP follows:

4.4.5. **Algorithm MAIN BFP**

**Input:** DT with attributes as before

Global attributes

BLIST: containing an ordering of B for backward flow.

BPL,BPL(G)

**Output:** BPL(G).

0. Initialize BPL.

#The main program lists the main passes of DT#

1. Determine BLIST

2. TREE(X,DT). #X is a temporary list, \( \overline{X} \) is \( \overline{\text{FPL TREE}} \#.

3. \( \text{BPL} \leftarrow \overline{X} \)

4. Compute cost of DT edges.

5. \text{UNSTRUCTURED BFP (BPL,DT)},

6. \( \text{BPL} \leftarrow \text{BPL} \wedge \overline{X} \).

7. \( \text{BPL}(G) \leftarrow \text{BPL} \).

Step 4 involves computing the cost of the dominator tree (DT) edges in a manner similar to that given in line 17 (Algorithm 4.4.3),
during a reverse depth first traversal of DT. The following sub-
algorithm gives details for processing the 'unstructured' edges of
G.

4.4.6. Algorithm UNSTRUCTURED BFP

Input: DT as before together with additional attributes--
\forall node i, attribute Cl(h,i) denoting the cost of
DT edge (h,i), h is the predecessor of i.

Output: BPL

1. Initialize TT' as DT with all the edges reversed.
   #Traverse DT in depth first order#

2. FOR i = 1 TO n
   DO Initialize a list L(i)
   WHILE edges in EDT (predecessor of i).AND.BO(i).AND.
   BT(i) have not all been examined.

3. DO IF e \in EDT (predecessor of i).AND.X(e) = i
   #Process DAG edges e with 'origin' node X(e)
   first#.
   THEN BE \leftarrow e

4. ELSE IF e \in (BO(i) OR BT(i)).AND.e is the next
   edge on BLIST

5. THEN BE \leftarrow e^1 e^k \ldots e^q is the sequence of
   back edges which have the same super-
   script as e. (If e has no super-
   script then BE = e);
   delete each edge placed in BE from
   BLIST;
6. Initialize a temporary list $L^T(i)$;

7. WHILE There is an unexamined edge $e$ in BE

8. DO $\text{FINAL} \leftarrow \begin{cases} \text{id}(\xi(e)) & \text{if } e \text{ is a DAG edge;} \\ T(e) & \text{if } e \in \text{BLIST} \end{cases}$

   $\#$'reverse compress' path $O(e)_{TT'}^\to \text{FINAL}$

9. Let $O(e) = a_1, a_2, \ldots, a_q = \text{FINAL}$ be the sequence of nodes on $O(e)_{TT'}^\to \text{FINAL}$;

10. VARIABLE_ANCESTOR $\leftarrow O(e)$;

11. FOR $j = 2$ to $q$

12. DO make VARIABLE_ANCESTOR the new ancestor of $a_j$ in $TT'$; Compute

13. $C_1(\text{VARIABLE_ANCESTOR}, a_j) \leftarrow C_1(\text{VARIABLE_ANCESTOR}, a_j-1) \cap C_1(a_j-1, a_j)$

14. $L^T(i) = L^T(i)_{a_j-1, a_j}$;

15. IF $BO(a_j) \cap \text{BLIST} \neq \emptyset$ OR $\text{MARK}(a_j) = 1$

   OR $e \in [EO(a_j) - BO(a_j)]$ AND $e \in \text{EDT}(h)$

   for some $h$, an ancestor of $j$

   THEN VARIABLE_ANCESTOR $\leftarrow a_j$ $\#$partial compression$\#$

   OD

16. IF $e$ is a DAG edge THEN $L(i) = L(i) \land \xi \land L^T(i)$

17. ELSE IF $e$ is not super-scripted

   THEN $L(i) = L(i) \land [\xi \land L^T(i)]^2$

   ELSE remove duplicates from

* To facilitate this at each node keep track of the unique successor of the node in $TT'$ which has descendants. Update this information by adding steps to line 17. To determine the nodes on a path trace from node $O(e)$ to FINAL using the successor information at each node.
from \( L^T(i) \); \#See Fig. 4.4.3#

\[ L(i) \leftarrow L(i) \land \left[ e_1^{k} e_2^{k} \ldots e_v^{k} \land L^T(i) \right]^2 \]

OD

18. \( \text{BPL} \leftarrow \text{BPL} \land L(i); \)

OD

We now use the Figure 4.4.1 to illustrate the Algorithm

MAIN BFP.

4.4.7. Example to determine \( \text{BPL}(G) \). (See Figure 4.4.1)

\( \text{BLIST} \) determined by step 1 of 4.4.5 is as follows:

\[ \text{BLIST} = (10,1)^3, (6,3)^3, (7,4)^3, (13,9)^{11}, (17,10)^{11}, (16,11)^{11} \]

\( i = 1, \quad \text{BT}(1) = (10,1) \)

\[ L(i) = [(1,10), (3,6), (4,7), (7,4), (6,5), (10,9), (9,4), (4,3), (3,2), (2,1)]^2 \]

See Figure 4.4.3a.

\( i = 2, \)
\( i = 3, \)
\( i = 4, \)
\( i = 7, \)
\( i = 8, \quad L(i) = (7,8), (8,4) \)
\( i = 5, \quad L(i) = (8,6), (6,4) \)
\( i = 6, \)
\( i = 9, \quad L(i) = (8,13), (13,11), (11,10), (10,4) \)

\[ L(i) = L(i) \left[ (9,13), (10,17), (11,16), (13,11), (17,15), (15,14), (14,11), (11,10), (10,9) \right]^2 \]

See Fig. 4.4.3b.
\[ i = 10, \]
\[ i = 11, \]
\[ i = 13, \]
\[ i = 12, \quad L(i) = (13, 12), (12, 11) \]
\[ i = 16, \]
\[ i = 14, \quad L(i) = (13, 15), (16, 15), (15, 11) \]
\[ i = 15, \]
\[ i = 17. \]

Figure 4.4.3. Values assumed by \( TT' \)
Therefore, $BPL = \overline{FPL_{\text{TREE}}}$

\[
\begin{align*}
(1, 10), (3, 6), (4, 7), (7, 4), (6, 5), (5, 4) \\
(10, 9), (9, 4), (4, 3), (3, 2), (2, 1) \end{align*} \]

\[
(7, 8), (8, 4) \\
(8, 6), (6, 4) \\
(8, 13), (13, 11), (11, 10), (10, 4) \\
(9, 13), (10, 17), (11, 16), (13, 11), (17, 15) \\
(15, 14), (14, 11), (11, 10), (10, 9) \end{align*} \]

\[
(13, 12), (12, 11) \\
(13, 15), (16, 15), (15, 11) \overline{FPL_{\text{TREE}}} 
\]

Let $\overline{p} = 3, 6, 5, 4, 7, 8, 13, 12, 11, 16$. Then,

$\overline{p}(1) = (3, 6), (6, 5), (5, 4), (4, 7), (7, 8), (8, 13), (13, 11), (11, 16)$.

$\overline{p}(2) = (13, 12), (12, 11)$.

$CR(\overline{p}) = \overline{p}(2)\overline{p}(1)$

If $\overline{p} = 4, 7, 8, 13, 12, 11, 10, 9$, then,

$\overline{p}(1) = (4, 7), (7, 8), (8, 13), (13, 11), (11, 10), (10, 9)$

$\overline{p}(2) = (13, 12), (12, 11)$.

$CR(\overline{p}) = \overline{p}(2)\overline{p}(1)$ (Compare with Example 4.4.4).

For the reducible flow graph in Figure 4.4.1, the lengths of $FPL(G)$ and $BPL(G)$ determined in Examples 4.4.4 and 4.4.7 are roughly equal. In general, this will not be the case. We note an important difference between the path listings determined for our flow graph. Let $p = 9, 10, 11, 12, 13, 8, 7, 4$ and $\overline{p} = 4, 7, 8, 13, 12, 11, 10, 9$. $CR(p) = p(3) \ p(2) \ p(1)$ where $p(3) = (11, 12), (12, 13)$, $p(2) = (9, 10), (10, 11), (11, 13)$ and $\overline{p}(1) = (9, 13), (13, 8), (7, 4)$. On the other hand $CR(\overline{p}) = \overline{p}(2)\overline{p}(1)$.
where $\overrightarrow{p}(2) = (13, 12), (12, 11)$ and $\overrightarrow{p}(1) = (4, 7), (7, 8), (8, 13), (13, 11), (11, 10), (10, 9)$. Partial compression in line 15 of 4.4.6 is responsible for the relatively longer representation $\overrightarrow{p}(1)$ as compared with $p(1)$, and the shorter sequence $CR(\overrightarrow{p})$ as compared with $CR(p)$. A long complete representation sequence $CR(p) = p(\ell)p(\ell-1)...p(1)$ is desirable for any path $p$ (the same is true for any reverse path $\overrightarrow{p}$) for the following reason. The sequences in $p(\ell), p(\ell-1)...p(2)$ of a complete representation of $p$ can be though of as determining intermediate results which are useful in determining the data flow from each node in $p$ to $T(p) (O(p)$ if we are considering reverse path $\overrightarrow{p})$. If most of the paths have long complete representations then each of these paths can be 'processed' using lots of intermediate results determined while 'processing' other paths (see previous section). That is, if most of the paths (or reverse paths) in a graph have long complete representation sequences then more intermediate results can be used thus avoiding recomputation. Another way of stating this is as follows: if most of the paths (or reverse paths) have long complete representation sequences then the path listing for the graph may be made 'short'. In the limit case, when $CR(p)$ is of length 1 for any path $p = x_1, x_2, ... , x_k$, we have $p(1) = S_{i_1}S_{i_2}...S_{i_k}$ where $S_{i_j} = (x_j, x_{j+1})$ is a segment in the path listing. That is, the path listing is similar to a node listing. An analogous situation is true for the reverse paths of a graph.
There is a large class of reducible flow graphs for which 'short' BPL's can be obtained. Roughly speaking, the $t_i$ reverse paths in the representation of any path (see Lemma 4.2.6) in a graph belonging to this class has the property that it can be written as $x_i \xrightarrow{ST} y_i \xrightarrow{DAG} z_i$. Figure 4.4.4 illustrates a graph in this class. In the following example we show that no partial compression occurs for this graph.
Example to determine a 'short' BPL(G)

\[ \text{BLIST} = (4,1), (7,5), (8,6), (9,5), (10,2) \]

\[ i = 1, \quad L^T(i) = [(1,4), (4,3), (3,2), (2,1)]^2 \]

\[ i = 2, \]

\[ i = 3, \quad L^T(i) = [(3,7), (7,6), (6,5), (5,4), (4,3)]^2 \]

\[ i = 4, \]

\[ i = 5, \quad L^T(i) = [(5,8), (8,7), (7,5)]^2 \]

\[ L^T(i) = [(5,9), (9,8), (8,5)]^2 \]

\[ i = 6, \]

\[ i = 7, \]

\[ i = 8, \]

\[ i = 9, \]

\[ i = 10, \quad L^T(i) = [(2,10), (10,9), (9,5), (5,4), (4,2)]^2 \]

\[ \text{BPL} = \underbrace{\text{FPL}}_{\text{TREE}} (1,4), (4,3), (3,2), (2,1)]^2 \[ (3,7), (7,6), (6,5), (5,4), (4,3)]^2 \]

\[ [(5,8), (8,7), (7,5)]^2 \[ (5,9), (9,8), (8,5)]^2 \]

\[ [(2,10), (10,9), (9,5), (5,4), (4,2)]^2 \]

\[ \text{For,} \]

\[ \overline{p} = 1,4,3,2,10,9,8,7,6,5 \]

\[ \text{CR}(\overline{p}): \]

\[ \overline{p}(1) = \{(1,4), (4,3), (3,2), (2,10), (10,9), (9,5)\} \]

\[ \overline{p}(2) = \{(9,8), (8,5)\} \]

\[ \overline{p}(3) = \{(8,7), (7,5)\} \]

\[ \overline{p}(4) = \{(7,6), (6,5)\} . \]
4.5 Proofs of Correctness.

We begin this section by examining the flow between cycles. In particular, we identify certain properties of the $t_i$ paths which channel data flow between cycles.

4.5.1 Lemma. Let $t_1, b_1, t_2, b_2, \ldots, t_k, b_k, t_k$ be the representation of any simple path $p$ as in Lemma 4.2.5. Let $t_i$ be a path, in DAG of $G$, of length greater than zero such that $t_i$ can be further represented as

$$x_i \xrightarrow{\text{DAG}} y_i \xrightarrow{\text{ST}} z_i, \quad y_i \in \text{ENTER}(b_i).$$

Then, the last edge in $x_i \xrightarrow{\text{DAG}} y_i$ is a cross edge $(a_i, y_i)$ if $(a_i, y_i) \in \text{EDT}(T(b_i))$.

Proof. If $t_i$ can be represented as $x_i \xrightarrow{\text{DAG}} y_i \xrightarrow{\text{ST}} z_i$ with $y_i \in \text{ENTER}(b_i)$ then $T(b_{i-1}) \notin \text{CYCLE}(b_i)$. That is $T(b_{i-1})$ does not dominate nodes in $\text{CYCLE}(b_i)$. Therefore, the path $x_i \xrightarrow{\text{DAG}} y_i$ must contain at least one cross edge $(a_i, y_i)$. There are two non intersecting paths from $T(b_i)$ to $y_i$. These are $T(b_i) \xrightarrow{\text{ST}} T(b_{i-1}) \xrightarrow{\text{DAG}} a_i \xrightarrow{\text{DAG}} y_i$ and $T(b_i) \xrightarrow{\text{ST}} y_i$. No node which dominates $T(b_i)$ can immediately dominate $y_i$ since $y_i \in \text{CYCLE}(b_i)$ and $G$ is reducible. Therefore, $T(b_i) = \text{id}(y_i)$. It follows that $(a_i, y_i) \in \text{EDT}(T(b_i))$. See Figure 4.5.1.
The following definition imposes a restriction on the sequence of compressions carried out during the forward flow algorithm. This restriction ensures that when an attempt is made to compress $i \rightarrow_T j$ it is actually connected. Figure 4.5.2 illustrates the need for this restriction which is the same as in [T5].

**4.5.2. Definition:** Let $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ be a sequence of 2-tuples such that $i_\ell \rightarrow^* D_T j_\ell$, $1 \leq \ell < k$.

The sequence is said to be correct for compression if node $i_{\ell+1}$ dominates node $i_\ell$.

In the remainder of this section we shall say that a path has been processed if appropriate 2-tuples to completely represent the path have been added to FPL. When all the paths from any node to certain specific nodes have been processed we shall say that the
a) \(i_1\) dominates \(i_2\) 

\[
\begin{array}{c}
\text{i}_1 \\
\text{i}_2 \\
\text{b} \\
\text{c} \\
\text{j}_1 \\
\text{j}_2
\end{array}
\]

b) Compress \(i_1 \rightarrow j_1\) (1st compression) 

\[
\begin{array}{c}
\text{i}_1 \\
\text{i}_2 \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{j}_1 \\
\text{j}_2
\end{array}
\]

Compress \(i_2 \rightarrow j_2\) (1st compression) 

\[
\begin{array}{c}
\text{i}_1 \\
\text{i}_2 \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{j}_1 \\
\text{j}_2
\end{array}
\]

\(i_2\) and \(j_2\) become disconnected. Therefore, compress \(i_2 \rightarrow j_2\) is invalid if it is preceded by compress \(i_1 \rightarrow j_1\).

c) \(i_1, j_1\) are still connected 

\[
\begin{array}{c}
\text{i}_1 \\
\text{i}_2 \\
\text{b} \\
\text{c} \\
\text{j}_1 \\
\text{j}_2
\end{array}
\]

d) Compress \(i_1 \rightarrow j_1\) (2nd compression) 

\[
\begin{array}{c}
\text{i}_1 \\
\text{i}_2 \\
\text{b} \\
\text{c} \\
\text{j}_1 \\
\text{j}_2
\end{array}
\]

Compress \(i_1 \rightarrow j_1\) can be carried out since \(i_1, j_1\) are connected after compress \(i_2 \rightarrow j_2\).

Figure 4.5.2. Proper compression order.
data flow to these nodes has been considered (or processed).

The overall strategy of algorithm 4.4.3 is to first process
the data flow to any cycle $e_a$. Then the data flow from cycle $e_a$
to cycle $e_b$ is processed. Here $e_a$ is a back edge which precedes
$e_b$ in the ordering of Lemma 4.2.7. After all the flow from
preceding cycles is processed, the flow around cycle $e_b$ is
processed. That is, the cycles of $G$ are processed on-line
according to the ordering of back edges given in Lemma 4.2.7.

Lemma 4.3.4 implies that if FPL is a sequence in which
every path $p$ of $G$ can be completely represented, then FSOLVE
applied to FPL determines the solution to the FFP. In view of
this, we must show that the cost of each segment in FPL is de-
termined and prove the following lemma.

4.5.3. **Lemma.** a) FPL is a value for FPL($G$) and
b) C2(FPL) is determined by Algorithm 4.4.1.

**Proof.** Lemma 4.2.4 shows a way of representing any simple
path $p$ as $t_1, b_1, t_2, b_2, \ldots, t_{k-1}, b_{k-1}, t_k$
such that $b_i$ is a back edge in $B$, $1 \leq i \leq k-1$, and $t_j$ is
a path in DAG from a node in CYCLE($b_{j-1}$) to a
node in CYCLE($b_j$), $2 \leq j \leq k-1$. Lemma 4.2.7 implies
that $T(b_j)$ dominates $T(b_{j-1})$. Algorithm 4.4.3
processes the back edges (i.e., the cycles of $G$) in
reverse depth first order according to the occurrence
of their target nodes in DT. Hence, CYCLE($b_{j-1}$)
is processed before CYCLE($b_j$) and the back edges
and 'parts' of the DAG paths are processed in the
order $t_1, b_1, t_2, b_2, \ldots, b_{k-1}, t_k$.

Intuitively, to process $\text{CYCLE}(b_j)$, $1 \leq j \leq k-1$, means data propagated to a node $u$, $u \in \text{ENTER}(b_j)$, along a DAG path $t_j$, is propagated to each node $v \in \text{EXIT}(b_j)$. This can ordinarily be done by two listing+ the nodes of $\text{CYCLE}(b_j)$. This is the reason for listing $L^T(u)$ twice in line 12 of algorithm 4.4.3. However, path compression, a device used to obtain long complete representations for paths, complicates the picture.

Path compression is carried out on $TT$, a structure initially denoting $DT$. Algorithm 4.4.3 determines the cost of each edge in $DT$ as follows. During its reverse depth first traversal of $DT$ (with index $i$) it examines the edges dominated by node $i$ (i.e., $\text{EDT}(i)$). Let $e$ be some edge in $\text{EDT}(i)$. The cost of $DT$ edge $(\text{id}(T(e)), T(e))$ is not known and must be computed. Line 17 (4.4.3) computes this cost which must reflect all the DAG paths from $\text{id}(T(e))$ to $T(e)$.

As $DT$ is traversed, the global attribute $TT$ may be altered several times. In the following discussion any reference to $TT$ will imply its most recent value.

Line 12 (4.4.3) implies the compression of path $p = a_1, a_2, \ldots, a_k$ in $TT$ causes edges $(a_1, a_2)(a_1, a_3), \ldots, (a_1, a_k)$ to be added to $TT$ after deleting the edges to the original ancestors of $a_3, a_4, \ldots, a_{k-1}, a_k$. At the same time the sequence $(a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k)$ is added to FPL and the cost of compressed paths $(a_1, a_2), (a_1, a_3), \ldots, (a_1, a_k)$ is computed (see lines 12, 13, and 14 of 4.4.3). That is, if $(a_1 a_\ell)$, $2 \leq \ell \leq k$, denoting compressed

+ For more details refer to [Ke1].
paths is added to TT then

a) $C_2'(a_1, a_2)$ is computed,

b) $(a_1, a_2)$ is represented in FPL.

If we assume that $(a_m, a_{m+1})$, $1 \leq m < k$, is completely represented in FPL then we can inductively conclude

c) $(a_1, a_2)$ is completely represented in FPL.

Therefore, if an edge $e$ in TT is added to FPL, we know that any path from $0(e)$ to $T(e)$ can be completely represented.

To determine the actual path listing, the algorithm 4.4.3 performs successive path compressions on TT correctly (according to 4.5.2) as we now show. We note that if $T(b_j)$ dominate $T(b_{j-1})$, for $b_j, b_{j-1} \in B$, then $T(b_j) \xrightarrow{TT} 0(b_{j-1})$ is compressed before $T(b_j) \xrightarrow{TT} 0(b_j)$. We must show that compressions stemming from DAG edges correctly intersperse the compressions stemming from back edges. This is obvious since DT is traversed in reverse depth first order with index $i$ and at each node $i$, EDT($i$) is examined and $x(e) \xrightarrow{TT} 0(e)$ is compressed for some $e$'s in EDT($i$). If $x(e)$ is $i$ then no compression occurs. On the other hand, if $x(e)$ is a successor of $i$ in DT then compression occurs.

We have shown that the cycles in the representation of any simple path are processed in the proper order. To show that each cycle in any representation is also processed properly, we note the following. If $(m,n)$ is a back edge then the sequence of compressed paths $(m,n)(n,a_2)(a_2,a_3)\ldots(a_{k-1},m)$, where $n = a_1, a_2, \ldots, a_k = m$ is the sequence of nodes on the path $n \xrightarrow{TT} m$ at stage $i$, is listed twice in FPL (line 16 of 4.4.3). Since we have inductively argued that the compressed paths in the above sequence can be completely repre-
sented, data flow from each node 'a' in CYCLE(m,n) to each node
\( s_j \in \text{CYCLE}(m,n) \), \( 1 \leq j \leq k \), is processed.

Algorithm 4.4.3 processes the cycles of G in an on-line
fashion according to the ordering of Lemma 4.2.7. Each cycle e
in the ordering is processed only after all the flow from each
cycle f preceding e in the ordering is processed. Next we show
that the flow between cycles along DAG paths is correctly processed.

If \( t_j \) is a DAG path such that \( T(b_{j-1}) \xrightarrow{\text{DAG}} y_j \xrightarrow{\text{ST}} o(b_j) \)
and \( y_j \in \text{ENTER}(b_j) \) then 4.4.3 ensures \( T(b_{j-1}) \xrightarrow{\text{TT}} o(b_{j-1}) \) is
compressed before \( T(b_j) \xrightarrow{\text{TT}} o(b_j) \). Consequently, if all the data
flow to \( \text{CYCLE}(b_{j-1}) \) is processed, then the flow from \( \text{CYCLE}(b_{j-1}) \)
to nodes in \( \text{CYCLE}(b_j) \) is processed automatically by adding the
appropriate 2-tuples to propagate the incoming flow around \( \text{CYCLE}(b_j) \).

Suppose \( t_j \) is the path \( T(b_{j-1}) \xrightarrow{\text{DAG}} y_j \xrightarrow{\text{DAG}} o(b_j) \) and
\( y_j \in \text{ENTER}(b_j) \). By Lemma 4.5.1, the last edge \( (a_j,y_j) \) in
\( T(b_{j-1}) \xrightarrow{\text{DAG}} y_j \) is a cross edge which also belongs to the set
\( \text{EDT}(T(b_j)) \). When node \( T(b_j) \) is traversed, \( x(a_j,y_j) \xrightarrow{\text{TT}} a_j \) is
compressed, the 2-tuples added to FPL propagate the flow from any
node on \( x(a_j,y_j) \xrightarrow{\text{DT}} a_j \) to \( y_j \). Lines 5 and 7 of 4.4.3 imply
this compression occurs before \( \text{CYCLE}(b_j) \) is processed. It remains
to be shown that as the subtree of DT with root \( T(b_j) \) is tra-
versed in reverse depth first order, 2-tuples added to FPL ensure
that the flow from nodes on \( t_j \) to the above path \( x(a_j,y_j) \xrightarrow{\text{TT}} a_j \)
is processed.

The path \( t_j \) can be written as \( d_1 e_1 d_2 e_2 \ldots e_d \)

a) \( d_i \) is a path (possibly of length zero) in the DAG

b) \( e_i \) is an edge on \( T(b_{j-1}) \xrightarrow{\text{DAG}} y_i \) such that
When node $\text{id}(T(e_i))$ is traversed the 2-tuples added to FPL ensure that the data flow from any node on $x(e_i)$ to $T(e_i)$ is processed. The 2-tuples added for the $e_i$ edges process the flow from each node on $t_j$ to nodes on $x(a_j, y_j)$ as we shall show next. (See Figure 4.5.3).

![Figure 4.5.3. Flow between cycles](image)

When node $\text{id}(T(e_i))$ is traversed, the 2-tuples added to FPL represent the segment $(\text{id}(T(e_i)), T(e_i))$ and the cost of the path $\text{id}(T(e_i)) \xrightarrow{\text{DAG}} x(e_i) \xrightarrow{\text{DAG}} 0(e_i) \xrightarrow{\text{DAG}} T(e_i)$ is computed. Now consider the positions of $e_{i+1}$ relative to $e_i$.

a) Suppose $\text{id}(T(e_{i+1}))$ dominates $\text{id}(T(e_i))$ (Figure 4.5.4a).

In this case compression of $x(e_i) \xrightarrow{\text{TT}} 0(e_i)$ results in the addition of 2-tuples representing the segment $\text{id}(T(e_i), T(e_i))$ which is an edge on $x(e_{i+1}) \xrightarrow{\text{TT}} 0(e_{i+1})$. Therefore the flow from each node on $x(e_i) \xrightarrow{\text{TT}} 0(e_{i+1})$ is processed.
b) Suppose $id(T(e_{i+1}))$ is dominated by $id(T(e_i))$. (Figure 4.5.4b). In this case the 2-tuples added to FPL during compressions $x(e_{i+1}) \xrightarrow{TT} 0(e_{i+1})$ and $x(e_i) \xrightarrow{TT} 0(e_i)$ serve as representations of DT edges $(id(T(e_i)), T(e_i))$ and $(id(T(e_{i+1})), T(e_{i+1}))$ which are on $x(e_h) \xrightarrow{TT} 0(e_h)$ where $T(b_{j}) \xrightarrow{DT} id(T(e_h)) \xrightarrow{DT} id(t(e_i))$, $i+1 < h \leq \ell$.

It therefore follows that $t_j$ is completely represented. Also, the flow between cycles of $p$ is processed in an 'on line' fashion. First the flow around the CYCLE($b_{j-1}$) is processed. By Lemma 4.5.1 $(a_j, y_j) \in EDT(T(b_j))$. Therefore, the flow from CYCLE($b_{j-1}$) to CYCLE($b_j$) along path $t_j$ is processed before the flow around CYCLE($b_j$) is processed (see lines 5 and 6 of 4.4.3). That is, assuming the flow to nodes in CYCLE($b_{j-1}$) is correctly processed we have argued that the flow to $y_j$ via path $T(b_{j-1}) \xrightarrow{\text{DAG}} y_j$ is correctly processed. Finally, the 2-listing of compressed paths of CYCLE($b_j$) ensures that the flow from $y_j$ to each node is CYCLE($b_j$) is correctly processed.

By requiring the initial and the final subsequencies of FPL to be FPL TREE we ensure $t_1$ and $t_k$ in the representation of any path can be represented in FPL.

Therefore, we have inductively proved that if the cycles and the $t_i$ paths are processed on line as shown, every simple path can be completely represented in FPL and FPL = FPL(G).
Figure 4.5.4a. Data Flow along $T_j$ (case a).
Figure 4.5.4b. Data Flow along $T_j$ (case b).
An important consequence of the Algorithm 4.4.3 is that for each segment $S$ added to FPL, $C(S)$ is also computed. Evaluation of $C(S)$ may occur in stages and is done efficiently by using values computed earlier associated with segments in the representation of $S$. As we have discussed earlier, in the case of the backward flow algorithm it is harder to obtain reusable intermediate values which help avoid recomputation. That is, for $S$ in BPL, evaluation of $C(S)$ must generally be done using fewer intermediate values since $CR(S)$ is a 'short sequence.

Before proceeding to the proof of $BPL(G) = BPL$, we examine the compression steps of Algorithm 4.4.5 and discuss their necessarily inefficient functioning. Let $a^3$ and $b^3$ be two superscripted edges in BLIST and let $c$ be a third back edge as shown in Figure 4.5.5a. The order of appearance of these edges in BLIST would be $a^3, b^3, c$. Suppose $a^3$ and $b^3$ are compressed as in Figure 4.5.5b and 2-tuples $[(1,8),(2,4),(4,3), (8,7),(7,6),(6,5),(5,3),(3,2),(2,1)]^2$ are added to BPL. When it is time to process the back edge $c$, the nodes 7 and 5 will be disconnected. Unfortunately, unlike the situation for forward flow, the BLIST ordering necessary for the BFP is not suitable for efficient compressions. There is, however, a compression algorithm, for BFP, which is efficient for a large class of graphs (see example in Figure 4.4.4). The following discussion indicates the solution. $8,7,6,5,3,2,1$ are nodes on path $0(a^3) \xrightarrow{TT} T(a^3)$. If $u$ is any node on this path such that $e \in B0(u)$, and $e$ appears after $a^3$ on BLIST, or $u$ has more
a) RFG G.

b) Nodes 7(=O(c)) and 5(=T(c)) are disconnected in this TT'.

c) TT' after correct by partial compression of $b^3, a^3$.

d) TT' with nodes 5 and 7 connected.

Figure 4.5.4. Partial Compression.
than one descendant in DT and \( u \) is marked (see line 4, algorithm 4.2.9), or \( e \in E_0(u) - B_0(u) \) and \( x(e) \) dominates \( u \), then it is specially treated. Specifically, node 7 in the example becomes the ancestor of nodes 6, 5, 3 thereby ensuring nodes 7 and 5 are connected. That is, the necessary value \( C_1(7, 5) \) is computed at the expense of full compression illustrated in Figure 4.5.4b.

4.5.4. **Lemma** a) BPL is a value for BPL(G) and
b) \( C_1(BPL) \) is determined by Algorithm 4.4.5.

**Proof.** Lemma 4.2.6 shows a way of representing any simple path \( \overline{p} \) as \( t_1 \overline{b_1} t_2 \overline{b_2}, \ldots, t_{k-1} \overline{b_{k-1}} t_k \) such that \( b_i \in B, \ 1 \leq i \leq k, \) and \( t_j \) is a reverse path from a node in \( CYCLE(b_{j-1}) \) to a node in \( CYCLE(b_j), \ 2 \leq j \leq k-1. \) Lemma 4.2.9 implies that if there is a reverse path \( t_j \) of length greater than or equal to zero, from \( CYCLE(b_{j-1}) \) to \( CYCLE(b_j) \) then either \( 0(b_{j-1}) \) dominates \( 0(b_j) \) or \( T(e_{j-1}) \) dominates \( T(e_j) \) and \( 0(e_{j-1}), 0(e_j) \) do not dominate each other. Since back edges of \( G \) are processed in the order specified by BLIST, the cycles and 'parts' of the DAG paths in the representation of any path \( \overline{p} \) are processed in the order \( t_1, \overline{b_1}, t_2, \overline{b_2}, \ldots, t_{k-1}, \overline{b_{k-1}}, t_k \) in an on-line fashion.

Path compression for the BFP is carried out on TT', a global attribute initially denoting DT with all its edges reversed. That is, if \( e \) is an edge in DT then \( \overline{e} \) is an edge in TT'. Algorithm 4.4.5 assumes that the cost of DT edges are known. The FFP algorithm (4.4.4) gives details for calculating these values. As before, the attribute TT' is modified several times during the run of the algorithm. Any reference to TT will imply its most recent value.
Only partial compression can be implemented for most of the reducible flow graphs. The discussion that precedes this lemma indicates the reason. More specifically, property b) of Lemma 4.2.9 implies that $0(b_{j-1}) \xrightarrow{\text{TT}} T(b_{j-1})$ must be compressed before $0(b_j) \xrightarrow{\text{TT}} T(b_j)$ if $T(b_{j-1})$ dominates $T(b_j)$ but $0(b_j), 0(b_{j-1})$ do not dominate each other. But when compressing $0(b_{j-1}) \xrightarrow{\text{TT}} T(b_{j-1})$ we must ensure that nodes on $0(b_j) \xrightarrow{\text{TT}} T(b_j)$ do not get disconnected. We must also ensure that if $e$ is a back edge which appears after $b_{j-1}$ on BLIST such that $0(e)$ dominates $0(b_{j-1})$ then the nodes on $0(e) \xrightarrow{\text{TT}} T(e)$ do not get disconnected. The updating of VARIABLE_ANCESTOR in line 15 of algorithm 4.4.6 is used to guarantee this as illustrated on Figure 4.5.4. That is, instead of making $0(b_{j-1})$ the new ancestor of each node on $0(b_{j-1}) \xrightarrow{\text{TT}} T(b_{j-1})$ the current value in VARIABLE_ANCESTOR is used to determine the new ancestor. This leads to only partial compression if VARIABLE_ANCESTOR $\neq 0(b_{j-1})$.

The flow around each cycle is once again processed by two listing the compressed paths in a manner similar to the forward flow algorithm. The cost of each segment added to BPL is either given as input or computed in line 13 of 4.4.6.

We now examine the data flow between cycles $b_j$ and $b_{j-1}$. Let $t_j$ be the reverse path $T(b_j) \xrightarrow{\text{ST}} y_j \xrightarrow{\text{DAG}} 0(b_{j-1})$ such that $y_j \in \text{EXIT}(b_j)$. Either

a) $y_j \xrightarrow{\text{ST}} 0(b_{j-1})$ is along the spanning tree (Figure 4.5.5a) else

b) $y_j \xrightarrow{\text{DAG}} a_j \xrightarrow{\text{ST}} 0(b_{j-1})$ with $a_j \in \text{ENTER}(b_{j-1})$ and the first
edge on the reverse path $y_j \overset{*}{\rightarrow} a_j$ is a reverse cross edge $(c_j, a_j)$ (Figure 4.5.5b).

If $y_j \overset{0}{\rightarrow} 0(b_{j-1})$, then $0(b_{j-1})$ dominates $0(b_j)$ and the flow from CYCLE($b_{j-1}$) to CYCLE($b_j$) is processed in a straightforward manner when $0(b_j) \overset{*}{\rightarrow} T(b_j)$ is compressed.

We now consider the situation when $0(b_{j-1}), 0(b_j)$ do not dominate each other. Recall DT is traversed in depth first order with index $i$. If $t_j$ is as in a) above then CYCLE($b_{j-1}$) and CYCLE($b_j$) are both processed in the same stage. That is $b_{j-1}, b_j$ have the same superscript and therefore the 2-tuples added to BPL are compressed paths which propagate data flow from each node of CYCLE($b_{j-1}$) and CYCLE($b_j$) to each node $d, d$ on path $0(b_{j-1}) \overset{TT}{\rightarrow} T(b_{j-1})$ or on path $0(b_j) \overset{TT}{\rightarrow} T(b_j)$. (See line 3, Algorithm 4.4.5).

Figure 4.5.5. Backward flow between cycles
If $t_j$ is as in b) and the superscripts of $b_{j-1}$ and $b_j$ are the same then cycles $b_{j-1}$ and $b_j$ are processed together (see lines 5 and 7 of 4.4.6). The flow from each node in $\text{CYCLE}(b_{j-1})$ is propagated to nodes in $\text{CYCLE}(b_j)$ after being modified by the cost of DT edge $(\text{id}(a_j), a_j)$, since $\text{Cl}(\text{id}(a_j), a_j) \geq \text{Cl}(x(e), a_i)$ for each $e \in \text{EDT}(\text{id}(a_j))$ and $x(e) \neq \text{id}(a_i)$. Finally, the data flow from each node in $\text{CYCLE}(b_{j-1})$ to each node $c$ on $\overline{\text{id}(a_j) - y_i}$, modified by $\overline{\text{Cl}(cy_j - a_j)}$, is processed by 2-tuples which are added by line 16 of 4.4.6. (See figure 4.5.5c).

On the other hand, suppose $b_{j-1}$ and $b_j$ do not have the same superscript (i.e., there is a path $\overline{\text{id}(a_j) \xrightarrow{DT} T(b_j)}$ as in Figure 4.5.5b). We shall show that 2-tuples added to BPL first process the flow along reverse path $t_j$, from $\text{CYCLE}(b_{j-1})$ to $\text{CYCLE}(b_j)$, then process the flow around $\text{CYCLE}(b_j)$. Let $\overline{e_1, e_2, \ldots, e_\ell}$ be the edges, taken in order, on $\overline{y_j \xrightarrow{\text{DAG}} a_j}$ such that $x(e_i) \neq \text{id}(T(e_i))$, $1 \leq i \leq \ell$. (See Figure 4.5.6a.) When $x(e_i)$ is traversed, the 2-tuples added to BPL, guarantee the data flow from $T(e_i)$ to each node on $\overline{0(e_i) \xrightarrow{TT} \text{id}(T(e_i))}$ is processed (refer to lines 3, 8 and 16 of 4.4.6). The DAG path $t_j$ can be written as $\overline{e_1 d_1 e_2 d_2 \ldots e_\ell d_\ell}$ where $d_i$ is a reverse path in $\text{DAG}$ of $G$ from $0(e_i)$ to $T(e_{i+1})$. If $\text{id}(T(e_i))$ dominates $\text{id}(T(e_{i+1}))$ and $d_i$ is as indicated in Figure 4.4.6b then 2-tuples added to BPL during compression of $\overline{0(e_i) \xrightarrow{TT} \text{id}(t(e_i))}$ process flow from $T(e_i)$ to $T(e_{i+1})$. Alternatively, $\text{id}(T(e_{i+1}))$ dominates $\text{id}(T(e_i))$ and $d_i$ is as indicated in Figure 4.5.6c. The values $\text{Cl}(T(e_i))$, $\text{id}(T(e_i))$ and $\text{Cl}(T(e_{i+1}), \text{id}(T(e_{i+1}))$ are inputs to algorithm 4.4.6. Therefore, when $e_h$ is processed, $1 \leq h < i$, data is propagated
Figure 4.5.6. Representation of DAG paths between cycles for BFP.
from id(T(e_i)) to T(e_{i+1}). Therefore, assuming that the flow to any node T(e_i) is processed, we have shown that the flow to node T(e_{i+1}) is processed. That is, the flow from any node on \( y_j \xrightarrow{*} 0(b_{j-1}) \) to \( y_j \) is processed by traversing nodes id(T(e_1)), id(T(e_2)), \ldots, id(T(e_k)), \ k \leq \lambda. \) Here \( e_k \) is the edge with the largest index in the sequence \( e_1, e_2, \ldots, e_\lambda \) such that there is no path \( T(b_j) \xrightarrow{+} DT \) id(T(e_k)). Lines 3 and 4 of 4.4.6 ensure that \( e_k \) is processed before CYCLE(b_j). That is, the flow around CYCLE(b_j) is processed after path \( t_j \) is processed.

Therefore, once again we have shown that assuming the flow to nodes in CYCLE(b_{j-1}) is correctly processed, we can argue that the flow to node \( y_j \), in CYCLE(b_j), along path \( t_j \) is correctly processed. Hence the flow around CYCLE(b_j) can be correctly processed by 2-listing the compressed paths on CYCLE(b_j).

By requiring the initial and final subsequences of BPL to be FPL_TREE we ensure that the segments \( t_1, t_k \) in the representation of any path can also be represented in BPL. Therefore, we have inductively proved that if the cycles and the \( t_i \) paths are processed on-line as shown, every reverse path can be represented in BPL and BPL(G) = BPL. Also, Cl(BPL) is determined since the cost of each compressed path added to TT' is determined by line 13 of 4.4.6 and only 2-tuples denoting edges of TT' are placed in BPL.

4.6 Conclusions.

In Section 4.4 we presented algorithms which used only path
compression techniques. The complexity of MAIN FFP (4.4.1) for RFG G, \( G = (N,E,n_o) \), is as follows. Let \( n = |N| \) and \( e = |E| \).

Step 1 takes \( O(e + n) \) steps.

Step 2 takes \( O(e \cdot \max(1, \log_2(n^2/e)/\log_2(2e/n))) \)

by a theorem proved in \([T5]\). This is a consequence of the fact \( O(e) \) compressions are required.

Step 3 takes \( O(e + n) \) steps.

The bound for determining the input structure DT (see Chapter 3) is \( O(n + e\alpha(e,n)) \). Therefore the path listing algorithm takes \( O(e \cdot \max(1, \log_2(n^2/e)/\log_2(2e/n))) \) steps and the length of FPL is of the same order. If balancing techniques are used as shown in \([T5]\, [AHU]\), this bound can be improved to \( O(n + e\alpha(e,n)) \). Since the complexity of the FSOLVE algorithm (4.3.2) depends on the length of FPL, the number of set operations is also of \( O(n + e\alpha(e,n)) \).

The complexity of MAIN BFP(4.4.5) for any RFG G is as follows.

Step 1 takes \( O(e + n) \) steps

Step 2 takes \( O(e + n) \) steps

Step 4 takes \( O(e + n) \) steps

Step 5 takes \( O(e \cdot n) \) steps.

Therefore, the resulting complexity is \( O(e \cdot n) \) steps. However, suppose DT is represented using a balanced tree together with additional balanced trees representing the compressed 'bad' paths of DT. Then, any compression takes at most \( \log n \) steps--\( \log n \) being the maximum height of the trees representing DT. Since the number of compressions is bounded by \( e \), the total number of
steps is of $O(e \cdot \log n)$. That is, the length of BPL is of $O(e \cdot \log n)$. Therefore, BSOLVE (4.3.3) carries out $O(e \log n)$ set operations.

The bound for MAINBFP can be further improved. In deriving the bound given above we did not take into consideration the fact that partial compression is carried out in MAINBFP.
5. Conclusions and Recommendations

The results given in the first three chapters primarily attempt to establish attribute grammars as a desirable tool for specifying a variety of translation processes. The theoretical groundwork laid in these chapters should be of help in any implementation of an optimizing compiler.

In Chapter 1, details of a conceptually simple method for evaluating attribute grammars are given. The evaluator described here works on a larger class of attribute grammars—the non-circular grammars—than the evaluator for the absolutely non-circular grammars in [KW]. The evaluator described in [KW], however, is more efficient during runtime. Further understanding of the advantages of each of the two methods can only come from actual implementations.

Chapter 2 shows how attribute grammars can be used to specify context sensitive semantics. As an example, the problem of global optimization of code generated for structural programs is studied. We find that attribute grammars may be used to specify efficient optimization of structured programs with widely varying control structures. In fact, the optimization process so specified has a linear bound, in terms of the length of the input program, on its time and space requirements. The control structure dealt with in this chapter are single entry, single exit regions. Similar results can be obtained for a less restrictive class of control structures (e.g., control structures with k entrances and k exits). The study of control structures defined on locally occurring parse tree symbols provides the handle for a definition of 'structured' programs.
This concept merits further study.

Attribute grammars given in Chapter 3 specify an algorithm for determining the dominator tree underlying a flow graph \( G \). These grammars illustrate the adaptibility of attribute grammars in specifying a variety of translation processes.

In Chapter 4 backward and forward flow problems are shown to have distinct properties which are reflected in the distinct bounds for their respective algorithms based on similar techniques. This poses the interesting question of whether the lower bounds for the backward and forward flow problems are distinct. There are also practical implications of the algorithms given in Chapter 4. Firstly, the algorithms can be specified using attribute grammars. Secondly, the algorithm for determining the path listing for the backward flow problem is conceptually simpler than the one by [AU] and as efficient. Therefore, it should be easier to implement. Finally, the class of problems handled by our algorithms can be enlarged. To do this a wider range of interpretations can be allowed by making the definitions of BFP and FFP more general.
BIBLIOGRAPHY


[AsM] Ashcroft, E. and Mana, Z., "The Translation of 'GOTO' Programs to 'WHILE' Programs," IFIP 71, Vol. 1


