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
ZUCCONI, Linda, 1951-
A GRAPH GRAMMATICAL APPROACH TO COMPILER
OPTIMIZATION.

Rice University, Ph.D., 1977
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
RICE UNIVERSITY

A GRAPH GRAMMATICAL APPROACH

TO

COMPILER OPTIMIZATION

by

Linda Zucconi

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

Doctor of Philosophy

Thesis Director's signature:

Kenneth W. Kennedy Jr.

Houston, Texas

April 1977
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter Zero</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Chapter One</td>
<td>CONTROL FLOW GRAPHS AND THE SEMI-STRUCTURED FLOW GRAPH GRAMMAR</td>
<td>5</td>
</tr>
<tr>
<td>Chapter Two</td>
<td>A FAST PARSE FOR $\mathcal{F}_{SSFG}$</td>
<td>25</td>
</tr>
<tr>
<td>Chapter Three</td>
<td>APPLICATION TO GLOBAL DATA FLOW ANALYSIS</td>
<td>52</td>
</tr>
<tr>
<td>Chapter Four</td>
<td>THE EMPIRICAL STUDY</td>
<td>70</td>
</tr>
<tr>
<td>Chapter Five</td>
<td>GENERALIZING SSFG AND NODE-SPLITTING</td>
<td>85</td>
</tr>
<tr>
<td>Chapter Six</td>
<td>SUMMARY, CONCLUSION AND POINTS FOR FURTHER STUDY</td>
<td>92</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>94</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

| Figure 1.1 | The Böhm-Jacopini Graph Grammar | 7 |
| Figure 1.2 | Semi-structured flow graph grammar \((\mathcal{C}_{SSFG})\) | 8 |
| Figure 1.3 | SSFG reduction of Zahn's control structures | 11 |
| Figure 1.4 | SSFG reduction of Dahl's and Friedman and Shapiro's loop structures | 12 |
| Figure 1.5 | SSFG-reduction of Friedman and Shapiro's Bohm and Jacopini irreducible flow graph | 13 |
| Figure 1.6 | Derivation of Martin's set of "natural" control structures | 14 |
| Figure 1.7 | Derivation of the counterexample of Washcroft and Manna | 15 |
| Figure 1.8 | SSFG-reduction of the Spiral Graph | 16 |
| Figure 1.9 | SSFG reduction of the double chain graph | 17 |
| Figure 1.10 | SSFG reduction of the climbing graph | 17 |
| Figure 1.11 | SSFG-reduction of the dual exit climbing graph | 18 |
| Figure 1.12 | SSFG-reduction of the Hecht-Ullman binary graph with exit from each loop | 18 |
| Figure 1.13 | BJ _n_ loop structure | 21 |
| Figure 1.14 | Kosaraju's hierarchy of control structures and SSFG | 23 |
| Figure 1.15 | SSFG-reduction of a DREC$_3$ flow graph | 24 |
| Figure 2.1 | The SSFG transformation forming \(=\) | 27 |
| Figure 2.2 | The height 1 graphs in \(F^+_SSFG\) | 30 |
Figure 2.3  \( P_3 \) proof diagram  
Figure 2.4  Diagrams for self-interacting transformations  
Figure 2.5a,b  Diagrams for cc with dc and cd  
Figure 2.6a,b  Diagrams for d with dd and cd  
Figure 2.7a,b,c  Diagrams for loop with dc, dd and d-loops  
Figure 2.8a,b  Diagrams for dc with dd and cd  
Figure 2.8c  Diagrams for dc with d-loop  
Figure 2.9  Diagram for cd and dd  
Figure 2.10  Diagram for d-loop and dd  
Figure 2.11  A reducible region  
Figure 3.1a  A flow chart with inside ( ) and thru [ ] sets given for live analysis  
Figure 3.1b  Reduction cd has been applied to nodes h and a  
Figure 3.1c  Reduction d-loop has been applied to nodes h and b  
Figure 3.1d  Reduction dc has been applied to nodes h and c  
Figure 3.13  Rules d and cc are applied with the result-inside set for the entire reduced graph containing the variables x and y  
Figure 3.1f  The original flow graph with the computed live sets given inside each basic block
Figure 3.2 Attributed $G_{SSFG}$ for "Profitability" 65
Figure 3.2 Continued 66
Figure 3.3 A profitability reduction sequence 67
Figure 3.4 Profitability production sequence 68
Figure 4.1 $SSFG_x$-extensions 76
Figure 4.2 Additional rules for $SSFG_3$ 77
Figure 4.2 Continued 78
Figure 4.3 Proper use of a Case Statement 82
Figure 4.4a,b Improper use of the computed GOTO 83
Figure 5.1 $\Sigma_{SSFG}$: Non SSFG-reducible, non-node splittable flow graph 87
Figure 5.2 Hecht-Ullman flow graph transformations 88
Figure 5.3 Reduction of $G_{SSFG_\infty}$ by Hecht-Ullman transformations 89
Figure 5.4 Hierarchy of Hecht-Ullman and $SSFG_\infty$ reducible flow graphs 90
CHAPTER ZERO

INTRODUCTION

Optimization of compiled code means the reduction of some cost function (such as the number of instructions, the execution time or the amount of required computed memory) to a minimum while maintaining program correctness. In this thesis, we shall relate the global optimization of computer programs to the theory of graphs and to the parsing of two-dimensional languages.

For the purpose of compiler optimization, the analysis of a computer program begins with the program expressed in some intermediate language which is scanned and divided into straight-line sequences of instructions called basic blocks. After the execution of the last instruction in a basic block, control may transfer to any one of a number of blocks called successors of the block just executed. If we represent basic blocks by nodes and possible transfers of control by edges, we derive the control flow graph model of a program which is useful in the solution of many optimization problems.

Published studies indicate that typical control flow graphs tend to fall into a restricted subclass of general graphs, i.e., the vast majority of program graphs have no multiple-entry loops, [AC, HU2, HU3, Knl, KZ]. Recent work on "structured programming" has suggested that "good" programs fall into an even more restricted subclass. In fact, purists recommend that all programs be synthesized from the three basic control structures first studied by Böhm and Jacopini [BJ]: if-then-else, sequential and while-do statements. Several authors, including Wirth [Wi], have suggested adding the repeat-until statement to this set. These control structures
are sufficient to program any computation if extra boolean variables may be introduced [BJ].

Flow graphs using only these four control structures are particularly easy to analyze since they admit only single-entry, single-exit regions [GW, HU2, Ke3] However, there are many situations in which regions with more than one exit are convenient for programming [Cl, FS, Ka, Kn1, KP2, Ma, Sh, We, Wi, WR, WRH, Z] The major objection to the four Böhm and Jacopini control structures is that they do not permit multiple exits from loops and thus force programmers to use somewhat artificial boolean variables to distinguish different loop exits.

In response to this complaint, we have investigated control structures which allow two exits from regions and sensibly constructed multiple exit loops. These investigations show that two-exit regions are commonly encountered in programs but they do not unnecessarily complicate flow analysis.

Our presentation is organized as follows: In the first chapter, we formally introduce control flow graphs and present Böhm and Jacopini's flow graph grammar and our extended flow graph grammar $G_{SSFG}$. The second chapter demonstrates that we can determine whether a flow graph $\Gamma$ is a member of the family $\mathcal{F}_{SSFG}$ of flow graphs generated by $G_{SSFG}$ in time linear in the size of $\Gamma$. In the third chapter, we show how the SSFG parse from chapter two can be used to perform global data flow analysis. Then we present global data flow analysis as an attributed version of $G_{SSFG}$. In chapter four the question of the naturalness of the semi-structured flow graph grammar for describing real programs is investigated through an empirical study of some of five-hundred FORTRAN routines. We also present some extensions to $G_{SSFG}$ suggested by that study. The fifth chapter generalizes SSFG to SSFG$_0$, a single-entry, arbitrary exit flow graph grammar and dis-
cusses why it appears that node-splitting (code-copying) cannot fix all SSFG-irreducible flow graphs. The final chapter summarizes the results, makes some conclusions and proposes areas for further research.

ACKNOWLEDGMENTS

CASE persons IN

† Rice people who donated programs for analysis †

Steve_Smith: Prof_F_Quiocho: George_Phillips:
Prof_E_Hayes: Prof_F_T_Wall:
Richard_Liedtke: John_Chen:
Prof_JC-deBremaecker: Prof_N_F_Lane:
Tom_Winter: Allen_Haggard: Prof_M_F_Wheeler:
Bruce_Darlow: Naresh_Garg: Prof_E_C_Holt:

Issue (Thank-you);
† For help on parsing algorithm †

Rod_Farrow: Issue (Special_thank-you);
† Fellow graduate students who banged heads with mine †

Jay_Ramanathan: Tom_Shields

Issue (Thank-you);
† My thesis committee †

Meera_Blattner: Rui-deFigueiredo:

Issue (Thank-you);

Ken_Kennedy: Issue (Very_special-thank-you);
† My friend †

Mike_Fazio: Issue (Thank-you);
The person responsible for my coming to Rice

Prof_E_Feustel: Issue (Thank-you);

A very kind lady

Mrs_Velma_Goodwin: Issue (Thank-you);

My typist

Jeanne_Fulton: Issue (Thank-you);

NSF grants DCR72-03609-A01 and DCR73-03365-A01

NSF: Issue (Thank-you);

My parents and best friends

Bruno_Zucconi: Candida_Zucconi:

Issue (Most_special_thank-you);

My research notes warmers

Dorothy: Issue (Fat_mice);

Hermie: Issue (A_warm_place_in_Cat_heaven).

ESAC
CHAPTER ONE

CONTROL FLOW GRAPHS AND THE SEMI-STRUCTURED FLOW GRAPH GRAMMAR

A control flow graph of a program is a triple \( \Gamma = (N, E, x_o) \) where

1. \( N \) is a finite set of basic blocks called nodes,
2. \( E \) is a finite set of directed edges \( E \subseteq N \times N \) representing possible block to block transfers, and
3. \( x_o \) is the program entry node which has no other node in \( N \) as a predecessor.

The successor set \( S(x) \) of a given node \( x \) is defined as
\[
S(x) = \{ y \in N \mid (x, y) \in E \}
\]
and the predecessor set \( P(x) \) of a node \( x \) is defined as
\[
P(x) = \{ y \in N \mid (y, x) \in E \}
\]

Published studies have shown that the form of typical control flow graphs indicate that such graphs tend to fall into a restricted subclass of general directed graphs; the vast majority of program graphs have no multiple-entry loops [AC, Hu2, Hu3, Kn1, KZ]. Recent work on structured programming has suggested that good programs fall into an even more restricted subclass. In fact, purists recommend that all programs be synthesized from three basic control structures: if-then-else statements, sequential statements and while-do loops [Di, Wi].

In 1966, Böhm and Jacopini [BJ] showed that these three control structures are sufficient to program any computation if extra boolean variables may be introduced. Several authors, such as Wirth [Wi], added the repeat-until loop, proposing a set of four standard control structures consisting of sequential statements, if-then-else statement, while-do loop and repeat-until loop for structured programming. The flow graphs of "structured"
programs can be described by the BJ-grammar, $G_{BJ}$, depicted in Figure 1.1.

Graphs using the BJ-grammar are particularly easy to analyze since that grammar admits only single-entry, single-exit regions [GW, HU2, Ke]. However, there are many situations in which regions with more than one exit are desired for programming. This has been a subject for extensive discussion in the literature [Cl, FS, Ka, Kn1, KP2, Ma, Sh, We, Wi, WR, WRH, Za]. The major objection to the BJ control structures is that they do not permit multiple exits from loops, thus forcing programmers to use somewhat artificial boolean variables to distinguish different loop exits.

In response to this complaint, we present the semi-structured flow graph grammar $G_{SSFG}$, given in Figure 1.2, which allows two exits from regions and multiple exit loops where the loops are constructed sensibly. Note the addition of a single-entry, two-exit region to our set of non-terminal nodes. The decision-decision, decision-loop and loop rules allow the graphs produced by $G_{SSFG}$ to contain regions with sensibly constructed multiple exit loops. It is the combination of the addition of the two-exit region and the extra rules that give the SSFG set of control structures much power over the Böhm and Jacopini set, as we shall demonstrate later on.

The family of semi-structured flow graphs $F_{SSFG}$ is the set of all control flow graphs by $G_{SSFG}$. A graph $\Gamma$ is said to be SSFG-reducible if $\Gamma \in F_{SSFG}$. It can be said without too much difficulty that $F_{SSFG}$ is a proper subset of the "reducible" flow graphs [HU2, HU3] and properly contains the family of graphs generated by the Böhm and Jacopini grammar.

To show that the semi-structured flow graphs are useful, we must show that $F_{SSFG}$ contains important flow structures which cannot be generated by the Böhm and Jacopini control structures.

Knuth [Kn1] has suggested that Zahn's [Z] multiple exit loop structures
Figure 1.1

The Böhm-Jacopini Grammar
Figure 1.2

Semi-structured flow graph grammar (G_{SSFG})
represented below are a meaningful enrichment to the Böhm and Jacopini set.

1. \[\text{loop until } \langle\text{event}\rangle_1 \text{ or } \ldots \text{ or } \langle\text{event}\rangle_n ;\]
   \[\langle\text{statement list}\rangle_0 ;\]
   \[\text{repeat} ;\]
   \[\text{then } \langle\text{event}\rangle_1 = \langle\text{statement list}\rangle_1 ;\]
   \[\ldots \]
   \[\langle\text{event}\rangle_n = \langle\text{statement list}\rangle_n ;\]
   \[\text{fi;}\]

2. \[\text{begin until } \langle\text{event}\rangle_1 \text{ or } \ldots \text{ or } \langle\text{event}\rangle_n ;\]
   \[\langle\text{statement list}\rangle_0 ;\]
   \[\text{end} ;\]
   \[\text{then } \langle\text{event}\rangle_1 = \langle\text{statement list}\rangle_1 ;\]
   \[\ldots \]
   \[\langle\text{event}\rangle_n = \langle\text{statement list}\rangle_n ;\]
   \[\text{fi;}\]

G\textsubscript{SSFG} multiple-exit loops similar to Zahn’s can be generated by G\textsubscript{SSFG} through repeated use of the decision-decision rule, as depicted in Figure 1.3.

Dahl [Kn1] proposed the following mid-loop exit control structure:

\[\text{loop} ; \langle\text{statement list}\rangle_0 ; \text{while } \langle\text{boolean}\rangle ;\]
\[\langle\text{statement list}\rangle_1 ; \text{repeat} ;\]

depicted in Figure 1.4 along with Friedman and Shapiro’s [FS] two test while-loop:
while <boolean>0 repeat <statement list> until <boolean>1; Friedman and Shapiro also presented a flow graph that cannot be derived using only sequential, if-then-else and repeat-until type statements. Its reduction via the semi-structured flow graph grammar is given in Figure 1.5.

The set of "natural" basic control structures defined by Martin [Ma] is a subset of $\mathcal{F}_{SSFG}$ as shown in Figure 1.6.

Perhaps the most famous examples of a flow graph not definable within the Böhm and Jacopini set is the graph with which Ashcroft and Manna [Ma] proved that the Böhm and Jacopini control structures are not sufficient if extra boolean variables are not allowed. It is similar to the Friedman and Shapiro control structure; a derivation is depicted in Figure 1.7.

The semi-structured flow graph grammar can also generate a number of subclasses of flow graph forms including spiral graphs, double chain graphs, climbing graphs, dual-exit climbing graphs and Hecht-Ullman binary graphs when each innermost loop exits to the same point. Examples of these graphs and their SSFG-reductions are given in Figure 1.8 through 1.12.

Several researchers including Kosaraju, Ledgard and Marcotty have analyzed the relative structural complexity of programs arising from the use of various control structures [Ko, LeM, AM] and have developed a genealogy of control structures. It is interesting to see where $\mathcal{F}_{SSFG}$ fits among the classes of structure charts.

Given two classes of programs $C_1$ and $C_2$, we say that $C_1$ is convertible to $C_2$ if and only if

P1: for every input, $C_2$ computes the same function as $C_1$ and
P2: the primitive actions and predicates of $C_2$ are precisely those of $C_1$ [LeM].
SSFG-reduction of Zahn's control structures
Figure 1.4

SSFG reduction of Dahl's and Friedman and Shapiro's loop structures
Figure 1.5

SSFG-reduction of Friedman and Shapiro's Bohm and Jacopini irreducible flow graph
Figure 1.6

Derivation of Martin's set of "natural" control structures
Figure 1.7

Derivation of the counterexample of Ashcroft and Manna
Figure 1.8
SSFG-reduction of the Spiral Graph
SSFG reduction of the double chain graph

SSFG-reduction of the climbing graph
Figure 1.11
SSFG-reduction of the dual exit climbing graph.

Figure 1.12
SSFG-reduction of the Hecht-Ullman binary graph with exit from each loop
A class of structures $C_1$ is **reducible** to a class $C_2$, denoted $C_1 \leq C_2$, if every structure in $C_1$ can be semantically converted to a structure in $C_2$, but not necessarily vice versa. A class of structure $C_1$ is **strictly reducible** to a class $C_2$ denoted $C_1 < C_2$ if every structure in $C_1$ can be semantically converted to a structure in $C_2$, but not vice versa [LeM].

Ledgard, Marcotty and Kosaraju break control structures into the following classes:

(a) **D-structures**, D for Dijkstra. A D-structure is any program constructed only from the following one-in, one-out primitive structure:

(i) basic actions such as assignment statements, procedure calls, input/output statements

(ii) compositions "$s_1; s_2$" of two D-structures

(iii) conditional statements, i.e. $\text{if } p \text{ then } s_1 \text{ else } s_2$

(iv) while loops such as $\text{while } p \text{ do } is$

(b) **$D^1$-structures**. These are D-structures with single branching if-statements, n-way case statements and repeat-until loops.

(c) **BJ$_n$-structures**. These are the Bohm and Jacopini structures mentioned above with the one-in, one-out control structures given below:

\[
\text{loop if } p_1 \text{ then exit ;}
\]

\[
s_1 ;
\]

\[
\text{if } p_2 \text{ then exit ;}
\]

\[
s_2 ;
\]

\[
\ldots
\]

\[
\ldots
\]
if \( p_n \) then exit ;

\( \frac{s_n}{\text{end}} \)

It has the flow graph structure given in Figure 1.13.

(d) \( \text{RE}_n, \text{REC}_n, \text{DRE}_n, \text{DREC}_n \) structures. An \( \text{RE}_n \) structure is composed of basic actions, compositions; \text{if-then-else} structures, exit commands of the form \( \text{exit}(i) \) where \( 1 \leq i \leq n \) (causes exit out of \( i \) enclosing loops) and \text{repeat-end} constructs as follows:

\[
\text{repeat } s_1 ;
\]
\[
\begin{align*}
\text{s}_2 ; \quad & \\
. \quad & \\
. \quad & \\
. \quad & \\
\text{s}_n \end{align*}
\]

\( \text{end} \)

A \( \text{REC}_n \) structure is a \( \text{RE}_n \) structure with the addition of a \text{cycle}(i) instruction for \( 1 \leq i \leq n \). A \text{cycle}(i) command is similar to an \text{exit}(i) command except that the \( i \)th enclosing loop is re-executed.

A \( \text{DRE}_n \) structure is an \( \text{RE}_n \)-structure with \text{while-do} loops added. A \( \text{DREC}_n \) structure is a \( \text{DRE}_n \) structure with \text{cycle}(i) for \( 1 \leq i \leq n \).

(e) \( \text{GP}_n \) and \( \text{L} \) structures. A \( \text{GP}_n \) structure is defined as any structure such that all one-in, one-out substructures have at most \( n \) different predicates. An \( \text{L} \) structure is any structure with no restrictions on the number or configuration of predicates, actions or transfers of control. An \( \text{L} \)-structure is the parallel of programs with free use of labels and \text{goto} statements.
Figure 1.13

BJ_n loop structure
Kosaraju's hierarchy of control structures is depicted in Figure 1.14. An upward line from class \( C_1 \) to class \( C_2 \) means \( C_1 < C_2 \), a dashed line upwards that \( BJ_\infty < SSFG \) and \( DREC_1 \leq SSFG \), however \( SSFG < DREC_\infty \equiv GP_\infty \equiv L \).

Yet there are flow graphs which are \( DREC_n \) for \( n \geq 2 \) that are also SSFG-reducible; see the flow graph in Figure 1.15. It is in the \( DREC_3 \) class, yet it is SSFG-reducible.

Hence, we see that \( \mathcal{F}_{SSFG} \) family appears to take a slice out of the \( DREC_n \) and \( GP_n \) structures yet it does not recognize all of them. That implies that SSFG structures allow exits and cycles out of arbitrary levels of nested loops only if the exits are shared and constructed conservatively.

SSFG is a powerful tool for describing the control flow of programs. However, we need to know, for SSFG to be useful, if given some flow graph \( \Gamma \), we can always determine if \( \Gamma \) is in \( \mathcal{F}_{SSFG} \).
Figure 1.14

Kosaraju's hierarchy of control structures and SSFG
CHAPTER TWO

A FAST PARSE FOR $\mathcal{F}_{SSFG}$

If we are to use $G_{SSFG}$ to analyze program structure, we must be able to parse an arbitrary program flow graph in $\mathcal{F}_{SSFG}$. The structural information provided by the parse can then be used in data flow analysis on the program. For languages which do not admit a GOTO-statement, the conventional syntactic parse of the program should provide much of the $G_{SSFG}$ parse since standard control structures in such languages will have known constructions. However, we shall seek a more general technique, applicable to well-written programs with GOTO-statements. An important step is to prove that reductions according to $G_{SSFG}$ taken as a "replacement system" have the Finite-Church-Rosser (FCR) property, i.e., if reductions are successively applied to a graph $\Gamma$ in any order until no more reductions are possible, the limit graph is unique. To put it another way, if there exists any sequence of $G_{SSFG}$ reductions which collapse $\Gamma$ to a single computation node, then the FCR property guarantees that we can collapse $\Gamma$ to a single node by applying $G_{SSFG}$ reductions in any order. In this chapter, we establish the FCR property for the family $\mathcal{F}_{SSFG}$.

A replacement system $S = (B, \rightarrow, \equiv)$ is a triple consisting of a set $B$, a binary relation $\rightarrow$ on $B$, and an equivalence relation $\equiv$ on $B$. We can define a replacement system on the family $\mathcal{F}_{SSFG}$ augmented by the graph "sentential forms," i.e., the family of graphs which may contain non-terminal nodes as well as terminal nodes generated by $G_{SSFG}$. We denote the family of SSFG sentential forms as $\mathcal{F}_{SSFG}^+$. We now define the replacement system

$$S = (\mathcal{F}_{SSFG}^+, \rightarrow, \equiv)$$

where $\equiv$ is graph isomorphism and $\rightarrow$ is the union of the nine transformations
which result from inverting the rules of $G_{SSFG}$ (Figure 2.1). Note that the transformations loop, dc, dd and d-loop also reduce mirror images of the regions shown on the left-hand sides of Figure 2.1. In the subsequent discussion we shall refer to specific transformations by the labels shown in Figure 2.1. For $\Gamma$ and $\Gamma' \in S^+_{SSFG}$ we say that $\Gamma \Rightarrow x \Gamma'$ if $\Gamma'$ can be obtained from $\Gamma$ by a single application of the transformation labelled $x$. $\Rightarrow^*$ denotes the reflective transitive closure of the composite relation $\Rightarrow$.

An instance of a reduction according to $S = (S^+_{SSFG}, \Rightarrow, \equiv)$ is a pair $I = (x, R)$ where $x$ is the rule name and $R$ is the region in $\Gamma$ which is isomorphic to the left-hand side (LHS) of rule $x$. If $I = (x, R)$ is an instance of $\Rightarrow$, the result of applying $I$ to $\Gamma$ is a graph $\Gamma'$ which is isomorphic to $\Gamma$ except for region $R$ which is replaced by the single non-terminal node on the right-hand side (RHS) of rule $x$.

A rule $x$ of $S$ is said to interact with rule $y$ of $S$ if there is a graph $\Gamma \in S^+_{SSFG}$ for which there is an instance $I_x = (x, R_x)$ of rule $x$ and an instance $I_y = (y, R_y)$ of rule $y$ and $R_x \cap R_y \neq \emptyset$.

In other words, two transformations interact if they can have distinct instances with a node in common. If this is not possible, the two transformations are disjoint.

A replacement system $(B, \rightarrow, \equiv)$ is said to be finite if for every $p \in B$ there exists a $k \geq 0$ such that if $q \in B$ can be obtained from $p$ by $i$ applications of $\rightarrow$, then $i \leq k$. The height of an element $p \in B$ is the maximum number of times that $\rightarrow$ can be applied to $p$.

**Lemma 2.1** $S = (S^+_{SSFG}, \Rightarrow)$ is finite.

**Proof:** Given any graph $\Gamma_{SSFG}^+$ with $n$ nodes and $e$ edges, we can apply rules c-blk and d-blk a maximum of $n$ times. Each of the other transformations in $\Rightarrow$ eliminates at least one edge, so these can be applied an aggre-
Figure 2.1

The SSFG transformations forming
gate of e times. Therefore, the desired k may be taken as (n + e) and the results follows.

A replacement system \((B, \rightarrow, \equiv)\) is **Finite Church-Rosser** (FCR) if it is finite and \(\forall \ p, q, r, s, B \) such that \(p \equiv q, p \rightarrow r\) and \(q \rightarrow s\) there exist \(\hat{r}, \hat{s} \in B\) such that \(r \rightarrow \hat{r}, s \rightarrow \hat{s}\) and \(\hat{r} \equiv \hat{s} [Ro, ASU]\).

Two simplified tests for the FCR property have been devised by Aho, Sethi and Ullman [ASU] and by Sethi [Se].

**Theorem 2.2** (Aho, Sethi, Ullman): The replacement system \((B, \rightarrow, \equiv)\) is FCR if \(\forall \ p, q, r, s \in B\) such that \(p \equiv q, p \rightarrow r\) and either \(q \rightarrow s\) or \(q = s\) there exist \(\hat{r} \rightarrow \hat{r}, s \rightarrow \hat{s}\) and \(\hat{r} \equiv \hat{s}\).

Sethi's necessary and sufficient condition can be expressed in terms of two properties [Se]:

**P1:** A replacement system \((B, \rightarrow, \equiv)\) has property \(P_3\) if \(u, v, w \in B, u \equiv w, u \rightarrow v\) imply that \(\exists y, z \in B\) such that \(w \rightarrow^* y, v \rightarrow^* z\) and \(y \equiv z\).

**P3:** A replacement system \((B, \rightarrow, \equiv)\) has property \(P_3\) if \(u, w, x \in B, u \rightarrow w\) and \(u \rightarrow x\) imply that \(\exists y, z \in B\) such that \(w \rightarrow^* y, x \rightarrow^* z\) and \(y \equiv z\).

**Theorem 2.3** (Sethi):

A replacement system \((B, \rightarrow, \equiv)\) is FCR iff it is finite and has properties P1 and P3.

**Theorem 2.4** (Main Result):

\[ S = (\mathcal{G}^{*}_{SSFG}, \rightarrow, \equiv) \] is FCR.

**Proof:** Lemma 2.1 establishes finiteness. We shall use Sethi's test.

**P1:** Let \(I_1\) and \(I_2\) be isomorphic graphs in \(\mathcal{G}^{*}_{SSFG}\). (Note that isomorphism preserves node types as well as structure). If \(I_1\) and \(I_2\) have height 0 then property \(P_1\) is vacuously true to assume they each have height > 0. Suppose \(I_1 \rightarrow I_3 \in \mathcal{G}^{*}_{SSFG}\) where \(I_1 = (x, R_1)\); i.e, the result of
applying $I_2$ to $\Gamma_1$ is $\Gamma_3$. Let $R_2$ be the isomorphic image (under $\equiv$) of $R_1$ in $\Gamma_2$. Then the result of applying $I_2 = (x, R_2)$ to $\Gamma_2$ is a graph $\Gamma_4$ which must be isomorphic to $\Gamma_3$. This establishes property P1.

P3: We must show that for $\Gamma, \Pi_1, \Pi_2 \in S_{SSFG}$, $\Gamma \rightarrow \Pi_1$ and $\Gamma \rightarrow \Pi_2$ imply that there exist $\Pi_3, \Pi_4 \in S_{SSFG}^+$ such that $\Pi_1 \rightarrow \Pi_3$, $\Pi_2 \rightarrow \Pi_4$, and $\Pi_3 \equiv \Pi_4$ and $\Pi_3 \equiv \Pi_4$. There are three cases.

CASE I. height ($\Gamma$) = 1.

In this case $\Gamma$ cannot be further reduced so P1 holds vacuously.

CASE II. height ($\Gamma$) = 1.

$\Gamma \in S_{SSFG}^+$ must be isomorphic to one of the graphs that can be generated from the start symbol of $G_{SSFG}$ by the application of a single production. These graphs are shown in Figure 2.2. If $\Gamma \in S_{SSFG}^+$ and $\Gamma \rightarrow^x \Gamma_1$, $\Gamma \rightarrow^y \Gamma_2$, then $x$ must equal $y$ and $\Gamma_1 \equiv \Gamma_2 \equiv \Phi$ is the trivial graph consisting of a single computation node and no edges (the start symbol of $G_{SSFG}$).

CASE III. height ($\Gamma$) > 1.

We will establish P3 by exhaustively "completing the proof diagram" in Figure 2.3. We will show that for any choice of $x$ and $y$ we can find $i$ and $j$ such that when $i$ is applied to $\Gamma_1$ and $j$ is applied to $\Gamma_2$, the result is the same graph $\Gamma_3$ (or at least isomorphic to $\Gamma_3$).

In practice we must deal with instances $I_x = (x, R_x)$ and $I_y = (y, R_y)$ of transformations $x$ and $y$. There are several subcases.

Case 1: $R_x \cap R_y = \emptyset$.

Consider the graph homomorphisms $\varphi_x$ and $\varphi_y$ induced by applying $I_x$ and $I_y$ to $\Gamma$ respectively. Since $R_x$ and $R_y$ are disjoint, $x$ takes $R_y$ into a region $R_y \mid \Gamma_1$ in $\Gamma_1$ such that $R_y \mid \Gamma_1$ is isomorphic to $R_x$. Clearly, if we apply transformation $y$ to
The height 1 graphs in $\mathcal{G}_{SSFG}$

Figure 2.3

$P_3$ proof diagram
and transformations x to \( R_\Gamma \) the results will be the same graph \( \Gamma \) (which can be derived from \( \Gamma \) by collapsing \( R_x \) by x and \( R_y \) by y). This completes the diagram in Figure 2.3.

Note that if x and y are "disjoint transformations" we have by definition \( R_x \cap R_y = \emptyset \) for any legal choices, \( R_x \) and \( R_y \). Thus we may restrict our consideration to transformations x and y which may interact.

Case 2: x and y may interact and \( R_x \cap R_y \neq \emptyset \).

Case 2a: \( x = y \).

Rules \texttt{cblk}, \texttt{dblk}, \texttt{d}, \texttt{loop} and \texttt{cd} cannot interact with themselves so we need only complete the diagrams for \texttt{cc}, \texttt{dc}, \texttt{dd} and \texttt{d-loop}. This is done in Figure 2.4. In this figure and all subsequent ones in the proof we show only the pertinent nodes in the graphs \( \Gamma, \Gamma_1, \Gamma_2, \Gamma_3 \). The context is not shown because it is left undisturbed by the subject transformations. In the diagrams, region \( R_x \) is defined by a broken line and \( R_y \) is defined by a dotted line.

Case 2b: \( x \neq y \).

(1) The \texttt{c-blk} and \texttt{d-blk} rules are disjoint with each other and all other rules.

(2) The \texttt{cc} rule interacts only with \texttt{cd} and \texttt{dc}. These are presented in Figure 2.5 a,b.

(3) Rule \texttt{d} interacts with rules \texttt{dd} and \texttt{cd} and these are presented in Figure 2.6 a,b.

(4) The \texttt{loop} rule interacts with rules \texttt{dc}, \texttt{dd} and \texttt{d-loop} and is given in Figure 2.7 a,b,c.
Figure 2.4 Diagrams for self-interacting transformations
Figure 2.5a,b

Diagrams for cc with dc and cd
Figure 2.6a,b

Diagrams for d with dd and cd
Figure 2.7a,b,c  Diagrams for loop with dc, dd and d-loop
(5) Rule \text{dc} interacts with three rules: \text{dd}, \text{cd} and \text{d-loop} as given in Figure 2.8 a,b,c.

(6) Rule \text{cd} interacts with \text{dd} in Figure 2.9

(7) Finally \text{d-loop} and \text{dd} interact in Figure 2.10.

This exhausts the choices for x and y and completes case 2b.

Property P3 is thus established.

By Sethi's theorem (Theorem 2.3) the replacement \( S = (\Sigma_{\text{SSFG}} \Rightarrow \equiv) \) is FCR, the desired result.

We point out that Theorem 2.4 is not obvious since it would not hold if rule \text{cd} were deleted from \( \Rightarrow \).

A graph \( \Gamma_0 \) is said to be \text{irreducible} under \( \Rightarrow \) if there exists no graph \( \Gamma_1 \) such that \( \Gamma_0 \Rightarrow \Gamma_1 \). Let \( \Gamma \) be an arbitrary graph with the node types of \( G_{\text{SSFG}} \) and let \( \hat{\Gamma} \) be any irreducible graph such that \( \Gamma \Rightarrow \hat{\Gamma} \). We have the following result.

**Corollary 2.5:**

\[ \Gamma \in \mathcal{F}^+_{\text{SSFG}} \] iff \( \hat{\Gamma} = Q \) where Q is the trivial graph consisting of a single computation node and no edges (Q is the start symbol of \( G_{\text{SSFG}} \)).

**Proof:** Since there is a sequence of productions to produce any graph in \( \mathcal{F}^+_{\text{SSFG}} \) from Q, \( \Gamma \in \mathcal{F}^+_{\text{SSFG}} \) implies \( \Gamma \Rightarrow \Gamma_0 \) such that \( \Gamma_0 \equiv Q \). But by Theorem 2.4, \( \hat{\Gamma} \) must be isomorphic to \( \Gamma_0 \) and hence to Q.

Suppose \( \Gamma \notin \mathcal{F}^+_{\text{SSFG}} \). Then there can be no sequence of applications of \( \Rightarrow \) which will reduce \( \Gamma \) to a graph isomorphic to Q because the inverse of such a sequence would comprise a \( G_{\text{SSFG}} \) derivation for \( \Gamma \). Thus \( \hat{\Gamma} \) cannot be isomorphic to Q.

This corollary provides a useful test for SSFG-reducibility: apply \( \Rightarrow \) repeatedly to the input graph until an irreducible graph \( \hat{\Gamma} \) is produced;
Figure 2.8a,b

Diagrams for dc with dd and cd
Figure 2.8c

Diagrams for dc with d-loop
Figure 2.9
Diagram for cd and dd

Figure 2.10
Diagram for d-loop and dd
if $\hat{\Gamma}$ is a single computation node with no edges, $\Gamma$ is SSFG-reducible, otherwise it is not.

Next we present an algorithm which parses arbitrary flow graphs in $\mathcal{F}_{SSFG}$. In developing such an algorithm, we might consider the "brute force" method suggested by Corollary 2.5 in which the graph is searched for any applicable reduction on each iteration. That method will always produce a correct parse because of the finite Church-Rosser property; however, it is too inefficient for practical purposes. Our aim is to bound the number of operations during the parse by some linear function of the number of nodes in the original graph. Therefore, we must find a method which is more systematic than "brute force".

The input to our algorithm will consist of:

1. A representation of the graph $\Gamma$, and

2. A list $L_u$ of the nodes of $\Gamma$, called the list of unvisited nodes. Nodes of $\Gamma$ appear in $L_u$ in a special order called straight order [EBA] which is equivalent to the reverse of the order in which the nodes would be last visited by a "depth-first search" on $\Gamma[\Gamma_1, \text{HUL}]$. Straight order has several appealing properties. First, by using depth-first search and a bucket sort, nodes of $\Gamma$ can be arranged into straight order in time linear in the number of nodes and edges in $\Gamma$. Second, straight order uncovers a natural set of backward branches, namely chose edges $(x,y)$ such that $y$ appears before $x$ in the ordering. For our purposes the important property of straight order is as follows: for every node $x$ in $\Gamma$ if $x$ is not the program entry node, there exists at least one edge $(y,x)$ in $\Gamma$ such that node $y$ appears before node $x$ in the ordering. In other words, if we list the nodes of $\Gamma$ in straight order:

a) the entry node of $\Gamma$ will be listed first and
b) if $x$ is not the entry node, at least one predecessor of $x$ in
$\Gamma$ will be listed before $x$ is listed.

In particular, if $(y, x)$ is the only edge of $\Gamma$ incident into $x$, $y$ will be
listed before $x$. This property is an easy consequence of the behavior of
depth-first search algorithms.

Any instance of a reduction under the replacement system $(\mathcal{G}^{+}_{\text{SSFG}}, =, \equiv)$
is a pair $(x, R_x)$ where $R_x$ is a region isomorphic to the right-hand-side
(RHS) of the rule $x$ in $G_{\text{SSFG}}$. Since each such region is single-entry, it
has an entry node which we call the header of $R_x$. For example, in Figure
2.11, node 1 is the header. We say that a node $a$ in some graph $\Gamma \in$
$\mathcal{G}^{+}_{\text{SSFG}}$ is a reduction candidate if $a$ is the header of a region $R$ isomorphic
to the RHS of some rule in $G_{\text{SSFG}}$. In Figure 2.11, node 1 is a reduction
candidate. Suppose that the region $R$ is actually reduced to a single node
$a^1$ to produce a new graph $\Gamma^1$; then $a^1$ might itself be a reduction candidate
in $\Gamma^1$, heading the region $R^1$. Consider the sequence of reductions in which
$R \in \Gamma$ headed by $a$ is reduced to $a^1$ to form $\Gamma^1$; then $R^1, \in \Gamma^1$ headed by
$a^1$ is reduced to $a^2$ to form $\Gamma^2$, etc. This sequence is guaranteed to be
finite since $(\mathcal{G}^{+}_{\text{SSFG}}, =, \equiv)$ is finite, so there must exist an $a^k$ which is the
end result of $k$ reductions and which is not itself a reduction candidate in
$\Gamma^k$. The sequence of reductions from $a$ to $a^k$ is called a collapse of $a$.

Our first subalgorithm performs a collapse at its given input node.

Algorithm C Collapse

Input: (1) the graph $\Gamma$
(2) a node $a$ in $\Gamma$
Figure 2.11

A reducible region
Output:  
1. the graph $\Gamma^1$ derived from $\Gamma$ by a collapse at $a$.
2. the sequence $P_a$ of reductions applied
3. a flag `success` which is true iff at least one reduction is applied.

Method:

begin $P_a := \emptyset$ ; success := false ; reducing := true ; $\Gamma^1 := \Gamma$ ;

while reducing do

for each production $p$ in $G_{SSFG}$ do

if RHS($p$) is isomorphic to a region $R$ in $\Gamma^1$ headed by $a$

then using $p$, reduce $R$ to a single node $\hat{a}$, thus forming a new graph $\hat{\Gamma}$ ;

add the production $p$ to $P_a$ along with some auxiliary information ;

success := true ;
$\Gamma^1 := \hat{\Gamma}$ ;
a := $\hat{a}$ ;

fi

od

reducing := false ;

od

end
Correctness and termination of Algorithm C are fairly obvious; they follow from the argument presented in the paragraph immediately before the algorithm. However, the test at step 4 is not specifically given. An implementation of the loop at step 3 and this test (which also locates the region R) would go as follows. First, the node $a$ and its associated edges are compared to the RHS of each production whose RHS has only a single node. If no match is found, $a$ and one successor are compared with the RHS of each two-node production; if $a$ has two successors and no match is found with the first successor chosen, the process is repeated for $a$ and its second successor. The comparisons can be made in any order because the FCR property guarantees that any correct reduction is part of a legal parse if a legal parse exists.

**Theorem 2.6:**

The total number of collapses performed by Algorithm C during a collapse is bounded by

$$(d + 1) (A + B) + A$$

where $A$ and $B$ are constants and $d$ is the number of edges eliminated by the collapse.

**Proof:** It takes a fixed number of operations to test a production at node $a$, so the maximum number of operations required to test all productions at $a$ is a constant. Call this constant $A$.

Once the test is performed it will be known whether or not a reduction can be performed. If a reduction is to be performed, the graph modification and addition of the reduction to $P_a$ take a fixed amount of time $B$. Since each reduction (except, possibly the first) eliminates at the least one edge, the total number of successful reductions $k$ is less than or equal to
Thus the number of operations required is the time for the successful reductions

\[ k(A + B) \leq (d + 1)(A + B) \]

plus the time \( A \) for the last unsuccessful test, for a total of

\[ (d + 1)(A + B) + A \]

This is the desired result.

With the "collapse" process as a subalgorithm, we now define the full parse process.

The algorithm begins by considering the program entry node first. It "visits" this node and then successively visits other nodes in the graph by removing them from the list of unvisited nodes.

Specifically, when a node \( a \) is "visited," we attempt to collapse it. If the collapse is unsuccessful (no reductions are performed), a link to \( a \) is inserted in a special field within each of its unvisited successors and the next unvisited node is visited. If the collapse is successful and \( a \) has been linked to one of its predecessors after a previous collapse, that predecessor is revisited to attempt further reductions, otherwise processing moves to the next unvisited node.

**Algorithm**

\[ G_{SSFG} \text{ Parse} \]

**Input:**

1. Representation of \( \Gamma \)
2. List of unvisited nodes; in straight order

**Output:** A parse \( P_{\Gamma} \) of \( \Gamma \) if one exists; otherwise a failure report.

**Method:**

1. \( \text{begin } a := \text{ the entry of } \Gamma; \ P_{\Gamma} = \emptyset; \)
2. \( \text{remove } a \text{ from the list of unvisited nodes;} \)
3 \underline{while} \ a \neq \text{null} \\
\underline{do} \\
4 \ \underline{collapse} \ a \ \underline{using} \ \text{Algorithm C} \\
\ \underline{\phi} \ \Gamma', \ P_a \ \underline{and} \ \text{the flag \ success \ are} \ \text{provided} \ \phi \\
5 \ \underline{make} \ \underline{a} \ \text{the unique linked predecessor of all unvisited} \\
\ \underline{successors \ of} \ \underline{a} \ \underline{in} \ \Gamma' \\
\ \underline{add} \ P_a \ \underline{to} \ P_\Gamma \\
6 \ \underline{if} \ \text{success} \ \phi \ \underline{a \ reduction \ was \ successful} \ \phi \\
\ \underline{and} \ \underline{a} \ \underline{is \ linked \ to \ a \ predecessor} \\
\ \underline{then} \ a := \underline{linked \ predecessor \ of} \ a \\
\underline{else if \ the \ list \ of \ unvisited \ nodes \ is \ empty} \\
\ \underline{then} \ a := \text{null} \\
\underline{else} \ a := \underline{the \ nexted \ unvisited \ node} \\
\ \underline{remove \ this \ node \ from \ the \ list \ of} \\
\ \underline{unvisited \ nodes} \\
\underline{fi} \\
\underline{fi} \\
\underline{od} \\
\underline{if} \ \Gamma \ \underline{is \ now \ a \ single \ computation \ node} \ (\Gamma \equiv Q) \\
\ \underline{then} \ \underline{the \ original \ graph \ is \ in \ the \ grammar} \ \underline{and} \ P_\Gamma \ \underline{is \ a \ parse} \\
\ \underline{else} \ \underline{the \ original \ graph \ is \ not \ in \ the \ grammar} \\
\underline{fi} \\
\underline{end}
Theorem 2.7:

Algorithm P terminates and finds a correct parse if one exists, failing if and only if a parse does not exist.

Proof: If Algorithm P reduces $\Gamma$ to a single node, then the sequence of reductions certainly forms a parse; $\Gamma$ is therefore in the language.

Suppose $\Gamma$ is not so reduced. Can $\Gamma$ nevertheless be in the language? Recall that reductions according to this grammar are finite Church-Rosser; thus if $\Gamma$ is in the language but Algorithm P terminates without finding a complete reduction, the resulting graph can be further reduced. We will prove the theorem by showing that if there is another reduction, Algorithm P will find it.

At any point during execution of the algorithm, the nodes in the current graph are subdivided into two classes:

1. the nodes which have been visited
2. the nodes which have not been visited, i.e. those which remain on the list of unvisited nodes.

Claim: Whenever a nodes is taken from the list of unvisited nodes, there are at that point no further reductions possible at any node that has been visited.

Proof of claim: by induction on the number k of nodes visited.

Basis: ($k = 1$)

After the program header node is visited and the next node is taken, all possible collapses have been performed at the header. Since no other nodes have been visited the claim holds.

Induction step:

Suppose the claim is true for $k \leq K$. We establish it for
$k = K + 1$.

Case 1:

If no reductions are performed at node $K + 1$, none are possible. The graph is left in exactly the same state as after the $K^{th}$ node visited so none of the visited nodes can be further reduced.

Case 2:

Reductions are performed at node $K + 1$. Certainly no further reductions are possible at node $K + 1$. However, the collapse at node $K + 1$ might make further reductions possible for nodes previously visited. This can be possible only if

1. a predecessor of $K + 1$ is one of the previously visited nodes (the reduction must take place at a predecessor and must involve node $K + 1$) and
2. node $K + 1$ has only one predecessor (there is no reduction involving two nodes where the second has a predecessor other than the first).

But if a previously visited node $J$ is the only predecessor of $K + 1$ then a link to $J$ was installed in $K + 1$ during the last visit to $J$ ($J$ must have been previously visited because the nodes are visited in straight order) and after $K + 1$ is collapsed the algorithm backs up to $J$ to try further reductions. Thus if any further reductions are possible they are done during this back-up phase.

This completes the induction and proves the claim.

If the algorithm terminates, it does so only after all nodes have been removed from the unvisited list. Therefore, upon termination, no further reductions are possible.
Note that on each iteration of the while loop at 3, either a node is removed from the unvisited list or a node is successfully collapsed. Nodes can be removed from the unvisited list at most times where \( n \) is the number of nodes in the original graph \( \Gamma \). Nodes can be collapsed at most \( e \) times, where \( e \) is the number of edges in \( \Gamma \), since each successful reduction eliminates at least one edge. Therefore, the finiteness of \( \Gamma \) implies termination.

\[ \square \]

**Theorem 2.8**

Algorithm P terminates on any graph after at most \( \mathcal{O}(e) \) operations where \( e \) is the number of edges in the original graph.

**Proof:** All parts of the while-loop (at step 3) take at most constant time except for the call to Algorithm C. On each iteration, either a node is collapsed or a node is removed from the list of unvisited nodes. The number of iterations is thus bounded by \( n + k \), where \( k \) is the number of times that some node is actually collapsed. From Theorem 2.6, the number of operations for each collapse is \( \mathcal{O}(d) \) where \( d \) is the number of edges removed. Thus the total number of operations during all \( k \) collapses is \( \mathcal{O}(e) \).

When a node is removed from the unvisited list, the number of operations is a constant. Thus the total number of operations is \( \mathcal{O}(n + e) \) or \( \mathcal{O}(e) \) since \( n \leq e \). Thus the algorithm has time complexity \( \mathcal{O}(e) \).

We point out here that in a practical implementation, the parse time would be improved by a constant factor if the rules c-blk and d-blk were not used. Use of them can be avoided if all basic blocks are initially assigned type "computation node" and all basic blocks with tests are assigned type "decision node".
Corollary 2.9

For graphs in $S_{SSFG}$, the parse is produced in time $O(n)$.

Proof: For such graphs $e \leq 2n$. \hfill \square

A few comments on the form of the parse produced would be appropriate here. A parse is a sequence

$$(r_1, r_2, \ldots, r_m)$$

where each $r_i$ is a reduction of the form

$$(\text{type, } n\text{-old, } n\text{-new, } e\text{-old, } e\text{-new})$$

where

- type - is the name of the production applied;
- n-old - is the set of nodes (header first) consumed;
- n-new - is the name of the composite node produced;
- e-old - is the set of edges in the graph which are to be replaced by edges in e-new;
- e-new - is the set of edges in the new graph which begin or end at n-new.

If a reduction $r_i$ of the above form is applied to a graph $\Gamma = (N, E, n_0)$ a new graph $\Gamma' = (N', E', n_0')$ is formed where

- $N' = (N - n\text{-old}) \cup (n\text{-new})$
- $E' = (E - e\text{-old}) \cup (e\text{-new})$
- $n_0' = \begin{cases} n_0 & \text{if } n_0 \in n\text{-old} \\ n\text{-new} & \text{if } n_0 \notin n\text{-old} \end{cases}$

If $(r_1, r_2, \ldots, r_m)$ forms a parse of graph $\Gamma$, the result of applying $r_1, r_2, \ldots, r_m$ to $\Gamma$ in sequence is the trivial graph $\Gamma_0$ consisting only of a single computation node (with no edges).

Actually the full form of the reductions above cannot be computed in
linear time so the following simplification must be used. Initially, each node in the graph is represented by a pair

\[(x, 0)\]

where \(x\) is the original node name. Whenever a reduction is performed if the header is named \((x, i)\) then the new composite node is named \((s, i + 1)\). Edges will be represented by pairs of original node names without the index numbers, so there is no need to replace edges coming into the header by edges into the new node. The set \(e_{\text{old}}\) then contains every edge leaving a node in the region \(\mathcal{R}\) collapsed and the set \(e_{\text{new}}\) is as follows.

\[e_{\text{new}} = \{(x, y) \mid (z, y) \in e_{\text{old}} \& y \in \mathcal{R}\}\]

where \(x\) is the new node name.

This simplification restricts the amount of work to be done in constructing \(e_{\text{new}}\) to a linear function of the number of edges leaving \(\mathcal{R}\). But since \(|\mathcal{R}| \leq 2\), the total number of operations is bounded by a constant.

Next we show how this fast parse is used to perform data flow analysis.
CHAPTER THREE

APPLICATION TO GLOBAL DATA FLOW ANALYSIS

We shall consider data flow analysis by studying a representative problem, that of locating "live" variables within a program. Given a variable $X$, which is defined at various points in a program, we wish to determine for each point $p$ in a program flow graph whether or not $X$ will be used after control leaves $p$. We say that $X$ is live at $p$ if it can be used again and dead at $p$ otherwise. The "live" information would be useful in register allocation for example, since the value of a variable which can never be used again need not be saved.

For simplicity, we formulate another version of this problem: for each block $b$ in the program, determine the set $\text{live}(b)$ of variables $X$ for which there is a path from the entry point of $b$ to a use of $X$, which path is $X$-clear (contains no redefinition of the variable $X$). We can not translate this problem to one of solving a system of boolean equations. Let $\text{inside}(b)$ be the set of variables $X$ which are live on entry to a block $b$ because there is a use of $X$ within which is not preceded by a redefinition. Let $\text{thru}(b)$ be the set of variables $X$ for which there exists an $X$-clear path through $b$. Note that the sets $\text{inside}(b)$ and $\text{thru}(b)$ can be computed by a local examination of block $b$.

Now there exists an $X$-clear path from the entry of $b$ to a use of $X$ if and only if there exists such a path to a use within $b$ or through $b$ to a successor of $b$ and from there to a use. In equation form:

\[ (*) \quad \text{live}(b) = \text{inside}(b) \cup \bigcup_{x \in S(b)} (\text{thru}(b) \cap \text{live}(x)) \]
The solution to this system of equations clearly provides a solution for our simplified live analysis program.

Several methods for the solution of (*) on reducible flow graphs have been proposed [GW, HU1, Ke1, Ke2, Ke3, Ki], all of these run in time proportional of \( n^2 \) or \( n \log n \) where \( n \) is the number of blocks in the program flow graph. Here time is measured in terms of the number of bit vector operations required (bit-vectors are used to implement sets). In this chapter, we show that for flow graphs in \( \mathcal{F}_{SSFG} \), live analysis can be performed in time proportional to \( n \).

We begin with a parse \((r_1, r_2, \ldots, r_m)\) and apply a modified version of Kennedy's live analysis algorithm [Ke1]. In the algorithm description given below we assume instructions "swap (old,new)" which replaces the old nodes and edges by the new node and edges associated with the current reduction. The instruction "swap (new,old)" does the reverse. For notational convenience, we shall assume that the set of successors \( S(x) \) and the set of predecessors \( P(x) \) of node \( x \) refer to the current edge set. Thus these sets are modified by a swap instruction. In a real implementation these sets would be realized by an iteration over edges incident to \( x \).

Given a graph \( \Gamma \in \mathcal{F}_{SSFG} \), live analysis is performed in two passes. The first pass iterates over the parse from \( 1 \) to \( m \) computing inside and thru sets for each composite region. The live set for the whole program can be assigned the value of inside for the single node representing that program. Pass two then computes live sets for each composite node and basic block in an iteration over the parse from \( m \) to \( 1 \).

Pass 1

Suppose that we have an algorithm which, given the thru and inside
sets for nodes in the region $R_x$ to be reduced to node $x$, computes thru and inside sets for $x$. Then that algorithm can be applied once for each step in the parse to compute inside and thru for each composite region. The algorithm is straight-forward.

Algorithm L1: Pass 1

Input: (1) A list $L$ of one or two nodes in the region to be reduced, the header $h$ of the region appearing first.
(2) The composite node $x$ to which the region is to be reduced.
(3) thru and inside sets for each composite node $x$.

Output: thru and inside sets for each composite node $x$.

Method: begin $S := \bigcup_{y \in L} S(y) - L$ ; $\notin$ successors of $x$ ;
for all $y \in S$ do thru $(x, y) + \emptyset$ od ;

$h := \text{header of } x$ ;
inside $(x) := \text{inside}(h)$ ;
for all $y \in S(h)$ such that $y \in L$ do thru $(x, y) := \text{thru}(h, y)$ od

if $L(2) \neq \text{nil}$ then
$z := L(2)$ ;
inside$(x) := \text{inside}(x) \cup$
$(\text{thru}(h, z) \cap \text{inside}(z))$ ;
for all $y \in S(x)$ do
thru$(x, y) := \text{thru}(x, y) \cup$
$(\text{thru}(h, z) \cap \text{thru}(z, y))$ ;

od
\[
\text{\textbf{fi}}
\]
\[
\text{swap (old,new) ;}
\]
\[
\text{end}
\]

This algorithm is applied once for each step in the parse. Let \( X_c \) be the node representing the whole program; then \( \text{live}(X_c) := \text{inside}(X_c) \) since \( X_c \) has no successors. After this assignment Pass two is applied.

**Pass 2**

On this pass, algorithm L2 (below) is applied once for each step in the parse in reverse order; i.e., from \( m \) to \( l \). At each stage of processing, the \textit{live} set for entry to every node in the current graph has been computed.

Algorithm L2 computes \textit{live} sets for the new nodes available after the reverse of a reduction is applied.

**Algorithm L2: Live sets**

**Input:**

1. A graph \( \Gamma \in \mathcal{S}_{SSFG}^+ \)
2. A node \( x \) in \( \Gamma \) to be expanded to region \( R_x \) by the current reduction (in reverse).
3. \textit{live} sets for each node in \( \Gamma \)

**Output:** \textit{live} sets for each node in the region \( R_x \) to which \( x \) is to be expanded.

**Method:**

\[
\text{begin}
\]
\[
h := \text{header of } x ;
\]
\[
\text{live}(h) := \text{live } (x) ;
\]
\[
\text{swap(new,old)} ;
\]
\[
\text{if } R_x \text{ contains a second node } z
\]
\[
\text{end}
\]
\begin{align*}
\text{then} \\
live(z) &= inside(z) \cup \\
&\quad \big( \cup \{ \text{thru}(z,y) \cap \text{lively(y))} \big) \\
&\quad \forall \in S(z) \\\n\text{fi} \\
\text{end}
\end{align*}

Since this algorithm is applied at each production \( r_m, r_{m-1}, r_z, r_1 \) the \textit{live} sets for all nodes in the current graph \( \Gamma \) are available when they are needed to compute \textit{live} sets in Algorithm L2.

\textbf{Theorem 3.1}

The total number of bit-vector operations (set unions and intersections) required by the iterated applications of L1 and L2 is \( \mathcal{O}(n) \) where \( n \) is the number of nodes in the original graph \( \Gamma \).

\textbf{Proof:} The total number of steps in the parse is \( \mathcal{O}(e + n) \) since each step consumes one edge or changes a basic block to a node. Since \( e \leq 2n \) for graphs in \( S_{SSFG} \), the total number of parse steps is \( \mathcal{O}(n) \).

On pass 1, the number of bit vector steps executed in Algorithm L1 is a constant since the number of successors of any node can be at the most two. On pass 2, the number of such steps in Algorithm L2 is a constant for the same reason.

The entire analysis requires a number of bit-vector operations proportional to the parse length which is \( \mathcal{O}(n) \).

In an actual implementation, data flow analysis could be made very efficient by using a table-driven scheme, with the parse as a table. Since the operations performed for a particular reduction type are fixed, the reduction type (\textit{cc}, \textit{cd}, \textit{loop}, etc.) could be used in an indexed jump to special code for that reduction. Data flow analysis is then performed in two
sweeps through the parse with a minimum number of tests. This leads to the attribute grammar scheme of the next section.

Suppose we consider the effect of algorithms L1 and L2 on the reduction of the flow graph in Figure 3.1. The inside set for each node is given in ( ) and the thru set for each node is given in [ ]. As an example, the header node h for the graph in Figure 3.1a has inside set consisting of the variables x and y. Its thru set is empty.

In reducing from the graph in Figure 3.1a to the one in Figure 3.1b, three c-blk, two d-blk and one dc productions are applied. Algorithm L1 computes the following during the transition when node a collapses into node h with the application of rule dc:

\[
\begin{align*}
\text{thru}(h, b) & := \emptyset ; \\
\text{thru}(h, d) & := \emptyset ; \\
\text{inside}(h) & := \text{inside}(h) \cup (\text{thru}(h, a) \cap \text{inside}(a)) \\
& = \{x, y\} \cup \emptyset \\
& = \{x, y\} ; \\
\text{thru}(h, b) & := \text{thru}(h, b) \cup (\text{thru}(h, a) \cap \text{thru}(a, b)) \\
& = \emptyset ;
\end{align*}
\]

similarly \text{thru}(h, d) := \emptyset ;

We next apply the reduction d-loop at node h, it collapses node b into node h in Figure 3.1c. The following sets result:

\[
\begin{align*}
\text{thru}(h, c) & := \emptyset ; \\
\text{inside}(h) & := \text{inside}(h) \cup (\text{thru}(h, b) \cap \text{inside}(b)) \\
& = \{x, y\} ; \\
\text{thru}(h, c) & := \text{thru}(h, c) \cup (\text{thru}(h, b) \cap \text{thru}(b, c)) \\
& := \emptyset ; \\
\text{thru}(h, d) & := \emptyset ;
\end{align*}
\]
A flow chart with inside ( ) and thru [ ] sets given for live analysis.

Reduction cd has been applied to nodes h and a.
Reduction d-loop has been applied to nodes h and b

Reduction dc has been applied to nodes h and c
We apply the reduction dc at node h and collapse node c into node h. The following sets result:

\[ \text{thru}(h,d) := \emptyset \]
\[ \text{inside}(h) := \text{inside}(h) \cup (\text{thru}(h,c) \cap \text{inside}(c)) = \{x,y\} \]
\[ \text{thru}(h,d) := \text{thru}(h,d) \cup (\text{thru}(h,c) \cap \text{thru}(c,d)) = \emptyset \]

We next apply rule d at node h and then rule cc at nodes h and d, see Figure 3.1e. The following sets result:

\[ \text{thru}(h,d) := \emptyset \]
\[ \text{inside}(h) := \text{inside}(h) \cup (\text{thru}(h,d) \cap \text{inside}(d)) = \{x,y\} ; \]

Now we go backward through the parse and compute the live sets for smaller and smaller regions in the flow graph. We obtain the following (see Figure 3.1f):

<table>
<thead>
<tr>
<th>Parse Step</th>
<th>Set Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>\text{h} := h;</td>
</tr>
<tr>
<td>\text{cc}</td>
<td>\text{live}(h) := \text{inside}(h) = {x,y}</td>
</tr>
<tr>
<td>\text{d}, dc</td>
<td>\text{live}(d) := \text{inside}(d) \cup \emptyset = \emptyset ;</td>
</tr>
<tr>
<td></td>
<td>\text{live}(c) := \text{inside}(c) \cup (\text{thru}(c,d) \cap \text{live}(d))</td>
</tr>
<tr>
<td></td>
<td>\quad = {y} \cup ([x,z] \cap \emptyset)</td>
</tr>
<tr>
<td></td>
<td>\quad = {z}</td>
</tr>
<tr>
<td>\text{d-loop}</td>
<td>\text{live}(h) := {x,y}</td>
</tr>
<tr>
<td></td>
<td>\text{live}(b) := \text{inside}(b) \cup ((\text{thru}(b,h) \cap \text{live}(h))</td>
</tr>
<tr>
<td></td>
<td>\quad \cup (\text{thru}(b,c) \cap \text{live}(c)))</td>
</tr>
<tr>
<td></td>
<td>\quad = {x,z} \cup ({y} \cup {y}) = {x,y,z}</td>
</tr>
</tbody>
</table>
Rules \( d \) and \( cc \) are applied with the resulting inside set for the entire reduced graph containing the variables \( x \) and \( y \).

The original flow graph with the computed live sets given inside each basic block.
Parse Step

dc

Set Computation

\[
\text{live}(h) := \{x,y\}
\]

\[
\text{live}(a) := \text{inside}(a) \cup (\text{thru}(a,b) \cap \text{live}(b))
\]

\[
= \{z\} \cup \{x,y\}
\]

\[
= \{x,y,z\}
\]

Summary of Results

\[
\text{live}(h) = \{x,y\}
\]

\[
\text{live}(a) = \{x,y,z\}
\]

\[
\text{live}(b) = \{x,y,z\}
\]

\[
\text{live}(c) = \{y\}
\]

\[
\text{live}(d) = \emptyset
\]

We now know that

(1) the variables \(x\) and \(y\) should be kept in registers during the execution of block \(h\).

(2) the variable \(z\), along with the variables \(x\) and \(y\) should be in registers during the execution of blocks \(a\) and \(b\).

(3) the registers containing \(x\) and \(z\) may be freed upon completion of execution of block \(b\) and only variable \(y\) need remain in a register for block \(c\)'s execution.

(4) no variables need be kept in registers during execution of block \(d\).

Thus, we have given a clear example of how the SSFG parse is used for global data flow analysis.

The grammatical description of program flow graphs allows a number of applications to be carried over from language theory. In particular, many algorithms on the flow graph can be specified via "attributes" on the graph grammar [Kn3,Kn4]. An attribute grammar is an ordinary context free
grammar augmented with attributes and semantic functions. For each symbol in the grammar, there are finite disjoint sets \( I(x) \) and \( S(x) \) of inherited and synthesized attributes respectively. Inherited attributes transmit information down the parse tree toward the leaves, while synthesized attributes transmit information up the tree toward the root. Each production rule in the grammar has a set of semantic functions associated with it. As an example, we show how one might specify the analysis of "profitability" computation on a program flow graph using an attributed version of \( G_{SSFG} \).

Optimizing compiler writers often perform automatic profitability analysis of the programs they are about to optimize. The optimization of seldom executed blocks of code in a program would cost more in time and effort than is gained by the optimization. Time and effort would be best spent optimizing blocks of code that the compiler has determined during previously performed profitability analysis, are going to be executed frequently.

Simply stated, the profitability problem is this. Suppose we are given for each branch \((x,y)\) in the program, the probability \( p(x,y) \) that that branch will be taken after block \( x \) is executed. We wish to determine the expected frequency \( f(x) \) of execution of each block \( x \) in the program during a single execution of the whole program. Under certain simplifying "Markov assumptions" [CK], the expected frequencies can be expressed as follows:

\[
f(n_0) = 1 \quad n_0, \text{ the program entry node}
\]

\[(*)\]

\[
f(x) = \sum_{(y,x) \in E} p(y,x) \cdot f(y) \text{ all other nodes}
\]
A solution-finding method for the system of equations (*) is described by the attribute grammar in Figure 3.2.

There are several points to notice about this graph grammar.

1) The attribute $p$ is a synthesized attribute; that is, its value for composite regions is based on its value for nodes within the region. The values of $p$ for terminal nodes are "given".

2) The attribute $f$ is an inherited attribute; that is, its value for nodes within a region is based upon its value for the region as a whole (and upon the values of $p$).

3) Because of the dependence of $f$ upon $p$, we are forced to evaluate $p$ first. Thus the attribute grammar in Figure 3.2 gives use to a two-pass algorithm. The first pass moves forward through the parse, computing $p$ for larger and larger regions; the second pass moves backward through the parse; computing $f$ for smaller and smaller regions.

4) The first value of $f$ is that for the whole program $n_0$ which is 1 by assumption (the whole program is executed once).

Figures 3.3 and 3.4 present an example profitability computation, with Figure 3.3 depicting the reduction pass and associated transition probability computations and Figure 3.4 depicting the production pass and frequency computations.

From the considerations above and the example, we can see that the attribute grammar specification gives rise to the classical algorithm for profitability [CK]. Furthermore, by applying analogs of techniques in [KW, KR] we can compile these attribute grammars into efficiently executing finite-state machines which use the parse (or its inverse) as input. The
Figure 3.2

Attributed $G_{SSFG}$ for "Profitability"
Figure 3.2 Continued

6) \[
\begin{align*}
p(z, v) &= p(y, v) \\
p(z, w) &= p(y, w) \\
f(x) &= f(z) \\
f(y) &= f(z)
\end{align*}
\]

7) \[
\begin{align*}
p(z, v) &= p(x, y) \\
p(z, w) &= p(x, w) \\
f(x) &= f(z) \\
f(y) &= f(z) \times p(x, y)
\end{align*}
\]

8) \[
\begin{align*}
p(z, v) &= p(x, v) + p(x, y) \times p(y, v) \\
p(z, w) &= p(x, y) \times p(y, w) \\
f(x) &= f(z) \\
f(y) &= f(z) \times p(x, y)
\end{align*}
\]

9) \[
\begin{align*}
p(z, v) &= p(x, v) / (1 - p(x, y) \times p(y, x)) \\
p(z, w) &= p(x, y) \times p(y, w) / (1 - p(x, y) \times p(y, x)) \\
f(x) &= f(z) / (1 - p(x, y) \times p(y, x)) \\
f(y) &= f(z) \times p(x, y) / (1 - p(x, y) \times p(y, x))
\end{align*}
\]
Figure 3.3
A profitability reduction sequence
Figure 3.4
Profitability production sequence
 eventual result may be a system to generate graph-based optimization algorithms. By analogy with [FKZ] these algorithms should be linear in the size of the input program.

We have presented a means for doing flow analysis in time linear in the size of the program flow graph on a large class of programs. How large is this class? In other words, how many programs, written with or without structured concepts, fall naturally into $\mathcal{SSFG}$? That question led us to the investigation described in the next chapter.
CHAPTER FOUR

THE EMPIRICAL STUDY

For our study of the structure of typical application programs, we decided to look at FORTRAN programs since there exist numerous scientific applications programmed in that language and since the control structures do not particularly encourage good programming.

The next step was to acquire programs for the study. Members of the Rice faculty generously provided us with ample supply which we subdivided into the following groups.

1. Sixty FORTRAN programs written by students in an engineering class where "structured programming" was not taught. The sample represented student solutions to two problems:
   a) find the roots of a cubic equation
   b) compute the time to target and the required angle from the plane for a projectile with given velocity and range.

2. Two hundred twenty-seven subroutines from a large Chemistry Dept. application: to construct the wave function for a compound during molecular scattering using the "stabilization" method. The routines were as much as ten years old and many had been written at other universities.

3. Sixty-two Physics Dept. routines used to calculate eigenvalues and wave functions for certain Hamiltonians in a molecular structure application.

4. Seventy-six Biochemistry Dept. routines comprising a large crystallographic system for analyzing X-ray diffraction data to deduce molecular structure.
5. Five routines used by the Chemistry Dept. to determine mean square end-to-end separation of molecules via a Monte Carlo simulation of macro molecular structure.

6. Thirty-six routines comprising two Mathematical Sciences Dept. applications:
   a) a finite element differential equation solver which uses a Galerkin procedure and isoparametric elements.
   b) a collection of H1 procedures for solving quadratic homogeneous partial differential equations.

7. Thirty-four routines from two Geology Dept. programs:
   a) a program which solves the free convection problem for a medium with variable viscosity by solving a system of coupled partial differential equations.
   b) a program which simulates the slow flow of liquids under given boundary conditions.

All in all, a total of five-hundred routines were analyzed to determine how many were Cocke-Allen reducible and how many were SSFG-reducible. The results are summarized in Table 4.1 below.
<table>
<thead>
<tr>
<th>Group</th>
<th>Number of Programs</th>
<th>Number of Cocke-Allen Reducible</th>
<th>% Cocke-Allen Reducible</th>
<th>Number SSFG-Reducible</th>
<th>% SSFG-Reducible</th>
<th>% of Cocke-Allen reducible programs SSFG-reducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>60</td>
<td>58</td>
<td>97</td>
<td>55</td>
<td>92</td>
<td>94.8</td>
</tr>
<tr>
<td>2</td>
<td>227</td>
<td>221</td>
<td>97</td>
<td>199</td>
<td>88</td>
<td>90.4</td>
</tr>
<tr>
<td>3</td>
<td>62</td>
<td>60</td>
<td>97</td>
<td>56</td>
<td>90</td>
<td>93</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
<td>62</td>
<td>81.5</td>
<td>41</td>
<td>54</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>33</td>
<td>92</td>
<td>30</td>
<td>83</td>
<td>91</td>
</tr>
<tr>
<td>7</td>
<td>34</td>
<td>32</td>
<td>94</td>
<td>30</td>
<td>88</td>
<td>94</td>
</tr>
<tr>
<td>Total</td>
<td>500</td>
<td>470</td>
<td>94</td>
<td>411</td>
<td>82</td>
<td>87.5</td>
</tr>
</tbody>
</table>

Table 4.1 Reducibility Analysis
The last column, indicating the percentage of Cocke-Allen reducible programs which are also SSFG-reducible is worth comment. Since most fast data flow analysis routines work for Cocke-Allen reducible programs, one should compare the linear-time SSFG method with those for applicability. The last column in Table 4.1 indicates that the SSFG method would work in nearly 90% of those cases which could be handled by one of the fast on-linear methods [GW, Ke3, AU3].

Overall, more than four out of five of these programs, all written without the benefits of structured programming, were SSFG-reducible. Though we were pleased with this result, we were disturbed by the intractability of groups 4 and 5, without which 88% of the programs would have been SSFG-reducible. We therefore undertook a study of the programs in these two groups to determine the reasons they failed to reduce so frequently.

The first step was to perform a static analysis on all the programs to determine average length and frequencies of various statement types. The results of this study appear in Table 4.2.
<table>
<thead>
<tr>
<th>Group</th>
<th>Average number of statements</th>
<th>Average number if-statements</th>
<th>Average number do-loops</th>
<th>Average number GOTO</th>
<th>Average number Computes GOTO</th>
<th>Average number labelled Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>41</td>
<td>2.6</td>
<td>0.4</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>97</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>0.28</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>66.5</td>
<td>5</td>
<td>6</td>
<td>2.5</td>
<td>0.25</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>31</td>
<td>11</td>
<td>15</td>
<td>1.5</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>320</td>
<td>50</td>
<td>10</td>
<td>20</td>
<td>3</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 4.2 STATIC ANALYSIS
From this table we can see that the program in groups 4 and 5 were extremely long and contained many goto-statements. If we compare the statistics for group 4 with those for group 2, which reduced more frequently, we see that group 4 programs were three times as long (on the average) but contained five times as many goto's, five times as many labelled statements and only twice as many do-loops. It seems likely that many of the loops in group 4 were implemented with goto's rather than do's resulting in a fairly complex control flow structure.

Our next step was to analyze programs in groups 4 and 5 to determine specifically why they failed and to see if a richer grammar might reduce them. For purposes of this analysis, we defined two new grammars. The first, extended the SSFG grammar by adding two new rules depicted in Figure 4.1. The resulting grammar is called SSFG and an analysis of groups 4 and 5 showed that an additional 7 programs reduced under these rules.

A more ambitious grammar admitting 3-exit regions was also considered. This grammar, named SSFG, consists of rules 1-7 of SSFG and the nine rules depicted in Figure 4.2. Groups 4 and 5 were also analyzed for reduction under this grammar. The results of these studies are summarized in Table 4.3. It can be seen that significant improvements in reduction percentages can be achieved by using the larger grammars.
Figure 4.1

SSFG -extensions
Figure 4.2

Additional rules for SSFG₃
Figure 4.2 Continued
<table>
<thead>
<tr>
<th></th>
<th># of Cocke-Allen reducible</th>
<th># SSFG reducible</th>
<th># SSFC\textsubscript{X} reducible</th>
<th>% Cocke Allen SSFC\textsubscript{3} reducible</th>
<th># SSFC\textsubscript{3} reducible</th>
<th>% Cocke-Allen which were SSFC\textsubscript{3} reducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Biochem</td>
<td>76</td>
<td>62</td>
<td>41</td>
<td>47</td>
<td>76</td>
<td>58</td>
</tr>
<tr>
<td>5. Chemistry</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>25</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.3
Close analysis of these failing programs showed that the common causes of irreducibility were that programmers.

1. avoided replicating a few (usually from one to five or six) lines of code by using a backward branch to a previous occurrence of that code and then test and branch from there forward to the instruction immediately following the first branch point.

2. used implicit loop structures (coded their own loops with if-statements and GOTOs) and jumped into loops other than through the loop head.

3. and used complex computed GOTOs which would have been SSFG-reducible if a Zahn type iterative case could have been used instead.

The computed GOTO problem needs more comment. The CASE control structure controls exits from itself. When there are no GOTOs used in the CASE structure, all exits from the structure are forced to be from a common point at the bottom. Also, entry to any particular case from outside the CASE structure is forbidden. Thus CASE structures are single entry, single exit regions. However, a computed GOTO does not control the exits from the labelled statements associated with the GOTO. In other words, from one labelled statement in the GOTO, control can transfer to any labelled statement in the routine. Even more, transfer to a GOTO associated labelled statement can be from any point in the routine. This unrestricted entry and exit capability of computed GOTOs allows more than one computed GOTO to share labelled statements. This gives computed GOTOs the potential of being used to generate multiple entry and exit regions.
The flow graph in Figure 4.3 is an example of the use of a CASE structure. Note that the CASE structure is treated by the SSFG rules as a chain of if-then-else statements which is how compilers sometimes implement CASE statements and computed GOTOs. The figure illustrates how the CASE structure forces every case reached from the structure to exit to a point common to all the cases which is the node immediately below the CASE structure in the flow graph (this point is the instruction immediately following the CASE statement in the code). It is interesting to note that the CASE structure is reduced by the \texttt{dd}, \texttt{dc} and \texttt{d} rules of SSFG.

Compare Figure 4.3 with the flow graphs of Figure 4.4a,b. The computed GOTO in Figure 4.4a has branches from its GOTO associated labelled statements crossing over each other to regions of the program outside the computed GOTO structure. Then these regions do not merge to a common point in the program until some point in determinately further down in the code. As there is no control over the exits from a computed GOTO, we cannot always determine where the computed GOTO structure terminates and thus from which point to travel backup the flow graph to find the start of the computed GOTO region. The flow graph in Figure 4.4b illustrates what happens to the flow of control in a program where two computed GOTOs have labels in common: this particular example flow graph exhibits the structure of Hecht and Ullman's counterexample flow graph that determines Cocke-Allen irreducibility.

From a compiler writer's point of view, programmers would be wise to follow Kernighan and Plauger's advice to be sure that "all CASEs should end by branching to a common point" and to "use GOTOs only to implement a fundamental [control] structure" [KPl, pp.40, 45].
Figure 4.3

Proper use of a Case Statement
Figure 4.4a,b Improper use of the computed GOIO
It is thus our belief that the programs in groups 4 and 5 have too many complex structures (in part due to FORTRAN's limitations) and that increasing the complexity of the SSFG grammar is not an appropriate way to deal with such programs. Can we perhaps copy code to turn SSFG irreducible programs into reducible ones?
CHAPTER FIVE

GENERALIZING SSFG AND NODE-SPLITTING

It is well known that Cocke-Allen irreducible flow graphs can be node-split, i.e. code can be copied, to become Cocke-Allen reducible. But this is an expensive process both in the time it takes and in the size of the graph produced. The worst case cost is exponential in $n$ where $n$ is the number of nodes in the irreducible flow graph we are splitting. So optimizing compiler writers seldom, if ever, use code copying as a method of allowing optimization of difficult programs.

However, the problem is still an interesting one. Given a Cocke-Allen reducible, SSFG-irreducible flow graph, can we always find a SSFG-reducible node splitting for it? The answer appears to be no.

The basic method to split nodes in an SSFG irreducible flow graph (we will assume that all graphs that are candidates for SSFG node splitting are Cocke-Allen reducible; if they are not, we can make them Cocke-Allen reducible) is to find nodes in the graph that have more than one predecessor and which are not loop headers and then to split these nodes so that each predecessor has its own unique copy of the split node. The reason for picking nodes for splitting this way becomes clear after examination of the $G_{SSFG}$ rules:

1. no $G_{SSFG}$ rule has a second node with more than one predecessor
2. the header node in each two node rule is the only node which may have more than one predecessor.

So the only candidates for SSFG node splitting are nodes which have at least $i$ predecessors where $j \ (2 \leq j \leq i)$ of the predecessors are not dominated by the candidate node. Thus copies of the candidate node become
candidates for reduction as second nodes in the two node $G_{SSFG}$ reduction rules. The predecessor node that just obtained its own copy of the split node is the header of the two node region to be reduced.

The restrictions imposed by SSFG node splitting on the candidate nodes suggests that there are Cocke-Allen reducible, SSFG irreducible flow graphs can not be split for successful SSFG reduction. The graph $\Gamma_{SSFG}$ in Figure 5.1, we believe, is not SSFG node-splittable. It does not contain any node that has more than one predecessor where two or more predecessors are not heads of backward edges. Thus it contains no candidate nodes for splitting. However, this is only conjecture and remains to be proven.

So it appears that code-copying will not always work as a method for fixing SSFG irreducible flow graphs. Even if it did, it would be prohibitively expensive.

Another interesting question is the relationship between rules and Hecht and Ullman $T_1$ and $T_2$ transformations. These are given in Figure 5.2 along with the edge absorbing transformation $T_3$, ordinarily implicitly contained in transformations $T_1$ and $T_2$. The $G_{SSFG}$ and $G_{SSFG_3}$ rules are exactly the $T_1$, $T_2$ and $T_3$ transformations explicitly given for one, two and three exit regions. So it is clear that if $G_{SSFG}$ were extended to $G_{SSFG_\infty}$ where any number of exits were allowed from each single entry region, then the SSFG$\infty$ grammar would be exactly the $T_1$, $T_2$ and $T_3$ transformations. Hence the class of graphs generated by the Hecht-Ullman transformations, denoted HU, is exactly the class generated by $G_{SSFG_\infty}$. Figure 5.3 presents SSFG$\infty$ reductions of the Hecht-Ullman transformations. Thus HU $\equiv$ SSFG$\infty$, depicted in Kosaraju's hierarchy of control structures given in Figure 5.4. Note that HU $\equiv$ SSFG$\infty$ $< G_P \equiv L$ as the L class includes programs with free use of GOTOs and labelled statements.
Figure 5.1

$\Gamma_{SSFG}$: Non SSFG-reducible, non-node splittable flow graph
Hecht-Ullman flow graph transformations
Figure 5.3

Reduction of $G_{SSF_G}$ by Hecht-Ullman transformations
Figure 5.4
Hierarchy of Hecht-Ullman and SSFG reducible flow graphs
Thus the Cocke-Allen reducible flow graphs are precisely the family of graphs by $G_{SSFG} : \mathcal{F}_{HU} \equiv \mathcal{F}_{SSFG}$. 
CHAPTER SIX

SUMMARY, CONCLUSION AND POINTS FOR FURTHER STUDY

We have introduced a simple graph grammar $G_{ssfg}$ (the Semi-Structured Flow Graph Grammar) which is significantly less restricted than the standard Böhm and Jacopini grammar for structured programming; in fact, $G_{ssfg}$ contains many of the control structures proposed in the literature.

The set of graph transformations defined by the reductions of $G_{ssfg}$ have the "finite Church Rosser" property, which allows us to apply reductions in any order and be assured of finding a valid parse if one exists. Taking advantage of the FCR property, we have derived a parsing algorithm which, through a specific search ordering strategy, finds a complete parse (if one exists) of an input graph in time linear in the number of nodes in the graph; the length of the parse is also linear in the number of nodes. In turn, the parse is used in a linear-time algorithm for global data flow analysis. Previous work [Ke3] has produced linear data-flow algorithms for the Böhm and Jacopini grammar, but this result extends linearity to a much larger class of graphs, comparable to but not the same as the class of graphs on which the Graham-Wegman data flow algorithm is linear [GW].

The $G_{ssfg}$ parse is useful not only for data flow analysis but also as a clear representation for the structure of a control flow graph. As such it can be used in a variety of optimization algorithms such as code motion, common subexpression elimination and reduction in strength. The structural information will allow these optimizations to be applied to loops from inner loops outward.

For languages with explicit control structures and no goto's the syntactic parse can also provide the $G_{ssfg}$ parse. For such languages,
"attribute grammars" might be used to perform automatic data flow analysis in a manner similar to the method described herein [Kn2, Kn3, J1, J2, KW, KR].

An empirical study has shown that programmers (even those untutored in "structured programming") tend to write programs which are derivable using $G_{SSFG}$. We conclude that a typical "good programmer" would not find rules based upon $G_{SSFG}$ too restrictive.

We have conjectured that it is not always possible to find an SSFG reducible node-splitting of arbitrary SSFG-irreducible flow graphs. To prove this result, more must be learned about the overall structure of SSFG reducible flow graphs.

Fosdick and Osterwell [FKO] describe methods for using data flow analysis procedures to improve software reliability. It would be an interesting problem to explore applications of SSFG flow analysis procedures to this problem.

Foremost, the SSFG linear flow analysis method must be implemented as a tool for optimization in production compilers to demonstrate finally its usefulness and practicality. The languages compiled could be designed to implement the SSFG control structures. The programmer will not find the resulting languages at all restricting. The compiler writer will find the SSFG parse simple to understand and reasonably easy to implement for global program flow analysis.
References


[Kn1] D. E. Knuth, "Structured programming with GOTO statements", 


AI Project Memo ATM-137, Computer Science Dept. Report No. 
CS-186, Stanford University, Stanford, Ca.


program synthesis", SIGPLAN Notices, Vol. 9, No. 4, April 1974.

[Lo] D. B. Loveman, "Program improvement by source to source transforma-
tion", Proc. 3rd ACM Symp. POPL, Atlanta, Ga., pp. 140-152, 
January 1976.


sequence in directed graphs", IBM Research Report RC 5477, 
Yorktown Heights, N. Y., July 1975.

[Ma] J. J. Martin, "The 'Natural' set of basic control structures", 

[Mat] D. Matuszek, "The case for the assert statement", SIGPLAN Notices, 
Vol. 11, No. 8, pp. 36-38, August 1976.

[McC] C. L. McClure, "Top-down, bottom-up and structured programming", 
Trans. on Software Engineering, Vol. SE-1, No. 4, pp. 397-402, 
December 1975.

[Mc] W. M. McKeeman, "On preventing programming languages from interfe-
ring with programming", Trans. on Software Engineering, Vol. SE-1, 

IBM Report, Gaithersburg, Md., February 1972.

[Na] P. Naur, "Programming languages, natural languages and mathematics", 


