INVERSE TRANSLATION:

THE THEORY OF PRACTICAL AUTOMATIC PROGRAMMING

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy
Research Report #113

by

Steven Peter Reiss

December, 1977

ABSTRACT

INVERSE TRANSLATION:
THE THEORY OF PRACTICAL AUTOMATIC PROGRAMMING

Steven Peter Reiss

Yale University, 1977

Programming is an expensive and time-consuming task that should be automated to the greatest extent possible. In this dissertation we determine what this extent is and thereby give bounds and directions for the task of practical automatic programming.

We first show that automatic programming is the problem of inverting a semantics-specifying mapping. From various automatic programming efforts we determine the essentials of the problem. Then, using a series of simplifications and formalisms, we show that a specific problem is defined by a mapping that takes the target language into the source, and that automatic programming is just the task of inverting this mapping.

This concise model is used to determine what practical automatic programming actually is. We show that the complexity of the problem depends on the type of mapping to be inverted. We then note that recursive mappings are too complex to be practical while gsm mappings

are too simple. Between these two extremes we investigate the natural and widely studied class of top-down finite state tree transducers. We prove the exact conditions under which these mappings cannot be tractably inverted. More importantly, we present reasonable algorithms for inversion in the tractable cases.

Based on our model of automatic programming and this complexity study of inverse translation, we conclude by characterizing practical automatic programming. We present examples of how practical problems can be defined using our mappings and algorithms. Moreover, we point out new directions for automatic programming efforts that should prove more successful than current ones.

In short, if you want to get the right answer from a computer, you have to plant your answer ahead of time, just as they do on quiz shows. The whole secret is to ask your question right. The problem must be presented in a series of logical steps which the machine can understand by access to its memory storage units. This process is called _programming_ after its discoverer, Professor J. Mellish Program of Harvard, who discovered it in the bottom drawer of his desk one day while looking for his overshoes.

A computer can be man's best friend if it is kept happy. Try to make your machine happy, and it will make you happy in return.

--_Corey Ford's Guide to Thimking_

Dedicated to Oma Irene

who always knew I would get a Phd.

4. INVERTING SIMPLE TREE TRANSDUCERS . . . 59
4.1 THE MODEL OF TRANSLATION . . . 59
4.1.1 Top-Down Finite-State Tree Transducers . . . 60
4.1.2 Syntax-Based Translations . . . 64
4.1.3 A Look Ahead . . . 70
4.2 LINEAR NONDELETING TREE TRANSDUCERS . . . 72
4.2.1 An Example . . . 72
4.2.2 The Basic Model For Inverting A t-fst . . . 74
4.2.3 Bottom-Up Tree Transducers . . . 75
4.2.4 Parsing The Target Grammar . . . 78
4.2.5 Parser-Transformers . . . 82
4.2.6 Using Parser-Transformers . . . 85
4.2.7 The Complexity Of Simple Parser-Transformers . . . 88
4.3 SYNTAX-DIRECTED TRANSLATION SCHEMATA . . . 90
4.3.1 Inverting An SDTS . . . 90
4.3.2 The Complexity Of Inverting An SDTS . . . 92
4.4 SUMMARY OF BASIC RESULTS . . . 93

5. INVERTING GENERAL TREE TRANSDUCERS . . . 95
5.1 DELETION . . . 95
5.2 NONLINEARITY . . . 100
5.2.1 Describing The Target Language . . . 101
5.2.2 Extended b-fsts . . . 103
5.2.3 Parser-Transformers For Nonlinear t-fsts . . . 106
5.2.4 Complexity Results For Nonlinear t-fsts . . . 109
5.3 GENERALIZED SYNTAX-BASED TRANSLATION SCHEMATA . . . 121
5.3.1 Inverting An Arbitrary GSDTS . . . 122
5.3.2 Determinism Versus Nondeterminism . . . 126
5.3.3 P-Grammars . . . 128
5.3.4 Parsing P-Grammars . . . 134
5.3.5 Common Parses . . . 138
5.3.6 An Algorithm To Parse P-grammars . . . 141
5.3.7 An Algorithm To Determine A Common Parse . . . 150
5.3.8 The Complexity Of The Algorithm . . . 156
5.3.9 Special Types Of GSDTS . . . 161
5.4 INVERSE TRANSLATION AND AUTOMATIC PROGRAMMING . . . 165

6. APPLICATIONS OF THE MODEL . . . 167
6.1 DIRECT APPLICATIONS . . . 168
6.2 AUTOMATIC CODE GENERATION: EXTENDING THE MODEL . . . 169
6.2.1 The Source Language . . . 170
6.2.2 The Basic Machine Description . . . 171
6.2.3 Code Generation With One Class Of Registers . . . 176
6.2.4 Code Generation For Arbitrary Machines . . . 180
6.3 AUTOMATIC CODE GENERATION: RESTRICTING THE PROBLEM . . . 185
6.3.1 The General Problem . . . 186
6.3.2 Restricting The Problem . . . 188
6.3.3 The Overall Model . . . 190
6.3.4 Automatic Code Generation . . . 198
6.4 GENERAL APPLICATIONS . . . 200

7. CONCLUSION . . . 203
7.1 A FIRM FOUNDATION . . . 203
7.2 DIRECTIONS FOR FURTHER STUDY . . . 205

BIBLIOGRAPHY . . . 208

INDEX . . . 222

TABLE OF FIGURES

3.1 Automatic Programming Problems in our Model 55

4.1 An Example t-fst 62

4.2 Restricted Forms of Tree Transducers 64

4.3 An Example GSDTS 68

4.4 An Example SDTS 70

4.5 A Linear Nondeleting Tree Transducer 73

4.6 An Example b-fst 77

4.7 The Target Grammar of the t-fst of Figure 4.5 79

4.8 An Example Parser-Transformer 84

5.1 A Deleting t-fst 97

5.2 A Nonlinear t-fst with a Range that is not Context Free . . 100

5.3 A Sample Extended b-fst 104

5.4 Example of Inversion Solving a 3-satisfiability Problem . . 117

5.5 An Example P-grammar 130

5.6 Inverting a GSDTS by Parsing Its P-Grammar 133

5.7 Inverting a GSDTS 144

6.1 The Language of Triples 171

6.2 A Simple Machine Description 173

6.3 Example of Code Generation with this Model 179

6.4 Machine Description 192

6.5 Sample Flowchart to String Coding 193

6.6 A GSDTS as a Semantic Mapping 196

6.7 Sample Code Generation 200

CHAPTER 1

INTRODUCTION

A computer that lets you state your problem in English and then does the programming for you...

A compiler that can generate code for any machine...

A program specified using only examples...

Are these science fiction or will they someday be fact? Will computers ever be able to do their own programming or will they need human intervention for all but the simplest programs? Will computers be easier to program in the future or are there limits to more and more sophisticated languages? What can we expect of the future?

1.1 THE PROBLEM OF AUTOMATIC PROGRAMMING

For over three decades people have been attempting to simplify the use of computers and have called their work automatic programming. These efforts have been essential to the development of computers and have produced the compilers, editors and loaders that are an integral part of all modern computer systems. But, as these particular tasks have been formalized and as computers and programs have become more sophisticated and the job of programming more complex, the meaning of the term automatic programming has changed. Today it no longer applies to any effort that just simplifies the use of computers. Rather it is used only for those efforts where the computer actually creates its own programs.

As such, automatic programming is an important problem of current interest. Even with all of the simplifications that have been introduced, programming remains a costly and time-consuming task. It still takes a programmer several weeks or even months of work to solve an average problem, and programs still have to be rewritten for each new computer or new language. All this, along with the ever increasing demand for computers, means that more and more programs will be needed in the future. One solution to this growing problem is to have the computer write its own programs, that is, to do automatic programming.

But this is possible only if automatic programming can actually be done. Many tasks have been found that cannot be handled by a machine. Others, such as theorem proving [32], are demonstrably intractable. And for other problems such as graph colorability [47], even getting a good

approximation quickly seems to be impossible. If an instance of automatic programming is similar to any of these problems we cannot realistically expect to solve it on a practical scale.

There is empirical evidence to show that while programming seems easy, automatic programming is difficult. Programming is simple enough to be taught to people of all ages and levels, from elementary school through college. Moreover, most of the effort of programming is spent on the simple, time-consuming and mechanical tasks of coding and debugging, and not on the more difficult problem of algorithm design. While a computer may never be able to practically design programs, it should at least be able to take a program design and automatically implement it. But this also seems out of reach. There have been a number of unsuccessful attempts at automatic programming. Researchers at MIT, ISI, Stanford, Yale and elsewhere have been working on the problem for a number of years. And, although these efforts have been noteworthy, they have not produced a system that can create anything but the simplest of programs. True automatic programming has not been achieved.

This contrast between the apparent mechanical nature of programming and the difficulty of automating the task raises several interesting and important issues. Automatic programming in the most general sense is known to be very hard, and yet there is no good feel for what makes it so. We first want to determine why automatic programming is hard so that we can concentrate on restricted forms of the problem that are still useful. It is these instances of practical automatic programming that are the focus of this dissertation. Our study determines what

kinds of automatic programming problems can realistically be solved. It shows what parts of the task of programming can be automated. And it illustrates how practical automatic programming should be done.

1.2 PRACTICAL AUTOMATIC PROGRAMMING

In order to study practical automatic programming we must first understand exactly what we mean by the term practical. A practical problem represents a compromise between the two conflicting criteria of tractability and the ability to do something worthwhile. A practical problem must be solvable within a realistic time frame and at the same time must be worthwhile in that it must be both nontrivial and useful. The goal of our research, characterizing practical automatic programming, is then accomplished by determining the widest class of automatic programming problems whose solution is feasible.

To determine such a class of problems we need a formal definition of both the term tractable and the term worthwhile. A problem is generally called tractable if it can be solved in polynomial time, that is, if there is a polynomial P such that given input x, the problem can be solved in time at worst $P(|x|)$ where $|x|$ indicates the size of x. This notion was introduced by Edmonds [39] and Karp [67,68] and is currently widely accepted. It is a reasonable one since if a problem can be solved in polynomial time, it can generally handle larger inputs within a realistic time frame. For example, doubling the size of a polynomial problem can only increase its time complexity by a constant factor. In contrast, a problem is called intractable if it requires at least

exponential time, that is, if there is no polynomial P such that the problem is solvable in time $P(|x|)$ on input x. A problem which requires exponential time cannot generally be solved for a large number of inputs as doubling the input size can square the run time of the program.

While many problems are known to be tractable or intractable, there are two classes of problems which are not, those that are NP-complete and those that are PSPACE-complete. A problem is in the class NP if we can solve it in nondeterministic polynomial time, that is, if we are allowed to guess the proper solution and only have to insure that this guess is correct. Clearly any problem that is solvable in polynomial time is in this class. We then say that a problem in NP is NP-complete if it is at least as hard as any problem in NP. Many of the problems that are NP-complete--the traveling salesman problem, satisfiability, integer programming, graph colorability--have been widely studied for a considerable time and no polynomial solutions are known. Along with other evidence, this has led to the belief that NP-complete problems cannot be solved in polynomial time and hence are intractable. The second class of problems we want to consider is the class PSPACE. A problem is in this class if it requires intermediate storage space polynomial in the size of the input. A problem is then called PSPACE-complete if it is in the class PSPACE and is at least as hard as any other problem of the class. PSPACE-complete problems include all problems that are NP-complete and possibly others as well, and are thus also considered to be intractable. In this light we say that an automatic programming problem is tractable if it is solvable in polynomial time

and is intractable if it is at least as hard as any problem in the class NP.

It is more difficult to find a definition of the term worthwhile. Since we are actually interested in finding the most useful automatic programming problem that is still tractable, the notion of worthwhile serves only to set a minimum standard for what we consider as automatic programming. We only want to insure that such a problem can involve an actual programming language, and thus, as almost all programming languages have an underlying structure that is at least context free, we say that an automatic programming problem is worthwhile if it at least involves programs in a context free language.

1.3 A LOOK AHEAD

The object of this dissertation is to study the complexity of automatic programming in order to determine what practical automatic programming is and what it is not. We first build a concise mathematical model for automatic programming that shows that it is just the problem of inverting a semantics-specifying mapping. Next we undertake a comprehensive study of the complexity of automatic programming through a study of the complexity of inversion. We show exactly which classes of semantics-specifying mappings can be inverted tractably and which cannot be. This study illustrates the classes of automatic programming problems that are practical and allows us to determine the complexity of any specific problem. Moreover, it actually presents algorithms for doing practical automatic programming. Finally we consider several

applications of our model. These demonstrate its strengths and weaknesses and show in more detail exactly what practical automatic programming is and how to do it.

This dissertation is completely self-contained. Chapter 2 gives all the definitions necessary to understand the technical aspects of the work. Chapter 3 develops and formalizes the mathematical model of automatic programming. The comprehensive study of the complexity of inverse translation is contained in chapter 4 which defines the appropriate translations and considers the simpler cases, and chapter 5 which extends these results and shows where the boundary between practical and impractical automatic programming lies. Applications of the model are presented in chapter 6. And finally chapter 7 concludes by discussing the meaning and uses of our results. A reader mainly interested in the theory of inverse translation should concentrate on chapters 4 and 5; a reader interested primarily in how our results relate to automatic programming should emphasize chapters 3, 6 and 7.

CHAPTER 2

NOTATION AND BACKGROUND

This chapter provides the notation and background information that is required to understand the techniques and results of this dissertation. This includes the mathematical concepts of sets, mappings, graphs and trees; the language theoretic concepts of strings, languages and grammars; and the concepts of reducibilities and complete problems from the theory of computational complexity. Moreover, since trees are used throughout the disseration as a way of describing both languages and mappings, a number of methods are introduced for describing and manipulating them. We begin with some basic notation.

2.1 GENERAL NOTATION

We generally use specific types of symbols for various objects. These uses can be summarized as

$\alpha,\beta,\gamma,\delta,\ldots$	Strings
$\Sigma,\Delta,\Gamma,\ldots$	Sets of symbols or strings
A,B,C,...	Nonterminal symbols
G	Grammars
P,R	Sets of productions or rules
Q	Sets of states
T	Transformations

$a,b,c,\ldots$	Terminal symbols
q	States
s	Strings of terminal symbols
t	Trees
x,y,z	Variables

We use subscripts, primes (′ or ″) and bars ($\bar{\ }$) to further distinguish between different symbols.

Most of our notations are standard. Braces ({}) are used to denote sets, and $\{x \mid R\}$ denotes the set of all objects satisfying the condition R. Moreover the notations $\lceil x \rceil$ and $\lfloor x \rfloor$ are used respectively to mean the smallest integer greater than or equal to x and the largest integer less than or equal to x. Finally $\binom{n}{m}$ denotes the number of ways of choosing m distinct objects from a set of n objects. It is known [74] that

$$\binom{n}{m} = \frac{n!}{(n-m)!\,m!}.$$

Moreover, when m is fixed as n increases, this means that

$$\binom{n}{m} \sim \frac{n^m}{m!}$$

where ~ means approximately equal.

2.2 MATHEMATICAL BACKGROUND

Several mathematical foundations are required throughout this dissertation. These include the concepts of sets, mappings and graphs.

2.2.1 Sets

A set is a collection of objects. We use $a \in A$ to denote the fact that an object a is a member of the set A, and $A \subseteq B$ to denote that the set A is contained in the set B, that is, every element of A is also an element of B. The set theoretic operations of *union*, *intersection*, and *Cartesian product* are defined respectively for sets A and B as

$$A \cup B = \{c \mid c \in A \text{ or } c \in B\}$$

$$A \cap B = \{c \mid c \in A \text{ and } c \in B\}$$

$$A \times B = \{\langle a,b\rangle \mid a \in A \text{ and } b \in B\} = \langle A,B\rangle$$

where the element $\langle a,b\rangle$ is called an ordered pair or tuple. All of these operations extend naturally to any finite number of sets.

2.2.2 Mappings And Inverse Mappings

In addition to sets, we work extensively with mappings. A *mapping* or *relation* R from a set A into a set B is a subset of $A \times B$ where we denote the fact that $\langle a,b\rangle \in R$ by $R:a \to b$ or by $R(a)=b$. Since mappings are an integral part of our work, we need a number of related terms. Let $R \subseteq A \times B$ be a mapping. The *domain* of R, denoted dom(R), is the set $\{a \in A \mid \langle a,b\rangle \in R \text{ for some } b \in B\}$, and the *range* of R, denoted range(R), is the set $\{b \in B \mid \langle a,b\rangle \in R \text{ for some } a \in A\}$. R is said to be defined over a set $C \subseteq A$ if $C \subseteq \mathrm{dom}(A)$. Finally if $R \subseteq A \times B$ and $S \subseteq B \times C$ are relations, then the *composition* of R and S is the relation

$$S(R)=\{\langle a,c\rangle \mid \text{there is a } b \in B \text{ with } R(a)=b \text{ and } S(b)=c\}.$$

There are several properties that a mapping $R \subseteq A \times A$, from the set A into itself, can exhibit. It is called *reflexive* if $<a,a> \in R$ for all $a \in A$; is called *transitive* if whenever $<a,b> \in R$ and $<b,c> \in R$, then $<a,c> \in R$; and is called *symmetric* if $<a,b> \in R$ implies that $<b,a> \in R$. A mapping that is reflexive, transitive and symmetric is called an *equivalence relation*.

Mappings can also be classified by how they map their domain into their range. A mapping R is called *many-one* if b=c whenever $R:a \rightarrow b$ and $R:a \rightarrow c$, and is called *one-many* if a=b whenever $R:a \rightarrow c$ and $R:b \rightarrow c$. A mapping that is both many-one and one-many is called *one-one* and a mapping that is neither is called *many-many*. Note that what is generally called a mapping, we call a many-one mapping or function.

Throughout this dissertation we consider the problem of inverting such mappings. Let $R \subseteq A \times B$ be a mapping. We define $S \subseteq B \times A$ as an *inverse* of R if S is defined over the range of R and if $S:b \rightarrow a$ implies that $R:a \rightarrow b$. This is a slightly weaker definition than is usual for the inverse of a relation [11] as we do not require that $S:b \rightarrow a$ if and only if $R:a \rightarrow b$. However it is precisely what is required for the problem of automatic programming because our interest lies in finding any element c such that $R:c \rightarrow b$.

2.2.3 Graphs

It is often convenient to formalize objects in terms of graphs. A *directed graph* G is an ordered pair $<V,E>$ where V is the set of vertices

or nodes and $E \subseteq V \times V$ is the set of directed edges. A labeling of a graph is a function f from the set of nodes into some set L. For a given node a, we call $f(a)=\ell$ the *label* of a. In this dissertation we use the term graph to mean a labeled directed graph. If $e=<a,b>$ is an edge from vertex a to vertex b in some graph, then e is said to *leave* a and *enter* b. We define the *indegree* of a node to be the number of edges entering it and the *outdegree* of a node to be the number of edges leaving it. Finally, a graph $G'=<V',E'>$ is called a *subgraph* of a graph $G=<V,E>$ if G′ is a graph and $V' \subseteq V$, $E' \subseteq E$.

We require several definitions concerning a restricted class of graphs, those without cycles. Let $G=<V,E>$ be a graph. A *path* of length $n \geq 1$ is a sequence of nodes $(a_0,a_1,\ldots,a_n)$ such that $<a_i,a_{i+1}> \in E$. If $a_0=a_n$, the path is called a *cycle*. A graph that contains no cycles is called a *directed acyclic graph* or *dag*. We are particularly interested in a restricted type of dag called a tree. This is a dag with a distinguished node called the root such that there is a unique path from the root to every other node.

2.2.4 Vector Addition Systems

In addition to graphs, we need the formalism of a vector addition system. An n-dimensional *vector addition system* (VAS) [69] is a pair $V=<S,d>$ where S is a finite set of integral n-vectors and d is a single integral n-vector. The principal question involving these is that of reachability. An n-vector a is said to be *reachable by* V if there is a

sequence of vectors from S, $s_1, s_2, \ldots, s_n$ and a corresponding sequence of sums $T_0=d$, $T_i=T_{i-1}+s_i$, such that $T_n=a$ and every component of every T_i is greater than or equal to zero. Testing for reachability is known to be intractable [30].

2.2.5 Infix And Prefix

Throughout this dissertation use examples that involve code generation for arithmetic expressions. An expression that is written in the ordinary way with parentheses and with each operator between its operands is said to be in infix notation. We often use a more concise way of presenting the same expression called prefix notation where there are no parentheses and where each operator is written immediately before its operands. In particular, if E is an infix expression then the corresponding <u>prefix expression</u> E' is defined recursively as

a) if E is a single operand then $E'=E$;

b) if $E=E_1\theta E_2$ for expressions E_1 and E_2 and operator θ, then $E'=\theta E_1'E_2'$ where E_1' and E_2' are the prefix forms of E_1 and E_2; and

c) if $E=(E_1)$ for an expression E_1, then $E'=E_1'$ where E_1' is the prefix form of E_1.

2.3 STRINGS, LANGUAGES AND GRAMMARS

Many of our results regarding practical automatic programming are based on formal language theory. In particular we make frequent use of

concerning the hierarchy of languages and the various methods of parsing.

2.3.1 Strings

A <u>string</u> is a sequence of elements from a set of symbols which we call an alphabet. For any string α we denote the ith element of α as α_i and the number of elements or length of α as $|\alpha|$. The string containing no symbols is called the null string and is represented by λ. If α and β are strings over some alphabet then we define the <u>concatenation</u> of α and β, $\alpha\beta$, as the string constructed by appending the string β to the string α. We represent repeated concatenations of a single string with superscripts. Thus for any string α, $\alpha^0=\lambda$ and for $i\geq 1$, $\alpha^i=\alpha\alpha^{i-1}$.

2.3.2 Languages

A set of strings all of whose symbols come from an alphabet Σ is called a <u>language over</u> Σ. This is a very broad definition that includes natural languages such as English as well as programming languages such as FORTRAN. We use languages as a way of describing the sets of input and output strings of an automatic programming problem, and hence we are primarily interested in methods of representing large languages in a concise manner. We consider two such methods of representation, expressions and grammars.

While a finite language can be easily represented by listing all of its strings, an infinite language cannot be. We therefore introduce three operations on languages that allow a finite representation of an infinite language. For arbitrary languages A and B we define

1. the <u>concatenation</u> of A and B, $AB=\{\alpha\beta \mid \alpha\in A,\ \beta\in B\}$;
2. the <u>union</u> of A and B, $A\cup B=\{\gamma \mid \gamma\in A \text{ or } \gamma\in B\}$; and
3. the <u>closure</u> of A, $A^{*}=\{\alpha^{i} \mid i\geq 0 \text{ and } \alpha\in A\}$.

Using these operators we can build up expressions that define complex languages in terms of simple ones. Languages that can be constructed from finite languages by means of these expressions are called regular and are said to be defined by regular expressions [11]. For example,

$$\{0\}\cup\{1\}(\{0\}\cup\{1\})^{*}$$

is a regular expression denoting the set of all binary integers. We often denote the language containing a one element string by that element. Thus an alternative for this example would be

$$0\cup 1(0\cup 1)^{*}$$

2.3.3 Grammars

Such expressions cannot describe all of the languages we are interested in. To do this we need the alternative representation of a grammar which provides a formal means for generating all the strings of a language. Formally, we define a <u>grammar</u> as a 4-tuple $G=\langle N,\Sigma,P,S\rangle$ where

1. N is a finite set of nonterminal symbols;
2. Σ is a finite set of terminal symbols disjoint from N;
3. P is a finite subset of $(N\cup\Sigma)^{*}N(N\cup\Sigma)^{*} \times (N\cup\Sigma)^{*}$ called the productions of G; and
4. $S\in N$ is a distinguished symbol called the start symbol.

We denote a production $\langle\alpha,\beta\rangle$ of P by $\alpha\to\beta$. As an example of a grammar consider

$$G_0 = \langle\{A,S\},\{0,1\},P,S\rangle$$

where P contains the rules

$S\to 0A1$
$A\to 0A1$
$A\to\lambda$

A grammar describes a language by being able to generate it. Let $G=\langle N,\Sigma,P,S\rangle$ be a grammar. Then we define the set of <u>sentential forms</u> of G recursively as

1. S is a sentential form;
2. For strings $\alpha,\beta,\gamma,\delta$ in $(N\cup\Sigma)^{*}$, if $\alpha\beta\gamma$ is a sentential form and $\beta\to\delta$ is in P, then $\alpha\delta\gamma$ is also a sentential form.

The language generated by the grammar G, denoted L[G], is then the set of all sentential forms of G containing only terminal symbols. We can represent the action of 2) above by the relation <u>directly derives</u> denoted $\underset{G}{\Longrightarrow}$. Thus for some production $\beta\to\gamma$ in P, we write $\alpha\beta\gamma\underset{G}{\Longrightarrow}\alpha\delta\gamma$. If G is understood here we omit it, using only $\Longrightarrow$. We denote a chain of one or more such operations by $\overset{+}{\underset{G}{\Longrightarrow}}$. Thus $\alpha\overset{+}{\underset{G}{\Longrightarrow}}\beta$ if and only if there are strings $\gamma_1,\ldots,\gamma_n$, $n\geq 1$, such that

$$\alpha \underset{G}{\Longrightarrow} \gamma_1 \underset{G}{\Longrightarrow} \cdots \underset{G}{\Longrightarrow} \gamma_n = \beta.$$

Finally, we say $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ if either $\alpha=\beta$ or $\alpha \overset{+}{\underset{G}{\Longrightarrow}} \beta$. A <u>derivation</u> of a string α from a grammar G is the series of strings $\alpha_1,\ldots,\alpha_n$ such that $\alpha_1=S$, $\alpha_n=\alpha$ and $\alpha_i \underset{G}{\Longrightarrow} \alpha_{i+1}$, $1\le i<n$. Some sample derivations of the grammar G_0 would then include

```
S ==> 0A1 ==> 01
S ==> 0A1 ==> 00A11 ==> 0011
S ==> 0A1 ==> 00A11 ==> 000A111 ==> 000111 .
```

Thus the set of sentential forms of G_0 includes S, 0A1, 01, 00A11, 0011, ... Moreover, the language generated by the grammar is easily seen to be $\{0^n 1^n \mid n\ge 1\}$.

One common operation that is performed with grammars is that of parsing. For a grammar G and a string α this is the problem of determining a derivation of G that yields α. We call a description of a derivation for α a <u>parse</u> of α and later show that trees are a convenient representation for parses. If ρ is a parse, then <u>applying</u> ρ to G is the operation of using the derivation described by ρ with G. We call a parse <u>top-down</u> if it describes a derivation by begining with the start symbol and ending with the actual string, and we call a parse <u>bottom-up</u> if it describes a derivation by begining with the string and ending with the start symbol. Finally, we make a subtle distinction between a parse and a derivation. For most of the grammars we consider, a given derivation can be changed into a second one for the same string by just modifying the order in which productions are applied. When this is the case we say the two derivations are <u>equivalent</u> and use a single parse to

describe them. Thus a parse can describe a large number of distinct derivations.

There are several properties of derivations and grammars that are important to us. A derivation is called <u>leftmost</u> if whenever $\alpha\beta\gamma \Longrightarrow \alpha\delta\gamma$ by a production $\beta\to\delta$, then $\alpha\in\Sigma^*$, i.e. the next production is applied at the leftmost possible point in the sentential form. We then call a parse <u>leftmost</u> or <u>left-to-right</u> if it can describe a leftmost derivation. Note that a leftmost derivation or parse can be uniquely described with only a sequence of productions. Next we say that a grammar contains a <u>cycle</u> if there is a nonterminal A such that $A \overset{+}{\Longrightarrow} A$. A grammar which contains no cycles is called <u>cycle-free</u>. Finally a grammar is called <u>ambiguous</u> if there is at least one string α in L[G] that has two distinct parses.

2.3.4 The Hierarchies Of Languages And Mappings

One of the principal contributions of formal language theory has been the definition by Chomsky [31] of a hierarchy of languages based on restricted types of grammars. Let $G=\langle N,\Sigma,P,S\rangle$ be a grammar. Then G is called <u>regular</u> if each production in P has the form $A\to\alpha B$ or $A\to\alpha$ where A and B are in N and $\alpha\in\Sigma^*$; G is called <u>context free</u> if each production in P has the form $A\to\beta$ for $A\in N$ and $\beta\in(N\cup\Sigma)^*$; G is called <u>context sensitive</u> if each production in P has the form $\alpha\to\beta$ for $\alpha,\beta\in(N\cup\Sigma)^*$ and $|\alpha|\le|\beta|$; and otherwise G is called <u>unrestricted</u>. The class of languages that can be generated with regular grammars is the class of regular languages which can also be described by regular expressions. The class of lan-

guages that can be generated by context free (context sensitive) grammars is called the context free (context sensitive) languages. Finally, the class of languages that can be generated by unrestricted grammars is called the recursively enumerable languages. In addition to these types of languages based directly on grammars, we say that a language L over and alphabet Σ is recursive if L is recursively enumerable and if its complement language,

$$\bar{L} = \{\alpha\in\Sigma^* \mid \alpha\notin L\},$$

is also recursively enumerable. It has been shown that the recursive languages include all context sensitive languages and form a proper subset of the recursively enumerable languages.

Given this hierarchy of languages, we can define an equivalent hierarchy of mappings. Let Σ and Δ be alphabets. We define a homomorphism from Σ into Δ as a mapping $h:\Sigma^*\rightarrow\Delta^*$ such that $h(\alpha\beta)=h(\alpha)h(\beta)$ for all strings α and β. Then we say a mapping R is characterized by a language L if there are homomorphisms h_1 and h_2 such that $R=\{<h_1(\alpha),h_2(\alpha)> \mid \alpha\in L\}$. R is said to be a regular (context free, context sensitive, recursive, recursively enumerable) mapping if and only if it can be characterized by a regular (context free, context sensitive, recursive, recursively enumerable) language. We often call a regular mapping a gsm mapping [61].

2.3.5 Turing Machines

Another way of defining the same hierarchy is through a hierarchy of restricted machines. The most common such machine is called a Turing

machine [61] and is defined as the 5-tuple $M=<Q,\Sigma,\Gamma,\delta,q_0>$ where

1. Q is a finite set of states;
2. $\Sigma\subseteq\Gamma$ is the input alphabet;
3. Γ is the tape or output alphabet;
4. $\delta \subseteq Q\times\Gamma\rightarrow\Gamma\times\{-1,0,1\}\times Q$ is the transition function; and
5. $q_0\in Q$ is the initial state.

Intuitively M consists of an infinite tape, a finite control, and a single tape head. The tape consists of a one-way infinite sequence of tape squares each of which can contain a single element of Γ. This tape can be read or written one square at a time by positioning the tape head over the square to be accessed. M then operates by making a series of moves or transitions based on a finite state control composed of the set of states and the transition function. Initially the tape contains a finite input string α followed by an infinite string of blanks, the tape head is positioned over the first character of α, and M is said to be in state q_0. Each move of M is determined from δ based on the current state and the contents of the tape cell under the tape head. If M is in state q and is reading the symbol a, and if $<q,a,b,n,q'>\in\delta$, then M can move by writing b on the current tape square, moving the tape head one square left if n=-1 or one square right if n=1, and going into state q'. If there are several possible transitions for a given state and tape symbol, we assume M chooses the "right" one and call M nondeterministic. Finally, if M enters a situation where it cannot make a move, then it is said to halt, and we say that the string β consisting of the tape con-

said to halt, and we say that the string β consisting of the tape contents up to the first blank is the output of M given α. We say that M maps α into β if there is a sequence of transitions of M such that given α initially, M halts with output β. It is known [61] that the class of mappings defined by Turing machines is precisely the class of recursive mappings. Moreover, it is also known [61] that any mapping that can be defined by a "reasonable" programming language can also be defined by a Turing machine.

One restricted form of Turing machine is also of interest to us. This is the class of <u>linear</u> <u>bounded</u> <u>automata</u> (LBA) where the tape is finite and contains exactly one cell for each character of the input string. It has been shown [61] that the class of languages definable by nondeterministic linear bounded automata is the class of context sensitive languages.

2.3.6 Parsing Context Free Grammars

Most of the work concerning grammars and parsing has involved context free grammars. These can be used to describe most useful languages including the underlying structure of most programming languages. Moreover, they exhibit several convenient properties [11]. Every parse of a context free grammar is leftmost, i.e. can be thought of as describing a leftmost derivation. Moreover, every context free grammar that contains cycles is effectively equivalent to another context free grammar that is cycle-free. Finally, context free grammars can be parsed relatively quickly.

Several algorithms have been proposed for parsing an arbitrary context free grammar and most of these construct a data structure that embodies all possible parses of the given string so that either the existence of a parse or any particular parse can easily be found. We call such a parsing algorithm <u>universal</u>. The most popular universal algorithm was originally proposed by Earley [38]. The basic element of his algorithm is an item which describes a potential parse starting at a specific point of the given string and using a specific production. It consists of a production with a marker called a dotted rule and an integer. We represent an item as

$$[A \to \alpha \bullet \beta, n].$$

where $A \to \alpha\beta$ is a production, $\bullet$ is the marker, and n is the integer denoting where in the source string the potential parse begins. The algorithm operates by constructing a list of all items that can possibly be used in a parse of each initial substring of the input string. It has been simply stated by Ullman [128] as

EARLEY'S ALGORITHM for Parsing a Context Free Grammar:

Given: A context free grammar $G = \langle N, \Sigma, P, S \rangle$ and a string $s = a_1 \cdots a_n$.

Construct: Item lists $\Xi_0, \ldots, \Xi_n$.

1. Construct Ξ_0 as follows:
   a) Add $[S \to \bullet \alpha, 0]$ to Ξ_0 for each $S \to \alpha$ in P.
   b) Apply the following closure operation with j=0. The closure rules for Ξ_j are
      i) If $[A \to \alpha \bullet B\beta, i]$ is on Ξ_j and $B \to \gamma$ is a production, add $[B \to \bullet \gamma, j]$ to Ξ_j;

ii) if $[A \to \alpha \cdot, i]$ is on Ξ_j then for all items on Ξ_i of the form $[B \to \beta \cdot A\gamma, k]$, add $[B \to \beta A \cdot \gamma, k]$ to Ξ_j.

2. Suppose Ξ_{j-1} has been constructed. Construct Ξ_j as follows:
   a) For all $[A \to \alpha \cdot a_j \beta, i]$ on Ξ_{j-1} add $[A \to \alpha a_j \cdot \beta, i]$ to Ξ_j;
   b) Apply the above closure rules to Ξ_j.

□

It has been shown [128] that

FACT: Item $[A \to \alpha \cdot \beta, i]$ is placed on Ξ_j if and only if there is a derivation $S \overset{*}{\Longrightarrow} a_1 \ldots a_i A\gamma \Longrightarrow a_1 \ldots a_i \alpha\beta\gamma \overset{*}{\Longrightarrow} a_1 \ldots a_j \beta\gamma$.

Thus, there is an item $[S \to \alpha \cdot, 0]$ in Ξ_n if and only if s is in L[G].

As an example of this algorithm consider the simple grammar G_0 on page 16 and the string s=0011. The corresponding item lists are

Ξ_0: $[S \to \cdot 0A1, 0]$

Ξ_1: $[S \to 0 \cdot A1, 0]$

$[A \to \cdot 0A1, 1]$
$[A \to \cdot, 1]$
$[S \to 0A \cdot 1, 0]$

Ξ_2: $[A \to 0 \cdot A1, 1]$

$[A \to \cdot 0A1, 2]$
$[A \to \cdot, 2]$
$[A \to 0A \cdot 1, 1]$

Ξ_3: $[A \to 0A1 \cdot, 1]$

$[S \to 0A \cdot 1, 0]$

Ξ_4: $[S \to 0A1 \cdot, 0]$

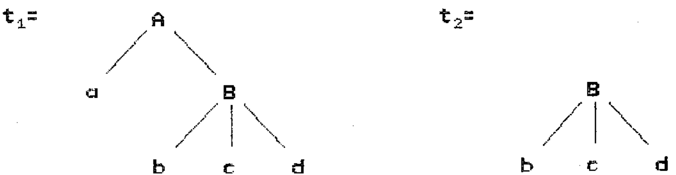
and the resultant derivation is

$$S \Longrightarrow 0A1 \Longrightarrow 00A11 \Longrightarrow 0011.$$

2.4 TREES

A <u>tree</u> is defined as a labeled directed acyclic graph with a distinguished node called the root such that there is a unique path from this node to every other node. It is easy to see that this definition implies that the root has indegree zero and that all other nodes have indegree one [11]. We call a node of a tree with outdegree zero a leaf or terminal node of the tree, and a node with positive outdegree an interior node. A <u>subtree</u> $T' = \langle V', E' \rangle$ of a tree $T = \langle V, E \rangle$ is a subgraph of T such that there is no edge in the tree T leaving a node of V′ and entering a node of V not in V′. A subtree is clearly a tree in itself and every node of a tree is the root of an unique subtree of the tree [11]. If an edge leaves node A and enters node B, then A is called the father of B and B is called the son of A. We often use the term son to refer to the unique subtree that is rooted at the corresponding node. Finally we say that a tree has k <u>levels</u> or is of <u>depth</u> k if the longest path from the root to a leaf has length k+1.

In addition to labeling our trees, we also order them. A tree is ordered if the sons of each node appear in a fixed order. We draw an ordered tree with the root on top and with all arcs going down. Each node is represented by its label and the sons of a node are ordered by their appearance from left to right. Two simple trees are illustrated below:



Here the root of t_1 is labeled A, and t_1 has two interior nodes, A and B, and four leaves, a,b,c and d. The sons of A are in order the nodes labeled a and B. Finally the tree t_2 is the subtree of the tree t_1 rooted at the node B.

2.4.1 Ranked Alphabets

Most of the trees we consider have labels drawn from a fixed finite alphabet in a limited way. For example, we might want A and B to only be labels of interior nodes and a,b,c and d to only be the labels of leaves. To accomplish this we distinguish the subsets of symbols that can be used to describe a node of the tree based on the outdegree of that node. Formally we say that an alphabet Σ is <u>ranked</u> [41] if there is a finite relation $R \subseteq \Sigma \times N$, where N is the set of natural numbers, such

that $\sigma \in \Sigma$ can be used to label a node with outdegree k if and only if $<\sigma,k> \in R$. We denote the set of all possible labels of nodes with outdegree k as

$$\Sigma_k = \{\sigma \in \Sigma \mid <\sigma,k> \in R\}.$$

A ranked alphabet is generally specified by a finite collection of subsets of this form, and a tree is said to be defined over such a ranked alphabet if its labels are restricted in this manner. Finally we denote the set of all trees that are defined over the ranked alphabet Σ by T_Σ.

2.4.2 Tree Expressions

While a ranked alphabet allows us to define some limited sets of trees, we often need more specific sets. For this purpose we introduce tree expressions [41] which are a way of building up sets of trees in a limited way. Let Σ be a ranked alphabet and N a set of symbols. Then we can formally define tree expressions and the sets of trees they denote recursively as

1. N is a tree expression denoting the set of single node trees labeled by an element of N;
2. T_Σ is a tree expression denoting the set of all trees defined over Σ;
3. if E and E′ are tree expressions denoting sets of trees T and T′ respectively, then
   a) E^k is a tree expression denoting the set of all trees in T that have k levels.
   b) N(E) is a tree expression denoting the set of all trees whose root is labeled with an element of N, has outdegree one, and whose only son is a tree from T;

c) $E \cup E'$ is a tree expression denoting $T \cup T'$;

d) $E[E']$ is a tree expression denoting the set of all trees that can be formed by taking a tree in T and replacing zero or more of its leaves with trees in T'; and

4. nothing else is a tree expression.

For example, if Σ and Δ are ranked alphabets and Q is a set of symbols, then $T^2_\Delta[Q(T_\Sigma)]$ is the set of all trees where the root is in Δ and where the sons of the root are either elements of Δ_0 (since it is a 2-level tree) or are trees whose root is in Q and whose only son is a tree over Σ.

2.4.3 Linear Tree Representation

Since it will often not be convenient to draw a tree every time we need to describe one, we introduce a linear or string representation for trees. We represent a leaf by its label and an interior node by a string composed of its label, the metasymbol '(', the representation of each of its subtrees in order, and the metasymbol ')'. A tree is then described by the representation of its root. Thus the tree t_1 presented earlier is represented as A(aB(bcd)) and the tree t_2 as B(bcd).

2.4.4 The Frontier Function

In addition to describing trees and sets of trees, we need to describe the string represented by a tree. We call the string composed of the labels of the leaf nodes of a tree in the left-to-right order in

which they appear the frontier of the tree. Formally, we define the <u>frontier function</u> of a tree t, fr(t), recursively as

1. if t is a single leaf labeled a, then fr(t)=a; and

2. if $t=A(t_1 t_2 \ldots t_k)$, a tree with root A and k sons, $t_1 \ldots t_k$, then $fr(t)= fr(t_1) \ldots fr(t_k)$.

Thus the frontier of the tree t_1 is the string 'abcd' and that of t_2 is 'bcd'.

If $t \in T_\Delta$ where Δ is the ranked alphabet constructed from a ranked alphabet Σ and a set of states Q such that

$\Delta_i = \Sigma_i$ for $i \neq 1$, $\Delta_1 = \Sigma_1 \cup Q$

then we recursively define the <u>state-frontier</u> of t, denoted $fr_Q(t)$, as

1. $fr_Q(a)=a$ if $a \in \Sigma_0$;

2. $fr_Q(q(t_1))=q$ if $q \in Q$; and

3. $fr_Q(A(t_1 \ldots t_k))=fr_Q(t_1) \ldots fr_Q(t_k)$ if $A \in \Sigma_k$.

Intuitively this is the frontier of the tree if we consider the states or elements of Q as terminal symbols and discard the portion of the tree below them.

2.4.5 Parse Trees

One of our most common uses for trees is to represent a parse of a context free grammar. Let $G=\langle N,\Sigma,P,S\rangle$ be a context free grammar. Then a <u>parse</u> or <u>derivation tree</u> for G is a tree where each leaf is labeled by a symbol from $\Sigma\cup\{\lambda\}$, where each interior node is labeled by a symbol from N, where the root is labeled with S, and where if an interior node is labeled A and has sons that are labeled in order $X_1,\ldots,X_n$, then the production $A\rightarrow X_1\ldots X_n$ is in P. A tree t is called a parse tree for a string s using G if t is a parse tree for G and fr(t)=s. It should be obvious how such a tree describes a number of equivalent derivations or a parse of the string s. Moveover, it is important to note the taking the frontier is an inverse operation to finding a parse.

As an example of a parse tree, consider the context free grammar

$G=\langle\{A,B\},\{a,+,*\},P,A\rangle$

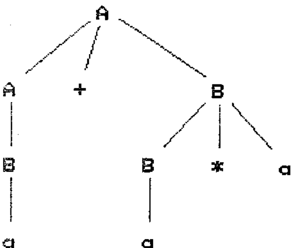
where P contains the rules

A→A+B
A→B
B→B*a
B→a

A simple derivation for the string 'a+a*a' would be

A ==> A+B ==> B+B ==> a+B ==> a+B*a ==> a+a*a

which would be represented by the parse tree

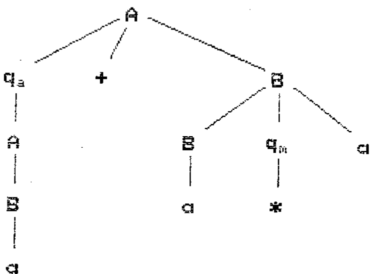


or equivalently the string A(A(B(a))+B(B(a)*a)).

2.4.6 Tree Rewriting Systems

Many of the semantic mappings that we use throughout this dissertation are based on parse trees. They start with a parse tree for a given string and transform it into a new or target tree whose frontier is the desired result. The specific types of mappings are defined throughout chapters 4, 5 and 6 and are all based on the framework of tree rewriting systems.

A <u>tree rewriting system</u> [41] is the equivalent of a grammar for trees. Formally it is a 5-tuple $\langle\Sigma,\Delta,Q,q,R\rangle$ where Σ and Δ are ranked alphabets describing the set of input and output trees respectively, Q is a finite set of states, $q\in Q$ is a distinguished state, and R is a finite set of rules. Each rule consists of two finite trees, one that must match a subtree of the original tree and one that describes how to modify the subtree that is matched. The finite set of states is used to control the mapping. A state $q_a\in Q$ is <u>associated</u> with a subtree t′ in a tree t if t′ is replaced with the tree $q_a(t')$ in t. Thus in



the subtree A(B(a)) is associated with the state q_a and the subtree * is associated with the state q_m.

A rule of a tree rewriting system consists of a source pattern and a replacement pattern. The source pattern is an arbitrary (finite) tree, but for our uses will generally involve at least one state, elements of Σ, and possibly tree variables. A tree variable is simply a variable that can represent a single arbitrary tree throughout the application of a rule and will be selected from the set $X=\{x_1,x_2,x_3,\ldots\}$. A source pattern is said to match a subtree if that pattern can be placed on top of the subtree so that it matches exactly except for tree variables, and so that each tree variable is associated with a unique tree. Formally, let r be a rule with source pattern p and let t be a subtree. We say that p <u>matches</u> t or that r is <u>applicable</u> to t if and only if

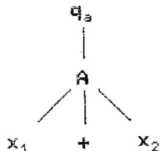
a) p consists of a single tree variable x_i, x_i is defined, and $x_i=t$;

b) p consists of a single tree variable x_i and x_i is undefined, in which case x_i is defined to be t;

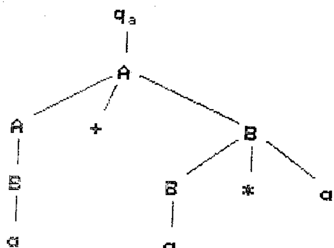
c) p consists of a single node which is not a tree variable and p=t; or

d) p consists of a tree with root A and k sons $p_1 \cdots p_k$ and t is also a tree with root A and with k sons $t_1 \cdots t_k$ such that p_i matches t_i for $1 \le i \le k$.

Thus the source pattern $q_a(A(x_1+x_2))$



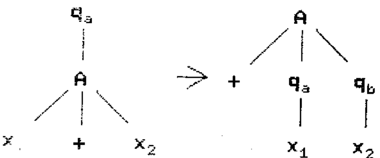
would match the tree $q_a(A(A(B(a))+B(B(a)*a)))$



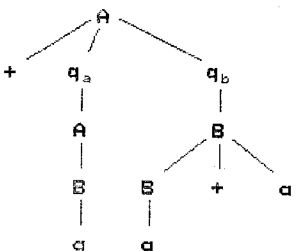
with the assignments $x_1=A(B(a))$ and $x_2=B(B(a)*a)$.

Once a rule is applicable to a given subtree, the rule can be <u>applied</u>. This consists of replacing the matched subtree with a new subtree that is based on the right part of the rule. This tree is created in the obvious way, taking the replacement pattern and substituting

for each tree variable the unique tree that was associated with that variable in the matching process. Thus for the match given above, if the complete rule was $q_a(A(x_1+x_2)) \rightarrow A(+q_a(x_1)q_b(x_2))$



then the result would be $A(+q_a(A(B(a)))q_b(B(B(a)+a)))$.



If a tree t can be transformed into a tree t′ by a rule of a tree rewriting system T, we say t <u>derives</u> t′ by T and denote this $t \underset{T}{\Longrightarrow} t'$. The notation $t \overset{*}{\underset{T}{\Longrightarrow}} t'$ ($t \overset{+}{\underset{T}{\Longrightarrow}} t'$) then means that t′ is derivable from t using zero or more (one or more) applications of rules of T. If there is no confusion we omit the T and just write $t \Longrightarrow t'$.

Finally, a tree rewriting system can be classified by how it uses the distinguished state q to start and end a transformation. If it uses q as an initial state, assigns it to the root of the source tree before any rules are applied, and then halts when no states remain in the tree, it is called <u>top-down</u>. It is called <u>bottom-up</u>, on the other hand, if it uses q as a final state and halts when q is assigned to the root of a tree containing no other states.

2.5 COMPLEXITY THEORY

Much of this dissertation is concerned with the complexity of inverse translation and its applications to automatic programming and we therefore require several concepts from the theory of computational complexity. We consider such complexity in reference to the realistic abstract model of a random access machine (RAM) where we have a potentially infinite number of registers (memory) that can store integer values up to some arbitrarily large fixed value and a finite control or program that can access any of these registers and perform simple operations--i.e. arithmetics, comparisons, loads and stores--with them. Such a model, described formally and in more detail in [4], closely approximates most current computers and hence allows our complexity results to be thought of in terms of real machines.

Given such a model of computation, we can consider the complexity of a given problem. We say that a problem is solvable in time t(n) if, for all inputs of length n, the number of primitive operations or RAM instructions needed to compute the solution is bounded by the function t(n). Similarly we say that a problem is solvable in space t(n) if for all inputs of length n, at most t(n) registers of the RAM are required for temporary storage in computing a solution. We introduce the approximate notion of order [74] and say that a problem has time (space)

complexity on the order of t(n), denoted O(t(n)), if there is a constant c, independent of n, such that the problem is always solvable in time (space) ct(n). Finally we say that a problem is unsolvable or (recursively) undecidable if there is no algorithm that can solve it.

One important feature of algorithms and machines is nondeterminism. We have already discussed nondeterministic Turing machines and can similarly consider nondeterministic algorithms where there can be several alternative next steps and the program can choose between them. We say that a nondeterministic algorithm halts successfully if there is some valid sequence of computations that would cause it to halt successfully. This makes nondeterminism seem quite powerful since a nondeterministic algorithm can guess the proper solution and then just has to check if it is correct. We say that a problem is solvable in nondeterministic time (space) t(n) if there is a nondeterministic algorithm that can solve it and which requires at most time (space) t(n).

We define a complexity class as the set of problems that are solvable in deterministic or nondeterministic time t(n) for some set of functions t(n). For example the class of polynomial time problems, denoted PTIME, is the set of all problems that can be solved in deterministic time t(n) for some polynomial t. Similarly we define the class NPTIME or just NP as the set of all problems solvable in nondeterministic polynomial time, PSPACE as the set of all problems solvable in nondeterministic polynomial space, and EXPTIME as the set of problems solvable in deterministic exponential time. It has been established [4] that

$$\text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

and that

$$\text{PTIME} \neq \text{EXPTIME}.$$

Finally we say that a problem is hard for a class C, denoted C-hard, if it is at least as difficult as any problem in C, and that a problem is C-complete if it is in the class C and is C-hard.

2.5.1 Reducibilities And Complete Problems

One way of showing that a problem is hard or complete for a class and of comparing the complexity of two problems in general is by means of reducibilities [67]. A problem A is polynomial reducible or just reducible to a problem B if an instance of problem A can be transformed in deterministic polynomial time into an instance of problem B such that any solution found for this new problem can be translated into a solution for problem A. It is clear that if a problem B is in any of the complexity classes mentioned above and if a problem A is reducible to B, then A is also in the class.

We can use this notion of reducibility to prove that a problem in some complexity class is complete for that class by showing that every other problem in the class is reducible to it. Once we have one complete problem however, we need only show that it is reducible to some second problem in the class to show that this new problem is also complete. This latter technique is the one we use in this dissertation. We note that the following problems are known [67] to be NP-complete:

3-SATISFIABILITY

> Given disjunctive clauses $D_1, D_2, \ldots, D_r$ each consisting of at most 3 literals of the set $\{u_1, u_2, \ldots u_m\} \cup \{\bar{u}_1, \bar{u}_2, \ldots \bar{u}_m\}$, determine if there is a truth assignment to $u_1 \ldots u_m$ such that the clauses are simultaneously satisfied.

KNAPSACK

> Given a set of values $a_1, a_2, \ldots a_r$ and an objective b, all integers, determine if there is some subset of the values that sums to b, i.e. $(\Sigma a_j x_j = b)$ has a 0-1 solution.

It is also known [4] that the problem of determining if a given string is in an arbitrary context sensitive language is PSPACE-complete. Moreover it has been shown [55] that there is a hardest context sensitive language, that is, a language L such that for any context sensitive language M there is a function f computable in linear time such that $x \in M$ if and only if $f(x) \in L$. Using these two facts we note that the problem

CONTEXT SENSITIVE RECOGNITION

> Given a string s, determine if s is in a fixed context sensitive language.

is PSPACE-complete.

2.5.2 The Complexity Of Context Free Parsing

In addition to proving problems NP-complete or PSPACE-complete and hence showing them intractable, we consider the more practical problem of parsing a context free grammar. Earley's algorithm discussed earlier requires time $O(n^2)$ if the grammar is unambiguous and time $O(n^3)$ in the general case, and is perhaps the best practical algorithm. We denote by $T_{parsing}(n)$ the time required to find a single parse of an arbitrary context free grammar for an arbitrary string of length n, and we denote by $T'_{parsing}(n)$ the time required to find all parses. For a given grammar and string of length n there can be an infinite number of parses if the grammar contains cycles and an exponential number if not [11]. We do not require that all parses be output here, but rather that a data structure be created that represents all parses. An example of such a structure is the set of item lists produced by Earley's algorithm. For both of these complexities, if n is implicit we will omit it and just write $T_{parsing}$ or $T'_{parsing}$. It is then known that

$$O(n^2) \leq T'_{parsing} \leq O(n^{2.81})$$

by the result of Valiant [130] and the fact that it can take $O(n^2)$ storage to describe all parses of a string. Little is known about $T_{parsing}$ other than the trivial fact that

$$O(n) \leq T_{parsing} \leq T'_{parsing}.$$

CHAPTER 3

A MODEL FOR AUTOMATIC PROGRAMMING

Before we can consider the complexity of automatic programming and before we can describe what practical automatic programming is and is not, we must provide a formal description of the problem. The purpose of this chapter is to develop a mathematical model of automatic programming. We first consider several instances of the problem and deduce their common elements. From these elements we construct a concise model that correctly reflects automatic programming. We next illustrate this model by showing how it fits a specific instance of the problem. Finally, we consider how this model reflects the complexity of automatic programming.

3.1 AUTOMATIC PROGRAMMING

Automatic programming is a label given to a variety of projects whose principal purpose is to automate the use of computers. These projects range from machine independent compilers to the use of everyday English as a programming language; from programming by examples to programming in the first order predicate calculus. In this section we

take a close look at what various people call automatic programming and deduce from it the essential elements of the problem.

There have been several efforts aimed at allowing the user to specify a problem in English and have the computer do the rest. The earliest work in this area involves Heidorn's efforts in generating GPSS simulation programs from an English dialogue description [56]. His system uses a semantic net both to represent the problem and to serve as a basis for generating the GPSS program. A more recent effort is that of Balzer [16]. He views automatic programming as a four step task: first understanding the problem through a natural language dialogue; then finding a high level process or algorithm for solving the problem; next verifying that this algorithm is the desired one; and finally coding the algorithm in some computer language. Another major effort in this area is being undertaken at MIT [86,108]. This involves the generation of business data processing systems based on a natural language dialogue. Once the problem is understood here it is translated into MAPL, a relational modeling language, and a PL/1 program is constructed from this MAPL description. A final effort at programming with natural language is the work being done at Stanford by Green [51]. Here a combination of English and examples is converted into a very high level program which in turn is coded in LISP or SAIL.

One major difficulty with these efforts is the complexity of understanding natural language. While this is an important problem in its own right, it is really not automatic programming. The automatic programming part of these systems consists only of the translation of

some formalism for English--semantic nets for Heidorn, a very high level programming language for Balzer and Green, MAPL for the MIT group--into some computer language. Other automatic programming efforts avoid this difficulty. One of the best known of these was the development of the HACKER system by Sussman [121]. He uses a very limited domain--several blocks on top of a table--and a formalism based on LISP and the predicate calculus. Automatic programming then consists of finding a sequence of commands for a mechanical arm that achieves a desired state of the set of blocks based on a given initial state.

Several other automatic programming efforts have been based on a formal specification of the problem. Early efforts in this area by Waldinger [81] and Manna [88] used the first order predicate calculus. Here a problem is represented by the relation between its input and output variables and automatic programming is accomplished by means of theorem proving. The theorem to be proved is simply the relationship that defines the problem and the available axioms specify what each construct of the target language does. A formal proof of the relationship using these axioms can easily be transformed into a valid program. Recent efforts along these same lines include further work of Manna and Waldinger [89] and the system of Darlington based on second order predicate calculus [34].

Another form of automatic programming offers a compromise between natural language and highly formalized problem specification. This is programming by example and is characterized by the work of Summers [119]. In this system a problem is specified by giving a number of

input-output pairs that illustrate what is to be computed. These pairs are then transformed into a set of recurrence relations from which a recursive LISP program is generated. Other systems that involve programming by example include the work of Biermann with Turing machines [22] and the early work of Green with LISP [51].

A final focus for automatic programming is the problem of automatic code generation. Here both a source program and a machine description are given and the object is to translate the program into machine language for the given machine. These efforts are best characterized by work continuing at Yale by Perlis, Fraser, Wick and others [134,46]. Here the program is given in a very simple programming language roughly equivalent to the intermediate language of a compiler, and the machine is specified using ISP (Instruction Set Processor), a common formalism introduced by Bell and Newell [20].

In order to develop a formal model of automatic programming we must determine the common elements of these examples. The clearest element is that of translation. All of these efforts are concerned with the problem of translating a program or problem in one language into a computer program in a second language. The natural language based systems use a formalism of English--either semantic nets, a very high level language, or MAPL--as the source language. The HACKER system uses a set of blocks world predicates. Manna and Waldinger use the predicate calculus. Summers' programming by example system uses the set of recurrence relations constructed from the given set of examples. And the

efforts at Yale in automatic code generation use a simple programming language as the source.

Translation is characteristic of many different problems. However, while most such problems are based on straightforward, well-defined mappings, the various automatic programming problems are not. Thus another common element of automatic programming problems is that they are based on a vaguely defined translation. Through further analysis we show this is actually the inverse of a well-defined translation.

3.2 SEMANTICS

We have shown that automatic programming can be viewed as the translation of a program in some source language into a corresponding program in some target language. This is not an arbitrary transformation, but rather one that preserves the meaning or semantics of the source program. The concept of formal semantics is central to the problem of automatic programming. The semantics of the source language and specifically of the source program must be clearly defined. Moreover the semantics of the target language are necessary to determine if the resultant program is a proper one.

The simplest approach to formal semantics is translational. Here a program is defined in terms of another, equivalent program in some standard well-defined language. The translational approach is characterized by the use of a formal translation that takes a program of the source language into an equivalent program in the base language. The

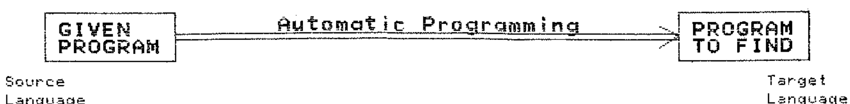
most common such translations are attributed grammars in work done by Knuth [75] and Lewis, Rosenkrantz and Stearns [83], and syntax-based translations in the work of Irons [63], Feldman and Gries [43], and Aho and Ullman [7,10,11]. Since this approach is the simplest formal approach to semantics, we use it as a foundation for our model of automatic programming.

Another approach to semantic specification is interpretive. This is characterized by the use of an interpretive mechanism--either a machine or a mathematical formalism--that takes the original program and executes it. Here the semantics of the program are represented by the resultant execution sequence. There are essentially three types of interpretive semantics: operational, denotational, and axiomatic. Of these, operational semantics are the most computer-oriented and the only ones we use. Here a formal machine is specified along with a simple translation of the source language into the input language for this machine. The semantics of a program are represented by the sequence of states the machine passes through while executing the program. The principal examples of this approach involve the use of VDL (Vienna Definition Language) [133], SECD (Store, Environment, Control, Dump) [78], and ISP [20] as the interpreting machine. Denotational semantics use recursive functions to execute the given program in a more abstract setting. These are characterized by the work of Scott [113], McCarthy [92] and Stratchey [118]. Finally, axiomatic semantics relate programs to well-formed formulas of some logic. The efforts of Floyd [45] and Hoare [58,59] are the principal examples of this abstract approach.

These formal methods of semantic specification have a number of common elements. They all represent a program as an element of a semantic domain. With translational semantics this is as a program in a base language, and for interpretive semantics this is as a sequence of machine states in the set of all such sequences. Moreover these methods all define an easily computable mapping that can take a program in some language into its semantic representation. Thus, regardless of which formal semantics we use, a program is represented by a form or structure in some semantic domain, and a programming language is represented by a mapping from that language into this semantic domain.

3.3 DEVELOPING THE MODEL

With this notion of semantics we can develop a model that reflects the common elements of the problem of automatic programming. A common basis is that of translation. Given a problem or program in a source language, the object of automatic programming is to generate an equivalent program in the target language. Thus the problem can be viewed as

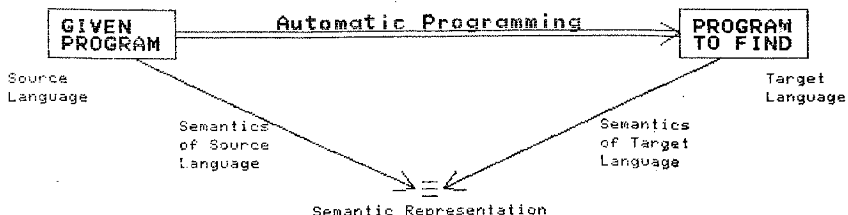


This simple model does not fully reflect the problem of automatic programming since it cannot guarantee that the source and target programs must have the same meaning. A complete model of automatic programming must represent the semantics of both the source and target

languages by means of mappings into a common semantic domain. When we include this notion of semantics the problem becomes

> Given a program in some source language and a semantic characterization of both the source and target languages in terms of some common semantic representation, find a program of the target language whose semantics are equivalent to that of the original program.

This can be pictured as

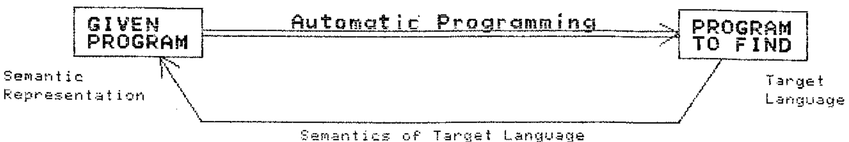


where the single arrows represent the semantic mappings and the double arrows represent the translation of automatic programming.

We can simplify this model by noting that semantics-specifying mappings are generally quite easy to compute. Since we are actually interested in the semantic representation or meaning of the source program rather than the program itself, we can consider this semantic representation as the input. This eliminates one of the three languages and one of the mappings of the model and lets us state the problem of automatic programming as

> Given the semantics of a program in some semantic representation and a semantic characterization of the target language in terms of this semantic representation, find a target language program whose semantics are equivalent to the given ones.

This can be depicted as

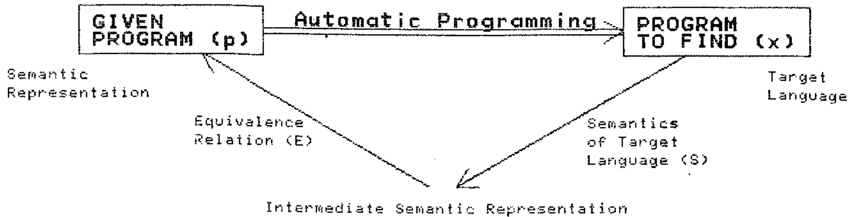


This is still not a complete model of automatic programming as the notion of equivalence is still not well-defined. The general definition of equivalence--that two programs are equivalent if they compute the same function--is recursively undecidable and represents an overly complex problem specification. To make the problem more meaningful we must allow for a more restricted notion of equivalence. We do this by having an appropriate equivalence relation as part of the problem input. This relation can be viewed as a mapping that can take any program into any equivalent program. Then the problem of automatic programming becomes

> Given the semantics of a program in some semantic representation, a semantic characterization of a target language in terms of this semantic representation, and a mapping defining the notion of equivalence, find a target language program whose semantics can be mapped into that of the original program by the equivalence mapping.

If we then allow for an intermediate program as the actual semantic

representation of the target program, we can view automatic programming as



Here the constraint that the source and target programs have the same meaning is reflected in the two mappings. In particular, the original program must be an image of the semantic representation of the target program which must in turn be the image of the actual target program. If we let S denote the semantic mapping, E the equivalence mapping, x the program to find, and p the given program, this condition can be stated mathematically as

$$p=E(S(x))$$

But then x can be obtained from p by inverting these mappings. In particular

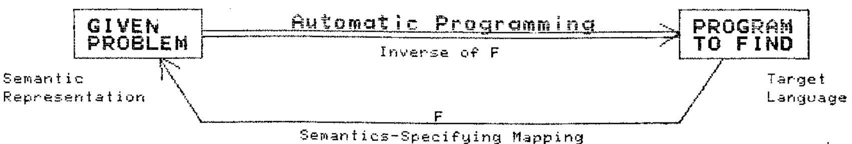
$$x=S^{-1}(E^{-1}(p))$$

Thus automatic programming can be viewed as the problem of inverting semantics-specifying mappings.

We can simplify this model one step further by composing the two mappings. Then the problem of automatic programming can be stated as

> Given the semantics of a problem in some semantic representation and a mapping defining the semantics of the target language in terms of this semantic representation, find a target program whose image under this mapping is the semantics of the original problem.

It is again clear that automatic programming is simply the problem of inverting a semantics-specifying mapping. This final concise model can be pictured as



The model we have developed for automatic programming is concise and accurate. It correctly represents the problem as a process of translation. Moreover it reflects the fact that the mapping is not well-defined since it is actually the inverse of a given mapping. Its real advantage however is how well it actually reflects automatic programming. In the next section and in the examples of chapter 6 we show that the model does fit the problem both in the sense that actual automatic programming problems can be easily considered in terms of the model and that results obtained through the model can be applied to actual automatic programming problems.

3.4 ILLUSTRATING THE MODEL

As an illustration of how the model reflects an actual automatic programming problem we consider the problem of automatic code generation. We first present the basic problem and then we simplify and formalize it to show that automatic code generation actually involves computing an inverse translation.

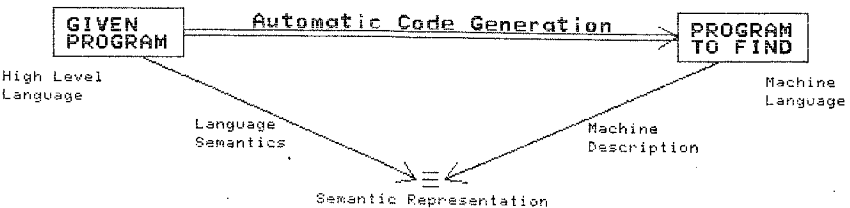
Automatic code generation is the problem

> Given a machine description, a fixed source language, and some program in that language, produce a machine language version of the given program for the given machine.

This problem is similar to that of a translator writing system [43]. In both cases the input consists of a description of the source language and a description of the target machine and the output consists of a mapping from the source into the machine language. However, while the only semantics given to a translator writing system are those of the source language in terms of the machine language, the semantics of both the source and machine languages are given in terms of a common semantic representation in an automatic code generation problem.

The given machine description is used to define the semantics of the machine language rather than to physically describe the machine. Thus the input to the problem actually consists of the source program and a semantic characterization of both the source and target languages. Both of these semantic characterizations are actually mappings from the appropriate language into a common semantic domain. Then the problem of

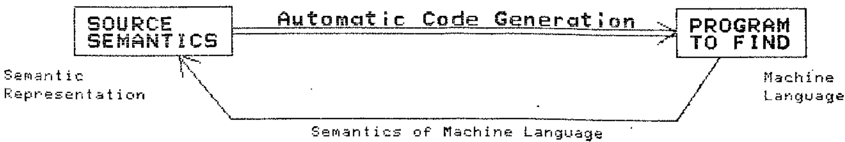
automatic code generation is simply that of finding a machine language program whose semantic representation is equivalent to that of the source program. This problem can be viewed as



The original source program is never actually used in this problem. Rather it is converted into the appropriate semantic representation which then serves as the source. This conversion is easy to compute since it just involves a semantic mapping. We can view it as separate from the problem of automatic code generation and accept as input the semantic representation of the source program rather than the program itself. Then the problem becomes

> Given a semantic description of a machine language in terms of some semantic representation and the semantics of a a program in this representation, find a machine language program whose semantics are equivalent to the given ones.

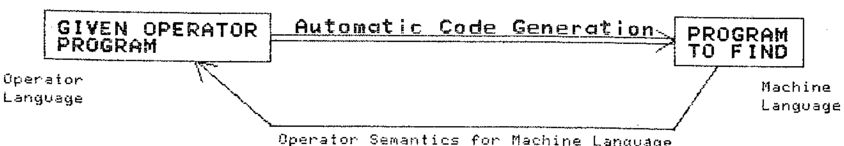
This can be pictured as



A simple semantic domain for the problem of automatic code generation is an operator language. Here the semantics of a program are given by a sequence of primitive operations. The semantics of a machine language are represented by a mapping that takes each machine instruction into the sequence of primitive operations it accomplishes. With such a semantic representation the problem of automatic code generation can be stated as

> Given a sequence of operations representing a program and a description of a machine language in terms of operations, find a machine language program whose sequence of operations is equivalent to that of the original program.

This can be pictured as

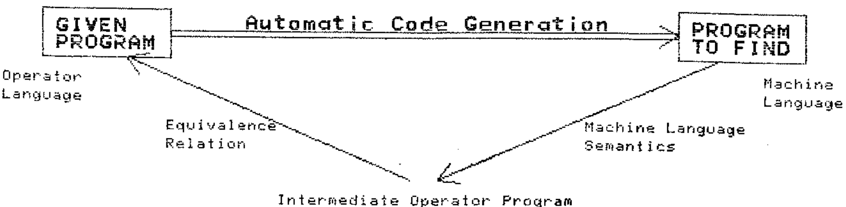


In order to formalize this description we have to add a formal notion of equivalence. We again do this with an appropriate equivalence relation as part of the problem input. If we consider this relation as

a mapping within the operator language, the problem of automatic code generation becomes

> Given a sequence of operations, a description of a machine language in terms of operations, and a mapping defining the notion of equivalence, find a machine language program whose sequence of operations can be mapped into the given one by the equivalence mapping.

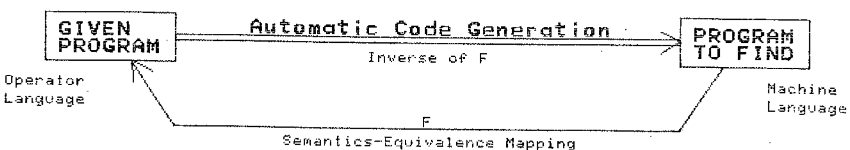
which can be represented as



Finally, we can simplify this model by composing the machine description mapping with the equivalence mapping. This yields the problem

> Given a sequence of operations and a mapping defining all equivalent operation sequences for a machine language program, find a machine language program whose image under this mapping can be the given sequence.

Pictorially, this is the model



But this shows that automatic code generation is the problem of inverting a semantics-specifying mapping. We can do a similar development for most other automatic programming problems. In each case we can reduce the problem to one of inverting a single semantics-specifying mapping. A sampling of such problems and their reduced versions is presented in figure 3.1.

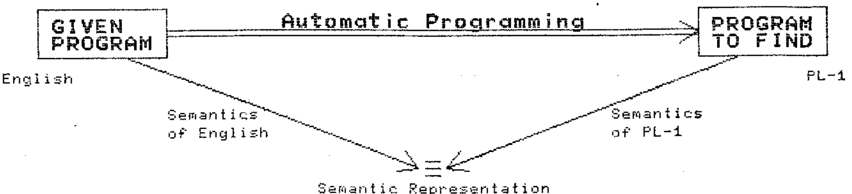
3.5 DEFINING THE MODEL--SEMANTIC REPRESENTATION

Our model shows that there are two factors that determine the complexity of automatic programming, the type of source language and the nature of the semantics-specifying mapping. We can use this to determine the complexity of a specific automatic programming problem by knowing what source language and semantic mapping are required by our model to accurately represent the problem. Moreover we can use our model to characterize practical automatic programming by determining which types of languages and classes of mappings make the problem both tractable and worthwhile.

The source language of our model actually defines the semantic representation of both the initial problem and the target language. Within the context of automatic programming, all the various forms of semantic representations tend to be one of two forms. In the first form only the relationship between the inputs and the outputs of a program is specified. Here automatic programming involves both finding an algorithm to effect this relationship and coding this algorithm in some programming language. We denote these as <u>input-output-form</u> semantics.

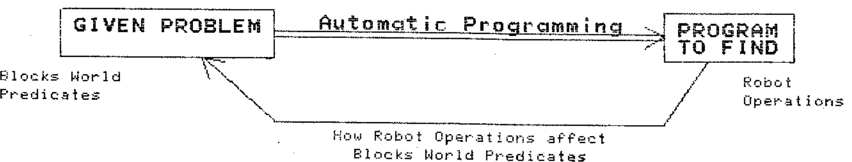
BALZER'S VIEW OF AUTOMATIC PROGRAMMING

PROBLEM: Given an English description of a problem, generate a PL-1 program to solve it.



SUSSMAN'S HACKER SYSTEM

PROBLEM: Given an initial and target set of predicates describing the blocks world, find a series of robot operations to transform the initial set into the desired one.



THE DECOMPILING PROBLEM

PROBLEM: Given a machine language program, find an equivalent FORTRAN program.

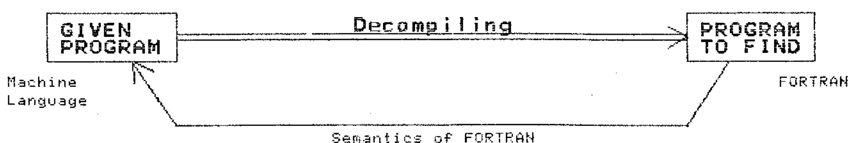


Figure 3.1: Automatic Programming Problems in our Model

The second form includes the algorithm as part of the specification. Here the problem of automatic programming only involves coding. We denote these as program-form semantics.

Input-output-form semantics represent the most general and ambitious form of automatic programming. They are charcteristic of Balzer's view of the problem where the user merely specifies his problem in English and the computer does the rest. They are also the basis of the automatic programming efforts of Manna, Waldinger and Darlington where the problem is given in terms of predicate calculus relations and a program is generated by theorem proving techniques. However, the inherent task of finding an algorithm for a problem is extremely hard.

It can be shown formally that automatic programming with input-output-form semantics is equivalent in complexity to theorem proving. Informally, we can view the semantic specification of a program, the input-output relationship, as the theorem to be proved. The semantics of a language then show how the execution of each program construct affects the given input-output relationship, and hence the various program constructs are axioms that can be used to establish the desired relationship. A proof using these constructs can be easily converted into a program. Similarly, a program that establishes the given relationship is equivalent to a proof of that relationship based on the given axioms.

But theorem proving is a hard problem. In many cases it is undecidable since many formal systems are incomplete, i.e. have theorems that can neither be proved nor disproved [101]. In simpler cases it

remains exponential [32]. Even in the simplest case, where the relationship is given in terms of the first order predicate calculus, the problem is known to be NP-complete [67] and hence most likely intractable. This implies that automatic programming with input-output-form semantics is not practical. We therefore restrict our attention to program-form semantics.

Program-form semantics include the algorithm as part of the specification and hence are actually programming languages. The automatic code generation problem described earlier uses program-form semantics in the form of an operator language. The recurrence relations of Summers' system and the machine language of decompiling also represent program-form semantics. When program-form semantics are used there is no real need to develop an algorithm to solve the problem. The complexity of automatic programming then depends solely on the type of mapping required for the semantic specification of the target language.

3.6 DEFINING THE MODEL--SEMANTIC MAPPING

Many types of mappings can be used for this specification. Those that are currently employed are generally recursive or at least primitive recursive relations. These are widespread because they are both easy to describe and quite powerful. However, as long as they are used, automatic programming remains an intractable problem. The simple question of determining if an inverse exists is undecidable when the mapping is recursive and is exponential when the mapping is primitive recursive [101].

If the full power of recursive functions were necessary for semantic characterization, the problem of automatic programming would be intractable. However, program-form semantics allow us to consider restricted classes of mappings. Perhaps the simplest such class is that defined by finite automata known as gsm mappings. These are quite simple and have been widely studied [61]. Moreover, as any given gsm mapping can be inverted in linear time, an automatic programming problem that used these would be quite tractable. However it would not represent worthwhile automatic programming. In particular, both the domain and the range of a gsm mapping are regular languages and, since a reasonable programming language is at least context free, a gsm mapping could not adequately describe it.

To achieve our goal of practical automatic programming we must consider transformations that are more powerful than gsm mappings and more restrictive than recursive functions. The next chapters of this dissertation are devoted to finding such a model of translation. We determine the most powerful mappings that can be used for semantic specification and still be inverted in polynomial time. In this way we find the most useful form of automatic programming that is still tractable and thus fully characterize practical automatic programming.

CHAPTER 4

INVERTING SIMPLE TREE TRANSDUCERS

The discussion of the previous chapter demonstrates that the type of semantics-specifying mapping used for automatic programming is quite important. In particular, it shows that in a practical problem such a mapping must be more general than a gsm mapping and yet not as powerful as a recursive one. In this chapter we investigate such practical mappings. We first define top-down finite state tree transducers and then consider the complexity of their inversion in two simple cases. Our results not only illustrate a simple instance of practical automatic programming, but also lay an excellent foundation for the more detailed results of the next chapter.

4.1 THE MODEL OF TRANSLATION

We choose top-down finite state tree transducers (t-fsts) as our model of translation. This model was first proposed by Doner [37] and by Thatcher and Wright [127] and has been developed by Thatcher [123, 124,125,126], Rounds [105], Engelfriet [41] and others [15,90,112]. It is a reasonable model for several reasons. Top-down tree transducers have been widely studied as a simple generalization of gsm mappings and

hence are a logical extension of that simple model. Moreover, the trees in the domain of a t-fst are in effect parse trees for some context free grammar [125] and thus t-fsts are quite useful for specifying the semantics of context free programming languages. An interesting subset of the class of t-fsts is the class of generalized syntax-directed translation schemata (GSDTS) introduced by Lewis and Stearns [85] and developed by Aho and Ullman [6,7,8,9,10,11]. This class has been widely studied in its own right.

4.1.1 Top-Down Finite-State Tree Transducers

A t-fst is a top-down tree rewriting system. It assigns the starting state to the root of the tree and each application of a rule takes a node with an assigned state and yields a new subtree with states assigned to the sons of the original node. In this way the states move down the tree from the root toward the leaves until the transformation succeeds, yielding a tree with no assigned states. Formally, a t-fst T is a tree rewriting system $<\Sigma,\Delta,Q,q_d,R>$ where each rule in R either has the form $q(A(y_1 \cdots y_k)) \rightarrow q(t)$ with each $y_i \in \Sigma_0 \cup X$, $q \in Q$, $A \in \Sigma_k$ and $t \in T_\Delta[Q(X)]$, or the form $q(a) \rightarrow b$ where $q \in Q$, $a \in \Sigma_0$, $b \in \Delta_0 \cup \{\lambda\}$. We require that each x_i in X, the set of tree variables, appears at most once in the left part of a rule, and that each x_i that appears in the right part of a rule must appear in the left part as well. Then a t-fst T translates a tree $t \in T_\Sigma$ into a tree $t' \in T_\Delta$ if and only if $q_d(t) \overset{*}{\underset{T}{\Rightarrow}} t'$ in the sense of a tree rewriting system.

While most semantic representations and programming languages are given in terms of strings, a t-fst defines a mapping between trees. We therefore generalize t-fsts in a natural way to take strings into strings. In particular, if T is a t-fst, the <u>string mapping</u> defined by T is just the relation

$$\{\langle s,s'\rangle \mid T{:}t\to t' \text{ and } s=fr(t),\ s'=fr(t')\}.$$

When we refer to the mapping of a tree transducer it will be clear from the context whether we mean the actual tree mapping or this associated string mapping.

We present an example of a t-fst in figure 4.1. Part 4.1a defines a mapping from the string a^n into the string a^{n^2} using the equivalence

$$n^2 = (n-1)^2+2(n-1)+1.$$

Here the rules of the t-fst are specified both in their tree form and in the equivalent linear form. In part 4.1b we show the intermediate steps of a transformation computed with this t-fst.

Several results are known concerning t-fsts. The principal one is that the language composed of the frontiers of all trees in the domain of a t-fst is context free and that every context free language can be so represented [123]. Moreover, the domain of any t-fst is either the set of parse trees for some context free grammar or is equivalent to such a set under a suitable relabeling of the interior nodes [125]. In the latter case there is another t-fst that computes the same transformation and whose domain is precisely the parse trees of some context free grammar. We call this grammar the <u>domain grammar</u> of the t-fst. It

FIGURE 4.1

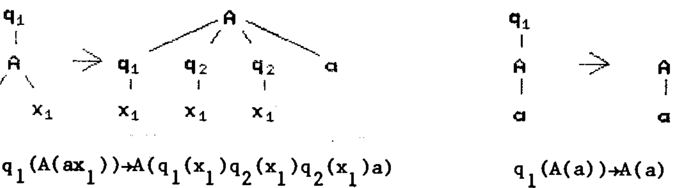
$T = \langle \Sigma,\Delta,Q,q_1,R\rangle$

$\Sigma_0 = \{a\}, \quad \Sigma_1=\Sigma_2=\{A\}$

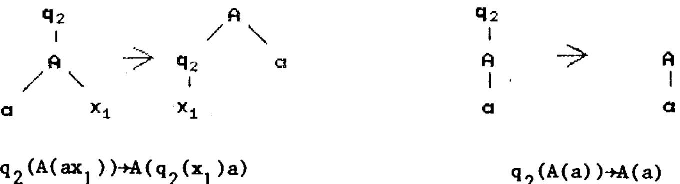
$\Delta_0 = \{a\}, \quad \Delta_1=\Delta_2=\Delta_4=\{A\}$

$Q = \{q_1,q_2\}$

R:

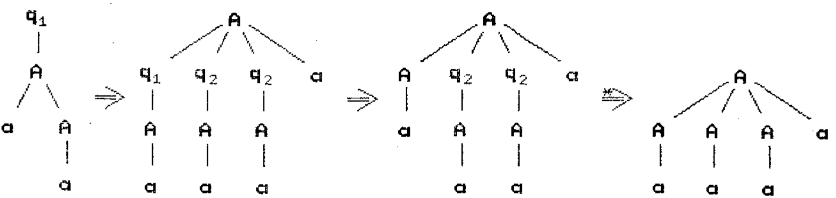


$q_1(A(ax_1))\to A(q_1(x_1)q_2(x_1)q_2(x_1)a)$ $\qquad$ $q_1(A(a))\to A(a)$



$q_2(A(ax_1))\to A(q_2(x_1)a)$ $\qquad$ $q_2(A(a))\to A(a)$

a) A t-fst to map a^n into a^{n^2}



b) A mapping of a^2 into a^4

Figure 4.1: An Example t-fst

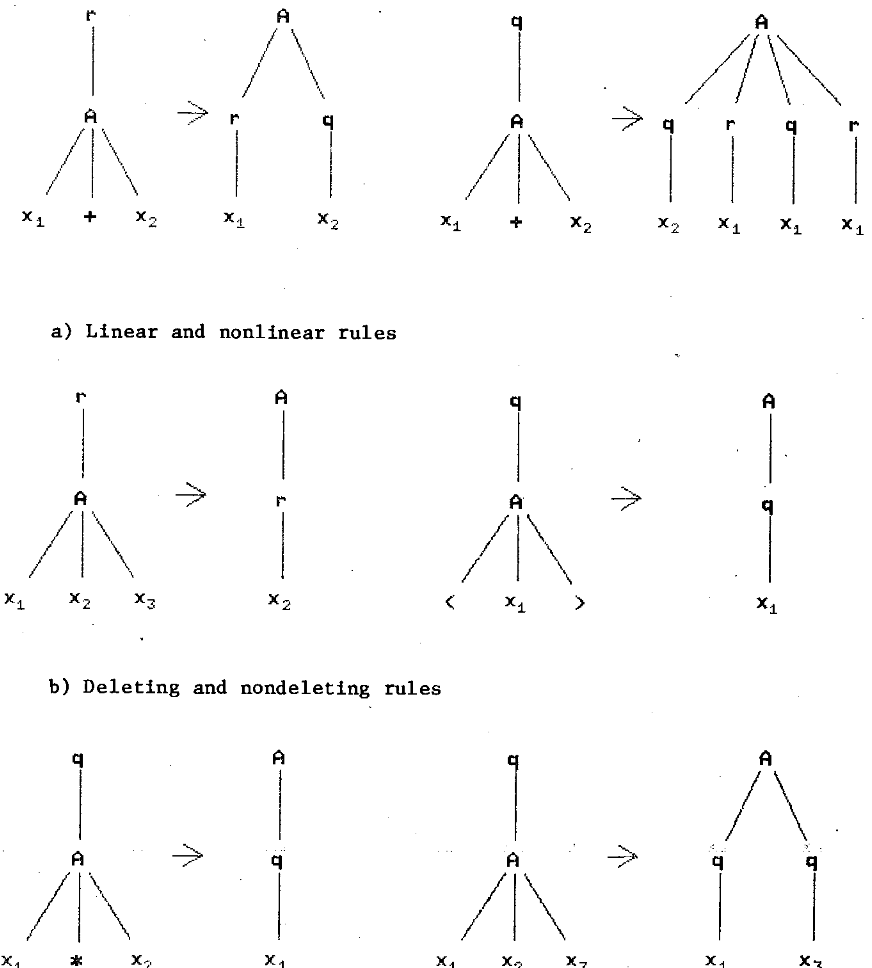
is easy to see from its source patterns that the t-fst of figure 4.1 has the context free domain grammar <{A},{a},{A→aA,A→a},A>.

We consider several restricted forms of t-fsts that are based on limiting the set of applicable rules. A t-fst is called <u>linear</u> if no subtree is ever duplicated, that is, if every tree variable x_i that appears in the left part of some rule appears at most once in the right part. Similarly a t-fst is called <u>nondeleting</u> if no arbitrary subtree is ever deleted, that is, if every x_i that appears in the left part of a rule appears at least once in the right part. Finally a t-fst is called <u>deterministic</u> if for every possible subtree with an associated state there is at most one applicable rule. Illustrations of these restrictions are given in figure 4.2.

4.1.2 Syntax-Based Translations

The class of deterministic t-fsts is equivalent to the class of syntax-based translations called generalized syntax-directed translation schemata (GSDTS) [90]. These translations can be conveniently represented by means of a pair of associated grammars, one describing the domain and the other describing the range of the translation. The first or source grammar is context free. For each nonterminal of this grammar, there is a finite set of associated nonterminals or translations, of the second or target grammar. Moreover, each production of the source grammar is paired with a unique production of the target grammar. While the source production represents a substitution for a single nonterminal, the target production represents a substitution, to be done in

FIGURE 4.2



a) Linear and nonlinear rules

b) Deleting and nondeleting rules

c) Rules that are simultaneously applicable

Figure 4.2: Restricted Forms of Tree Transducers

parallel, for each of the associated nonterminals. A translation is then defined by parsing the input according to the source grammar and using the associated parse with the target grammar to generate the target string.

Formally a GSDTS is a 6-tuple $\langle N,\Sigma,\Delta,\Gamma,R,S\rangle$ where N, Σ and Δ are finite alphabets of nonterminals, input symbols and output symbols respectively, $S\in N$ is the start symbol, Γ is a set of transition symbols of the form A_i for $A\in N$ and $i\geq 0$, and R is a set of rules of the form

$$(A\rightarrow\alpha \ , \ A_1=\beta_1 \ , \ \dots \ , \ A_m=\beta_m)$$

such that $A\in N$, $\alpha\in(N\cup\Sigma)^*$, $A_i\in\Gamma$ for $1\leq i\leq m$, and each symbol of each β_i is either in Δ or is a symbol $B_k\in\Gamma$ such that $B\in N$ appears in α. We require that $S_1\in\Gamma$ since it will be used as the output of the mapping. Moreover, if α contains several occurrences of some nonterminal B in some rule, we require that superscripts be used to uniquely associate each B_i in some β_k of the rule with one of these occurrences. In particular, we use the superscript j to indicate the jth occurrence of B in α. Figure 4.3a presents a GSDTS that computes the same mapping as the t-fst of figure 4.1.

There are several definitions we need concerning GSDTS. Let $T=\langle N,\Sigma,\Delta,\Gamma,R,S\rangle$ be a GSDTS. For each rule of T we call $A\rightarrow\alpha$ the <u>source production</u> of the rule. Let P be the set of all source productions. Then the <u>source grammar</u> of T is just the context free grammar $\langle N,\Sigma,P,S\rangle$. We call each A_k a <u>translation</u> of A and each pair $A_i=\beta_i$ a <u>translate</u>. The set of all translates of a rule is called its <u>target production</u>.

The operation of a GSDTS is based on the notion of an associated derivation. Let α be the source string for a GSDTS $T=\langle N,\Sigma,\Delta,\Gamma,R,S\rangle$ and let G be the source grammar of T. Then by definition α has a derivation using the grammar G,

$$S=\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \dots \Longrightarrow \alpha_n=\alpha \ .$$

An <u>associated derivation</u> here is the sequence $(\gamma_0,\gamma_1,\dots,\gamma_n)$ where each γ_i is related to the corresponding α_i. In particular, each α_i can be written as $\delta_0 X_1 \delta_1 X_2 \dots X_k \delta_k$ with $\delta_j\in\Sigma^*$ and $X_j\in N$, and each γ_i can be written as $\sigma_0 Y_1 \sigma_1 Y_2 \dots Y_m \sigma_m$ with $\sigma_j\in\Delta^*$ and $Y_j=A_k\in\Gamma$ for some $A\in N$, such that Y_j is associated uniquely (using superscripts if necesssary) with some $X_\ell=A$ in α_i. This relationship is defined inductively by

1. initially $\alpha_0=S$ and $\gamma_0=S_1$; and

2. if $\alpha_i\Longrightarrow\alpha_{i+1}$ using the rule $(A\rightarrow\alpha, A_1=\beta_1,\dots,A_n=\beta_n)$ to replace $X_j=A$ in α_i, then γ_{i+1} is constructed from γ_i by substituting the string β_k for all $Y_\ell=A_k$ associated with X_j.

The association between the symbols of α_{i+1} and γ_{i+1} is then maintained by retaining the association between all the items in α_i and γ_i except for X_j and adding to this all the associations inherent in the rule that is used. This is illustrated in figure 4.3b.

Using this notion of an associated derivation, we say that the GSDTS T transforms a string s into a string s′ if and only if

1. s is in the domain of T; and
2. there is a parse of s using the source grammar of T,
$S=\alpha_0 \Longrightarrow \ldots \Longrightarrow \alpha_n=s$,
such that $(\gamma_0,\ldots,\gamma_n)$ is the associated derivation and $\gamma_n=s'$.

As α_n contains only terminal symbols, $\gamma_n \in \Delta^*$ and hence s′ is indeed a string in the output language. The construction of such a transformation for the example GSDTS of figure 4.3a is presented in figure 4.3b.

We can define restricted classes of GSDTS just as we did with tree transducers. A GSDTS, for example, is called nondeleting if and only if each translate of each rule contains at least one translation of each nonterminal in the source production of the rule. However, rather than defining all of the various restricted classes, we introduce the more common notion of syntax-directed translation schemata (SDTS). These have only a single translation of each nonterminal symbol and a single translate in each rule. Moreover, the nonterminal elements in the translate of a rule must be a permutation of the nonterminals in the source production. Because there is only one translation for each nonterminal there is no need for the alphabet Γ and we can formally represent an SDTS T as a 5-tuple $T=\langle N,\Sigma,\Delta,R,S\rangle$ where N is the set of nonterminals, Σ is the source alphabet, Δ is the target alphabet, $S \in N$ is the start symbol, and R is the set of rules. Each rule in R has the form $A \to \alpha,\beta$ where A is a nonterminal, α is the source string, and β is the target string. The underlying source production here is $A \to \alpha$ while the target production is $A \to \beta$. Let R″ be the set of all target productions

$G = \langle N,\Sigma,\Delta,\Gamma,R,A\rangle$

$N=\{A\}$
$\Sigma=\{a\}$
$\Delta=\{a\}$
$\Gamma=\{A_1,A_2\}$

R: $A \to aA$, $A_1=A_1A_2A_2a$, $A_2=A_2a$
$A \to a$, $A_1=a$, $A_2=a$

a) A GSDTS to map a^n into a^{n^2}

Parse of a^3:

$A \Longrightarrow aA \Longrightarrow aaA \Longrightarrow aaa$

Associated Derivation

$A_1 \Longrightarrow A_1A_2A_2a$
$\Longrightarrow A_1A_2A_2aA_2aA_2aa$
$\Longrightarrow aaaaaaaaa$

b) Mapping of a^3 into a^9

Figure 4.3: An Example GSDTS

of T. Then the <u>target grammar</u> of T is just the context free grammar $\langle N,\Sigma,R'',S\rangle$. We present an example of an SDTS in figure 4.4a.

Let T be an SDTS and s a string. We define the operation of an SDTS as the obvious restriction of the operation of a GSDTS. Thus T(s) is computed by finding a parse for s using the source grammar of T and then constructing the associated derivation of this parse. An example of such a translation is presented in figure 4.4b. Note that each element γ_i of the associated derivation contains a permutation of the nonterminals of its associated α_i. Moreover, it is clear that

$$S \Longrightarrow \gamma_0 \Longrightarrow \gamma_1 \Longrightarrow \dots \Longrightarrow \gamma_n = s'$$

is actually a derivation of s′ using the target grammar of the SDTS. We make substantial use of these observations when we consider the problem of inverting an SDTS mapping.

4.1.3 A Look Ahead

When tree transducers and syntax-based translations are used to specify the semantics of a programming language, automatic programming is just the problem of computing their inverses. We continue our study of the complexity of automatic programming by considering in detail the complexity of inverting these mappings. We look at two classes of problems. In the first class the mapping and the string to invert are given as input. Here we show that the problem is practical for a whole class of mappings. This is unrealistic at times since automatic programming often uses a fixed or restricted mapping in which case we only want to know how the complexity of the problem varies with the size of

$T=\langle N,\Sigma,\Delta,R,E\rangle$

- N= {E,T,F}
- Σ= {a,<,>,+,*}
- Δ= {a,+,*}
- R:
  - E→E+T , +ET
  - E→T , T
  - T→T*F , *TF
  - T→F , F
  - F→<E> , E
  - F→a , a

a) An SDTS to map infix expressions into prefix expressions

Parse of 'a*<a+a>'	Associated Derivation
E==> T	E==> T
==> T*F	==> *TF
==> F^1*F^2	==> $*F^1F^2$
==> a*F	==> *aF
==> a*<E>	==> *aE
==> a*<E+T>	==> *a+ET
==> $a*\langle T^1+T^2\rangle$	==> $*a+T^1T^2$
==> a*<F+T>	==> *a+FT
==> a*<a+T>	==> *a+aT
==> a*<a+F>	==> *a+aF
==> a*<a+a>	==> *a+aa

b) Sample mapping of a*<a+a> into *a+aa

Figure 4.4: An Example SDTS

the string to invert. Our second class of problems considers precisely this complexity since it assumes that the mapping is fixed and the only input is the string to invert.

Using this framework for studying the complexity of automatic programming we show

MAIN THEOREM: Within the class of top-down finite state tree transducers, considering the restrictions of linearity, nondeletion, and determinism, the following are the maximal tractable cases:
1) For the case of a fixed translation, a deterministic t-fst;
2) For the case of an arbitrary translation, a linear t-fst.

In the following sections we consider the simplest kinds of t-fsts, those that are both linear and nondeleting, and introduce the ideas that are needed to invert them. We first show that the problem of inversion here is tractable by showing that it is equivalent to the problem of parsing an arbitrary context free grammar. Here we present algorithms for computing the inverse of both this class of t-fsts and for the related class of SDTS. Then in the next chapter we consider more complex mappings. We first show that every deleting t-fst is effectively equivalent to a nondeleting one of about the same complexity, and thus that the restriction of nondeletion does not affect the asymptotic complexity of the problem. We then consider nonlinear tree transducers and show that the problem of inversion is intractable for even a fixed translation. However for the case of a fixed deterministic t-fst, or equivalently a fixed GSDTS, we prove that the problem is tractable by presenting a practical algorithm to compute the inverse.

4.2 LINEAR NONDELETING TREE TRANSDUCERS

We begin our study of the complexity of inverting semantics-specifying mappings with a simple case, the string mappings of linear nondeleting tree transducers. Our main result will be to show that these can be inverted tractably and hence can provide a basis for practical automatic programming. We start our investigation with a semantics-oriented example of a linear nondeleting tree transducer. We then show how such a mapping can be inverted and determine the complexity of computing its inverse.

4.2.1 An Example

A relatively simple problem related to the semantic specification of a programming language is the translation of infix expressions into prefix notation. In figure 4.5a we give a linear nondeleting t-fst to accomplish this task for a limited set of expressions. It operates by parsing the original equation in order to determine precedence and eliminate parentheses, and then correctly rearranging the resultant symbols. This is demonstrated in figure 4.5b where an example is worked out. The general problem of finding an inverse is that of finding some string that the transducer would map into the given string. In this case it is just the problem of translating from prefix notation back into infix notation.

$T = \langle \Sigma, \Delta, Q, q_e, R \rangle$

$\Sigma_0 = \{a,+,*,<,>\}$, $\Sigma_1 = \Sigma_3 = \{E,T,F\}$

$\Delta_0 = \{a,+,*\}$, $\Delta_1 = \{E,T,F\}$, $\Delta_3 = \{E,T\}$

$Q = \{q_e, q_t, q_f\}$

R: $q_e(E(x_1+x_2)) \rightarrow E(+q_e(x_1)q_t(x_2))$

$q_e(E(x_1)) \rightarrow E(q_t(x_1))$

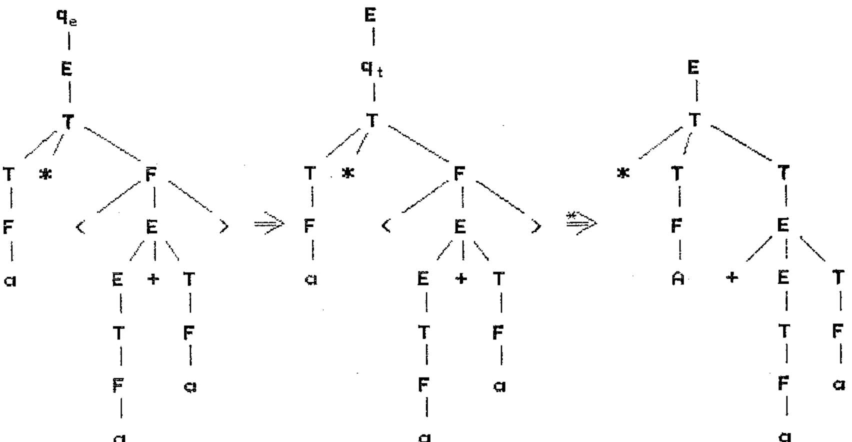
$q_t(T(x_1*x_2)) \rightarrow T(*q_t(x_1)q_f(x_2))$

$q_t(T(x_1)) \rightarrow T(q_f(x_1))$

$q_f(F(<x_1>)) \rightarrow F(q_e(x_1))$

$q_f(F(a)) \rightarrow F(a)$

a) A linear, nondeleting t-fst to map infix into prefix



b) Sample mapping of 'a*<a+a>' into '*a+aa'

Figure 4.5: A Linear Nondeleting Tree Transducer

4.2.2 The Basic Model For Inverting A t-fst

Before we can consider inverting a t-fst, we must understand the three distinct steps of the forward translation, parsing, translating and taking the frontier. In the first step, the input string is parsed using the context free source grammar. The second step takes the resultant parse tree into a target tree. And the third step takes the frontier of this tree and yields the result of the translation. We invert such a mapping by inverting each of these steps in turn. Thus, to find the inverse translation of a given target string we first find a valid target tree with this string as its frontier by parsing the given string using a target grammar. Once we have constructed such a tree we next undo the effects of the top-down tree translation using another tree transducer, this one operating from the frontier of the tree to the root. The result is a parse tree of the domain grammar for some string. The final step of the inversion process involves determining this string by taking the frontier of the parse tree.

Most of the complexity of this inversion process occurs in the first step, parsing the given string to get a valid target tree. The final step, taking the frontier, is a simple operation, and the second one, finding a tree transformation which inverts the given t-fst, is relatively straightforward once we are given a valid target tree. The first step however is difficult because the grammar we must parse need not be context free. However, for the simple case where the transducer is both linear and nondeleting, we show that the grammar is context free and thus that the target string is easy to parse.

In this section we consider this simple class of mappings and construct a new class of transformations that can invert them. We do this by first considering separately the operations of inverting the actual tree transduction and parsing the target string, and then combining the results.

4.2.3 Bottom-Up Tree Transducers

To invert the actual tree mapping of a t-fst we introduce a different type of tree transducer that can perform the same type of operations in the reverse order. Where a top-down tree transducer operates from the root of a tree to the leaves, this new transducer operates from the leaves to the root. Where the top-down transducer begins by assigning a specific starting state to the root of the tree, our new transducer accepts, or terminates successfully, if a specific final state is assigned to the root. And where the rules of a top-down transducer specify a transition that takes a subtree with a state assigned to the root into a new subtree where the sons are either terminal nodes or subtrees with assigned states, this new transducer does just the opposite. This different type of tree transformation is called a bottom-up finite state tree transducer (b-fst) and has been studied by Engelfriet [41], Thatcher [126], and others [15,106].

Formally a b-fst, B, is a bottom-up tree rewriting system $<\Sigma,\Delta,Q,q_d,R>$ where each rule in R has one of the forms

1) $A(y_1 \cdots y_k) \to q(t)$ where $A \in \Sigma_k$, $q \in Q$, $t \in T_\Delta^2[Q(X)]$, and where either $y_i \in \Sigma_0$ or y_i is $q_j(x_k)$ for some $q_j \in Q$ and $x_k \in X$; or

2) $a \to q(t)$ where $a \in \Sigma_0 \cup \{\lambda\}$, $q \in Q$, $t \in T_\Delta^1 \cup T_\Delta^2$.

We again require that each x_i appear at most once in the left part of a rule and that each x_i that appears in the right part of a rule must also appear in the left part. An example of a b-fst is given in figure 4.6a. Figure 4.6b then demonstrates the operation of this b-fst.

As with a t-fst, the frontier of the domain of a b-fst describes a context free language [41]. Furthermore we can define restricted classes of b-fsts just as we did with t-fsts. We call a b-fst <u>linear</u> if no x_i appears more than once in the right part of any rule. We call a b-fst <u>nondeleting</u> if each x_i that appears in the left part of a rule also appears in the right part. And we call a b-fst <u>deterministic</u> if no two rules are applicable at the same node at the same time.

Given a linear nondeleting t-fst, $T=<\Sigma,\Delta,Q,q_d,R>$, we can use a very simple construction to construct the b-fst B that inverts it. In particular, we use the b-fst $B=<\Delta,\Sigma,Q,q_d,R'>$ where R' contains the rule $\beta \to \alpha$ if and only if R contains the rule $\alpha \to \beta$. This construction is demonstrated in figure 4.6 where the example b-fst inverts the t-fst of figure 4.5.

To show that this simple construction yields a b-fst that actually inverts the given t-fst we prove

$B = <\Sigma,\Delta,Q,q_e,R>$

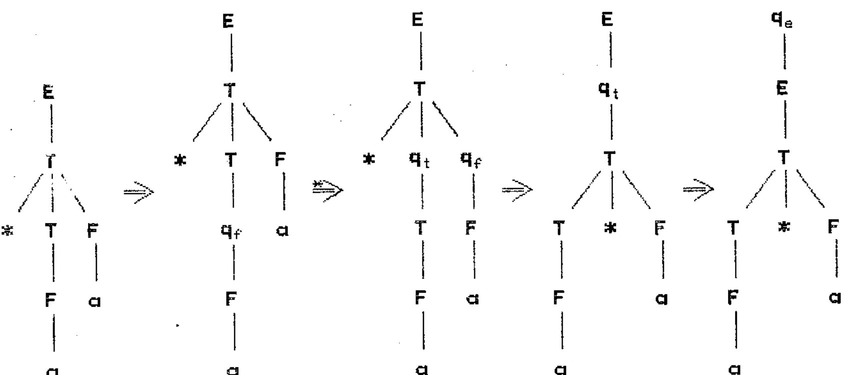
$\Sigma_0 = \{a,+,*\}$, $\Sigma_1 = \{E,T,F\}$, $\Sigma_3 = \{E,T\}$

$\Delta_0 = \{a,+,*,<,>\}$, $\Delta_1 = \Delta_3 = \{E,T,F\}$

$Q = \{q_e,q_t,q_f\}$

R:
$E(+q_e(x_1)q_t(x_2)) \rightarrow q_e(E(x_1+x_2))$
$E(q_t(x_1)) \rightarrow q_e(E(x_1))$
$T(*q_t(x_1)q_f(x_2)) \rightarrow q_t(T(x_1*x_2))$
$T(q_f(x_1)) \rightarrow q_t(T(x_1))$
$F(q_e(x_1)) \rightarrow q_f(F(<x_1>))$
$F(a) \rightarrow q_f(F(a))$

a) A b-fst to map prefix expressions into infix expressions



b) An example mapping of '*aa' into 'a*a'

Figure 4.6: An Example b-fst

LEMMA 4.1: Let $T=<\Sigma,\Delta,Q,q_d,R>$ be a linear nondeleting t-fst and let $B=<\Delta,\Sigma,Q,q_d,R'>$ be constructed as above. Then B is a b-fst, and for each $t \in range(T)$ there is a tree t' such that $B:t \rightarrow t'$ and $T:t' \rightarrow t$, i.e. B inverts T.

Proof: We first claim that B is actually a b-fst. This follows immediately from the definitions of the rules of a t-fst and a b-fst and from the fact that T is linear and nondeleting and hence each x_i appearing in the left part of a rule of T appears exactly once in the right part.

We show that B inverts T by inductively proving that $t \overset{*}{\underset{B}{\Longrightarrow}} t'$ if and only if $t' \overset{*}{\underset{T}{\Longrightarrow}} t$. To do this we show for arbitrary trees t_1 and t_2 that $t_1 \underset{T}{\Longrightarrow} t_2$ if and only if $t_2 \underset{B}{\Longrightarrow} t_1$. Let $t_1 \underset{T}{\Longrightarrow} t_2$ by a rule $\alpha \rightarrow \beta$. Then the node in t_2 that results from this application has a form which is acceptable for the rule $\beta \rightarrow \alpha$ of B. Moreover, applying this rule to this node must yield the tree t_1 and hence $t_1 \underset{T}{\Longrightarrow} t_2 \underset{B}{\Longrightarrow} t_1$. By using this fact repeatedly it follows that $t_1 \overset{*}{\underset{T}{\Longrightarrow}} t_2 \overset{*}{\underset{B}{\Longrightarrow}} t_1$ and hence that $t' \overset{*}{\underset{T}{\Longrightarrow}} t$ if and only if $t' \overset{*}{\underset{B}{\Longrightarrow}} t$.

□

4.2.4 Parsing The Target Grammar

This demonstrates one way of performing the tree transducer part of the inversion process. We next consider the parsing step and show that the range of any linear nondeleting t-fst is described by a context free grammar. This allows us to easily parse the target string and compute the inverse translation.

We first show how to determine this context free grammar. Let $T=\langle\Sigma,\Delta,Q,q_d,R\rangle$ be any t-fst. We define the <u>target grammar</u> of T, G(T), as the 4-tuple $\langle Q,\Delta_0,P,q_d\rangle$ where a production $q\to\alpha$ is in P for $q\in Q$, $\alpha\in(Q\cup\Delta_0)^*$ if and only if there is a rule of the form $q(t)\to t'$ in R such that $fr_Q(t')=\alpha$. For example, the target grammar of figure 4.5 is given below.

$G=\langle N,\Sigma,P,q_e\rangle$

where

$N=\{q_e,q_t,q_f\}$

$\Sigma=\{a,+,*\}$

P: $q_e\to +q_eq_t$

$q_e\to q_t$

$q_t\to *q_tq_f$

$q_t\to q_f$

$q_f\to q_e$

$q_f\to a$

Figure 4.7: The Target Grammar of the t-fst of Figure 4.5

It is easy to show that

LEMMA 4.2: G(T) is context free for any t-fst T.

Proof: Clearly by definition G(T) is a grammar. Hence we need only show that all the rules are of the form $A\to\alpha$ for some nonterminal A and string of nonterminals and terminals α. But this follows immediately from the construction.

□

While G(T) does not exactly describe the target trees of a t-fst T, it comes quite close. Let $L_t(T)$ represent the target language of T, that is,

$$L_t(T) = \{s \mid t\in range(T) \text{ and } s=fr(t)\}.$$

Then we can show

LEMMA 4.3: Let $T=\langle\Sigma,\Delta,Q,q_d,R\rangle$ be a linear nondeleting t-fst and let $G=G(T)$. Then $L[G]=L_t(T)$.

Proof: Let

$$q_d(t'')=\alpha_0 \underset{T}{\Longrightarrow} \alpha_1 \underset{T}{\Longrightarrow} \alpha_2 \underset{T}{\Longrightarrow} \cdots \underset{T}{\Longrightarrow} \alpha_n=t' \qquad (*)$$

be an application of T to t'' yielding t' and let $\gamma_i=fr_Q(\alpha_i)$ for $1\le i\le n$. We first prove that $L_t(T)\subseteq L[G]$ by showing that every tree translation corresponds to a derivation of G. In particular, we prove that (*) implies that there is a derivation of G,

$$q_d=\gamma_0 \overset{*}{\underset{G}{\Longrightarrow}} \gamma_n=fr(t') \quad ,$$

and hence that $fr(t')\in L[G]$. We do this inductively by showing that $\alpha_0 \overset{*}{\underset{T}{\Longrightarrow}} \alpha_i$ implies that $\gamma_0 \overset{*}{\underset{G}{\Longrightarrow}} \gamma_i$ for $0\le i\le n$. This holds trivially when $i=0$. For $1\le i\le n$, it then suffices to show that $\alpha_i \underset{T}{\Longrightarrow} \alpha_{i+1}$ implies $\gamma_i \underset{G}{\Longrightarrow} \gamma_{i+1}$. Let the rule of T used here be $q(t)\to s$. In the string

$\gamma_i = fr_Q(\alpha_i)$ there is a symbol q corresponding to the node being modified by this rule of T. After the rule is applied, the state-frontier changes only at this q and in effect replaces this q with the state-frontier of the applied rule. But this is exactly the effect of applying the production of G derived from this rule, $q \to fr_Q(t')$, to this q in γ_i and hence $\gamma_i \overset{}{\underset{G}{==>}} \gamma_{i+1}$.

We next show that given some derivation of G,

$$q_d = \gamma_0 \underset{G}{==>} \gamma_1 \underset{G}{==>} \cdots \underset{G}{==>} \gamma_n \quad ,$$

there is a tree t" such that T(t")=t′ and $fr(t') = \gamma_n$. We do this inductively by creating a set of approximations to t" in $T_\Sigma[Q]$, $\alpha_0 \cdots \alpha_n$, such that $\alpha_i \overset{*}{\underset{T}{==>}} \mu_i$ where $fr_Q(\mu_i) = \gamma_i$ and where there is a one-to-one correspondence between the elements of Q in α_i and in γ_i. For the case i=n, this shows that $t'' = \alpha_n \overset{*}{\underset{T}{==>}} \mu_n = t'$ where $fr(t') = \gamma_n$, and hence that $L[G] \subseteq \mathcal{L}_t(T)$. Initially we have $\alpha_0 = q_d = \mu_0 = \gamma_0$ and the hypothesis holds. Suppose at the ith stage, $\gamma_i \underset{G}{==>} \gamma_{i+1}$, we replace a nonterminal q using a production corresponding to the rule of T $q(A(z_1 \cdots z_n)) \to t$. Let α_{i+1} be constructed from α_i by replacing the node associated with this q by the tree $A(y_1, \ldots, y_n)$ where $y_j = z_j$ if $z_j \in \Sigma_0$ and $y_j = q_k$ if $z_j = x_\ell$ and x_ℓ appears in t associated with the state q_k. This is well-defined since each x_ℓ appears exactly once in t as T is both linear and nondeleting. Moreover, it is then clear from the construction of G(T) that $\alpha_{i+1} \overset{*}{\underset{T}{==>}} \mu_{i+1}$ such that $fr_Q(\mu_{i+1}) = \gamma_{i+1}$.

□

4.2.5 Parser-Transformers

This lemma shows that G(T) describes the target language of T. Moreover, since the second part of the the proof shows how to simulate a tree transducer without using the nonterminal symbols of its target alphabet, the lemma also proves that these symbols are irrelevant to the problem. We use this fact to combine the operations of parsing and bottom-up tree transduction into a single transformation that can invert the string mapping defined by a linear nondeleting t-fst. In particular, we construct a parser-transformer (pt) which applies rewriting rules to parse trees as they are constructed in a bottom-up parse. We view the input to a pt as a sequence of trees. Initially each of these trees consists of a single node. Then at each stage of the transformation a rule is applied to a subsequence of these trees and transforms that subsequence into a single tree with an associated state. The pt terminates when only a single tree with an associated accepting state remains.

Formally a parser-transformer is a 5-tuple $<\Sigma, \Delta, Q, q_d, R>$ where Σ is the input alphabet, Δ the ranked output alphabet, Q the set of states, $q_d \in Q$ is a distinguished state, and R is the set of rules. A pt is similar to a b-fst except that a rule is applied to a sequence of trees to produce a combined tree rather than to a node and its sons to produce a replacement for that node. The rules of a pt have the form

$$y_1, y_2, \ldots, y_k \to q(t)$$

where $k \geq 0$ and for each y_i either $y_i \in \Sigma$ or $y_i = r(x_j)$ for some $r \in Q$ and x_j a tree variable. A rule is applicable to a consecutive sequence of trees

$t_1 \cdots t_k$ if for $1 \le i \le k$, t_i matches y_i in the sense of a tree rewriting system. When a rule is applied, the sequence of trees $t_1 \cdots t_k$ that is matched is replaced by the single tree t' defined from the rule by substituting for each tree variable in t the appropriate tree that matched this variable. We denote the fact that a sequence of trees $\gamma_2=(t_1 \cdots t_k)$ can be derived from a sequence $\gamma_1=(t_1' \cdots t_\ell')$ using a pt P by $\gamma_1 \underset{P}{\Longrightarrow} \gamma_2$. The notation $\overset{*}{\underset{P}{\Longrightarrow}}$ ($\overset{+}{\underset{P}{\Longrightarrow}}$) then signifies zero (one) or more applications of rules of P in this manner. An example of a pt is given in figure 4.8a with an example transformation worked out in figure 4.8b.

There are several interesting restricted types of parser-transformers. A pt is called <u>linear</u> if no tree variable appears more than once in the right part of any rule. Similarly a pt is called <u>nondeleting</u> if each tree variable that appears in the left part of a rule also appears in the right part. We use parser-transformers that are both linear and nondeleting and denote these as <u>regular</u>. We also allow pts to have tree variables that occur more than once in the left part of a rule. Such a rule is applicable if and only if each occurrence of the same tree variable matches the same tree. A regular pt in which this situation does not occur, that is, where each tree variable appears exactly once in the left part of a rule, is called <u>simple</u>.

A parser-transformer is the combination of a parser and a b-fst. The purpose of the parser is to select the next rule to be applied as well as the sequence of trees to which it should be applied. The b-fst then acts as if this sequence of trees were the immediate subtrees of

$P = \langle \Sigma, \Delta, Q, q_e, R \rangle$

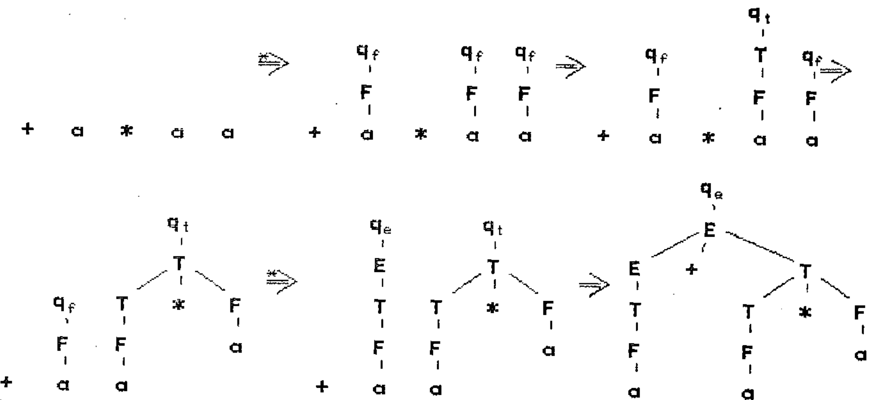
$\Sigma = \{a,+,*\}$

$\Delta_0 = \{a,+,*,<,>\}$, $\Delta_1 = \Delta_3 = \{E,T,F\}$

$Q = \{q_e, q_t, q_f\}$

R:
$+, q_e(x_1), q_t(x_2) \rightarrow q_e(E(x_1 + x_2))$
$q_t(x_1) \rightarrow q_e(E(x_1))$
$*, q_t(x_1), q_f(x_2) \rightarrow q_t(T(x_1 * x_2))$
$q_f(x_1) \rightarrow q_t(T(x_1))$
$q_e(x_1) \rightarrow q_f(F(<x_1>))$
$a \rightarrow q_f(F(a))$

a) A pt to invert the t-fst of figure 4.5



b) Mapping of '+a*aa' into 'a+a*a'

$+a*aa \Longrightarrow +q_f*aa \Longrightarrow +q_f*q_fa \Longrightarrow +q_f*q_fq_f \Longrightarrow +q_f*q_tq_f$
$\Longrightarrow +q_fq_t$
$\Longrightarrow +q_tq_t \Longrightarrow +q_eq_t$
$\Longrightarrow q_e$

c) The corresponding parse

Figure 4.8: An Example Parser-Transformer

some node and applies a production to form the subtree that would result in this case. This new tree then replaces the original sequence of trees. Determining the parse to be used is the most complicated part of the process. We usually assume that a parse is given to the parser-transformer which then checks to make sure that this parse is a proper one. This is equivalent to nondeterministically guessing which parse to use.

4.2.6 Using Parser-Transformers

Constructing a parser-transformer to invert a linear nondeleting t-fst is straightforward. Let $T=\langle\Sigma,\Omega,Q,q_d,R\rangle$ be such a t-fst. We define the corresponding pt $P=\langle\Delta,\Sigma,Q,q_d,R'\rangle$ where $\Delta=\Omega_0$ and where R' contains the rule $t_1,\ldots,t_n \rightarrow q(A(y_1\ldots y_k))$ if and only if the rule $q(A(y_1\ldots y_k))\rightarrow B(t_1\ldots t_n)$ is in R for some B. Then we claim

LEMMA 4.4: For a given linear nondeleting t-fst, T, this construction yields a simple pt.

Proof: It suffices to show that all the derived rules are of the proper form for a parser-transformer and that each x_i appears exactly once on each side of a derived rule. But all of this follows immediately from the construction and the fact that T is both linear and nondeleting and hence each x_i appears exactly once in each side of each of its rules.

□

We next show how to use a bottom-up parse of G(T) to direct a parser-transformer inverting the t-fst T. We first prove

LEMMA 4.5: Let T be a linear nondeleting t-fst. Let G(T) be the target grammar of T and let P be the pt constructed from T as above. Then a bottom-up parse of G(T) for some string s identifies a transformation of P on s.

Proof: We prove this inductively by demonstrating a one-to-one relationship between sequences of trees during a transformation of P and sentential forms during a derivation of G.

Let $\gamma=(t_1\ldots t_n)$ be a sequence of trees that arises during a transformation of P. We define $g(\gamma)$ as the string $s_1\ldots s_n$ where $s_i=t_i$ if t_i is one of the initial single node trees with no associated state, and $s_i=q_i$ if t_i is a derived tree associated with state q_i. Let γ_1 be some sequence of trees and suppose $\alpha \underset{G}{\Longrightarrow} g(\gamma_1)$ by some production of G derived from rule r of T. Then we claim that the rule of P derived from r can be applied to γ_1 to produce a sequence of trees γ_2 such that $g(\gamma_2)=\alpha$. This follows from the construction of P and G since the rules of P depend only on the state for derived trees and the node for initial trees when T is both linear and nondeleting.

Hence, if we have a derivation of s using G,

$$q=\alpha_0 \underset{G}{\Longrightarrow} \alpha_1 \underset{G}{\Longrightarrow} \cdots \underset{G}{\Longrightarrow} \alpha_n=s$$

there is a transformation using P

$$s \underset{P}{\Longrightarrow} \gamma_n \underset{P}{\Longrightarrow} \cdots \underset{P}{\Longrightarrow} \gamma_0$$

where $g(\gamma_i)=\alpha_i$. But then P accepts s by this transformation since γ_0 must be a single tree with the proper associated state.

□

This relation between a parse of G(T) and the operation of P is illustrated in figure 4.8c where the parse corresponding to the mapping of figure 4.8b is presented.

We next show that the pt constructed in this manner for a linear nondeleting t-fst is the proper one by

THEOREM 4.6: Let T be a linear nondeleting t-fst. Given <u>any</u> parse of a string s using G(T) we can use the simple pt P constructed from T to invert T, i.e. to find a tree t′ such that $T:t' \to t$ and $fr(t)=s$.

Proof: We show by induction that for any string s, if $s \overset{*}{\underset{P}{\Longrightarrow}} t$, then $t \overset{*}{\underset{T}{\Longrightarrow}} t'$ such that $fr(t')=s$. This holds trivially for the initial case where s is a single node and $t=t'=s$. It is then sufficient to show that if a rule r of P creates a tree $z=q(A(y_1 \cdots y_n))$ from a sequence of trees $z_1 \cdots z_m$, then the corresponding rule of T can be applied to z to yield a tree $B(z_1 \cdots z_m)$ for some B. But this is immediate from our construction of the rules of P from those of T.

□

These two lemmas give us a practical method of inverting a linear nondeleting t-fst T. In particular, given T we can easily construct G(T) and P as above. Then, for a string s we need only parse s using G(T) and then apply this parse to P to yield a tree in the domain of T that can be transformed into s. This shows that simple parser-transformers are powerful enough to invert such t-fsts. We can also show that they are not too powerful by

LEMMA 4.7: Every simple pt inverts some linear nondeleting t-fst.

Proof: We show that for every simple pt, P, there is a t-fst, T, such that the above construction applied to T yields precisely P. This is done by showing how to find a rule r of T for each rule r′ of P such that the construction applied to r yields r′. For any given rule of P,

$$y_1, \ldots, y_n \to q(t) \ ,$$

we construct the rule of T

$$q(t) \to A(y_1 \cdots y_n)$$

where A is arbitrary. This is clearly a valid rule of a linear nondeleting t-fst since P is simple and trivially satisfies the necessary conditions.

□

4.2.7 The Complexity Of Simple Parser-Transformers

Now that we have determined how to invert a linear nondeleting t-fst, we consider the complexity of the process. We first show that the problem is at least as hard as parsing an arbitrary context free grammar by

LEMMA 4.8: Inverting a linear nondeleting t-fst requires time $T_{parsing}$.

Proof: We show that we can define a t-fst, T, that is the identity on parse trees of an arbitrary context free grammar, G, and is undefined elsewhere. Thus some string s has an inverse s′ with respect to T if

and only if $s=s'$ and $s \in L[G]$ and hence the problem of context free recognition is reducible to that of inverting a linear nondeleting t-fst.

Let $G=\langle N,\Sigma,P,S\rangle$ and let $T=\langle\Delta,\Delta,N,S,R\rangle$ where Δ is the ranked alphabet with $\Delta_0=\Sigma$ and $\Delta_i=\{\bar{A} \mid A \in N,\ A\to\beta \in P \text{ and } |\beta|=i\}$ and where the rule $A(\bar{A}(y_1 \cdots y_m))\to\bar{A}(z_1 \cdots z_m)$ is in R if and only if $A\to c_1 \cdots c_m$ is in P such that for all $c_i \in \Sigma$, $y_i=z_i=c_i$, and for all $c_i \in N$, $y_i=x_i$ and $z_i=c_i(x_i)$. It is clear that T acts as the identity function on its domain. Moreover, T is only defined on parse trees of G since $A \in N$ occurs as a state if and only if the tree to which it is assigned is a derivation of A by G. This follows easily by induction from the construction of T. Finally we note that T is trivially linear and nondeleting.

□

This gives us a lower bound on the complexity of the problem. In the next section we improve this bound by showing it also holds for deterministic linear nondeleting t-fsts. We next show that this lower bound is achievable by

THEOREM 4.9: The problem of inverting a linear nondeleting t-fst T on a string s has time complexity $T_{parsing}(n)$ where $n=|s|$.

Proof: By lemma 4.8 it is enough to show how to invert such a t-fst in time $T_{parsing}$. We can find a parse of the string s by grammar G(T) in this time since G(T) is context free. By lemma 4.7 we can use this parse to direct a pt to invert T. Thus it suffices to show that the pt can be simulated in time $\le T_{parsing}$ given this directing parse.

Since any parse of G(T) for s will suffice, we can use a cycle-free one whose size is at worst O(n). Then, as each application of a pt rule requires only a constant amount of time, the total time required to simulate a pt is O(n). But $O(n) \le T_{parsing}$.

□

We can also determine such a directing parse while the parser-transformer is operating. In this mode the next rule to apply is determined by the next step of a deterministic bottom-up parsing algorithm.

4.3 SYNTAX-DIRECTED TRANSLATION SCHEMATA

In addition to linear nondeleting tree transducers we have proposed syntax-directed translation schemata (SDTS) as a model of simple semantics-specifying translations. Since these are equivalent to deterministic linear nondeleting t-fsts [125], theorem 4.9 indicates that inverting an SDTS might take time $T_{parsing}$. In this section we show that this is indeed the case with a proof based on the properties of an SDTS. The ideas and techniques we develop here will be used to later invert generalized syntax-directed translation schemata.

4.3.1 Inverting An SDTS

An SDTS consists of a pair of associated grammars and operates by determining a parse of the source string using the source grammar and

applying this parse with the target grammar to produce the desired translation. Because of the close association between the two grammars we can easily show that for any parse of either grammar there is an associated parse of the other. Thus, if s has a parse ρ of the source grammar and ρ applied to the target grammar yields the string s′, then ρ is a valid parse for s′ using the target grammar and ρ applied to the source grammar yields s again. Using this we can invert an SDTS by constructing another SDTS where the roles of the source and target grammars are reversed. Formally

Lemma 4.10: Let $T=\langle N,\Sigma,\Delta,R,S\rangle$ be an SDTS. Let R′ be derived from R by replacing each production of the form $A\to\alpha,\beta$ by the production $A\to\beta,\alpha$. Then $T'=\langle N,\Delta,\Sigma,R',S\rangle$ is an SDTS and moreover T′ inverts T.

Proof: To show that T′ is an SDTS it suffices to show that each rule in R′ is of the proper form. But this follows immediately from the construction. To show that T′ inverts T it suffices to prove that $s' \overset{*}{\underset{T'}{\Longrightarrow}} s$ implies $s \overset{*}{\underset{T}{\Longrightarrow}} s'$ is possible. But this follows from the symmetry between the source and target grammars since if $T{:}s\to s'$, then s and s′ have a common parse, s with the source grammar and s′ with the target grammar. But then finding this parse of s′ with the target grammar also determines a parse of s with the source grammar and, moreover, applying this parse to the source grammar must yield the string s. This is illustrated in figure 4.4b for the example of figure 4.4a.

□

A syntax-directed translation is a two step process, requiring first that a parse be determined and second that this parse be applied to generate a string. Similarly, inverting an SDTS involves the reversal of these two stages, first parsing the target string and then applying this parse to the source grammar. This concept of applying a parse of the target grammar to the source grammar is not restricted to SDTS. In the next chapter we show that this technique is also useful for inverting generalized syntax-directed translation schemata.

4.3.2 The Complexity Of Inverting An SDTS

We next consider the complexity of inverting an SDTS. Since the cost of applying a parse to a context free grammar requires only time proportional to the size of the parse, the complexity of finding an inverse is just that of finding a parse for the target grammar. In particular,

THEOREM 4.11: The problem of inverting an SDTS has time complexity $T_{parsing}$.

Proof: We need to show both that the problem requires time $T_{parsing}$ and that it can be done within this bound. The latter follows immediately by the above discussion since the target grammar is context free and $T_{parsing}$ is the maximum time required to parse a context free grammar.

To show the problem requires time $T_{parsing}$, let $G=\langle N,\Sigma,P,S\rangle$ be an arbitrary context free grammar. Consider the SDTS $T=\langle N,\Sigma,\Sigma,P',S\rangle$ where P' is derived by taking each production $A\rightarrow\alpha$ of P and adding the rule $A\rightarrow\alpha,\alpha$ to P'. It is clear that T is simply the identity on L[G] and is undefined elsewhere. Thus T has an inverse s′ for a string s if and only if $s=s'$ and $s\in L[G]$. But this shows that the problem of context free recognition is reducible to that of inverting an SDTS and hence that the latter requires $T_{parsing}$.

□

Because of the relationship between SDTS and t-fsts it follows that

COROLLARY 4.12: The problem of inverting a deterministic linear nondeleting t-fst requires time $T_{parsing}$.

□

This is an improvement over lemma 4.8 of the previous section.

4.4 SUMMARY OF BASIC RESULTS

This completes our discussion on inverting simple translations. The important results are summarized in theorems 4.6, 4.9 and 4.11 in that the problems of inverting both SDTS and linear nondeleting t-fsts have time complexity $T_{parsing}$. More important however are the concepts and methods that we used. To invert a t-fst we introduced the model of a parser-transformer that both finds and transforms a parse tree of the target language. To invert an SDTS we showed that it was sufficient to

simply determine a parse of the target grammar. As we consider more general transformations in the following chapters, we expand the model of a parser-transformer and consider parsing grammars that are no longer context free. In either case the processes will be simpler and more understandable since they will be based on the simple models we have introduced in this chapter.

CHAPTER 5

INVERTING GENERAL TREE TRANSDUCERS

If the simple transducers discussed in the previous chapter were powerful enough to describe the semantics of most computer languages, then practical automatic programming would be a reality. Unfortunately this is not the case. Most semantic representations cannot be described as the range of one of these simple mappings since such representations are not context free while the ranges are. In order to consider a more realistic semantic representation we must consider more powerful mappings. In this chapter we do just that, extending our simple translations by first adding the ability to delete subtrees and then the ability to copy subtrees. For these more sophisticated mappings we determine when the inverse can be computed tractably and when it cannot be.

5.1 DELETION

Of all the restrictions we placed on tree transducers in the previous chapter, nondeletion is the least limiting. In this section we show that deleting tree transducers are no more powerful than nondeleting ones and hence that any results that apply to the one also apply to the other. We do this by showing how to effectively construct a

nondeleting t-fst from a deleting one without substantially increasing the complexity.

A tree transducer is called deleting if it contains a rule with a tree variable in the left part that does not appear in the right part. When such a rule is applied, the subtree represented by this tree variable does not appear in the transformed tree, and, because of the top-down nature of the tree transducer, no state is ever assigned to this subtree and hence no checking is done on its contents. However, in the case of a nonlinear tree transducer, the subtree being deleted may have been copied and that copy can still be checked. This allows deletion to simplify the description of a t-fst as illustrated in figure 5.1. We show that this is all it can do by

THEOREM 5.1: Given a deleting t-fst, T, there is a nondeleting t-fst T′ such that $T:t \to t'$ if and only if $T':t \to t'$ and the size of T′ is linear in the size of T. Moreover, T′ is linear if and only if T is linear and T′ is deterministic if and only if T is deterministic.

Proof: Let $T=\langle \Sigma,\Delta,Q,q_d,R \rangle$. We construct $T'=\langle \Sigma,\Delta',Q',q_d,R' \rangle$ from T by first adding a new state and a set of rules and then modifying all the deleting rules of T. Let the new state be q_λ. Then we add to R′ the rules

$$q_\lambda(a) \to \lambda$$

for all $a \in \Sigma_0$ and

$$q_\lambda(A(x_1 \ldots x_k)) \to A(q_\lambda(x_1) \ldots q_\lambda(x_k))$$

for all $A \in \Sigma_k$ such that $q(A(y_1 \ldots y_k)) \to t$ is a rule in R for some $q \in Q$ and $y_1 \ldots y_k$ arbitrary. Clearly $q_\lambda(t) \overset{*}{\Longrightarrow} t'$ such that $fr(t')=\lambda$ for all $t \in dom(T)$.

$T = \langle \Sigma, \Delta, Q, q_0, R \rangle$

$\Sigma_0 = \{a, b, \lambda\}$, $\Sigma_1 = \{S, A\}$, $\Sigma_2 = \{A\}$

$\Delta_0 = \{1, \$, \lambda\}$, $\Delta_1 = \Delta_2 = \Delta_3 = \{A\}$

$Q = \{q_0, q_1, q_2\}$

R: $q_0(S(x_1)) \rightarrow A(q_1(x_1)\$q_2(x_1))$

$q_1(A(x_1 a)) \rightarrow A(1q_1(x_1))$

$q_2(A(x_1 a)) \rightarrow A(1q_2(x_1))$

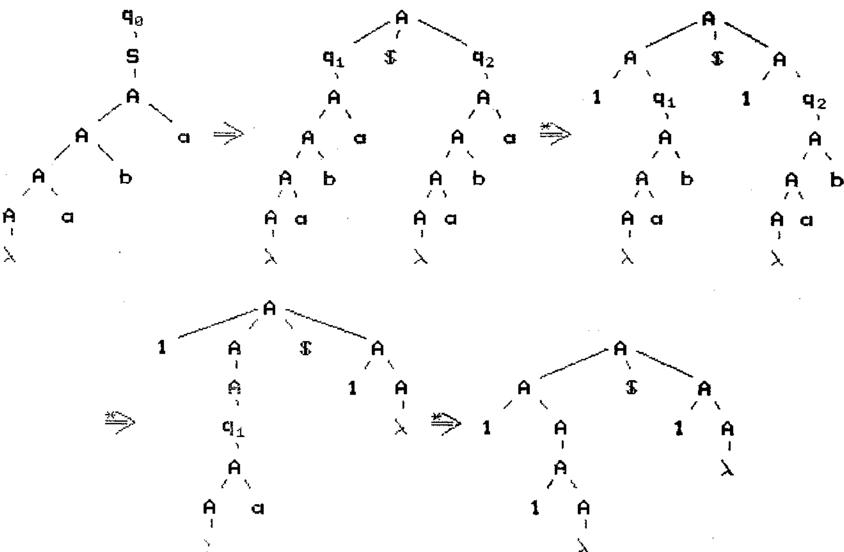
$q_1(A(x_1 b)) \rightarrow A(q_1(x_1))$

$q_2(A(x_1 b)) \rightarrow A(\lambda)$

$q_1(A(\lambda)) \rightarrow A(\lambda)$

$q_2(A(\lambda)) \rightarrow A(\lambda)$

a) Deleting t-fst to map $\{\langle \alpha, 1^n \$ 1^m \rangle \mid \alpha \in \{a,b\}^*$ contains n a's and m a's after the last b$\}$.



b) Mapping of 'aba' into '11$1'

$T' = \langle \Sigma, \Delta', Q', q, R' \rangle$

$\Delta'_0 = \{1, \$, \lambda\}$, $\Delta'_1 = \{A, S\}$, $\Delta'_2 = \Delta'_3 = \{A\}$

$Q' = \{q_0, q_1, q_2, q_\lambda\}$

R: $q_0(S(x_1)) \rightarrow A(q_1(x_1)\$q_2(x_1))$

$q_1(A(x_1 a)) \rightarrow A(1q_1(x_1))$

$q_2(A(x_1 a)) \rightarrow A(1q_2(x_1))$

$q_1(A(x_1 b)) \rightarrow A(q_1(x_1))$

$q_2(A(x_1 b)) \rightarrow A(q_\lambda(x_1)\lambda)$

$q_1(A(\lambda)) \rightarrow A(\lambda)$

$q_2(A(\lambda)) \rightarrow A(\lambda)$

$q_\lambda(S(x_1)) \rightarrow S(q_\lambda(x_1))$

$q_\lambda(A(x_1 x_2)) \rightarrow A(q_\lambda(x_1)q_\lambda(x_2))$

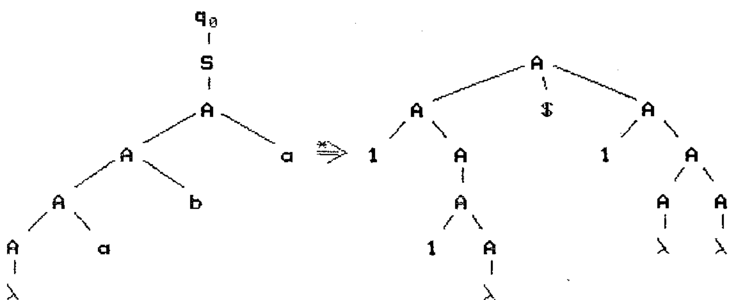
$q_\lambda(A(x_1)) \rightarrow A(q_\lambda(x_1))$

$q_\lambda(a) \rightarrow \lambda$

$q_\lambda(b) \rightarrow \lambda$

$q_\lambda(\lambda) \rightarrow \lambda$

c) Nondeleting version of this t-fst



d) The mapping with T'

Figure 5.1: A Deleting t-fst

Now for each rule of T where deletion occurs, we add to the right part of the corresponding rule of T′ the pair $q_\lambda(x_i)$ for each deleted tree variable x_i. Since $q_\lambda(t)$ yields no string and accepts any tree and since a deleted variable is not considered any further and thus can represent any tree consistent with the rest of the parse, it follows that T and T′ compute the same translation and that the number of rules added to T′ is at worst linear in the size of T. Moreover, since the construction does not introduce or eliminate nonlinearity, T′ is linear if and only T is; and, since there is only one rule applicable for any subtree involving the state q_λ, T′ is deterministic if and only if T is. The nondeleting version of the t-fst of figure 5.1a appears in figure 5.1c as an example of this construction.

□

From this theorem we see that our previous results hold for deleting t-fsts as well as nondeleting ones. Thus

COROLLARY 5.2:

a) Let T be a linear t-fst. Then there is a b-fst that inverts T.
b) Let T be a linear t-fst. Then there is a context free grammar G and a simple pt P such that P will compute the string inverse of T given any parse of G.
c) The problem of inverting a linear t-fst has time complexity $T_{parsing}$.

□

5.2 NONLINEARITY

While it can be shown that deletion does not increase the power of a t-fst, nonlinearity does. Whereas linear tree transducers have simple context free ranges, figure 5.2 gives a trivial example of a deterministic nonlinear t-fst that defines the mapping $\{<\alpha,\alpha\alpha> \mid \alpha \in \{a,b\}^*\}$ whose range is not context free [61]. This shows that nonlinear tree transducers are strictly more powerful than linear ones. It also suggests that inverting a nonlinear t-fst is more difficult than inverting a linear one.

$T = <\Sigma,\Delta,Q,q,R>$

$\sigma_0 = \{a,b,\lambda\}, \quad \Sigma_1 = \{S,A\}, \quad \Sigma_2 = \{A\}$

$\Delta_0 = \{a,b,\lambda\}, \quad \Delta_1 = \{A\}, \quad \Delta_2 = \{S,A\}$

$Q = \{q\}$

R: $q(S(x_1)) \to S(q(x_1)q(x_1))$

$q(A(ax_1)) \to A(aq(x_1))$

$q(A(bx_1)) \to A(bq(x_1))$

$q(A(\lambda)) \to A(\lambda)$

Figure 5.2: A Nonlinear t-fst with a Range that is not Context Free

The lack of a context free target language is the principal obstacle to inverting nonlinear tree transducers. In particular, for a nonlinear t-fst T we can no longer use any parse of the context free grammar G(T) to direct the inverse process. We show however that the target language is actually a subset of L[G(T)] and reduce the problem to finding a parse of G(T) that defines an actual target tree of the forward translation. There are two straightforward methods of attacking this reduced problem. We could first assume that the parse is proper, find the inverse, and then apply the forward translation to this inverse to check if the result is the initial string. This method presents several difficulties, especially in the case where the forward translation is one-many. The second method is to restrict proper parses to those in the domain of the inverse tree transformation of T. This is the method we use in this section. We introduce the notion of an extended b-fst whose domain is precisely the range of a nonlinear t-fst. We then use this notion to build a parser-transformer that inverts a nonlinear t-fst. Our main result is that the problem of inverting a nonlinear t-fst is intractable if the t-fst is arbitrary or nondeterministic.

5.2.1 Describing The Target Language

With a linear t-fst, T, we showed how to construct the context free grammar G(T) and proved that it actually described the target language. The situation is somewhat more complex for nonlinear t-fsts. The same construction still allows us to define, given a nonlinear t-fst T, the

grammar G(T) which by lemma 4.2 is still context free. While this grammar no longer defines the target language of T, it does describe a superset of this language. In particular

LEMMA 5.3: Let $T=\langle N,\Sigma,\Delta,R,S\rangle$ be a nonlinear t-fst and let G(T) be the target grammar derived from T. Then $L_t(T)\subseteq L[G(T)]$.

Proof: To see that $L_t(T)\subseteq L[G(T)]$ we note that the only difference from the case where T is linear is that there can be a tree variable, x, that appears several times in the right side of a rule, say as $q_1(x)$, $q_2(x)$, ..., $q_k(x)$. But then for each $1\leq i\leq k$ the frontier of the tree derived from $q_i(x)$ is a string of the language derivable from the nonterminal q_i in G(T). Hence the actual target language is a subset of L[G(T)].

□

We have to determine what restrictions must be placed on a parse of G(T) so that it can define only trees in the range of T and thus successfully direct a parser-transformer inverting T. When a nonlinear rule is applied in the forward translation a subtree is replicated and the resultant copies are processed independently. When inverting the tree mapping at this point, it is enough to insure that the inverses of each of these subtrees are identical since then they could be derived from the same original subtree. This check can be done with a parser-transformer where these inverse trees are computed at the same time as the parse and thus the condition can be tested before a production is applied. Later we show that a pt that makes this check will succeed if

the input is invertible. We also show that it is in general difficult to find a proper directing parse.

5.2.2 Extended b-fsts

While we eventually use a parser-transformer to invert a nonlinear t-fst T, we again want to consider parsing and tree transduction separately. We first consider the problem of actually inverting the tree mapping of T. We assume that we are given a proper parse and hence a tree in the range of T. We then define an extension of a b-fst that computes the inverse of the given tree and insures that the constraints imposed by nonlinearity are met.

Our previous construction took the t-fst, $T=\langle\Sigma,\Delta,Q,q_d,R\rangle$, and formed the b-fst $B=\langle\Delta,\Sigma,Q,q_d,R'\rangle$ where R' contained the rule $t'\to q(t)$ if and only if R contained the rule $q(t)\to t'$. We use the same construction here but the result is no longer a b-fst since a rule can be constructed that has a tree variable appearing more than once in the left part. We call such a tree rewriting system an extended b-fst (eb-fst) and say that a rule is applicable if and only if all of the trees that match occurrences of a single tree variable are the same. This is exactly the restriction implied by the nonlinear forward translation. Figure 5.3 provides an example of an extended b-fst derived from a nonlinear t-fst.

We show that the eb-fst of this construction can actually invert the nonlinear t-fst by

FIGURE 5.3

$T = \langle\Sigma,\Delta,Q,r_e,R\rangle$

$\Sigma_0=\{+,*,a,<,>\}, \quad \Sigma_1=\Sigma_3=\{E,T,F\}$

$\Delta_0=\{+,*,a,<,>,1\}, \quad \Delta_1=\Delta_3=\{E,T,F\}, \quad \Delta_9=\{T\}$

$Q=\{q_e,q_t,q_f,r_e,r_t,r_f\}$

R:
$q_e(E(x_1+x_2)) \to E(q_e(x_1)+q_t(x_2))$
$q_e(E(x_1)) \to E(q_t(x_1))$
$q_t(T(x_1*x_2)) \to T(q_t(x_1)*q_t(x_2))$
$q_t(T(x_1)) \to T(q_f(x_1))$
$q_f(F(a)) \to F(a)$
$q_f(F(<x_1>)) \to F(<q_e(x_1)>)$
$r_e(E(x_1+x_2)) \to E(r_e(x_1)+r_t(x_2))$
$r_e(E(x_1)) \to E(r_t(x_1))$
$r_t(T(x_1*x_2)) \to T(<q_t(x_1)*r_f(x_2)+q_f(x_2)*r_t(x_1)>)$
$r_t(T(x_1)) \to T(r_f(x_1))$
$r_f(F(a)) \to F(1)$
$r_f(F(<x_1>)) \to F(<r_e(x_1)>)$

a) A nonlinear t-fst that takes an expression into its derivative

$B = \langle\Delta,\Sigma,Q,q_d,R'\rangle$

R:
$E(q_e(x_1)+q_t(x_2)) \to q_e(E(x_1+x_2))$
$E(q_t(x_1)) \to q_e(E(x_1))$
$T(q_t(x_1)*q_f(x_2)) \to q_t(T(x_1*x_2))$
$T(q_f(x_1)) \to q_t(T(x_1))$
$F(a) \to q_f(F(a))$
$F(<q_e(x_1)>) \to q_f(F(<x_1>))$
$E(r_e(x_1)+r_t(x_2)) \to r_e(E(x_1+x_2))$
$E(r_t(x_1)) \to r_e(E(x_1))$

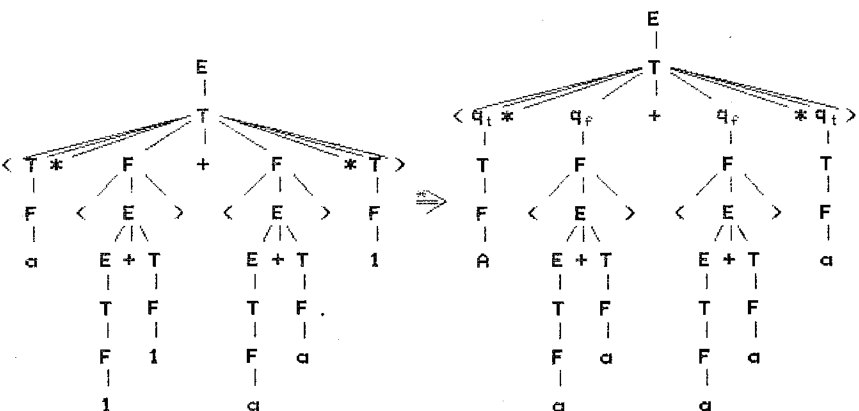
$T(<q_t(x_1)*r_f(x_2)+q_f(x_2)*r_t(x_1)>) \rightarrow r_t(T(x_1*x_2))$

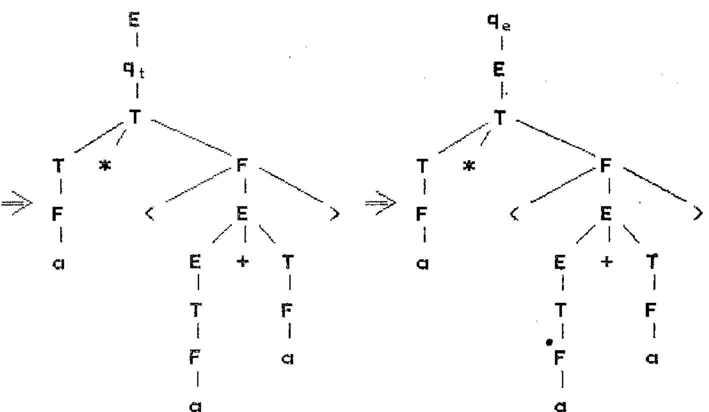
$T(r_f(x_1)) \rightarrow r_t(T(x_1))$

$F(1) \rightarrow r_f(F(a))$

$F(<r_e(x_1)>) \rightarrow r_f(F(<x_1>))$

b) The extended b-fst to invert this t-fst





c) Example mapping of B

Figure 5.3: A Sample Extended b-fst

LEMMA 5.4: Let $T=<\Sigma,\Delta,Q,q_d,R>$ be a nonlinear t-fst and let B be the extended b-fst derived from T as above. Then for trees t and t′, $B:t\rightarrow t'$ if and only if $t\epsilon range(T)$ and $T:t'\rightarrow t$.

Proof: By induction on the mappings defined by T and B it suffices to show for trees t and t′ that $t \underset{T}{\Rightarrow} t'$ if and only if $t' \underset{B}{\Rightarrow} t$ since this shows that $t \overset{*}{\underset{T}{\Rightarrow}} t'$ if and only if $t' \overset{*}{\underset{B}{\Rightarrow}} t$ and hence that B is defined over the range of T and that B finds an inverse of T when it is defined. Let $q(t)\rightarrow A(y_1 \cdots y_n)$ be a production of T where each $y_i \epsilon \Delta_0$ or $y_i = q_i(x_j)$. The derived rule is then $A(y_1 \cdots y_n)\rightarrow q(t)$. It clearly suffices to show that if several y_i's refer to the same tree variable then the derived rule can be applied if and only if the trees represented by this variable are identical. But the definition of an extended b-fst insures just that.

□

5.2.3 Parser-Transformers For Nonlinear t-fsts

We can use this notion of an extended b-fst to construct a parser-transformer that can invert a nonlinear t-fst. Let $T=<\Sigma,\Delta,Q,q_d,R>$ be a nonlinear t-fst. Then we construct $P=<\Omega,\Sigma,\Gamma,q_d,R'>$ where $\Omega=\Delta_0$ and R′ is a set of rules derived from R. A rule,

$$y_1,y_2,\ldots,y_k\rightarrow q(t)$$

is in R′ if and only if R contains the rule $q(t)\rightarrow t'$ such that either

1. $t' \in \Delta_0$, $k=1$ and $y_1=t'$; or

2. $t'=A(y_1 \cdots y_k)$ for some $A \in \Delta_k$.

This is essentially the same construction we used to create a simple pt from a linear t-fst. We can show

LEMMA 5.5: Let T be a nonlinear t-fst and let P be the pt constructed as above. Then P is a regular pt.

Proof: Without loss of generality we can assume that T is nondeleting. Then it suffices to show that the rules of P have the proper form, that is, that each tree variable appearing on the left side of a rule appears exactly once on the right side and that no tree variable appears on the right side without also appearing on the left. The right side of a pt rule is simply the left side of a t-fst rule and hence each variable that appears there appears exactly once. Moreover, since T is nondeleting, every variable that appears in the right part of a rule of P must also appear in the left part.

□

We want to show that the regular pt constructed this way actually computes the string inverse of a nonlinear t-fst T. We first show that some parse of G(T) can be used to direct this computation. Since each production of G(T) and each rule of the pt constructed from T is derived from a single rule of T, there is an obvious association between the rules of P, the productions of G(T), and the rules of T. Using this association we show

LEMMA 5.6: Let T be a nonlinear t-fst, P the pt constructed from it, and G(T) its target grammar. If $T:t \to t'$ for trees t and t′ and $s=fr(t')$ then there is a parse of s using G(T) which when applied to P for s yields the tree t.

Proof: Let the transformation yielding t′ be

$$t=t_0 \Longrightarrow t_1 \Longrightarrow \ldots \Longrightarrow t_n=t'$$

and let $\alpha_i=fr_Q(t_i)$ for $0 \le i \le n$. We claim that $t \overset{*}{\underset{T}{\Longrightarrow}} t_i$ if and only if $\alpha_0 \overset{*}{\underset{G}{\Longrightarrow}} \alpha_i$ and hence that $t \overset{*}{\underset{T}{\Longrightarrow}} t'$ if and only if $s \in L[G(T)]$. Clearly this holds for the case $i=0$. We use induction to prove that it holds for $1 \le i \le n$ by showing that if $t_i \underset{T}{\Longrightarrow} t_{i+1}$ using some production of T, then the corresponding production of G(T) takes $\alpha_i \underset{G}{\Longrightarrow} \alpha_{i+1}$. But this follows immediately from lemma 4.5 since nonlinearity does not affect the state-frontier.

□

Using this lemma we can show that a regular pt will invert a nonlinear t-fst by

THEOREM 5.7: Let T be a nonlinear t-fst. Then there is a context free grammar G and a regular parser-transformer P such that P computes the string inverse of T given <u>some</u> parse of the target string using G.

Proof: Let P be the pt constructed from T and let G be G(T). Then lemma 5.6 shows that P is defined over the range of T and it suffices to show that when T is defined P can invert it. This follows by induction from the fact that if $\gamma \underset{P}{\Longrightarrow} \gamma'$ for sequences of trees γ and γ', then T can take a tree whose state-frontier corresponds to the sequence γ' into one with sequence γ. If the production of P used here is

$y_1,\ldots,y_m \to q(t)$, then it is clear that the corresponding production of T, $q(t) \to A(y_1 \cdots y_m)$, modifies the state-frontier correctly.

□

We can also show that in some sense a regular pt is the proper model for inverting a nondeleting t-fst. In particular

THEOREM 5.8: For any regular pt P, there is a nondeleting t-fst T such that P computes the string inverse of T.

Proof: Let $P=\langle \Delta,\Sigma,Q,q_d,R\rangle$ and construct the t-fst $T=\langle \Sigma,\Delta',Q,q_d,R'\rangle$ where $\Delta_0'=\Delta$, $\Delta_i'=\{\bar{q} \mid q\in Q \text{ and } y_1,\ldots,y_i \to q(t) \in R\}$, and R′ is derived from R by

$$q(t)\to\alpha \in R' \text{ if and only if } \alpha\to q(t) \in R,$$

and

$$q(t)\to\bar{q}(y_1\cdots y_n) \in R' \text{ if and only if } y_1,\ldots,y_n\to q(t) \in R.$$

It is clear that T is a nondeleting t-fst. Thus it is enough to show that if we construct the pt that inverts T we get precisely P. But this again follows immediately from the construction.

□

5.2.4 Complexity Results For Nonlinear t-fsts

Theorem 5.7 differs from the corresponding theorem for linear tree transducers in one significant detail. While a linear tree transducer is invertible for a string s given <u>any</u> parse of s using G(T), a nonlin-

ear one is only invertible for <u>some</u> parse of s using G(T). Thus it is no longer enough to determine a single parse of G(T). Instead we must consider all the parses of G(T) for s and check each one to see if an inverse of s can be found.

While a string can in theory have an infinite or exponential number of parses under some context free grammar, it is more common in practice for it to have only a small number. In this case it would be practical to test all of these parses to determine if an inverse to the string exists. Let $A_G(n)$ represent the maximum number of parses of any string of length n using the context free grammar G. Then we can show

LEMMA 5.9: Let T be a nonlinear t-fst and let G=G(T) be its derived target grammar. Then given a string s of length n we can determine a tree t such that $T:t\to t'$ and $fr(t')=s$ in time $O(n^2 A_G(n))+T'_{parsing}$.

Proof: We assume that the grammar is first parsed with a universal algorithm such as Earley's so that all parses are available. This takes time $T'_{parsing}$. Then for each of the at most $A_G(n)$ parses that are discovered, we use the pt P derived from T to attempt to find an inverse. It is thus enough to show that this pt can operate in time $O(n^2)$.

If $A_G(n)$ is infinite, the theorem is trivial. If it is bounded then G must be cycle-free for all strings of length n. But then any parse tree of s can only have O(n) nodes. As a pt does one operation for each node of this tree it is thus sufficient to show that each step of a pt translation takes time at worst O(n). The most that a single step can do though is to compare a constant number of trees for equal-

ity. As each such tree has at worst O(n) nodes, this takes time at worst O(n).

□

While this result may be useful in some cases, many practical target grammars are ambiguous or contain cycles and thus have an exponential or infinite number of parses. Moreover, the question of whether a given grammar has a polynomial bounded A_G is undecidable [61] and hence even for a specific case we may not know whether testing all parses is a tractable or intractable approach. We would like an algorithm that does not have to consider all parses and that would run in polynomial time. Unfortunately we will show that we are not likely to find such an algorithm.

We first present an upper bound on the complexity of inverting a nonlinear t-fst. Although this upper bound is not particularly practical, we later show that it is the best we can do for a general t-fst.

LEMMA 5.10: The problem of inverting a t-fst is in the class PSPACE.

Proof: The target language of a t-fst is context sensitive [15] and thus determining if a string is in the target language is reducible to determining membership in a context sensitive language. But this latter problem is in the class PSPACE [4] and hence so is that of inverting a t-fst.

□

We can improve this upper bound for those t-fsts that are useful for expressing semantics. These insure that the information content of the target string is at least as great as that of the source string. In particular, a t-fst T is called <u>regular</u> if its domain is cycle-free and if there is a constant c dependent only on T such that if $T: t \to t'$ then $|fr(t')| \geq c|fr(t)|$. The first condition insures that a parse of the source string can only describe the source string and cannot code other information in the cycles of the grammar. The second condition insures that the size of the target string is at least linear in the size of the source string.

This restriction is a natural one for this class of semantic mappings. Most grammars that are used to describe programming languages, the source of a semantic mapping, are cycle-free. Moreover, it is easy to take a context free grammar containing cycles and replace it with an equivalent cycle-free one. Aho and Ullman have also shown [10] that if the size of the output frontier of a deterministic t-fst T is not at least linear in the size of its input frontier, then the range of T is finite. Such a mapping would not be useful for specifying semantics.

For the class of regular t-fsts we can show the improved result

LEMMA 5.11: The problem of inverting a regular t-fst T is in the class NP.

Proof: Based on the nondeterministic model of a regular parser-transformer already presented and the proof of lemma 5.9 which shows that if we guess a parse we can check it in polynomial time, it suffices to show that the size of a parse, if one exists, is polynomial in the

size of the target string. Suppose $T: t \to t'$. Let $\ell' = |fr(t')|$ and $\ell = |fr(t)|$. Then since T has a cycle-free domain grammar by assumption, the size of an accepting parse is $O(\ell)$. But, as T is regular, $O(\ell) \le O(\ell')$.

□

We next show that the upper bounds of lemmas 5.10 and 5.11 are tight. We first show

LEMMA 5.12: The problem of inverting a fixed regular t-fst, T, is NP-hard.

Proof: We show that the NP-complete problem of 3-satisfiability can be reduced to the problem of inverting a fixed regular t-fst by presenting a fixed t-fst T and a construction that gives a string s for a given 3-satisfiability problem such that T inverts s if and only if the given problem has a positive solution.

Let $T = \langle \Sigma, \Omega, Q, q_s, P \rangle$ where

$\Sigma_0 = \{1, \$, t, f\}$, $\Sigma_1 = \{S, B\}$, $\Sigma_2 = \{A, B\}$

$\Omega_0 = \{0, \$, a, b, c\}$, $\Omega_1 = \{A\}$, $\Omega_2 = \{B, S\}$, $\Omega_3 = \{A, B\}$, $\Omega_4 = \Omega_5 = \{B\}$

$Q = \{q_s, q_a, q_b\}$

and where P contains the rules

$q_s(S(x_1)) \to S(q_a(x_1)\$)$
$q_a(A(1x_1)) \to A(q_a(x_1)\$q_a(x_1))$
$q_a(A(\$x_1)) \to A(q_b(x_1))$
$q_b(B(tx_1)) \to B(0q_b(x_1)a)$

$q_b(B(tx_1)) \to B(000q_b(x_1)b)$
$q_b(B(tx_1)) \to B(00q_b(x_1)c)$
$q_b(B(fx_1)) \to B(0q_b(x_1)a)$
$q_b(B(fx_1)) \to B(000q_b(x_1)c)$
$q_b(B(fx_1)) \to B(00q_b(x_1)b)$
$q_b(B(\$)) \to B(0\$)$
$q_b(B(\$)) \to B(00\$)$
$q_b(B(\$)) \to B(000\$)$.

It is not hard to verify that the domain grammar of this translation is simply

$S \to A$
$A \to 1A$
$A \to \$B$
$B \to tB$
$B \to fB$
$B \to \$$.

This is an unambiguous regular grammar describing strings of the form $1^*\$(t \cup f)^*\$$, that is, a sequence of 1's followed by a sequence of t's and f's.

Let S be a 3-satisfiability problem with clauses $D_1, \ldots, D_r$ over the variables $u_1, \ldots, u_m$ and their complements. We can assume, without loss of generality, that each clause contains exactly 3 distinct variables and that $r = 2^k$ for some integer k. For each clause D_i we construct a string $s_i = 0^{m+7}\$\alpha^R$ where α^R is the reverse of α and where $|\alpha| = m$ and the jth component of α, α_j, is determined from u_j as

1. if $u_j \in D_i$ then $\alpha_j = b$;

2. if $\bar{u}_j \in D_i$ then $\alpha_j = c$;

3. otherwise $\alpha_j = a$.

It is clear from our assumptions on the clauses that each s_i is well defined and moreover contains exactly three symbols that are either b or c. We construct the string s to be the concatenation of these s_i, that is, $s = s_1 s_2 \ldots s_r$. It is clear that s describes the set of clauses $D_1, \ldots, D_r$ and that the size of s is $O(rm)$ and hence is polynomial in the size of the problem S.

Then we claim

CLAIM: There is a tree t such that $T: t \to t'$ with $fr(t') = s$ if and only if $D_1, \ldots, D_r$ are simultaneously satisfied.

It should be clear that proving this claim proves the theorem since given any 3-satisfiability problem we can form the string s in polynomial time and then, if we could also determine if an inverse to s exists in polynomial time, we would be able to determine if the set of clauses is satisfiable in polynomial time.

We first show that if there is an assignment of truth values such that all the clauses are satisfiable, then there is a tree t which establishes the claim. Suppose we have such an assignment. Let S and s be as before and let $r = 2^k$. Then let $\beta = 1^k \$ \gamma \$$ where $|\gamma| = m$ and the ith component of γ, γ_i, is t if u_i is assigned the value true and is f if u_i

is assigned the value false. Let t be the unique parse tree of β with the domain grammar of T. Then we need to show it is possible that that $T: t \to t'$ where $fr(t') = s$. T first takes the series of k ones in t and constructs $2^k = r$ copies of subtrees of the form $q_a(A(\$B(zx_1)))$ for $z \in \{t,f\}$. These in turn are transformed into r copies of $A(q_b(B(zx_1)))$, each of which is processed independently. It is then enough to show that the ith such subtree can be mapped into the string s_i.

Every translate of $q_b(B(zx_1))$ is of the form $0^{\ell} \$ \alpha$ where α is an arbitrary member of $(a \cup b \cup c)^m$. Thus we need only show that for the particular α of s_i we can form an initial string of m+7 zeros. Exactly m-3 of the elements of α must be a and hence contribute m-3 zeros to the translate. The remaining elements are either b if the corresponding variable is used in a positive sense in D_i or c if the corresponding variable is used in a negative sense. If we assign the value true to this variable we add three zeros for b and two for c, while if we assign false to it we add two zeros for b and three for c. Thus we add exactly six zeros if the clause is not satisfied and seven, eight or nine zeros if it is. Finally the translation of $B \to \$$ can add either one, two or three more zeros. Then it is clear that at most m+6 zeros can be present if the clause is not satisfied while m+7 zeros can be achieved whenever the clause is satisfied. Thus the translation is possible when the set of clauses is satisfiable. An example of such a translation is given in figure 5.4.

FIGURE 5.4

Variables: u_1, u_2, u_3

Clauses: $(u_1 \vee \bar{u}_2 \vee u_3)$, $(u_1 \vee u_2 \vee \bar{u}_3)$

Problem Coding: 0000000000$bcb0000000000$cbb
Solution: u_1=t, u_2=u_3=f

Solution Coding: 1tff

a) A 3-satisfiability problem and its codings

b) Mapping of the given solution into the problem

Figure 5.4: Example of Inversion Solving a 3-satisfiability Problem

It remains to show the other direction of the claim, that if an inverse translation (and hence a forward translation) exists, there is a satisfiable assignment. We first note that the frontier of the inverse of s must have the form $1^k\$\beta\$$ where β is a string of the form $(t \cup f)^m$. We claim that the corresponding assignment, where u_i is assigned the value true if and only if β_i is t, satisfies all the clauses. Suppose not. Then there is a translation of a subtree with frontier $\beta\$$ of the form $B(zx_1)$ into some s_i corresponding to a clause D_i that is not satisfied. By our previous arguments, we see that this translate must have the form $0^\ell\$\gamma\$$ where γ is determined by s_i and it is enough to show that ℓ cannot equal m+7. But ℓ=m+7 is possible only if D_i is satisfiable. This in turn shows that the claim holds and proves the theorem.

□

For an arbitrary nonregular t-fst we improve this result by

LEMMA 5.13: The problem of inverting an arbitrary deterministic t-fst is PSPACE-hard.

Proof: Let s be a positive integer and let M be the linear bounded automata that recognizes the hardest context sensitive language. Then it is known that the problem of determining if a string of length s is accepted by M, i.e. is in L[M], is PSPACE-complete. To prove the lemma we reduce this recognition problem to that of inverting a deterministic t-fst by defining a t-fst whose operation in effect simulates a linear bounded automata. Such a t-fst takes as input the initial string and the sequence of operations of the LBA and yields a string composed of

the initial string and a constant string if and only if the LBA accepts the initial string using the given sequence of operations. This t-fst then has an inverse if and only if there is a sequence of operations whereby the LBA can accept the initial string.

Let $L[M]$ be a language over the alphabet Ω. By definition M has rules of the form $<\sigma,a,b,n,\sigma'>$ where σ and σ' are members of the finite set of states P, $a,b\in\Omega$, and $n\in\{-1,0,1\}$. Moreover, without loss of generality we can assume that M has an initial state σ_0 and accepts only if it enters the final state σ_f with the tape containing all $0\in\Omega$. Let the set of rules of M be $R=\{r_0,r_1,\dots,r_n\}$.

Given M and s as above we construct the t-fst $T=<\Sigma,\Delta,Q,q_d,R>$ where

1. $\Sigma_0=\Omega\cup\{\$,z\}$, $\Sigma_1=\{R_i \mid 0\le i\le|R|\}\cup\{S\}$, $\Sigma_2=\{B,C\}$;

2. $\Delta_0=\Omega\cup\{\$,z\}$, $\Delta_1=\Delta_2=\Delta_3=\{A\}$;

3. $Q\subseteq(\{0,\dots,s-1\}\times P\times\Omega)\cup\{q_i \mid 0\le i<s\}\cup\{p_{\omega,i} \mid \omega\in\Omega,\ 0\le i<s\}\cup\{q_d\}$.

The domain grammar of T is a subset of the context free grammar $<\Sigma_0,N,R,S>$ where $N=\Sigma_1\cup\Sigma_2$ and R contains the productions

$S\to B$	
$B\to aB$	for all $a\in\Omega$
$B\to aC$	for all $a\in\Omega$
$C\to \$R_i$	for $0\le i\le n$
$R_i\to R_j z$	for $0\le i,j\le n$
$R_i\to \$$	for $0\le i\le n$.

Intuitively this domain consists of a copy of the source string α followed by \$ followed by z^k where k is the number of steps M takes in accepting α. The parse of $\$z^k$ is a chain of nodes labeled R_i indicating the order of application of the rules of M. T operates by first making s copies of this tree, each copy representing a single tape square. Then it simulates M using the state tuples to represent the state of M, the tape position and the tape contents. The first element of each tuple represents the head position by the mod s distance of this tape square from the head, the second element represents the current state of M, and the final element represents the contents of the tape square. A translation is produced only if M enters its final state with all tape symbols being 0.

The rules of T are split into several groups. The rules to do the proper initialization are

$q_d(S(x_1))\to A(q_0(x_1))$

$q_i(B(ax_1))\to A(ap_{a,i}(x_1)q_{i+1}(x_1))$ for all $a\in\Omega$, $0\le i\le s-2$

$q_{s-1}(B(ax_1))\to A(ap_{a,s-1}(x_1))$ for all $a\in\Omega$

$p_{b,i}(B(ax_1))\to A(p_{b,i}(x_1))$ for all $a,b\in\Omega$, $0\le i\le s-2$

$p_{a,i}(C(\$x_1))\to A(<i,\sigma_0,a>(x_1))$ for all $a\in\Omega$, $0\le i\le s-1$

The q_i states just accept each input symbol in turn and create a copy of the parse tree for the corresponding tape square. The $p_{\sigma,i}$ states remember the input symbol for some tape square and just ingore the rest of the source tape description. The second set of rules actually simulate M. Let $r_j=<\sigma,a,b,n,\sigma'>$ be a rule of M with $\sigma'\ne\sigma_f$. Then we add to T the rules

$<0,\sigma,a>(R_j(x_1z))\to A(<i,\sigma',b>(x_1))$ for $i=(-n \bmod s)$; and

$$<i,\sigma,c>(R_j(x_1z)) \to A(<i',\sigma',c>(x_1)) \text{ for } 1 \le i \le s-1,\ i'=((i-n) \bmod s) \text{ and all } c \in \Omega.$$

It is clear that these rules change the state so as to simulate M performing the action of rule r_j. Finally, the last set of rules insure that M terminates properly. These are

$$<i,\sigma_f,0>(\$) \to \$ \text{ for } 0 \le i \le s-1.$$

Then T is clearly a deterministic t-fst that takes the proper parse tree for $\alpha\$z^k$ into the string $\alpha\$^s$ if and only if M accepts α with k operations. Moreover if $z=\lambda$, then the size of the source string is linear in the size of the target string and the domain grammar of T contains cycles, while if $z \ne \lambda$, the domain grammar is cycle-free but the source string can be exponentially larger than the target string.

□

We can summarize the complexity results of this section by

THEOREM 5.14: The problem of inversion is
1. PSPACE-complete for an arbitrary t-fst;
2. NP-complete for a fixed regular t-fst.

□

5.3 GENERALIZED SYNTAX-BASED TRANSLATION SCHEMATA

The results of the previous section are quite negative in terms of automatic programming since they say that an automatic programming problem is intractable if it requires a fixed nondeterministic t-fst or an arbitrary nonregular t-fst for its semantic specification. This is

quite a contrast to the results of the previous chapter where we showed that automatic programming was practical if linear tree mappings were used. In order to reconcile these differences and to complete our study of practical automatic programming, we consider fixed deterministic t-fsts in this section and show that they can usually provide a practical semantic characterization.

We actually consider the class of generalized syntax-directed translation schemata as they form a more practical description of the class of deterministic tree transducers [90]. We first prove that the problem of inverting an arbitrary GSDTS is intractable. This shows that the best algorithm for inverting a fixed GSDTS we could realistically expect to find would have polynomial time complexity where the degree would depend on the GSDTS. Using determinism and a characterization of the class of context sensitive grammars that describe the target language, we then present an algorithm that achieves this bound.

5.3.1 Inverting An Arbitrary GSDTS

Since GSDTS are equivalent to the class of deterministic top-down tree transducers, the results of the previous section show the problem of inverting an arbitrary GSDTS is PSPACE-complete. This problem becomes more interesting when we consider the more natural class of regular GSDTS. A GSDTS is called regular if its source grammar is cycle-free and if the size of the target string grows with the size of the source string. The results of Aho and Ullman [10] show that if a GSDTS is not regular then either it is based on a grammar containing

cycles, which is unlikely, or its range is finite, in which case it is useless for specifying semantics. Moreover, they show that if the size of the target string is not bounded by a constant, it must be at least linear in the size of the source string.

Therefore our primary interest lies in the complexity of inverting regular GSDTS. These form a superset of the class of SDTS, and thus there is a trivial lower bound of $T_{parsing}$. We can improve this bound by a method similar to that used in lemma 5.12.

THEOREM 5.15: The problem of inverting an arbitrary regular GSDTS is NP-complete.

Proof: Since GSDTS are just deterministic tree transducers, lemma 5.11 shows that the problem of inverting an arbitrary regular GSDTS is in the class NP. Hence it is sufficient to show that it is NP-hard. We do this by reducing the problem of 3-satisfiability to that of inverting an arbitrary GSDTS. For a given set of clauses of a 3-satisfiability problem we construct a regular GSDTS T and a string s_0 such that T has an inverse for s_0 if and only if the set of clauses is satisfiable.

Let P be a 3-satisfiability problem with clauses $D_1,\ldots,D_r$ over the variables $u_1,\ldots,u_m$, and suppose, without loss of generality, that each clause contains exactly three distinct variables. We define $\alpha_{i,j}$ for $1 \le i \le r$ and $1 \le j \le m$ uniquely by

a) if u_j is used positively in D_i then $\alpha_{i,j}=1$;

b) if u_j is used negatively in D_i then $\alpha_{i,j}=2$; and

c) otherwise, if u_j not used in D_i, then $\alpha_{i,j}=3$.

Using this definition we construct the regular GSDTS $T=\langle N,\Sigma,\Delta,\Gamma,R,S\rangle$ where

$N = \{S,A,B,C\}$
$\Sigma = \{t,f,\$,a,b,c,d\}$
$\Delta = \{x,\$,d\}$
$\Gamma = \{S_1,B_1,C_1,C_2,C_3\} \cup \{A_i \mid 1 \le i \le r\}$

and where R contains the rules

1) $S \to dS$, $S_1=dS_1$
2) $S \to A$, $S_1=A_1\$A_2\$\ldots\$A_r\$$
3) $A \to C^1\ldots C^m B^1\ldots B^r$, $A_i=C^1_{\alpha_{i,1}}\ldots C^m_{\alpha_{i,m}} B^i_1$ for $1 \le i \le r$
4) $B \to a$, $B_1=x$
5) $B \to b$, $B_1=xx$
6) $B \to c$, $B_1=xxx$
7) $C \to t$, $C_1=xxx$, $C_2=xx$, $C_3=x$
8) $C \to f$, $C_1=xx$, $C_2=xxx$, $C_3=x$.

It is clear that this GSDTS is regular. Moreover, its size is polynomial in r and m and hence in the size of the 3-satisfiability problem P.

Rule 1 of this mapping allows the acceptance of infinite source and target strings and serves only to insure that the schema is regular. Otherwise, considering only rules 2 through 8, T accepts source strings of the form $(t \cup f)^m (a \cup b \cup c)^r$. Intuitively, the first part of this string is a truth assignment to the m variables and the second part is arbitrary. T operates by copying this truth assignment once for each clause

of P and then creating a string of x's based on the clause such that this string can be of a certain length if and only if the clause is satisfied by the truth assignment. Formally we show

CLAIM: There is a mapping of A_i into x^{m+7} if and only if the truth assignment specified by the parses of C^j in rule 3 for $1 \le j \le m$ satisfies the clause D_i.

To prove this claim we note that the only translate for A_i is

$$A_i \rightarrow C^1_{\alpha_{i,1}} \ldots C^m_{\alpha_{i,m}} B^i_1 .$$

Since the $\alpha_{i,j}$ are dependent on D_i, all but three of them must have the value 3. Hence, since any parse of C^i_3 yields x and since B^i appears only in this translate and hence the corresponding parse of B is arbitrary and can yield x, x^2 or x^3, it is sufficient to show that the three C^i corresponding to the variables used in D_i can together yield one of the strings x^7, x^8 or x^9 if and only if D_i is satisfiable. The translate used for C^i is C^i_1 if u_i is used positively in D_i and is C^i_2 if it is used negatively. Also, C^i_1 yields x^3 if $C \rightarrow t$ is the parse corresponding to this C^i and x^2 otherwise, while C^i_2 yields x^3 if $C \rightarrow f$ is the corresponding parse and x^2 otherwise. Hence x^3 is generated for some C^i if and only if the truth assignment makes the corresponding literal true in D_i and the claim follows.

Now let $s_0=(x^{m+7}\$)^r$. Clearly $|s_0|$ is polynomial in the size of the 3-satisfiability problem. We claim that T maps some string into s_0 if and only if each D_i of P is satisfied by a common truth assignment. This follows since rule 2 insures that the same truth assignment is used to test each clause and the claim shows that x^{m+7} is achievable if and

only if the corresponding clause is satisfied by the truth assigment.

□

This shows that the problem of inverting a regular GSDTS is hard in general. However, many of the translations we are interested in are fixed or at worst fairly simple, so it is reasonable to ask whether the problem is still hard for a fixed regular GSDTS. While it might be possible to find an infinite class of algorithms such that every regular GSDTS can be inverted in fixed polynomial time by some unknown algorithm in this class, this seems both unlikely and impractical. If we restrict ourselves to one algorithm (or equivalently a finite set of algorithms) then the above theorem says that the best we can hope to achieve for a fixed regular GSDTS is an algorithm that runs in polynomial time where the degree of the polynomial is dependent on the particular translation schema. In the following sections we show that this bound is actually achievable. We also demonstrate restricted classes of GSDTS for which this algorithm is quite practical.

5.3.2 Determinism Versus Nondeterminism

In the previous section we showed that the problem of inverting a fixed regular nondeterministic t-fst is NP-complete. We now want to show is that the same problem for a deterministic t-fst is solvable in polynomial time. In order to do this we must demonstrate some basic difference between the two models that allows this contrast in complexity. This difference can be summarized quite simply as follows: a

nondeterministic tree transducer can duplicate a subtree and then process the copies in an exponential number of ways while a deterministic tree transducer can duplicate a subtree and then process the copies only in a constant number of ways.

The major problem that arises when inverting a top-down tree transducer is that of nonlinearity where a subtree is copied in the forward translation and the copies are processed independently. In the inverse translation the results of these copies are processed independently and we must insure that they all yield identical trees. In a nondeterministic tree transducer this problem is quite complex. In particular each copy can be processed nondeterministically and hence, in the worst case, can yield an exponential number of different trees. Lemma 5.12 shows that resolving this problem is NP-complete. For a deterministic transformation, on the other hand, the number of different translations possible for a given subtree is bounded by the number of initial states that can be assigned to that subtree. This is easily seen since for a fixed state and subtree, determinism insures that the translation is always the same. Thus we can bound the number of possible translates of a subtree by the total number of states or more specifically by the maximum number of different states assigned in any one rule to a copied subtree. This greatly simplifies the task of inverting a deterministic mapping.

5.3.3 P-Grammars

To invert a syntax-directed translation schema we applied a parse of the target string to the source grammar. We use a similar technique for a generalized schema. To do this we first must determine the type of grammar needed to describe the target language. We define a grammar whose productions are just the target productions of the GSDTS and are applied in the same manner that a GSDTS applies its rules. Such a grammar describes a context sensitive language since Baker has shown that any t-fst target language is context sensitive [15]. We call this type of grammar a parallel or P-grammar since the productions consist of a set of rules to be applied in parallel.

Formally a <u>P-grammar</u> is a 4-tuple $\langle N,\Sigma,S_1,P\rangle$ where N is a finite alphabet of nonterminal symbols, Σ is a finite alphabet of terminal symbols, $S_1 \in N$ is the initial symbol, and P is a finite set of productions. The elements of N have the form A_i where i is a positive integer and the productions of P are of the form

$$(A_1 \rightarrow \beta_1 , \ldots , A_m \rightarrow \beta_m)$$

where each $A_i \in N$ and each $\beta \in (N \cup \Sigma)^*$. For such a production we call the elements A_i the <u>sources</u>, the strings β_i the <u>targets</u>, and the pairs $A_i \rightarrow \beta_i$ the <u>translates</u>. We require that whenever A_i and A_j are in N, $i \neq j$, and A_i appears as a source of some production, then A_j is also a source of that production. Finally we say that two nonterminals appearing as translates of some production are <u>associated</u> if they are of the form B_i and B_j for some i and j. To denote nonassociated occurrences of the

same variable we allow superscripting. Thus B_i^k and B_j^ℓ are associated in some production if and only if $k=\ell$.

The sentential forms of a P-grammar are strings in $((N \times Z) \cup \Sigma)^*$ where Z is the set of positive integers. We denote the nonterminal elements of these strings as A_j^k where $A_j \in N$ and $k>0$ is the <u>association integer</u>. We say that $s \Longrightarrow s'$ by a production $(A_1 \to \beta_1, \ldots, A_m \to \beta_m)$ if and only if

1. there is some i and some k such that $A_i^k \in s$; and

2. s′ is derived from s by replacing each occurrence of $A_j^k \in s$ for any j and a fixed k, by the string β_j modified so that each set of associated nonterminals of the production is assigned unique association integer not appearing in s.

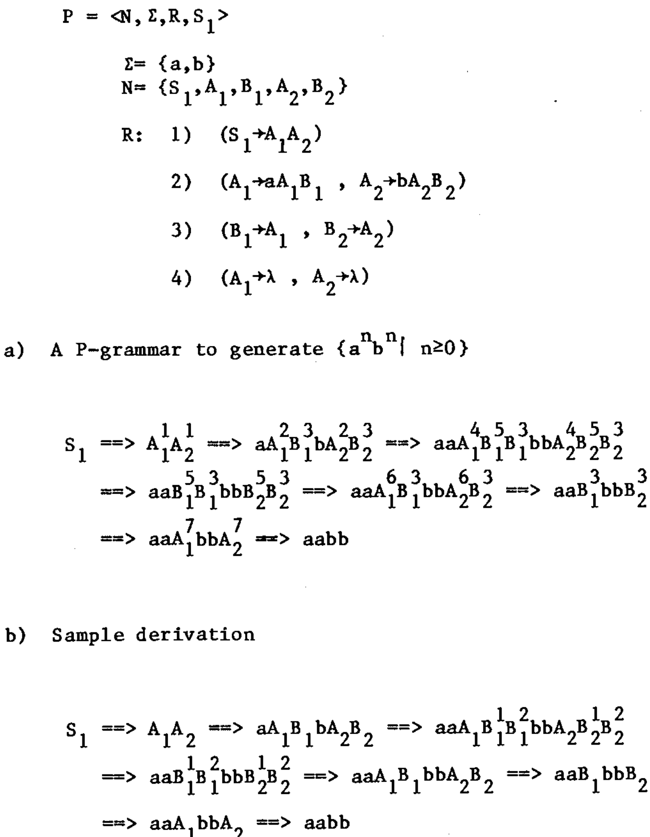
An example of a P-grammar and such a derivation is presented in figure 5.5. We will normally omit the association integers when no ambiguity can arise and thus we present the derivation of figure 5.5b as shown in figure 5.5c.

This definition of a P-grammar is closely related to the operation of a GSDTS. We can use this relationship to show

LEMMA 5.16: A P-grammar
- a) describes the range of a GSDTS;
- b) is the target grammar of a GSDTS;
- c) describes the target language of a deterministic t-fst;
- d) is context-sensitive; and
- e) is not in general context free.

Proof: If we have a GSDTS $T=\langle N,\Sigma,\Delta,\Gamma,R,S\rangle$, we can easily derive its target grammar as $\langle \Gamma,\Delta,S_1,R'\rangle$ where R' is the set of target productions of T. Then it is clear from the definition that this target

FIGURE 5.5



$P = \langle N,\Sigma,R,S_1\rangle$

$\Sigma = \{a,b\}$

$N = \{S_1,A_1,B_1,A_2,B_2\}$

R:
1) $(S_1 \to A_1A_2)$
2) $(A_1 \to aA_1B_1\ ,\ A_2 \to bA_2B_2)$
3) $(B_1 \to A_1\ ,\ B_2 \to A_2)$
4) $(A_1 \to \lambda\ ,\ A_2 \to \lambda)$

a) A P-grammar to generate $\{a^nb^n \mid n \geq 0\}$

$$S_1 \Longrightarrow A_1^1A_2^1 \Longrightarrow aA_1^2B_1^3bA_2^2B_2^3 \Longrightarrow aaA_1^4B_1^5B_1^3bbA_2^4B_2^5B_2^3$$
$$\Longrightarrow aaB_1^5B_1^3bbB_2^5B_2^3 \Longrightarrow aaA_1^6B_1^3bbA_2^6B_2^3 \Longrightarrow aaB_1^3bbB_2^3$$
$$\Longrightarrow aaA_1^7bbA_2^7 \Longrightarrow aabb$$

b) Sample derivation

$$S_1 \Longrightarrow A_1A_2 \Longrightarrow aA_1B_1bA_2B_2 \Longrightarrow aaA_1B_1^1B_1^2bbA_2B_2^1B_2^2$$
$$\Longrightarrow aaB_1^1B_1^2bbB_2^1B_2^2 \Longrightarrow aaA_1B_1bbA_2B_2 \Longrightarrow aaB_1bbB_2$$
$$\Longrightarrow aaA_1bbA_2 \Longrightarrow aabb$$

c) Simplified derivation

Figure 5.5: An Example P-grammar

grammar is precisely a P-grammar. This shows points a) and b). Point c) follows since for any deterministic t-fst an equivalent GSDTS can be effectively found and for any GSDTS an equivalent deterministic t-fst can be effectively found. Point d) follows from the work of Baker who showed that the range of a t-fst mapping is context sensitive [15]. Finally point e) follows from point c) and the example of figure 5.2.

□

In the previous chapter we inverted an SDTS by parsing its target grammar and applying this parse to the source grammar. We can use the same technique for a GSDTS and hence reduce the problem of inversion to that of parsing a P-grammar. In particular

LEMMA 5.17: Let T be a nondeleting GSDTS. Let G be its context free domain grammar and P its target P-grammar. If ρ is a parse of a string s using P then ρ applied to G yields a string s′ such that $T{:}s' \rightarrow s$.

Proof: Let

$$S_1 = \beta_1 \underset{P}{\Longrightarrow} \beta_2 \underset{P}{\Longrightarrow} \cdots \underset{P}{\Longrightarrow} \beta_n$$

be the parse ρ applied to the grammar P and let $\gamma_1 = S$. We then claim that $\beta_1 \overset{*}{\underset{P}{\Longrightarrow}} \beta_i$ implies $\gamma_1 \overset{*}{\underset{G}{\Longrightarrow}} \gamma_i$ for some γ_i such that there a one-to-one correspondence between nonterminals in γ_i and sets of associated nonterminals in β_i. Clearly this holds for i=1. Then it suffices to show that applying a production of P to β_i and applying the corresponding production of G to γ_i preserves the condition. By induction there is a one-to-one correspondence between the variables expanded by a production of P in β_i and some variable in γ_i. Then the definition of a GSDTS

implies that the corresponding production of G expands just this variable in γ_i, and moreover, that the application preserves this one-to-one relationship. Finally γ_n is actually a terminal string since it contains a nonterminal only if β_n does. Finally, by the definition of a GSDTS, as ρ is a parse of s′, ρ applied to P is a translation of s′ using T and hence $T{:}s' \rightarrow s$.

□

Figure 5.6 gives an example of a GSDTS, its target P-grammar, and the use of a parse of the P-grammar to invert the GSDTS.

5.3.4 Parsing P-Grammars

A P-grammar is difficult to parse because the application of productions can be spread throughout the entire string. To avoid this problem we determine a context free grammar for a given P-grammar such that a parse of the context free grammar meeting certain conditions defines a parse of the P-grammar. We then use a modification of Earley's parsing algorithm to determine and check a parse of this context free grammar and thus to produce a parse of the P-grammar. This method is analogous to inverting a nonlinear t-fst by finding a proper parse of its context free target grammar. The major difference is that determinism in this case allows us to find a polynomial algorithm for parsing a fixed P-grammar.

$T = \langle N, \Sigma, \Delta, \Omega, R, S \rangle$

$N = \{S, A\}$

$\Sigma = \{0, 1\}$

$\Delta = \{a\}$

$\Omega = \{S_1, A_1, A_2\}$

R: 1) $(S \to 1A,\ S_1 = A_1A_2)$

2) $(A \to 1A,\ A_1 = A_1A_2\ ,\ A_2 = A_2A_2)$

3) $(A \to 0A,\ A_1 = A_1\ ,\ A_2 = A_2A_2)$

4) $(A \to \lambda,\ A_1 = \lambda\ ,\ A_2 = a\ )$

a) An example GSDTS to map $\{\langle x, a^n \rangle \mid x \text{ is } n \text{ in binary}\}$

$P = \langle \Omega, \Delta, R', S_1 \rangle$

R: 1) $(S_1 \to A_1A_2)$

2) $(A_1 \to A_1A_2\ ,\ A_2 \to A_2A_2)$

3) $(A_1 \to A_1\ ,\ A_2 \to A_2A_2)$

4) $(A_1 \to \lambda\ ,\ A_2 \to a\ )$

b) The derived P-grammar

$S_1 \Longrightarrow A_1A_2$ by production 1

$\Longrightarrow A_1A_2A_2$ by production 3

$\Longrightarrow A_1A_2A_2A_2A_2$ by production 2

$\Longrightarrow aaaaa$ by production 4

$S \Longrightarrow 1A \Longrightarrow 10A \Longrightarrow 101A \Longrightarrow 101$

c) Using a parse of P to invert T

$G = \langle \Omega, \Delta, P, S_1 \rangle$

P: $S_1 \to A_1A_2$

$A_1 \to A_1A_2$

$A_2 \to A_2A_2$ [from rule 2]

$A_2 \to A_2A_2$ [from rule 3]

$A_1 \to A_1$

$A_1 \to \lambda$

$A_2 \to a$

d) The derived context free grammar of P

Figure 5.6: Inverting a GSDTS by Parsing Its P-Grammar

We derive a context free grammar from a P-grammar by splitting the productions of the P-grammar in the obvious way. Let $(A_1 \to \beta_1, \ldots, A_m \to \beta_m)$ be a production of a P-grammar, P. Then the <u>derived context free grammar</u>, G(P), contains the m productions $A_1 \to \beta_1$, $A_2 \to \beta_2$, ..., $A_m \to \beta_m$. Formally, let $P = \langle N, \Sigma, S, R \rangle$ be a P-grammar. Then G(P), the grammar derived from P, is $\langle N, \Sigma, S, R' \rangle$ where R' contains each translate of each production in R. This is illustrated in figure 5.6d for the P-grammar of figure 5.6b. We can then show

<u>LEMMA</u> 5.18: Let P be a P-grammar. Then $G = G(P)$ is context free and $L[P] \subseteq L[G]$.

Proof: To show G is context free we need only show that all productions have the form $A \to \alpha$ where A is a nonterminal. This follows immediately from the construction of G from P.

To show that $L[P] \subseteq L[G(P)]$ it is sufficient, by induction on the derivations of P, to show for any sentential form γ, that $\gamma \underset{P}{\Longrightarrow} \gamma'$ implies $\gamma \overset{*}{\underset{G}{\Longrightarrow}} \gamma'$. Let $\gamma \underset{P}{\Longrightarrow} \gamma'$ by a production ρ of P. Then we can derive γ' from γ using G by applying each production derived from ρ to each nonterminal of γ which was changed.

□

This proof illustrates the relationship between P and G(P). To parse P using G(P) we must determine when a parse of G(P) also describes a parse of P. It is not difficult to see that this occurs if and only if

Condition A: Whenever $A_1, \ldots, A_m$ are associated in an intermediate parse string of P, then the productions used to expand $A_1, \ldots, A_m$ are all derived from a common production of P.

We are primarily interested in bottom-up parsing and condition A is useful only in a top-down manner since a bottom-up parse could not know which nonterminals are actually associated until the parse was complete. We define a mapping function μ that takes a production of G(P) and returns the rule of P from which the production was derived. Then μ extends naturally to a series of productions of G(P) and hence to derivations of G(P). We say that a set of nonterminals $A_1, \ldots, A_n$ of G(P) has a <u>common parse</u> if and only if for $1 \le i \le n$ there is a parse ρ_i of A_i such that $\mu(\rho_1) = \mu(\rho_2) = \ldots = \mu(\rho_n)$. Given this notion, we can introduce a bottom-up condition that insures that a parse of G(P) is a parse of P:

Condition B: Whenever a production of G(P), $A \to \beta$, is applied in the parse, all associated nonterminals in β have a common parse.

We can show that both condition A and condition B are sufficient to insure that a parse of G(P) identifies a parse of P by

<u>LEMMA</u> 5.19: Let P be a P-grammar and G(P) be its derived context free grammar. Suppose we have a string s and a parse of s using G(P). Then either of the following is a sufficient condition for the parse of G(P) to identify a parse of P:
1) Condition A holds for the parse of G(P);
2) Condition B holds for the parse of G(P).

Proof: Let ρ be a parse of G(P). As G(P) is context free, we can assume without loss of generality that ρ is applied so that whenever a set of nonterminals are associated in some intermediate parse string, the productions for each of them are applied essentially simultaneously. If all sets of associated nonterminals are expanded using productions derived from a common production of P, then the parse ρ applied in this manner is exactly a parse of P. This shows condition A is sufficient.

To show condition B is sufficient we show that it implies condition A. A set of nonterminals have a common parse if and only if every set of associated nonterminals derived from the original set satisfies condition A. Also the definition of associated implies that a set of nonterminals is associated if and only if they are derived from a set of nonterminals associated in some production of P. This shows that condition A holds whenever condition B does.

□

This reduces the problem of parsing a P-grammar to that of checking if a set of nonterminals has a common parse. If G(P) is unambiguous then this is simple. However, the general case seems to require exponential time. Using the fact that a GSDTS is deterministic, we can simplify the problem by noting that two identical nonterminals have a common parse if and only if they produce a common string. Assuming we can determine what string is associated with a nonterminal, we need only compare the parses of associated but different nonterminals and then use string comparisons to check the associated occurrences of the same nonterminals. The important thing here is that the maximum number of different but associated nonterminals is a constant for any given P-grammar.

This is the basic underlying concept for parsing a P-grammar. In the next sections we consider the problem of checking if a set of nonterminals have a common parse. We do this by showing what modifications of Earley's parsing algorithm are necessary to make this check. Using these modifications we construct a complete algorithm to parse a P-grammar which can then be used to invert a GSDTS.

5.3.5 Common Parses

We attempt to check for common parses in the framework alloted by Earley's parsing algorithm. That is, we assume that we are given the items and item lists previously computed by an Earley-like algorithm and want to determine whether a set of nonterminals in some new item shares a common parse. In this section we present a recursive method of checking for a common parse and show what extra information is necessary for its implementation. Then in the next two sections we use this to construct an algorithm to parse a P-grammar.

We want to determine, for some production $A_i \to \beta$ of a potential parse, if all associated nonterminals in β have a common parse. Consider the item

$$\pi = [A_i \to \alpha \cdot B_j \beta, k]$$

and suppose an item for B_j is complete and would normally be used to

process the item π. Then it suffices to check if B_j has a common parse with all the nonterminals in α that are associated with it. We do not work directly with B_j, but rather consider the parse associated with the item used to complete B_j. Similarly, we consider the completing items for each of the associated nonterminals. If more than one item can represent a nonterminal, we try all possibilities separately. Thus we actually are considering the problem of determining if a set of items represents a common parse. We can show

LEMMA 5.20: A set of items shares a common parse if and only if
1) they are all derived from the same production; and
2) all of their associated nonterminals share a common parse.

Proof: A set of items represents a set of potential parses for a sequence of nonterminals and thus share a common parse if and only if the parses they define for the P-grammar are the same. But this is true if and only if their parse trees have the same root and have corresponding sons that define common parses. But these are just the two conditions of the lemma.

□

This recursive definition naturally suggests a recursive algorithm to check for a common parse among a set of items. Such an algorithm would check to see if the items were all derived from the same production of P and then would call itself recursively for all associated nonterminals. This is the intuitive design of the algorithm we propose.

To formulate such an algorithm we need to enhance the items of Earley's algorithm. We must first identify the production of the P-grammar from which the context free production used by the item is derived. This facilitates the basic checking procedure. We must also be able to determine which items were used to complete the various non-terminals of a given item, a process which is made quite complex by ambiguity in G(P). We solve the problem by identifying items with their history, that is, by recording for each item the item lists in which it previously appeared. This allows us to determine a relatively small set of items that were used to complete a given item and identifies the segments of the source string from which each of these items was derived.

Formally, an item for our algorithm is composed of a dotted rule, a derived production number, and a history string denoting where the item has appeared before. It is represented as

$$[A_i \to \alpha \cdot \beta,\ j,\ i_0:\ldots:i_n]$$

where $A_i \to \alpha\beta$ is a translate of production j of the P-grammar, $n=|\alpha|$, and where the item initially appears in the item list Ξ_{i_0}, next appears in the list Ξ_{i_1}, and so on. The problem of finding the items that correspond to a given nonterminal B_i in α is simply that of finding all items of the form

$$[B_i \to \gamma \cdot,\ k,\ \ell:\ldots:m]$$

where ℓ and m are fixed. We call the set of all such items for a given B_i, ℓ and m a <u>list of potential items</u> and denote it as $(B_i, \ell @ m)$.

5.3.6 An Algorithm To Parse P-grammars

We parse a P-grammar P by restricting the parse of its derived context free grammar $G=G(P)$. To do this we use Earley's parsing algorithm with our extended items and modify it so that an item is added to an item list if and only if there is a common parse of all associated nonterminals of the item that have already been completed. Thus we only add an item if the part we have looked at has a P-parse, that is, a parse with the given P-grammar.

To check for a common parse we introduce the function PARSECHECK of one argument, an extended item. We define this algorithm in the next section and show

LEMMA 5.25: PARSECHECK($\pi=[A_i \to \alpha X \cdot \beta, p, i_0 : \ldots : i_m]$) returns TRUE for the item π in some item list if and only if all sets of associated nonterminals in αX as used to complete this item have a common parse.

Using PARSECHECK we can parse a P-grammar with the algorithm

ALGORITHM PPARSE: Parsing a P-Grammar.

Given: A P-grammar $P=\langle N, \Sigma, S_1, R \rangle$, a string $a_1 \ldots a_n$, $G=G(P)$.

Build: A set of item lists $\Xi_0 \ldots \Xi_n$

1. {initialization} Let $\ell=0$. For each production of G of the form $S_1 \to \alpha$ derived from a rule p of P add to Ξ_0 the item

$$[S_1 \to \cdot \alpha, p, 0].$$

2. Repeat steps a and b until no more items can be added to Ξ_ℓ.

a) {completer} If $[A_i \to \gamma \cdot, p, k : \ldots : \ell] \in \Xi_\ell$ then for each item in Ξ_k of the form

$$[B_j \to \alpha \cdot A_i \beta, p', j_0 : \ldots : j_m]$$

construct $\pi=[B_j \to \alpha A_i \cdot \beta, p', j_0 : \ldots : j_m \ell]$ and if π is not in Ξ_ℓ and if PARSECHECK(π) returns TRUE then add π to Ξ_ℓ.

b) {predictor} If $[A_i \to \alpha \cdot B_j \beta, p, i_0 : \ldots : \ell] \in \Xi_1$, then for each production of G, $B_j \to \gamma$, derived from production r of P, add to Ξ_ℓ all new items of the form

$$[B_j \to \cdot \gamma, r, \ell].$$

3. If $\ell=n$ then halt.

4. {scanner} Set ℓ to $\ell+1$. For all items in $\Xi_{\ell-1}$ of the form

$$[A_i \to \alpha \cdot a_\ell \beta, p, i_0 : \ldots : i_m]$$

add to Ξ_ℓ the item

$$[A_i \to \alpha a_\ell \cdot \beta, p, i_0 : \ldots : i_m \ell].$$

Then go back to step 2.

□

This algorithm is essentially Earley's algorithm except that before an item is constructed in the completer stage, a test is made to determine if the item actually defines a parse of P. Moreover items constructed with the same production but in different ways are not always merged as they normally would be with Earley's algorithm. We want to prove that the algorithm correctly finds a parse of a P-grammar for a

string. We show first that the algorithm treats items as we would expect. In particular we observe

LEMMA 5.21: Let $[A_i \to X_1 \ldots X_j \bullet \beta, p, i_0 : \ldots : i_j]$ be an item added to some item list Ξ_ℓ by the algorithm. Then

1) $\ell = i_j$;
2) $A_i \to X_1 \ldots X_j \beta$ is a translate of rule p of P.
3) X_k represents a parse of $a_{i_{k-1}+1} \ldots a_{i_k}$.

Proof: All of these follow simply from the specification of the algorithm.

□

An example of the item lists generated by PPARSE is given in figure 5.7c.

We next prove that the algorithm works by presenting the exact conditions under which an item is placed in an item list.

THEOREM 5.22: After PPARSE executes, the item $[A_i \to \alpha \bullet \beta, p, j : \ldots : \ell] \in \Xi_\ell$ if and only if $S_1 \overset{*}{\underset{G}{\Longrightarrow}} \gamma A_i \delta$ for strings γ, δ such that $\gamma \overset{*}{\underset{P}{\Longrightarrow}} a_1 \ldots a_j$ and $\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_{j+1} \ldots a_\ell$ where α can contain associated nonterminals.

Proof: We prove this by induction on how an item is added to an item list. Initially we need only consider items of the form

$[S_1 \to \bullet \alpha, p, 0]$.

Clearly $S_1 \overset{*}{\underset{G}{\Longrightarrow}} S_1$, $\lambda \overset{*}{\underset{P}{\Longrightarrow}} \lambda$ and hence the conditions of the lemma hold.

FIGURE 5.7

$T = \langle N, \Sigma, \Delta, \Omega, R, A_1 \rangle$

$N = \{A\}$
$\Sigma = \{a\}$
$\Delta = \{a\}$
$\Omega = \{A_1, A_2\}$

R: 1) $(A \to aA,\ A_1 = A_1 A_2 A_2 a\ ,\ A_2 = A_2 a)$

2) $(A \to a,\ A_1 = a\ ,\ A_2 = a\ )$

a) GSDTS to map a^n into a^{n^2}

$P = \langle \Omega, \Delta, P, A_1 \rangle$

R: 1) $(A_1 \to A_1 A_2 A_2 a\ ,\ A_2 \to A_2 a)$

2) $(A_1 = a\ ,\ A_2 = a\ )$

b) Its P-Grammar

Ξ_0: $[A_1 \to \bullet A_1 A_2 A_2 a, 1, 0]$

$[A_1 \to \bullet a, 2, 0]$

Ξ_1: $[A_1 \to a \bullet, 2, 0:1]$

$[A_1 \to A_1 \bullet A_2 A_2 a, 1, 0:1]$

$[A_2 \to \bullet A_2 a, 1, 1]$

$[A_2 \to \bullet a, 2, 1]$

Ξ_2: $[A_2 \to a \bullet, 2, 1:2]$

$[A_2 \to A_2 \bullet a, 1, 1:2]$

$[A_1 \to A_1 A_2 \bullet A_2 a, 1, 0:1:2]$

$[A_2 \to \bullet A_2 a, 1, 2]$

$[A_2 \to \bullet a, 2, 2]$

Ξ_3: $[A_2 \to A_2 a \bullet, 1, 1{:}2{:}3]$

$[A_2 \to a \bullet, 2, 2{:}3]$

$[A_2 \to A_2 \bullet a, 1, 1{:}3]$

$[A_1 \to A_1 A_2 A_2 \bullet a, 1, 0{:}1{:}2{:}3]$

$[A_2 \to A_2 \bullet a, 1, 2{:}3]$

Ξ_4: $[A_2 \to A_2 a \bullet, 1, 1{:}3{:}4]$

$[A_1 \to A_1 A_2 A_2 a \bullet, 1, 0{:}1{:}2{:}3{:}4]$

$[A_2 \to A_2 a \bullet, 1, 2{:}3{:}4]$

$[A_2 \to A_2 \bullet a, 1, 2{:}4]$

$[A_2 \to A_{_} \bullet a, 1, 1{:}4]$

$[A_1 \to A_1 \bullet A_2 A_2 a, 1, 0{:}4]$

$[A_2 \to \bullet A_2 a, 1{:}4]$

$[A_2 \to \bullet a, 2, 4]$

Ξ_5: $[A_2 \to A_2 a \bullet, 1, 2{:}4{:}5]$

$[A_2 \to A_2 a \bullet, 1, 1{:}4{:}5]$

$[A_2 \to A_2 \bullet a, 1, 2{:}5]$

$[A_2 \to A_2 \bullet a, 1, 1{:}5]$

$[A_2 \to a \bullet, 2, 4{:}5]$

$[A_2 \to A_2 \bullet a, 1, 4{:}5]$

Ξ_6: $[A_2 \to A_2 a \bullet, 1, 2{:}5{:}6]$

$[A_2 \to A_2 a \bullet, 1, 1{:}5{:}6]$

$[A_2 \to A_2 a \bullet, 1, 4{:}5{:}6]$

$[A_2 \to A_2 \bullet a, 1, 2{:}6]$

$[A_2 \to A_2 \bullet a, 1, 1{:}6]$

$[A_1 \to A_1 A_2 \bullet A_2 a, 1, 0{:}4{:}6]$

$[A_2 \to A_2 \bullet a, 1, 4{:}6]$

$[A_2 \to \bullet A_2 a, 1, 6]$

$[A_2 \to \bullet a, 2, 6]$

Ξ_7: $[A_2 \to A_2 a \bullet, 1, 2{:}6{:}7]$

$[A_2 \to A_2 a \bullet, 1, 1{:}6{:}7]$

$[A_2 \to A_2 a \bullet, 1, 4{:}6{:}7]$

$[A_2 \to A_2 \bullet a, 1, 6{:}7]$

$[A_2 \to a \bullet, 2, 6{:}7]$

$[A_2 \to A_2 \bullet a, 1, 1{:}7]$

$[A_2 \to A_2 \bullet a, 1, 2{:}7]$

$[A_2 \to A_2 \bullet a, 1, 4{:}7]$

Ξ_8: $[A_2 \to A_2 a \bullet, 1, 4{:}7{:}8]$

$[A_2 \to A_2 a \bullet, 1, 6{:}7{:}8]$

$[A_2 \to A_2 a \bullet, 1, 1{:}7{:}8]$

$[A_2 \to A_2 a \bullet, 1, 2{:}7{:}8]$

$[A_2 \to A_2 \bullet a, 1, 1{:}8]$

$[A_2 \to A_2 \bullet a, 1, 2{:}8]$

$[A_2 \to A_2 \bullet a, 1, 4{:}8]$

$[A_2 \to A_2 \bullet a, 1, 6{:}8]$

$[A_2 \to A_1 A_2 A_2 \bullet a, 1, 0{:}4{:}6{:}8]$

Ξ_9: $[A_2 \to A_2 a \bullet, 1, 1{:}8{:}9]$

$[A_2 \to A_2 a \bullet, 1, 2{:}8{:}9]$

$[A_2 \to A_2 a \bullet, 1, 4{:}8{:}9]$

$[A_2 \to A_2 a \bullet, 1, 6{:}8{:}9]$

$[A_1 \to A_1 A_2 A_2 a \bullet, 1, 0{:}4{:}6{:}8{:}9]$

$[A_2 \to A_2 \bullet a, 1, 1{:}9]$

$[A_2 \to A_2 \bullet a, 1, 2{:}9]$

$[A_2 \to A_2 \bullet a, 1, 4{:}9]$

$[A_2 \to A_2 \bullet a, 1, 6{:}9]$

$[A_1 \to A_1 \bullet A_2 A_2 a, 1, 0{:}9]$

$[A_2 \to \bullet A_2 a, 1, 9]$

$[A_2 \to \bullet a, 2, 9]$

c) Item Lists for Parsing a^9

ICHECK($[A_1 \to A_1 A_2 A_2 a \bullet, 1, 0{:}4{:}6{:}8{:}9]$)

t= <A,1>
LCHECK[$(A_1, 0@4), (A_2, 4@6), (A_2, 6@8)$]

$a_5 a_6 = a_7 a_8$ so eliminate $(A_2, 6@8)$

ICHECK($[A_1 \to A_1 A_2 A_2 a \bullet, 1, 0{:}1{:}2{:}3{:}4], [A_2 \to A_2 a \bullet, 1, 4{:}5{:}6]$)

t= <A,1>
LCHECK[$(A_1, 0@1), (A_2, 1@2), (A_2, 2@3), (A_2, 4@5)$]

$a_2 = a_3$ so eliminate $(A_2, 2@3)$

$a_2 = a_5$ so eliminate $(A_2, 4@5)$

ICHECK($[A_1 \to a \bullet, 2, 0{:}1], [A_2 \to a \bullet, 2, 1{:}2]$)

t=<A,2>
ICHECK returns <A,2>
LCHECK returns <A,2>
t= <A,1>(<A,2>)
ICHECK returns <A,1>(<A,2>)
LCHECK returns <A,1>(<A,2>)
t= <A,1>(<A,1>(<A,2>))
ICHECK returns <A,1>(<A,1>(<A,2>))

d) Example Execution of LCHECK and ICHECK

$A_1 \Longrightarrow A_1 A_2 A_2 a$ by production 1

$\Longrightarrow A_1 A_2 A_2 a A_2 a A_2 aa$ by production 1

$\Longrightarrow$ aaaaaaaaa by production 2

e) The String Produced by the Parse <A,1>(<A,1>(<A,2>))

A ==> aA ==> aaA ==> aaa

f) The source string produced by this parse

Figure 5.7: Inverting a GSDTS

There are three ways that an item can be constructed from previous items and added to an item list. We show that each of these preserves the conditions of the lemma and hence by induction we show that the lemma holds.

CASE 1: $[B_j \to \cdot\gamma, r, \ell]$ is added in the predictor stage because of the item $[A_i \to \alpha \cdot B_j \beta, p, j{:}\ldots{:}\ell]$. By induction

$$S \overset{*}{\underset{G}{\Longrightarrow}} \gamma A_i \delta \underset{G}{\Longrightarrow} \gamma\alpha B_j \beta\delta.$$

Moreover,

$$\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_{j+1}\cdots a_\ell \text{ and } \gamma \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_j,$$

and hence

$$\gamma\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_j a_{j+1}\cdots a_\ell.$$

CASE 2: $[A_i \to \alpha a_\ell \cdot \beta, p, j{:}\ldots{:}\ell]$ is added to Ξ_ℓ in the scanner stage because of the item $\pi = [A_i \to \alpha \cdot a_\ell \beta, p, j{:}\ldots{:}(\ell-1)]$ in $\Xi_{\ell-1}$. Clearly $S_1 \overset{*}{\underset{G}{\Longrightarrow}} \gamma A_i \delta$ and $\gamma \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_j$ since the item π is in $\Xi_{\ell-1}$. By induction $\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_j\cdots a_{\ell-1}$. Then

$$\alpha a_\ell \overset{*}{\underset{P}{\Longrightarrow}} a_j\cdots a_{\ell-1} a_\ell.$$

CASE 3: $[A_i \to \alpha B_m \cdot \beta, p, j{:}\ldots{:}k\ell]$ is added to Ξ_ℓ in the completer phase because of the items

$$\pi_1 = [A_i \to \alpha \cdot B_m \beta, p_1, j{:}\ldots{:}k] \in \Xi_k$$

and

$$\pi_2 = [B_m \to \gamma\cdot, p_2, k{:}\ldots{:}\ell] \in \Xi_\ell.$$

Then, by induction, the existence of π_1 insures that $S_1 \overset{*}{\underset{G}{\Longrightarrow}} \gamma A_i \delta$ for some γ and δ such that $\gamma \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_j$. We also know by induction that

$$\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_{j+1}\cdots a_k \text{ and } B_m \overset{*}{\underset{P}{\Longrightarrow}} a_{k+1}\cdots a_\ell$$

and hence

$$\alpha B_m \overset{*}{\underset{P}{\Longrightarrow}} a_{j+1}\cdots a_\ell$$

if and only if B_m has a common parse with all nonterminals in α. But this is precisely the condition checked by PARSECHECK and hence by lemma 5.25, the theorem follows.

□

This theorem shows that the algorithm is correct as

COROLLARY 5.23: The item $[S_1 \to \alpha\cdot, p, 0{:}\ldots{:}n] \in \Xi_n$ after executing PPARSE if and only if $S_1 \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_n$.

Proof: $S_1 \overset{*}{\underset{P}{\Longrightarrow}} \alpha$ trivially and theorem 5.22 shows that $\alpha \overset{*}{\underset{P}{\Longrightarrow}} a_1\cdots a_n$.

□

5.3.7 An Algorithm To Determine A Common Parse

We next consider the problem of testing if a set of items have a common parse. We define the algorithm LCHECK($\Psi_1,\ldots,\Psi_n$) that determines if the given lists of potential items, $\Psi_1,\ldots,\Psi_n$, all share a common parse. This algorithm is then used both to determine the actual parse and to elaborate the function PARSECHECK used in the previous section.

The algorithm LCHECK is basically an efficient implementation of the recursive method suggested by lemma 5.20. It first insures that there is only one set of potential items corresponding to each B_ℓ. Here determinism is used to allow simple string comparisons to eliminate duplicate lists. The algorithm then considers all possible choices of a set of items, one for each B_ℓ, corresponding to a single production of the P-grammar P. There is a common parse of the original items if and only if one of these sets also has a common parse. Thus, for each set, the algorithm is applied recursively to check all sets of associated nonterminals of these items. This last step is handled by the algorithm ICHECK. In order to insure that both algorithms run as fast as possible, the result of each recursive call to LCHECK is stored so that when the same result is needed again it does not have to be recomputed.

Formally the algorithm for LCHECK is

ALGORITHM LCHECK: Check for a Common Parse.

Given: Lists of potential items for an associated nonterminal, $\Psi_i=(B_{\ell_i},m_i@n_i)$, $1\le i\le q$.

Determine: A common parse if one exists, FALSE if not.

1. Repeat until no list can be eliminated:

   If $B_{\ell_i}=B_{\ell_j}$ for $i<j$ then if $a_{m_i+1}\cdots a_{n_i} \ne a_{m_j+1}\cdots a_{n_j}$ then return FALSE, otherwise eliminate the list Ψ_j.

2. Let the remaining lists be $\Psi_i=(B_{\ell_i},m_i@n_i)$, $1\le i\le r$. If no value has been stored for this set of lists or if some element of the set has been changed since a value was stored, then store a value of UNDEFINED and go on to step 3. Otherwise let t be the stored value. If t is the value UNDEFINED then return FALSE and otherwise return t.

3. Apply the algorithm ICHECK for each set of items $\pi_1,\ldots,\pi_r$ such that
   a) $\pi_i \in \Psi_i$;
   b) all π_i are based on a single rule p of the P-grammar P.

   If any such application returns a value other than FALSE, return this value; if all applications return FALSE, then return FALSE. In either case, replace the value of UNDEFINED set in setp 2 for this set of lists with this result.

□

The algorithm ICHECK is then given as

ALGORITHM ICHECK: Check a Set of Items for a Common Parse.

Given: A set of items, $\pi_i=[B_{\ell_i}\to\alpha_i\bullet,p,m_i:\ldots:n_i]$, $1\le i\le r$.

Determine: A common parse if one exists, FALSE if not.

1. Let t be a tree consisting of a single node with label <B,p>.

2. Let the rule p of the P-grammar P be

   $$(B_1\to\alpha_1\ ,\ B_2\to\alpha_2\ ,\ \ldots\ ,\ B_n\to\alpha_n)$$

   For each set of associated nonterminals that occur in this rule do steps a and b:

   a) If there are no occurrences of any nonterminal of this set in P, go on to the next set. Otherwise determine the lists of potential items $\Psi_1,\ldots,\Psi_j$ that correspond to occurrences of the nonterminals of this set in p.

   b) Apply LCHECK to this set of lists. If it returns FALSE then return FALSE. Otherwise add the returned tree as a son of t.

3. Return the completed tree t.

□

An example of these two algorithms is given in figure 5.7d.

We next show that these two algorithms can be used to determine a common parse. Such a parse is contained in the tree constructed by ICHECK where a node labeled <A,p> indicates that production p is used to parse the set of nonterminals associated with A. The sons of such a node indicate how each set of associated nonterminals of production p is to be parsed. This is illustrated in figure 5.7e. It is clear that such a tree with root S represents a complete parse of the given string. To show that it is a proper parse of the P-grammar we prove

LEMMA 5.24: Working in conjunction with PPARSE for some P-grammar P,
a) LCHECK returns a common parse if one exists for a set of lists of potential items and returns FALSE otherwise; and
b) ICHECK returns a common parse if one exists for a set of items and returns FALSE otherwise.

Proof: The proof procedes by induction. We assume that any item already in some potential item list represents a valid parse. Then we need to show that calls to LCHECK and ICHECK work correctly assuming that all the recursive calls they make also work correctly.

We first claim that the algorithm terminates. It is clear that the only way this could fail to occur would be if the two algorithms would call each other an unbounded number of times. But this cannot happen since the number of sets of lists of potential items that can appear in LCHECK after step 1 has been executed is bounded. Then, as step 3, and

hence all recursive calls, can only be executed once for each of these sets because of step 2, the algorithm cannot loop.

Now consider the algorithm ICHECK. It is given a set of items derived from a common production p of P and checks, using a recursive call to LCHECK, whether all the associated nonterminals of these items have a common parse. By condition B this guarantees that the original items represent a common parse. Moreover the tree set up by ICHECK indicates the production used for the appropriate set of nonterminals and, by induction, correctly identifies a complete parse of this non-terminal.

Next consider the algorithm LCHECK. It is called to determine if a common parse exists among a set of lists of potential items. Each such list represents a number of ways of parsing a given string for some nonterminal. By lemma 5.20, a common parse exists if and only if there is a selection of one item from each such list such that all items are derived from a common production and all items have a common parse. If two lists of potential items represent the same nonterminal and have a common parse then this parse must derive a single terminal string. Hence these lists must both represent this string and, it is not hard to see that all parses of these two lists must be identical. But then it is enough to check one of these lists and to insure that both are derived from a common string. This is precisely what LCHECK does in steps 1 and 2. Moreover, a set of lists has a parse with a cycle and hence will access the value UNDEFINED at step 2 if and only if it has a parse without a cycle and hence returning FALSE in this case does not

adversely affect the correctness of the algorithm. Finally, as step 3 just tries out all valid choices of items to find a common parse, the validity of ICHECK shows that this algorithm also works.

□

Using these two algorithms we can easily specify the function PARSECHECK described earlier as

ALGORITHM PARSECHECK: Check if an Item has a P-Parse.

Given: The item $\pi=[A_i \rightarrow \alpha X \bullet \beta, j, i_0 : \ldots : i_n]$.

Determine: If all nonterminals of α associated with X have a common parse.

1. If X is a terminal symbol, return TRUE.
2. If X has no associated occurrences in α, return TRUE.
3. Let $B_1, \ldots, B_k = X$ be the nonterminals associated with X in αX. Let $\Psi_1, \ldots, \Psi_k$ be the corresponding lists of potential items. Then apply LCHECK to $\Psi_1, \ldots, \Psi_k$ and return TRUE if it returns a tree and return FALSE if it returns FALSE.

□

Then using lemma 5.24 we can prove that PARSECHECK works as claimed by

LEMMA 5.25: PARSECHECK($\pi=[A_i \rightarrow \alpha X \bullet \beta, p, i_0 : \ldots : i_n]$) returns TRUE for the item π in some item list if and only if all sets of associated nonterminals in αX as used to complete this item have a common parse.

Proof: Since π is an item in a parse list, algorithm PPARSE shows by induction that all nonterminals of α not associated with X share a common parse. Thus it suffices to check if those associated with X do. But this is precisely what PARSECHECK does, calling the algorithm LCHECK for any necessary tests. Then as LCHECK works correctly by lemma 5.24, so does PARSECHECK.

□

Lemma 5.24 also shows that the output of ICHECK determines a common parse of the calling items if it returns successfully. We can use this fact to both return the parse determined by the algorithm and also to invert a GSDTS. A complete parse of the given string can be determined by simply calling the function ICHECK with a single item of the form $[S_1 \rightarrow \alpha \bullet, j, 0 : \ldots : n]$ from the item list Ξ_n. This cannot fail since the item has already been added to Ξ_n. By lemma 5.17 it suffices to apply this returned parse using the source grammar to yield the inverse of a GSDTS for the given string. Thus the above algorithm can be used to invert a GSDTS mapping. This is illustrated by the example in figure 5.7 where the GSDTS of part 5.7a is inverted for the string a^9 as shown in part 5.7f.

5.3.8 The Complexity Of The Algorithm

In this section we show that these algorithms can be used for practical automatic programming by proving that they run in polynomial time for a fixed GSDTS. Let T be a fixed GSDTS and P its target P-

grammar. We define two parameters of the grammar P, μ and ν, where μ is the maximum number of nonterminals in a production of G(P) and ν is the maximum number of translates of a production of P. Then we can show

THEOREM 5.26: The total time spent parsing the P-grammar P for a string of length n using the algorithms PPARSE, LCHECK, ICHECK and PARSECHECK is at worst $O(n^{2\mu+2\nu+\nu\mu})$.

Proof: We prove this by showing that the total time spent in each of the four algorithms separately is at worst $O(n^{2\mu+2\nu+\mu\nu})$. We first establish that there is a polynomial bound on the number of items and the number of lists of potential items. Thus

CLAIM 1:

a) The total number of items is bounded by $c_1 n^\mu$ for some constant c_1;

b) The total number of lists of potential items is bounded by $c_2 n^2$ for some constant c_2; and

c) The total number of items in a potential item list is bounded by $c_3 n^\mu$ for some constant c_3.

Let $\pi = [A_1 \to \alpha \cdot \beta, j, i_1 : \ldots : i_m]$ be an arbitrary item. Then part a follows since there are at most a constant number of ways of choosing a translate of the production of P to form the item and $\binom{n}{m}$ ways of selecting the history string for the item. Thus, as these completely determine the item, the number of items is bounded by

$$c\binom{n}{m} \sim c_1 n^m \leq c_1 n^\mu$$

as m is bounded by μ. Similarly, part b follows since the number of

ways of selecting a list of potential items, $(A_1, j@k)$, is just the number of nonterminals of G(P), a constant, times the number of ways of choosing j and k, $\binom{n}{2}$, and hence the number of lists of potential items is bounded by

$$c\binom{n}{2} \sim c_2 n^2$$

Finally part c follows immediately from part a since the number of items in a list of potential items must be less than the total number of items.

Using this claim we can show that the total time spent in PPARSE, the main algorithm, is polynomial. To do this we charge each processing step of the algorithm to some item and then bound the maximum time that can be charged to a single item. The time required in step 1 and step 2b to create a new item is charged directly to that item. Similarly an item is charged each time it is completed either successfully or unsuccessfully in step 2a and each time it is scanned in step 4. Thus the maximum time that can be charged to an item is

$$1+\mu\delta+c$$

where δ is the maximum number of times the item can be tested for completion for a single nonterminal and c is the maximum number of terminals in any production of G(P) and is a constant. Clearly δ is bounded by $c_3 n^\mu$ since the item can only be completed once for each member of some potential item list. This then gives us an upper bound on the processing time per item of

$$1+\mu c_3 n^\mu + c \sim cn^\mu$$

and allows us to prove

CLAIM 2: The total time spent executing PPARSE is bounded by $c_4n^{2\mu}$.

This follows from the above discussion since the total number of items is bounded by c_1n^{μ} and hence the total running time is bounded by

$$(c_1n^{\mu})(cn^{\mu}) \sim cc_1n^{2\mu} \sim c_4n^{2\mu}.$$

As a simple corollary to this claim we get that the maximum number of calls of the algorithm PARSECHECK is bounded by $cn^{2\mu}$. As PARSECHECK does only a constant amount of work per call (on the order of μ), this shows that the total amount of time spent executing PARSECHECK is bounded by $c_5n^{2\mu}$ for some constant c_5.

Finally, to determine the total time spent processing LCHECK and ICHECK we must determine the number of times each is called. Here we must take into account the storing of previous results whenever possible and the number of possible calls. We first show

CLAIM 3:

a) The maximum number of calls to ICHECK is bounded by $c_6n^{2\nu+\mu}$;

b) The maximum number of calls to LCHECK is bounded by $c_7n^{2\nu+\mu}$.

The number of possible calls to ICHECK is bounded by the number of sets of lists of potential items that can be used in a call. As there are cn^2 lists of potential items and since the most that can occur in a single call is bounded by the maximum number of translates of any rule of P, ν, this number is bounded by $(cn^2)^{\nu}$ or $cn^{2\nu}$. However these calls might have to be repeated once for each item that could be a possible completer since this could alter one of the lists of potential items. As the number of such items is bounded by cn^{μ}, the total number of calls

to ICHECK is bounded by

$$(cn^{2\nu})(cn^{\mu}) \sim c_6n^{2\nu+\mu}.$$

There are two ways of calling LCHECK. Each time PARSECHECK is called it can call LCHECK at most once, and each time ICHECK is called, LCHECK can be called once for each nonterminal, or at most a constant $c' \leq \mu\nu$ number of times. Hence the maximum number of calls to LCHECK is bounded by

$$c_5n^{2\mu}+c'c_6n^{2\nu+\mu} < c_7n^{2\mu+2\nu}$$

for some constant c_7.

Given these two bounds on the number of calls to LCHECK and ICHECK, we can determine the total time spent executing each by simply determining the maximum time spent on each call. For ICHECK this value is a constant, proportional to $\mu\nu$. LCHECK requires at most cn time to check equivalent strings and a constant amount for each choice of a set of items in step 3. As there are at most c_3n^{μ} items in each such list and as there are at most ν lists remaining, the total number of such choices is bounded by $(c_3n^{\mu})^{\nu} \sim c_3n^{\mu\nu}$. Hence

CLAIM 4:

a) The total time spent in executing LCHECK is bounded by $c_8n^{2\mu+2\nu+\mu\nu}$;

b) The total time spent in executing ICHECK is bounded by $c_9n^{2\mu+2\nu}$.

Point b follows immediately from the above discussion and claim 3a. Point a follows as the time spent in one execution of LCHECK is bounded by

$$cn+c_3n^{\mu\nu} < c'n^{\mu\nu}$$

and hence the total time spent is bounded by

$$(c' n^{\mu\nu})(c_7 n^{2\mu+2\nu}) \sim c_8 n^{2\mu+2\nu+\mu\nu}.$$

Then, from claim 2, the discussion regarding PARSECHECK, and claim 4, we get a bound on the total execution time of the combined algorithms of

$$c_4 n^{2\mu} + c_5 n^{2\mu} + c_8 n^{2\mu+2\nu+\mu\nu} + c_9 n^{2\mu+2\nu} \sim O(n^{2\mu+2\nu+\mu\nu}).$$

□

From this result, and the fact that every deterministic t-fst is effectively equivalent to a GSDTS, it is easy to see that

COROLLARY 5.27:
- a) The problem of inverting a fixed GSDTS has time complexity at worst $O(n^k)$ where k is a constant dependent on the GSDTS.
- b) The problem of inverting a fixed deterministic t-fst has time complexity $O(n^k)$ where k is a constant dependent on the t-fst. □

5.3.9 Special Types Of GSDTS

So far we have determined the worst case time complexity of the algorithms presented above. However the algorithms are in reality better than these results show as they will run appreciably faster than the given worst case bounds in most cases. To illustrate this we consider two common restrictions of P-grammars and show that both of these significantly reduce the worst case complexity of the algorithm.

We first consider ambiguity. Since we are in effect parsing the grammar G(P), we consider the question of whether this context free grammar is ambiguous. If it is not ambiguous we can show

LEMMA 5.28: If T is a GSDTS with target P-grammar P such that G(P) is not ambiguous, then T can be inverted in time $O(n^2)$ for a string of length n.

Proof: Since G(P) is unambiguous, we can determine the unique parse of the input string in time $O(n^2)$. Using this parse we can apply an algorithm such as LCHECK combined with ICHECK to the resultant parse. The number of calls of such an algorithm is bounded by the number of nodes in the parse tree which in turn is O(n) since G(P) is not ambiguous. Moreover, the amount of work required for each call is bounded by O(n) and hence the total time needed to check the parse is $O(n^2)$.

□

This lemma should be compared with lemma 5.9.

While complete unambiguity may be unrealistic, Earley [38] introduced the more common notion of <u>bounded ambiguity</u> in which the number of ways an item can be completed is bounded by a constant. It is easy to see that this implies that each list of potential items can contain at most a constant number of elements. Applying this to the proof of theorem 5.26 we get

LEMMA 5.29: Let T be a GSDTS with target P-grammar P such that G(P) exhibits bounded ambiguity. Then T can be inverted in time $O(n^{2\nu+1})$ for a string of length n.

Proof: From the bounded ambiguity of G(P), it follows that the amount of processing per item in PPARSE is a constant and that the total number of items is bound by $O(n^2)$. Hence the execution time of PPARSE is bounded by $O(n^2)$. Similarly the number of calls to PARSECHECK is bounded by a constant times the number of items and hence the execution time of PARSECHECK is bounded by $O(n^2)$.

For ICHECK and LCHECK we note that bounded ambiguity implies that we only reconsider some set of lists of potential items a constant number of times. Hence the number of calls to each of these routines is bounded by $O(n^{2\nu})$. Finally, the execution time of LCHECK is bounded by O(n) plus the number of ways of selecting a set of items from a set of lists of potential items. But this later value is a constant since the size of each such list is bounded by a constant. Hence the execution time of ICHECK is bounded by $O(n^{2\nu})$ and that of LCHECK by $O(n^{2\nu+1})$. As $\nu \geq 1$, $2\nu+1 \geq 2$ and the lemma follows.

□

From this lemma and the fact that a GSDTS with one translate per rule is effectively equivalent to a deterministic b-fst, we can show

COROLLARY 5.30: A deterministic b-fst can be inverted in time $O(n^3)$. □

Restrictions on the ambiguity of G(P) reduce the complexity of the problem of inversion considerably. However the problem of determining whether a given grammar is unambiguous or if it exhibits bounded ambi-

guity is undecidable in general and hence these two lemmas cannot be used in a practical sense. A decidable and hence more reasonable property of a GSDTS or a P-grammar is that of regularity. We can show a regular GSDTS can be inverted somewhat faster than a nonregular one by

LEMMA 5.31: Let T be a regular GSDTS. Then T can be inverted in time $O(n^{\mu\nu+\ell})$ where $\ell=\max(\mu,2\nu)$.

Proof: Since T is regular its domain grammar contains no cycles and any target string must be at least linear in the length of the input string. This is enough to show that G(P) contains no cycles. But then we can establish a partial ordering on the nonterminals of G(P) such that we never modify a list of potential items after it is used. In PPARSE this means that each item need only be checked once and hence we need only use a constant amount of processing per item. Thus PPARSE and PARSECHECK will execute in time $O(n^{\nu})$. Moreover the number of possible recursive calls of ICHECK and LCHECK is $O(n^{2\nu})$, a reduction by a factor of n^{μ}. Then the total number of calls of these functions is bounded by $O(n^{2\nu}+n^{\mu})$ and hence the execution time of ICHECK and LCHECK together is bounded by

$$O(n^{\mu\nu})O(n^{2\nu}+n^{\mu}) \sim O(n^{\mu\nu+2\nu}+n^{\mu\nu+\mu}) \sim O(n^{\mu\nu+\max(\mu,2\nu)}).$$

□

These lemmas illustrate several common cases where the algorithms presented in this chapter have a better worst case bound then that given in theorem 5.26. In practice the actual time complexity is even better since these worst case bounds assume that all translates are equally

bad. For example, the time complexity of parsing the P-grammar of figure 5.7 with our algorithms is actually bounded by $O(n^{2.5})$ while theorem 5.26 yields $O(n^{16})$, lemmas 5.28 and 5.29 are not applicable, and lemma 5.31 yields $O(n^{10})$. Such occurrences are common since one translate can be ambiguous while another sharing a common parse with it is completely unambiguous. In this case the unambiguous parse can be applied to the grammar to generate a string representing the ambiguous one and the proper parse can then be derived quickly by simple string comparisons. In general, the ambiguity of a parse as it effects the running time of our algorithm is limited by that of the least ambiguous translate. This makes the algorithm we have presented practical for many realistic semantics-specifying GSDTS mappings.

5.4 INVERSE TRANSLATION AND AUTOMATIC PROGRAMMING

This completes our study of the complexity of inverse translation. We have taken a fairly simple class of mappings, top-down finite state tree transducers, and determined when they can be inverted tractably and when they cannot be. As these tree transducers are one of the simplest types of semantics-specifying mappings and as we have shown that even these are intractable in many cases, we have taken a large step toward fully characterizing all semantics-specifying mappings that can be tractably inverted. This result can be summarized by

MAIN RESULT I: A top-down finite state tree transducer can be inverted tractably if and only if
1) it is a fixed deterministic t-fst; or
2) it is an arbitrary linear t-fst.

□

The object of this study on inverse mappings was to fully characterize practical automatic programming. We have achieved this goal since our model shows that automatic programming can be looked at in terms of an inverse mapping. We can then take a specific automatic programming problem and determine its tractability by noting the type of semantic specification it requires and using our study to determine the complexity of the resultant inversion problem. Moreover, we can restate the previous theorem in terms of automatic programming as

MAIN RESULT II: An automatic programming problem is tractable if and only if the semantic specification of the target language
1) is fixed and can be modeled by a deterministic t-fst; or
2) is not fixed and can be modeled by a linear t-fst.

□

In the next chapter we look further at how this study relates to automatic programming. In particular we see how the results and algorithms we have presented can be used to do practical automatic programming.

CHAPTER 6

APPLICATIONS OF THE MODEL

Our comprehensive study of the complexity of inverse translation along with our concise mathematical model provide a strong foundation for doing practical automatic programming. We illustrate this foundation in this chapter by presenting several instances of automatic programming based on the model and showing that they are worthwhile and tractable.

There are several ways of constructing automatic programming problems from our model. In the first section we consider the direct approach and look at automatic programming problems that actually consist of inverting top-down finite state tree transducers. While these are the obvious ones to consider, most problems cannot be cast in these terms. A more reasonable approach involves either extending the model or redefining the problem and the next two sections show how these two techniques can be applied to simple instances of the specific problem of automatic code generation. In the second section we consider an extension of our simple mappings that provides an interesting formalism for code generation, while in the third section we show how a general code generation problem can be redefined into an instance of practical auto-

matic programming. Finally, we conclude by discussing how our model and results can be applied to more sophisticated and complex automatic programming problems.

6.1 DIRECT APPLICATIONS

The model we have developed considers the problem of automatic programming as inverting a semantics-specifying mapping. While any type of mapping can be used in principle, we have shown that syntax-based translations, in particular top-down finite state tree transducers (t-fsts) and generalized syntax-directed translation schemata (GSDTS), are the most practical. To show that our model is realistic and that the complexity results we have derived from it are actually meaningful, we have to show that these mappings can be applied to automatic programming problems.

One such problem is that of decompiling where we are given a machine language description of some high level target language such as FORTRAN and the object is to produce a FORTRAN program equivalent to a given machine language program. Generalized syntax-directed translation schemata are a natural way of describing the semantics here as they were originally designed to define mappings that take FORTRAN programs into their machine language equivalents. With them our model shows that the problem of decompiling is actually that of inverting a GSDTS mapping, and we can directly apply our complexity results. This immediately solves the problem when it is practical and indicates when it is not.

We can also use t-fsts or GSDTS as semantic mappings for several other automatic programming problems and thus get a good estimate of their complexity and a practical method of attack. However, these transformations are not flexible enough to provide the concise and accurate semantic characterization desired for even simple problems. To get such a complete specification we have to consider either an extended model of translation or a restriced version of the problem. Often, the necessary extensions are quite simple and are actually equivalent to the original model. Moreover, it is generally possible to find useful restricted forms of a problem that are tractable. In the next two sections we consider these methods of applying our model to the problem of automatic code generation. We then generalize this approach in the final section and show how it could be used to achieve practical automatic programming for more complex problems.

6.2 AUTOMATIC CODE GENERATION: EXTENDING THE MODEL

The problem of automatic code generation is to take a program and a description of a machine and to form the machine language program whose semantics are equivalent to those of the original program. We discussed this problem in terms of our model in chapter 3 and showed that the original program could be given in terms of an operator language such that the machine description would show what each machine instruction did in terms of primitive operators. In this section we develop a particular instance of this problem in more detail and show that it can be practical.

6.2.1 The Source Language

One of the more common operator languages is that of triples. Here a program is represented as a sequence of operations each of which consists of a primitive operator and zero, one or two operands. These operands can either be the result of previous triples or can be program variables or constants. This is illustrated in figure 6.1a where an arithmetic expression is given in terms of triples. The triple representation of a program can be conveniently viewed as a graph where each node represents some component of the program. Here there are nodes for each variable, constant and operation, and there are edges connecting each operator to each of its operands. Figure 6.1b shows the graph representation of the triples in figure 6.1a. In general, this graph representation is a directed acyclic graph (dag). However, if each computed value is used only once and if program constants and variables are assigned a new node for each use, then this graph is actually a tree. We consider this simpler program tree representation as a basis for a code generation problem.

6.2.2 The Basic Machine Description

To fully define this problem we must describe the target language in terms of our semantic representation of program trees. Most machine instructions compute exactly one of the primitive operators that characterize these trees and hence as a first step we can represent the semantics of a machine language by giving this operation for each instruction. We do this with a linear bottom-up finite state tree

FIGURE 6.1

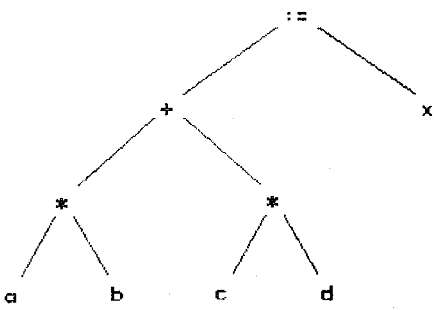
1) * , a , b

2) * , c , d

3) + ,(1),(2)

4) :=,(2), x

a) Triple representation of 'x:=a*b+c*d'



b) Tree representation of 'x:=a*b+c*d'

Figure 6.1: The Language of Triples

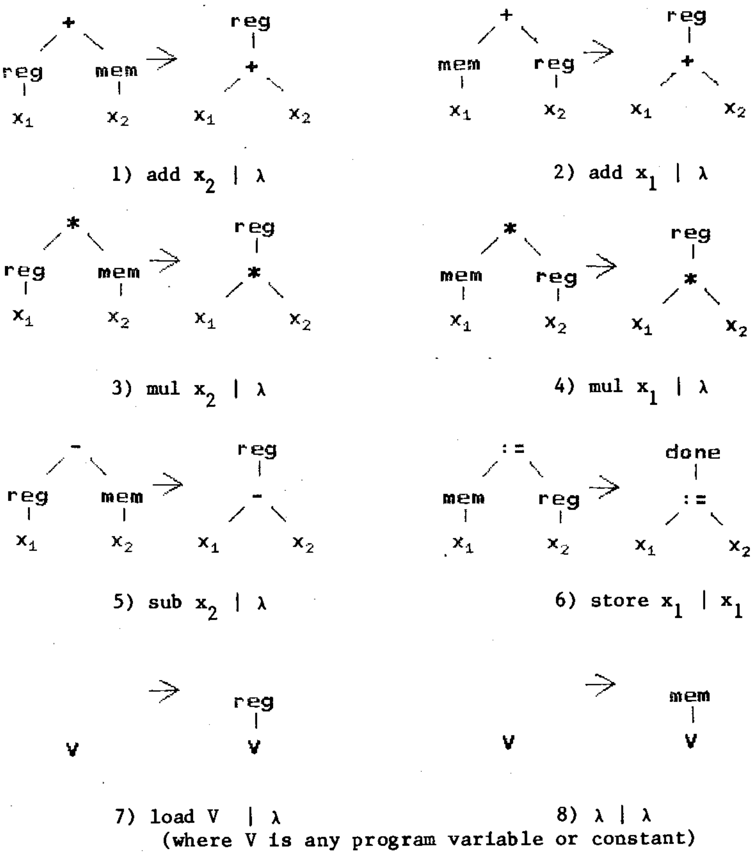
transducer (b-fst) that operates on the tree programs. Each rule of the transducer describes the operation of a single machine instruction and has associated with it the name of the instruction and the location of the result. Then the operation of this b-fst on the given program tree determines what code is generated.

In order to fill in this oversimplified description, we have to consider the possible features of the target machine. For an actual machine it might take several instructions to complete a single primitive operation and it might at the same time take several primitive operations to describe a single instruction. Moreover, the applicability of many of the instructions is limited since they may require their operands to be in registers and memory before they can be executed. All of this can be embodied in a linear b-fst. We use the finite set of states to represent the location or status of each computation by indicating when the result has been computed and whether it is in a register or in memory. These states can also encode information concerning partially completed instructions and thus can easily describe those instructions that compute several primitive operations in one step. Finally, by associating a sequence of instructions rather than a single instruction with each b-fst rule, we can describe those operations which require several instructions. Figure 6.2a shows how simple machine language can be represented by a linear b-fst using some of these techniques.

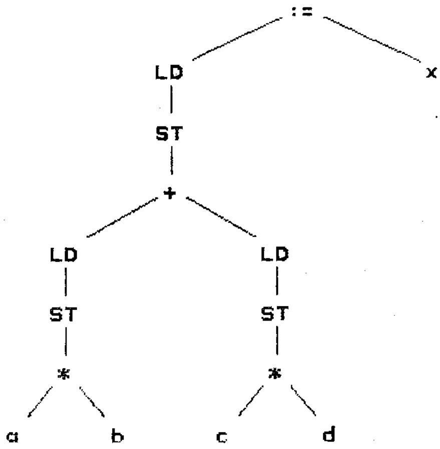
STATES

mem -- Value is in Memory
reg -- Value is in Register
done-- Value is assigned

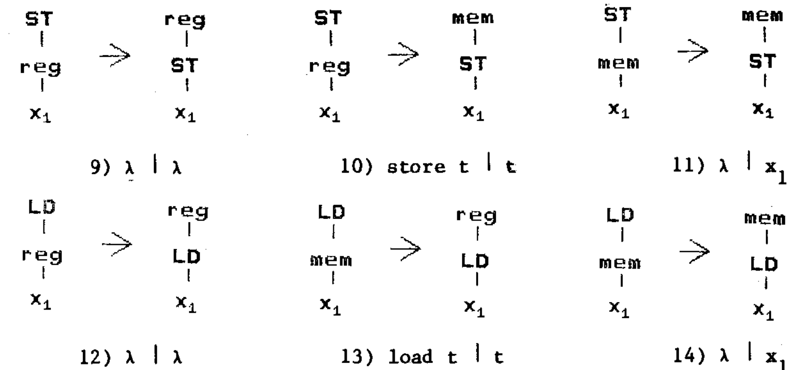
RULES



a) Basic rules describing machine language



b) Tree expression with transition nodes for 'x:=a*b+c*d'



c) Additional rules for transition nodes

VALUE	ASSIGNED TO RULES
-1	7,13
0	1,2,3,4,5,8,9,11,12,14
+1	6,10

d) The values assigned to each rule

Figure 6.2: A Simple Machine Description

This model is incomplete since it does not provide facilities for transfering data between registers and memory in order to perform the various operations. For instance the add instruction of our example requires one of its two operands in a register, and, if both were initially in memory, a load instruction would have to be generated. This can be included in our simple model by including a b-fst rule for every possible combination of states for each operator. For example, the add instrucion could be specified by giving b-fst rules for the cases where none, one or both operands are initially in registers. This however greatly increases the size of the b-fst needed to describe a given machine and since these instructions always occur between an operator and its operands, a preferred solution is to expand our tree program by adding transition nodes between each operator and its nonprimitive operands. Because we might want to do a store and then a load to free a register which must later be used, we place exactly two nodes between an operator and each of its operands. The first node, labeled LD, allows only a load, while the second, labeled ST, allows only a store. We enforce these restrictions by only introducing the appropriate rules for these nodes in our machine description. This is illustrated in figure 6.2b where the expression of figure 6.1 is rewritten with transition nodes, and in figure 6.2c where the additional rules for the simple machine of figure 6.2a are given.

In addition to these simple machine features, there are generally several other restrictions that affect our model. The most obvious of these is the fact that all machines have only a fixed number of registers of various types. An arbitrary choice of instructions or an arbitrary ordering could easily require more registers than are available or use the same register simultaneously for two distinct purposes. Unlike the previous machine features however, register usage is a global condition that cannot be handled directly by a tree transducer. We therefore extend our model of a linear b-fst to handle this global condition. We look first at the simple case where the registers are homogeneous and then consider the general case where there can be arbitrary classes of registers.

6.2.3 Code Generation With One Class Of Registers

To handle simple register usage we augment our standard model of a b-fst with a counter which keeps track of the number of available registers. Formally we define a <u>counter</u> <u>b-fst</u> as $B=<\Sigma,\Delta,Q,q_d,R,v_0)$ where Σ,Δ,Q,q_d assume their normal b-fst definitions; where v_0 is an integer that bounds and initializes a counter; and where R is a set of rules each of which consists of a standard b-fst rule and an integer. The operation of a counter b-fst B is determined by both the tree and a value v called the <u>counter</u> of B. Initially v is set to v_0. Then a rule of B is <u>applicable</u> if it is applicable in the normal sense of a tree rewriting system and if the integer associated with the rule, v_r, can be added to v so that the result satisfies $0 \leq v+v_r \leq v_0$. If a rule of B is applied, not only is the tree modified in the normal manner but also the counter is updated to be $v+v_r$.

Such a counter b-fst is clearly capable of insuring that we only use a fixed number of registers throughout a computation. We assign an integer n to a rule whose associated instructions use n registers that were not used before, and similarly assign -n to a rule whose associated instructions free n registers that were in use. Typically n will have the value one. For the example of figure 6.2, figure 6.2e contains the integers assigned to the various rules.

We can use a linear counter b-fst to do automatic code generation for our problem using a nondeterministic algorithm such as

ALGORITHM CODER: Generate Code Based on a Linear Counter B-fst.

Given: A program tree t; the number of available registers v_0; and a b-fst $B=\langle\Sigma,\Delta,Q,q_d,R\rangle$ such that each rule r has an associated code string c_r, a value string s_r, and a counter v_r.

Determine: A valid code sequence for t.

1. Let $v=v_0$. Let CODE=λ.
2. Nondeterministically choose a rule r of R and a node x of t such that r is applicable to t at x and $0\leq v+v_r\leq v_0$. If no such rule and node exist then return FAIL.
3. Apply r to t; add v_r to v; compute c'_r and s'_r by substituting the values of the sons of x as necessary in c_r and s_r. Then append c'_r to CODE and let the value of the node x be s'_r.

4. If q_d is assigned to the root of t then return CODE. Otherwise go back to step 2.

□

An example of the execution of this algorithm to produce a possible code sequence is given in figure 6.3.

This method provides a basis for practical code generation but is not practical in itself since it relies on nondeterminism. To make it practical we must make the further assumption that any result in a register can be stored in memory and then reloaded into the register. This is quite natural and allows us to consider the algorithm deterministically since a register can always be released if necessary and hence any method that properly executes the operations described by the program tree can be made to work.

The simplest approach that takes advantage of this assumption involves generating code for each operator by first generating all the code for the first son, then all the code for the second, and so on. In each case, code is generated up to but not including the topmost transition nodes. Then, when more registers are required the ST transition node of one of the previous subtrees can be used to release a register, and, as all previous registers can be released in this manner, we can ignore the register restrictions while selecting the next rule to use. This allows us to use a deterministic linear time simulation of the nondeterministic b-fst to find a set of instructions that works and then fill in the necessary stores and loads based on the register needs of this sequence. This is essentially how most compilers operate.

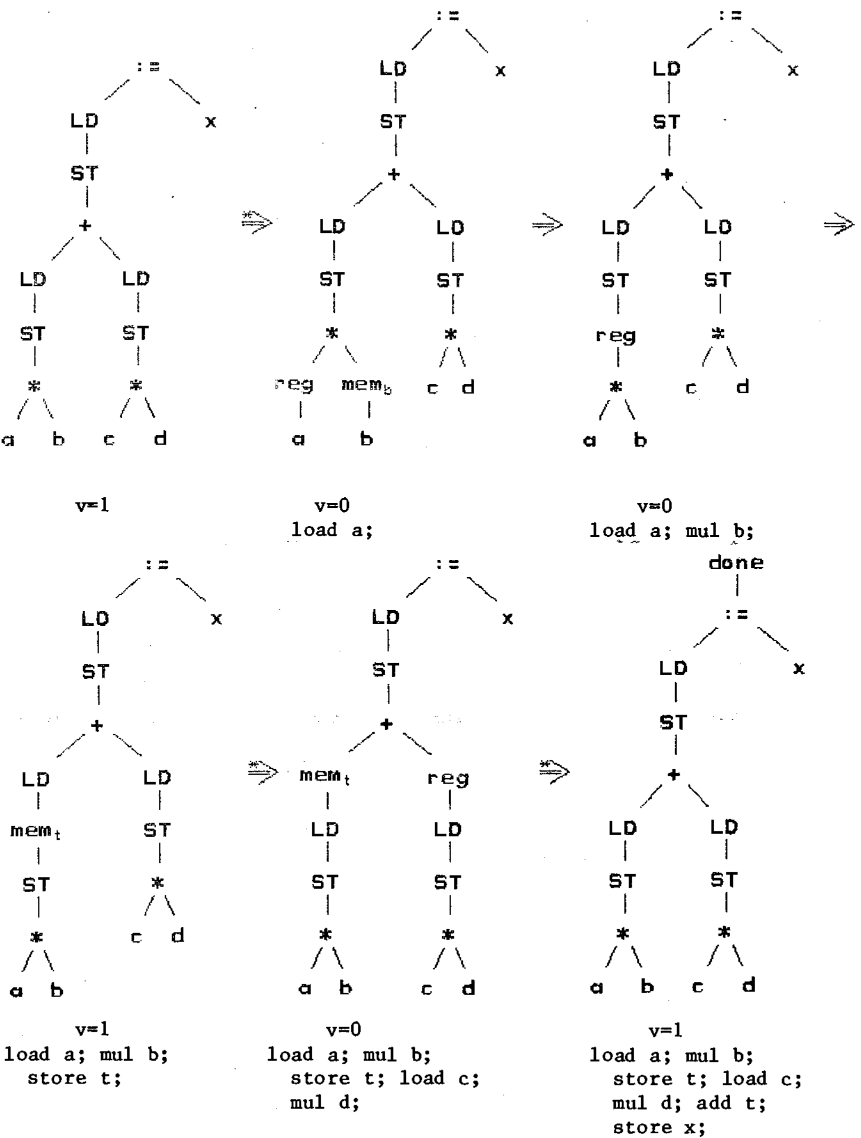


Figure 6.3: Example of Code Generation with this Model

This method of code generation for tree programs is quite good in a theoretical sense. It can be made to produce optimal code by computing the various subtrees of a node in the proper order. Aho and Ullman [11] present a method of ordering that requires only linear time and that yields the best ordering in the sense of requiring the fewest number of loads and stores. Moreover they show for any reasonable cost measure and simple machine that this method generates optimal code. Counter b-fsts provide an interesting theoretic extension of these practical results, allowing them to be applied to simple machines while still being able to model more sophisticated ones.

6.2.4 Code Generation For Arbitrary Machines

So far we have shown that our model is practical for simple machines with one class of registers. A lot of research has involved the problem of code generation for more complex machines. Most of the previous formal methods of code generation are difficult to extend to such machines. However, our methods extend quite naturally to handle them.

The states of a linear b-fst allow us to describe many types of restricted machines. By using different states for different classes of registers and using exclusive states to allow a register to be split apart, we can naturally describe such machine features as index or floating point registers and even-odd register pairs. To facilitate such descriptions we have to extend our model of a counter b-fst so that

several classes of registers can be restricted at once. We do this by generalizing the single counter v as a vector of counters V.

We assume there are n classes of registers. Then we define a <u>vector b-fst</u> as $B=\langle\Sigma,\Delta,Q,q_d,P,V_0\rangle$ where Σ, Δ, Q and q_d assume their normal definitions; where V_0 is an n-vector that represents the number of registers of each class that are available; and where P is the set of rules. Each rule in P consists of a normal b-fst rule and an n-vector. To utilize such a vector b-fst, we introduce an n-vector V called the <u>counters</u> of B. Initially we set V to V_0. Then a rule is applicable at a node if and only if the normal tree matching criteria apply and the vector associated with the rule can be added to V such that no component of the sum is negative. When such a rule is applied the tree is modified and V is changed to be the vector sum of V and the vector associated with the rule.

A vector b-fst is equivalent to a simplified vector addition system (VAS) since deciding if a valid computation exists for a given program tree is the same as determining if a given vector is reachable by a restricted VAS. While the general question of reachability for a VAS requires at least exponential time [30], vector b-fsts form a very restricted case. Because most machines only have a small number of register classes, we can limit the dimension of the VAS. Then as we restrict ourselves to program trees and only allow one addition for each node of the tree, we can also bound the number of additions that can be used to achieve reachability. Finally, as we are really talking about registers, we can give finite bounds on each coordinate of the system.

We have already seen that when there is only one class of registers, this reduced problem is easy to solve. We can generalize that result for restricted vector addition systems and hence arbitrary machines by

LEMMA 6.1: Let $V=\langle S,d\rangle$ be a n-dimensional VAS, e_0 a given vector, x the unary representation of some number. Then

a) determining if e_0 is reachable by V in x or fewer steps is NP-complete even if n=1; and

b) if each coordinate i of V is bounded by some constant N_i, then determining if e_0 is reachable by V in x or fewer steps can be done in $O(xN|S|)$ time where $N=\prod_{i=1}^{n}(N_i+1)$.

Proof: We first show that a) holds for n=3 and then show we can reduce this to the case n=1. The problem is clearly in NP as, given a sequence of at most x additions, it is easy to check if e_0 is reached in polynomial time. To show that the problem is NP-hard, we reduce the knapsack problem to it.

Consider an arbitrary knapsack problem with values $\alpha_1 \ldots \alpha_m$ and objective β. Let x=m and $e_0=\langle\beta,2^m-1,m\rangle$. Then we construct the three-dimensional vector addition system $V=\langle S,d\rangle$ where

$$d=\langle 0,0,0\rangle$$

$$S=\{\langle\alpha_i,2^{i-1},1\rangle \mid 0\le i\le m\} \cup \{\langle 0,2^{i-1},1\rangle \mid 0\le i\le m\}.$$

We claim that e_0 is reachable by V if and only if the given problem has a solution. To prove this, it suffices to show that e_0 is only reachable by a set of vectors of the form

$$\{\langle\gamma,w^{i-1},1\rangle \mid (\gamma=\alpha_i \text{ or } 0) \text{ for } 1\le i\le m\}.$$

But this follows as the only way of summing m elements of the set

$\{2^0, 2^1, \ldots, 2^{m-1}\}$ to get 2^m-1 is by using each element exactly once. To see this we note that in a sum totaling 2^m-1 based solely on powers of 2, each 2^k, $k<m$, must be achievable with $\gamma_k \geq 1$ components of the sum such that no component is counted twice. But then as there are m terms,

$$m=\gamma_0+\ldots+\gamma_{m-1}$$

which is true only if $\gamma_i=1$ for all i.

Next we show how to code the three-dimensional vectors used by V into one-dimensional vectors. It is clear that no addition can decrease the value of any coordinate as all elements of S are nonnegative. Given a vector $\langle a,b,c\rangle$ we construct the new vector $\langle z\rangle$ as

$$z= a+2^{\lceil \log \beta \rceil +\gamma}(b+c2^{m+\gamma})$$

where $\gamma= 1+\lceil \log m \rceil$. Intuitively, we are packing the three components of the original vector into a single value where enough space is provided so that no component can interfere with any other in a reachable series of additions. To show this we note that for all elements of S, c=1 and hence no more than m additions can ever be performed in a reachable series. Moreover, with m or fewer additions, the maximum number of bits that a value can propagate is $\lceil \log m \rceil$. Hence the fields are protected and the packing preserves the problem.

To show part b) of the lemma, it suffices to show that we can decide reachability in this time bound. We do this by constructing a directed graph. We first consider all (x+1)N of the (m+1)-tuples $\langle k,\gamma_1,\ldots,\gamma_m\rangle$ such that $0\leq k\leq x$ and $0\leq\gamma_i\leq N_i$ for $1\leq i\leq m$. Each such tuple will represent a node of the graph. Then for each vector $\langle \beta_1,\ldots,\beta_m\rangle$ of S, we add all directed edges from the node $\langle k,\gamma_1,\ldots,\gamma_m\rangle$ to the node

$\langle k+1, \gamma_1+\beta_1,\ldots,\gamma_m+\beta_m\rangle$ such that $k<x$ and $0\leq\gamma_i+\beta_i\leq N_i$ for all $1\leq i\leq m$. This requires time about $O(xnm|S|)$ and creates at most $(k+1)N|S|$ edges. The resultant graph is a dag since each edge is directed to a node with an increased first coordinate.

Given such a graph we can easily determine all reachable nodes. We first note that $\langle 0,d\rangle$ is reachable. Then for each $k\geq 1$ in turn, we can determine all reachable nodes with first coordinate k by considering only those nodes that are connected by an edge from some reachable node whose first coordinate is k-1. This method need only consider each edge once and hence takes time $O(xN|S|)$. Finally, given that each node is marked as reachable or not, it is simple to determine if there is a node $\langle \ell,e_0\rangle$ for some $0\leq\ell\leq x$ and hence if e_0 is reachable.

□

When we apply this lemma to the problem of code generation for an arbitrary machine specified by a vector b-fst we get

COROLLARY 6.2: Let $B=\langle\Sigma,\Delta,Q,q_d,P,V_0\rangle$ be a vector b-fst, t a program tree. Then

a) deciding if t is in the domain of B is NP-complete; and
b) if the counters V of B always satisfy $0\leq V\leq V_0$, then deciding if t is in the domain of B can be done is time $O(|t|V'|B|)$ where for $V_0=(v_1,\ldots,v_n)$, $V'=\prod_{i=1}^{n}(1+v_i)$.

Proof: These follow immediately from lemma 6.1 since a vector b-fst can make only |t| additions and since the tree t and the underlying grammar of B are arbitrary and hence any restricted VAS of the

type used by the lemma can be represented.

□

While this shows that the problem of code generation for an arbitrary machine is theoretically complex, it also shows that it is often practical. In particular it gives a linear bound for machines with one class of registers and a relatively good bound for machines with only a small number of registers of a small number of types. Thus, it shows one way of applying our model to do practial code generation for arbitrary machines.

6.3 AUTOMATIC CODE GENERATION: RESTRICTING THE PROBLEM

The approach taken in the previous section to the problem of automatic code generation is both clean and simple and yields results that illustrate what is practical for both common and arbitrary machines. But, because we approached the task by extending the model to fit the problem, these results cannot be easily generalized to a domain other than tree programs or to machines that cannot be described by linear vector b-fsts. If we want to consider such enhanced domains we must take a different approach. Rather than fixing a simple problem and extending our model, we consider the model as fixed and restrict the problem to fit it. We again consider the problem of automatic code generation and show how generalized syntax-directed translation schemata can be used to define a broad class of problems that are tractable.

To be useful an automatic code generation problem must have a powerful method of specifying both the source program and the target machine. We consider the problem where the initial program is given in terms of a flowchart and where the machine language is specified using an operational semantics such as ISP or VDL. While such a problem description is clearly ample for any practical application, it does not immediately represent an instance of practical automatic programming since both ISP and VDL are powerful enough to simulate an arbitrary Turing machine and hence can define arbitrary recursive mappings where the problem of deciding if an inverse exists is known to be undecidable [101]. To convert this into a practical problem we have to restrict the semantic specification so that the mapping that defines the target language can be given in terms of a GSDTS. We do this with a series of simple restrictions which are quite natural and which do not adversely affect the generality of the problem.

6.3.1 The General Problem

One common way of specifying an algorithm is through a flowchart or program schema. This consists of a number of blocks representing actions and connections between these blocks which represent flow of control. While there are many types of actions that can be used in a flowchart, a common simplification is to only consider three types, assignments, tests and halts. Each assignment block represents the computation of a single expression which is then assigned to a program variable. Each test block contains a single boolean expression and has

two possible exits, one that is taken if the expression evaluates TRUE and one taken when it is FALSE. And each halt block denotes the end of an execution path. While simple in nature, such flowcharts are equivalent in power to Turing machines and hence can represent an arbitrary program.

Along with this general method of representing programs, we need a powerful and useful method of defining the target machine and hence the target language. Here we use an abstract operational approach where we describe the action of the target machine in terms of a fixed semantic representation.

The primary function of any operational description of a machine is to execute each machine language instruction in the proper order and to record the effects of this execution on the state of the machine. This state is reflected in a variety of data elements. It must first include the value of all registers of the machine. Most machines have at least one accumulator and many have several. Moreover most machines have a number of internal registers containing such information as the condition code or the program counter. Secondly the machine state must include the contents of each relevant memory location. We assume that the program distinguishes between instructions and data, and thus that the state includes only the value of each program variable and temporary.

The information contained in the machine state can be divided into two categories, global and local. Global information is relevant not only to the next instruction, but also to the whole program. It

includes the contents of memory as well as any registers used for long term storage. Local information on the other hand is restricted to a smaller context either by the machine or by the nature of programming. Registers and memory used for short term temporary storage as well as internal registers such as the program counter or condition code are usually put in this class.

6.3.2 Restricting The Problem

There are two types of restrictions that have to be imposed to make our general problem a tractable one. While much of the state information is global, the type of mapping we have shown to be practical for automatic programming can only handle local information and then only by incorporating it in the finite state control or by nonlinearity. Thus the largest class of restrictions involve either eliminating global information or converting it into a small amount of local information. The second class of restrictions deal with the fact that operational semantics involve actually executing a program to determine its meaning. Since the number of steps involved here can be exponential or even infinite for very simple programs that involve loops, such an approach cannot be tractable. We must restrict our operational semantics so that they give a more static characterization of a given program and only need to give an operational description of code without loops.

Our first restriction is suggested by the nature of a flowchart where a program is represented by a number of connected blocks. We restrict our problem by generating code separately for each of these

blocks and then arranging the resultant code to form a program. Such a restriction greatly simplifies the problem. First of all it allows flow of control to be isolated as simple local information. Each statement need only indicate which one or two statements can be executed next and no flow information is needed within each statement as the corresponding machine instructions are executed in sequence.

The second restriction we consider involves symbolically executing each statement rather than actually computing the resultant values. This is necessary for several reasons. First of all, automatic code generation would be intractable if we allowed our semantics to deal with questions involving computed values. Even ordinary arithmetic with the natural numbers involves simple questions that are not recursively decidable [101]. Secondly, symbolic execution is consistent with a string representation of the flowchart blocks and the resultant code sequence. Thirdly, symbolic execution allows us to look at the problem in static rather than dynamic terms since every execution of a code string yields the identical result. Another important simplification this restriction allows is that we no longer have to keep track of the value of each program variable. Because our flowchart programs only allow a single variable to be changed at the end of a statement, we need only consider the names of these variables. This eliminates most of the global information of the original program.

However some global information does remain. In particular registers and temporaries can range over the entire program. These can be looked at as local information by merely restricting their range to a

single statement. This is a natural restriction and can be easily accomplished by assuming that each such register or temporary has an undefined value upon entering the statement and must be defined before it is used. By employing such a restriction we can guarantee that our mapping deals only with a small amount of local information.

6.3.3 The Overall Model

Given these natural restrictions to our general code generation problem, we can use a generalized syntax-directed translation schema to define the semantics of a machine language in terms of flowchart blocks. We first show how to describe a sequence of machine instructions as a string using a context free grammar and how to encode a flowchart block into a string containing all the relevant information. Then we describe the actual GSDTS mapping from the instruction string into the flowchart string and show how it is computed by simulating the machine. Throughout these formalisms we enforce the various restrictions that were discussed above. We conclude by presenting a complete GSDTS that defines the semantics of a relatively simple machine and shows that automatic code generation is actually practical within this framework.

Our first step is to characterize the context free grammar describing the machine language so that most meaningless code sequences are not defined. The grammar consists of two classes of productions. The first class describes the program primitives with rules that define each of the program variables, constants, temporaries and statement labels. The second class of rules describes the instruction set. Here

each rule allows a string of instructions to precede a single new instruction. While most rules in this class are used to define arbitrary straight line code, several of them are used to specify the operations of assignment, conditional branch and halting which characterize the three types of flowchart blocks we consider. A grammar describing the valid instruction strings of the simple machine of figure 6.4a is given in figure 6.4b.

In additon to defining the target machine language, we must characterize our semantic form. Here we start with a flowchart which can be represented by describing each of its blocks and their connections to other blocks. We first label each block of the flowchart uniquely. Then we can encode the whole flowchart as a set of strings with one string representing each block. One element of this encoding is the type of statement contained in the block. Another element is the label for the block. A third element would be the labels of any subsequent blocks, none in the case of a halt block, one for an assignment, and two for a test. Finally the string must include the semantics of the actual statement. For an assignment this means indicating the variable and the expression that are involved, while for a test it means showing the conditional expression. A possible encoding of the three types of statements is shown in figure 6.5.

Finally, given descriptions of the source and target languages we can define the GSDTS. It consists of three stages, initialization, encoding and simulation. The initialization stage describes the string representation of the various program variables, statement labels, reg-

FIGURE 6.4

```
Register:  r
Instructions:
    add m     ::  add m to r
    sub m     ::  subtract m from r
    mul m     ::  multiply m times r
    load m    ::  load m into r
    store m   ::  store r in m
    jump l    ::  jump to label l
    test m    ::  if r<m then skip
    stop      ::  halt
```

a) Machine description

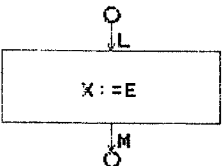
```
Primitive:
    M→K
    M→a              program variables
    M→b
    M→c
    M→d
    K→0              program constants
    K→1
    N→t              program temporaries
    L→1              program labels
    L→2
    L→3
    L→4

Straight Line Instructions:
    A→A add M;
    A→A add N;
    A→A sub M;
    A→A sub N;
    A→A mul M;
    A→A mul N;
    A→B load M;
    B→A store N;
    B→>

Statement Instructions:
    S→L:  T
    T→stop<
    T→U jump L<
    U→A store M;
    U→V jump L;
    V→A test M;
    V→A test N;
```

b) Context free grammar describing machine language

Figure 6.4: Machine Description

FIGURE 6.5

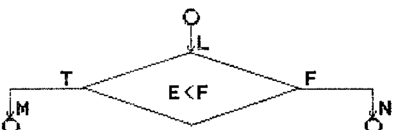
ASSIGNMENT:



Variable: X
Prefix Expression: E
Statement Type: 1
Statement Label: L
Next Statement Label: M

Coding: L1XEM

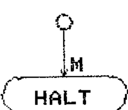
TEST:



Left Expression: E
Right Expression: F
Statement Type: 2
Statement Label: L
True Label: M
False Label: N

Coding: L2$?EF$MN

HALT:



Statement Type: 3
Statement Label: L

Coding: L3$

Figure 6.5: Sample Flowchart to String Coding

isters and temporaries. The encoding stage creates the actual target string to represent a block based on the initial series of statements, whatever special statements are needed for the particular block type, and a final branch to the next block. Finally the simulation stage interprets any proper straight line sequence of machine instructions and symbolically indicates the result of the execution.

Of these three stages, simulation is the most involved. In order to symbolically execute a series of machine instructions we must maintain the symbolic contents of each local register and temporary and indicate how these are changed by each instruction. This can be easily accomplished with a GSDTS based on the context free grammar we have described by employing one set of translates for each local item. Each production of the GSDTS involves a single machine instruction and an initial sequence of instructions. The previous register contents are available as translates of the initial sequence and the new contents of the local data are easy to describe using the resultant translates. For example, a rule for the add instruction might be

$$A \to A \text{ add } M; \quad A_1 = +A_1M_1 \quad A_2 = A_2$$

where the first translate represents a register and the second a temporary. Here the GSDTS indicates in prefix notation that after executing the add instruction, the register contains its old contents, A_1, plus the contents of the memory location described by M, and that the temporary is unchanged. We can legitimately have several equivalent interpretations of a single instruction. Thus

$$A \to A \text{ add } M; \quad A_1 = +M_1A_1 \quad A_2 = A_2$$

might also be a valid rule of the same GSDTS. This is one way in which

the notion of equivalence can be incorporated into the mapping. The simulation rules of the machine and encoding described in figures 6.4 and 6.5 are given in figure 6.6a.

There are two interesting features of simulation rules. First of all, the context free grammar they are based on insures that the evaluation order of the GSDTS is proper. This avoids all the difficulties of the previous section where order was an essential component of the solution. Secondly, this set of rules contains a single initial instruction that sets the various translates to the error symbol ϕ. This symbol does not appear in any valid semantic string and hence none of these translates can be used in an actual translation. This insures that the local data is defined before it is used.

Unlike the very general simulation rules, the GSDTS rules for encoding flowchart blocks present a very definite instruction sequence for each block type. An assignment block involves an arbitrary simulation followed by a single store instruction; a test block involves an arbitrary simulation followed by a test instruction and then a jump instruction; and a halt block involves executing only a stop instruction. Moreover, both assignment and test blocks are immediately followed by a jump instruction leading to the next block of the flowchart. The function of the encoding rules is to gather all of the appropriate information and place it into a single string of the semantic domain. The appropriate encoding rules for the example of this section are presented in figure 6.6b.

FIGURE 6.6

$A \to$ A add M;	$A_1 = +A_1 M_1$	$A_2 = A_2$
$A \to$ A add M;	$A_1 = +M_1 A_1$	$A_2 = A_2$
$A \to$ A add N;	$A_1 = +A_1 A_2$	$A_2 = A_2$
$A \to$ A add N;	$A_1 = +A_2 A_1$	$A_2 = A_2$
$A \to$ A sub M;	$A_1 = -A_1 M_1$	$A_2 = A_2$
$A \to$ A sub N;	$A_1 = -A_1 A_2$	$A_2 = A_2$
$A \to$ A mul M;	$A_1 = *A_1 M_1$	$A_2 = A_2$
$A \to$ A mul M;	$A_1 = *M_1 A_1$	$A_2 = A_2$
$A \to$ A mul N;	$A_1 = *A_1 A_2$	$A_2 = A_2$
$A \to$ A mul N;	$A_1 = *A_2 A_1$	$A_2 = A_2$
$A \to$ B load M;	$A_1 = M_1$	$A_2 = B_2$
$B \to$ A store M;	$B_1 = A_1$	$B_2 = A_2$
$B \to$ >	$B_1 = \phi$	$B_2 = \phi$

a) Simulation rules

$S \to$ L:T	$S_1 = \$ L_1 T_1$
$T \to$ stop<	$T_1 = \$ 3 \$$
$T \to$ U jump L<	$T_1 = U_1 L_1 \$$
$U \to$ A store M;	$U_1 = \$ 1 \$ M_1 \$ A_1 \$$
$U \to$ V jump L;	$U_1 = \$ 2 \$ V_1 \$ L_1 \$$
$V \to$ A test M;	$V_1 = ?A_1 M_1$
$V \to$ A test N;	$V_1 = ?A_1 A_2$

b) Encoding rules

FIGURE 6.6

$M \rightarrow K$	$M_1 = K_1$
$M \rightarrow a$	$M_1 = a$
$M \rightarrow b$	$M_1 = b$
$M \rightarrow c$	$M_1 = c$
$M \rightarrow d$	$M_1 = d$
$N \rightarrow t$	$N_1 = \phi$
$L \rightarrow 1$	$L_1 = 1$
$L \rightarrow 2$	$L_1 = 2$
$L \rightarrow 3$	$L_1 = 3$
$L \rightarrow 4$	$L_1 = 4$
$K \rightarrow 0$	$K_1 = 0$
$K \rightarrow 1$	$K_1 = 1$

c) Initialization rules

CODE: 1: >load d; add c; add b; store a; jump 2<

TRANSLATION: $1$1a+b+cd2

d) Sample use as a semantic mapping

Figure 6.6: A GSDTS as a Semantic Mapping

The final class of GSDTS rules involve the initialization process. These name all the program variables, temporaries, statement labels and constants for the purpose of symbolic execution. In the case of program variables, labels and constants they return the proper constant string. Temporaries however involve the proper choice of a translate and thus must be handled by the simulation rules. The specific initialization rules of our example are presented in figure 6.6c.

These three sets of rules form a GSDTS that is capable of defining the semantics of most sensible instruction sequences of the machine presented in figure 6.4. For example, figure 6.6d shows the mapping of a simple code sequence into its proper semantic representation.

6.3.4 Automatic Code Generation

We did not describe this GSDTS however to merely define the semantics of a sequence of machine instructions. Rather our objective is to compute the inverse mapping, going from an arbitrary semantic string representing a flowchart block and mapping it into the appropriate code sequence. Indeed, our mapping is better suited to this since its range includes most semantic strings while its domain is limited by the use of labels and jumps and the various register allocation constraints.

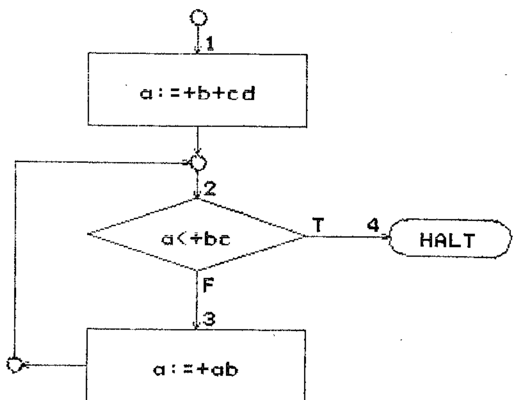
To do practical automatic code generation here we need to apply the model and results of this dissertation. We start with the flowchart representing the source program as in figure 6.7a. We next represent this program by a set of strings, one encoding each statement of the

flowchart. This is shown in figure 6.7b. Then for each of these strings we apply the algorithm of chapter 5 for inverting a GSDTS to do the actual code generation. Figure 6.7c illustrates a possible machine language translation of the whole program.

6.4 GENERAL APPLICATIONS

It should be clear from the examples of this chapter that the formal techniques of this dissertation can be used to actually solve simple automatic programming problems such as basic automatic code generation. A natural question then is how our model and results can be applied to more complex problems which are known to be intractable and yet must be solved. We conclude our discussion on the applications of our results by briefly considering this important class of problems and showing how our results can be used as a guide to achieve practical automatic programming.

Most automatic programming problems are stated in terms that make them intractable. A simple problem such as decompiling typically involves a large number of source and machine level equivalences and an optimizing compiler, and hence can not be directly viewed as a GSDTS. Automatic code generation typically involves another large set of equivalences, a high level source description, a recursive machine description, and such problems as register and memory allocation, all of which can not be easily tied together in a simple model such as we are proposing. Finally, the more numerous and sophisticated automatic programming efforts such as those of Green, Balzer and Manna are clearly



a) Sample program

1) $ 1 $ 1 $ a $ +b+cd $ 2 $

2) $ 2 $ 2 $a+bc $ 4 $ 3 $

3) $ 3 $ 1 $ a $ +ab $ 2 $

4) $ 4 $ 3 $

b) Statement codings

1) 1: >load d; add c; add b; store a; jump 2<

2) 2: >load c; add b; test a; jump 4; jump 3<

3) 3: >load b; add a; store a; jump 2<

4) 4: stop<

c) Resultant code

Figure 6.7: Sample Code Generation

dealing with problems that are at least NP-complete and hence intractable.

However, if they are handled properly all of these problems can be solved. Most current automatic programming efforts fix a semantic representation because it is convenient or seems appropriate, and then concentrate on finding heuristics that can be used with this representation. Our results show that the complexity of the problem is highly dependent on the specific semantic representation and hence that the choice of semantics and not the determination of heuristics is the crux of the problem. This shows that more effort should be put into determining a good, practical semantic representation so that considerably less effort would be required for heuristics. This is the means whereby practical automatic programming can actually be achieved.

In order to emphasize such a simple semantic approach, much of the work of an automatic programming effort will involve converting a given intractable problem into a suitable tractable subproblem. Such an effort would make use of heuristics, simplifications, world knowledge, knowledge of specific instances of the problem, simple extensions of our semantic models, and so on, in order to do the actual conversion. We have illustrated a few such conversions in the examples of this chapter, but more research is needed before more complex or sophisticated problems can be solved on a practical scale. One of the principal conclusions one can draw from the results of this dissertation is that this research would be worthwhile, that developing a practical semantic rep-

resentation for a given automatic programming problem is perhaps the best method of actually solving that problem.

CHAPTER 7
CONCLUSION

If one is going to work in the area of automatic programming there are several questions that have to be answered and several more that should be. It is both the function and the justification of theory, and particularly theory in an applied area such as computer science, to provide these answers. In this dissertation we have presented the necessary model and methods for this task.

7.1 A FIRM FOUNDATION

Automatic programming, like many other problems, is ill-defined. Before we can hope to achieve automatic programming, we must understand completely what the problem is all about. We must know what makes programming easy and what makes it difficult. We must realize what questions must be answered. In this study we address these issues. The model we construct in chapter 3 shows that programming and hence automatic programming is the problem of inverting a semantics-specifying mapping. We show that the problem is difficult because most mappings are difficult to invert. And we show that the complexity of the problem

depends almost totally on the methods used to describe the semantics of the source and target languages.

Once we know what the problem of automatic programming is all about, we must know when it is practical and when it is not. We must determine what types of automatic programming can realistically be achieved and what types are hopeless. We must decide when there is a simple, tractable algorithm that should be used and when we must resort to heuristic methods. Again, this dissertation has addressed these topics. The complexity study of inverse translation contained in chapters 4 and 5, while interesting in itself, shows precisely where the boundary between practical and impractical automatic programming lies. It demonstrates those classes of mappings that can be used for practical automatic programming and those classes that cannot be. And it lets us take any specific instance of automatic programming, determine its complexity by determining what type of mapping must be inverted, and then decide whether there is a tractable solution or whether we must apply heuristics.

Once we have determined what types of automatic programming are practical, we must make use of this knowledge. We must develop methods for taking a problem that we want to solve and converting it into one that can be solved. We must determine fast and efficient methods of doing automatic programming when it is practical. We must develop a foundation upon which heuristic methods can be applied when they are required. And again our study has provided the answers. The model of chapter 3 shows that any practical implementation of automatic program-

ming must concentrate on the semantic representation of both the original problem and the target language. The algorithms presented throughout the study provide an efficient approach to practial automatic programming. And the example applications of chapter 6 provide a direction as well as methods for redefining a given automatic programming problem so that it becomes tractable. These show how simple extensions to the model, along with restrictions of the problem, can provide a natural basis for automatic programming.

7.2 DIRECTIONS FOR FURTHER STUDY

But while this study provides the necessary foundation for automatic programming, it raises more questions than it settles. There are two principal directions for further research that could both clarify and possibly solve the problem of automatic programming. The first direction involves extending the theory that has been built up throughout this dissertation, while the second direction involves actually applying our results to automatic programming problems.

There are several theoretic questions we have not addressed. Most of our results are worst case results and, as the discussion of section 5.3.9 points out, this is often unrealistic. It would be interesting to develop results that would give the exact complexity for a specific problem or that would give better results over some restricted class of problems. Moreover, there are several other classes of mappings that might yield practical automatic programming. In particular, the class of attributed translations has been proposed [75] for semantic specifi-

cation and at the same time is quite close to the t-fst mappings we have considered. Another possibly practical extension would involve the use of dags instead of trees as a basis for mappings. While it is not obvious how or if this could be done, it could solve many current problems in the area of code generation and compiler optimization such as determining when and how optimal code can be generated from a dag representation in polynomial time [129]. Finally, the theory we have built up throughout this study supposes that the range of the semantics-specifying mapping covers the domain we are interested in. The question of how easy this is to show for any specific mapping and what should be done about it merits further attention.

Along the more practical direction, our study has provided several results that can be the basis of a real automatic programming system. Chapter 6 introduces a very elementary model for doing automatic code generation. It would definitely be worthwhile to develop this model further and achieve a practical system that would compile reasonable code for an arbitrary machine since this is an interesting and important problem and since the approach we have taken seems to be able to solve it in a practical manner. However, more research is needed to extend the simplifications and conversions that we have proposed into a truly useful system. In particular, one would like to incorporate a wider set of operators and machine instructions, and to tackle the somewhat more difficult questions of register and memory allocation. It would seem that all of this could be built into a practical system.

In addition to automatic code generation, other automatic programming problems can be handled using the methods suggested in this dissertation. One such problem is that of decompiling. This is an interesting problem that arises when machine dependent software must be transferred to another machine. By defining this problem in terms of a simple compiler along with a set of local equivalences and optimizing transformations, it would seem possible to look at and solve this problem from the simple and practical viewpoint of our model.

Finally, while it might be possible to take some of the more ambitious automatic programming efforts and cast them in terms of tree transducers and syntax-directed translations, such an approach seems somewhat unrealistic. In order to do practical automatic programming in these terms, research must be done toward finding a semantic representation which is both natural for the problem and yet simple in terms of our model. This is the line of research that holds perhaps the most potential in terms of both interesting results and possible payoff in actually achieving practical automatic programming.

BIBLIOGRAPHY

1] Alfred V. Aho.
Indexed grammars: An extension of context-free grammars.
Journal of the Association for Computing Machinery 15(4):647-671, October 1968.

2] Alfred V. Aho.
Nested stack automata.
Journal of the Association for Computing Machinery 16(3):383-406, July 1969.

3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
A general theory of translation.
Mathematical Systems Theory 3(3):193-221, September 1969.

4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

5] Alfred V. Aho, S. C. Johnson, and Jeffrey D. Ullman.
Code generation for expressions with common subexpressions.
Conference Record of the Third ACM Symposium on Principles of Programming Languages, 19-31, Association for Computing Machinery, 1976.

6] Alfred V. Aho and Jeffrey D. Ullman.
Syntax directed translations and the pushdown assembler.
Journal of Computer and Systems Science 3(1);37-56, February 1969.

7] Alfred V. Aho and Jeffrey D. Ullman.
Properties of syntax directed translation.
Journal of Computer and Systems Science 3(3):319-334, August 1969.

8] Alfred V. Aho and Jeffrey D. Ullman.
Transformations on straight line programs.
In _Proceedings of the Second Annual ACM Symposium on Theory of Computing_, 136-148, Association for Computing Machinery, 1970.

9] Alfred V. Aho and Jeffrey D. Ullman.
Characterizations and extensions of pushdown translations.
Mathematical Systems Theory 5(2):172-192, July 1971.

10] Alfred V. Aho and Jeffrey D. Ullman.
Translations on a context free grammar.
Information and Control 19(5):439-475, December 1971.

11] Alfred V. Aho and Jeffrey D. Ullman.
The Theory of Parsing, Translating and Compiling.
Two volumes. Prentice-Hall, 1973.

12] Suad Alagic.
Natural state transformations.
Journal of Computer and Systems Science 10(2):266-307, April 1975.

13] Saul Amarel.
Representations and modelling in problems of program formation.
In Bernard Meltzer and Donald Michie, editors, Machine Intelligence 6, 411-466, American Elsevier, 1971.

14] Brenda S. Baker.
Tree transductions and families of tree languages.
In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, 200-206, Association for Computing Machinery, 1973.

15] Brenda S. Baker.
Tree Transductions and Families of Tree Languages.
PhD dissertation, Harvard University, 1973. Department of Computer Science Technical Report #TR 9-73.

16] Robert M. Balzer.
A global view of automatic programming.
In Proceedings of the Third International Joint Conference on Artificial Intelligence, 494-499, Stanford Research Institute, 1973.

17] Robert M. Balzer.
Imprecise program specification.
Information Sciences Institute Research Report RR-75-36, 1975.

18] Robert M. Balzer and D. J. Farber.
APAREL: A parse request language.
Communications of the Association for Computing Machinery 12(11):624-631, November 1969.

19] F. L. Bauer and J. Eickel.
Compiler Construction: An Advanced Course.
Lecture notes in Computer Science, Volume 21, Springer-Verlag, 1976.

20] C. G. Bell and A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, 1971.

21] David B. Benson.
Syntax and semantics: A categorical view.
Information and Control 17(2):145-160, September 1970.

22] Alan W. Biermann.
On the inference of Turing machines from sample computations.
Artificial Intelligence, 3:181-198, 1972.

23] Alan W. Biermann.
Approaches to automatic programming.
In Morris Rubinoff and Marshall C. Yovits, editors, Advances in Computers, Volume 15, Academic Press, 1976.

24] Alan W. Biermann and Ramachandran Krishnaswamy.
Constructing programs from example computations.
IEEE Transactions on Software Engineering, SE-2(3):141-153, 1976.

25] E. K. Blum.
Towards a theory of semantics and compilers for programming languages.
Journal of Computer and Systems Science 3(3):248-275, August 1969.

26] Gregor V. Bochmann.
Semantic evaluation from left to right.
Communications of the Association for Computing Machinery 19(2):55-62, February 1976.

27] Ronald V. Book.
Topics in formal language theory.
In Alfred V. Aho, editor, Currents in the Theory of Computing, 1-34, Prentice-Hall, 1973.

28] Walter S. Brainerd.
The minimalization of tree automata.
Information and Control 13(5):484-491, November 1968.

29] Walter S. Brainerd.
Tree generating regular systems.
Information and Control 14(2):217-231, February 1969.

30] E. Cardoza, R. Lipton, and A. R. Meyer.
Exponential space complete problems for Petri nets and commutative subgroups.
Proceedings of the Eighth Annual ACM Symposium on Theory of Computing, 50-54, Association for Computing Machinery, 1976.

31] N. Chomsky.
On certain formal properties of grammars.
Information and Control 2(2):137-167, June 1959.

32] S. Cook.
On the complexity of theorem proving.
Proceedings of the Third Annual ACM Symposium on Theory of Computing, 151-158, Association for Computing Machinery, 1971.

33] James A. Craig, Susan C. Berezner, Homer C. Carney, and Christopher R. Longyear.
DEACON: Direct English access and control.
Proceedings of the Fall Joint Computer Conference, 365-380, American Federation of Information Processing Societies, Spartan Books, 1966.

34] Jared L. Darlington.
Automatic program synthesis in second-order logic.
In Proceedings of the Third International Joint Conference on Artificial Intelligence, 537-542, Stanford Research Institute, 1973.

35] Hamish Dewar, Paul Bratley, James Peter Thorne.
A program for the syntactic analysis of English sentences.
Communications of the Association for Computing Machinery 12(8): 476-479, August 1969.

36] Edsger W. Dijkstra.
Guarded commands, nondeterminacy and formal derivation of programs.
Communications of the Association for Computing Machinery 18(8): 453-457, August 1975.

37] John Doner.
Tree acceptors and some of their applications.
Journal of Computer and Systems Science 4(5):406-451, October 1970.

38] Jay Earley.
An Efficient Context Free Parsing Algorithm.
Phd dissertation, Carnegie-Mellon University, 1968.

39] J. Edmonds.
Paths, trees and flowers.
Canadian Journal of Mathematics, 17:449-467, 1965.

40] Clarence A. Ellis.
Probabilistic tree automata.
Information and Control 19(5):401-416, December 1971.

41] Joost Engelfriet.
Bottom-up and top-down tree transformations: A comparison.
Mathematical Systems Theory 9(3):198-231, 1975.

42] Jerome A. Feldman.
A formal semantics for computer languages and its application in a compiler-compiler.
Communications of the Association for Computing Machinery 9(1): 3-9, 1966.

43] Jerome A. Feldman and David Gries.
Translator writing systems.
Communications of the Association for Computing Machinery 11(2): 77-113, February 1968.

44] Michael J. Fischer.
Grammars with Macro-like Productions.
PhD dissertation, Harvard University, 1968.

45] R. W. Floyd.
Assigning meaning to programs.
In Schwartz, editor, Mathematical Aspects of Computer Science, proceedings of the Symposium of Applied Mathematics, number 19, 19-32.

46] Christopher Fraser.
Automatic Generation of Code Generators.
PhD dissertation, Yale University, expected 1977.

47] M. R. Garey and D. S. Johnson.
The complexity of near-optimal graph coloring.
Journal of the Association for Computing Machinery 23(1):43-49, January 1976.

48] Seymour Ginsburg.
The Mathematical Theory of Context Free Languages.
McGraw-Hill, 1966.

49] Abraham Ginzburg.
Algebraic Theory of Automata.
Academic Press, 1968.

50] James N. Gray and Michael A. Harrison.
On the covering and reduction problems for context free grammars.
Journal of the Association for Computing Machinery 19(4):675-698, October 1972.

51] C. Cordell Green, Richard J. Waldinger, David R. Barstow, Robert Elschlager, Douglas B. Lenat, Brian P. McCune, David E. Shaw, and Louis I. Steinberg.
Progress report on program understanding systems.
Stanford University Department of Computer Science Technical Report STAN-CS-74-444, 1974.

52] Sheila A. Greibach.
Inverses of Phase Structure Generators.
PhD dissertation, Harvard University, 1963.

53] Sheila A. Greibach.
Theory of Program Structures: Schemes, Semantics, Verification.
Lecture Notes in Computer Science, Volume 36, Springer-Verlag, 1975.

54] Sheila A. Greibach and John Hopcroft.
Scattered context grammars.
Journal of Computer and Systems Science 3(3):233-247, August 1969.

55] J. Hartmanis and H. B. Hunt, III.
The LBA problem and its importance in the theory of computing.
In SIAM-AMS Proceedings, Volume 7, 1-25, American Mathematical Society, 1974.

56] George E. Heidorn.
Automatic programming through natural language dialogue: A survey.
International Business Machines Research Report #RC-6074, 1976.

57] Peter Henderson.
Finite state modelling in program development.
In Proceedings of the International Conference on Reliable Software, published in SIGPLAN Notices 10(6), 221-227, 1975.

58] C. A. R. Hoare.
An axiomatic basis for computer programming.
Communications of the Association for Computing Machinery 12(10):576-583, October 1969.

59] C. A. R. Hoare.
Procedures and parameters: An axiomatic approach.
In E. Engeler, editor, Symposium on the Semantics of Algebraic Languages, 102-116, Springer-Verlag, 1971.

60] C. A. R. Hoare and P. E. Lauer.
Consistent and complementary formal theories of the semantics of programming languages.
Acta Informatica 3:135-153, 1974.

61] John E. Hopcroft and J. D. Ullman.
Formal Languages and their Relation to Automata.
Addison-Wesley, 1969.

62] Oscar H. Ibarra.
Characterizations of transductions defined by abstract families of transducers.
Mathematical Systems Theory 5(3):271-281, September 1971.

63] Edgar T. Irons.
A syntax directed compiler for ALGOL 60.
Communications of the Association for Computing Machinery 4(1): 51-55, January 1961.

64] Edgar T. Irons.
The structure and use of the syntax directed compiler.
In Richard Goodman, editor, Annual Review in Automatic Programming, Volume 3, 207-227, MacMillan, 1963.

65] Mehdi Jazayeri, William F. Odgen, and William C. Rounds.
The intrinsically exponential complexity of the circularity problem for attribute grammars.
Communications of the Association for Computing Machinery 18(12):697-706, December 1975.

66] Neil D. Jones and Steven S. Muchnick.
Even simple programs are hard to analyze.
Conference Record of the Second ACM Symposium on Principles of Programming Languages, 106-118, Association for Computing Machinery, 1975.

67] Richard M. Karp.
Reducibility among combinatorial problems.
In Raymond E. Miller and James W. Thatcher, editors, Complexity of Computer Computations, Plenum Press, 1972, 85-104.

68] Richard M. Karp.
On the computational complexity of combinatorial problems.
Networks, 5(1):45-68, January 1975.

69] Richard M Karp and Raymond E. Miller.
Parallel program schemata.
Journal of Computer and Systems Science 3(2):147-195, May 1969.

70] Shmuel Katz and Zohar Manna.
Logical analysis of programs.
Communications of the Association for Computing Machinery 19(4): 188-206, April 1976.

71] Charles H. Kellogg.
A natural language compiler for on-line data management.
In Proceedings of the Fall Joint Computer Conference, Volume 33, Number 1, 473-492, American Federation of Information Processing Societies, Thompson Book Company, 1968.

72] Ken Kennedy and Scott K. Warren.
Automatic generation of efficient evaluators for attribute grammars.
Conference Record of the Third ACM Symposium on Principles of Programming Languages, 32-49, Association for Computing Machinery, 1976.

73] James C. King.
Symbolic execution and program testing.
Communications of the Association for Computing Machinery 19(7): 385-394, July 1976.

74] Donald E. Knuth.
The Art of Computer Programming: Fundamental Algorithms.
Addison-Wesley, 1968.

75] Donald E. Knuth.
Semantics of context free languages.
Mathematical Systems Theory 2(2):127-145, July 1968. Errata in Mathematical Systems Theory 5(1):95-96, March 1971.

76] Donald E. Knuth.
Examples of formal semantics.
In E. Engeler, editor, Symposium on Semantics of Algorithmic Languages, 212-235, Springer-Verlag, 1971.

77] H. P. Kreigel and H. A. Maurer
Formal translations and the containment problem for Szilard languages.
In G Goos and J. Hartmanis, editors, Automata Theory and Formal Languages, Lecture Notes in Computer Science, Volume 33, 233-238, Springer-Verlag, 1975.

78] P. J. Landen.
A formal description of ALGOL 60.
In T. B. Steel, Jr., editor, Formal Language Description Languages for Computer Programming, 266-294, North-Holland, 1966.

79] Sandra A. Leach and Will D. Gillett.
An extended syntax-directed translation scheme.
University of Illinois at Urbana-Champaign, Department of Computer Science Technical Report UIUCDCS-R-76-795, 1976.

80] Henry F. Ledgard.
A Formal System for Defining the Syntax and Semantics of Computer Languages.
Phd dissertation, Massachusetts Institute of Technology, 1969.

81] R. C. T. Lee, C. L. Chang, and Richard J. Waldinger.
An improved program-synthesized algorithm and its correctness.
Communications of the Association for Computing Machinery 17(4): 211-217, April 1974.

82] Bruce P. Lester.
Cost analysis of debugging systems.
Masters thesis, Massachusetts Institute of Technology, Project MAC Technical Report #TR-90, 1971.

83] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns.
Attributed translations.
Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, 160-171, Association for Computing Machinery, 1973.

84] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns.
Compiler Design Theory.
Addison Wesley, 1976.

85] P. M. Lewis and R. E. Stearns.
Syntax-directed translation.
Journal of the Association for Computing Machinery 15(3):465-488, July 1968.

86] A. Malhotra.
Design criteria for a knowledge-based English language system for management: An experimental analysis.
Massachusetts Institute of Technology Project MAC Technical Report TR-146, 1975.

87] Zohar Manna.
Mathematical Theory of Computation.
McGraw-Hill, 1974.

88] Zohar Manna and Richard J. Waldinger.
Toward automatic program synthesis.
Communications of the Association for Computing Machinery 14(3): 151-165, March 1971.

89] Zohar Manna and Richard J. Waldinger.
Knowledge and reasoning in program synthesis.
Artificial Intelligence, 6(2):157-174, 1975.

90] David F. Martin and Steven A. Vere.
On syntax-directed transduction and tree transducers.
Proceedings of the Second Annual ACM Symposium on Theory of Computing, 129-135, Association for Computing Machinery, 1970.

91] William A. Martin.
Computer input/output of mathematical expressions.
In Stanley Roy Petrick, editor, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, 78-87, Association for Computing Machinery, 1971.

92] J. McCarthy.
Toward a mathematical science of computation.
In Proceedings of the International Federation Information Processing Conference, 21-28, 1962.

114] R. Simmons and J. Slocum.
Generating English discourse from semantic networks.
Communications of the Association for Computing Machinery 15(10):891-905, October 1972.

115] J. Sklansky, M. Finkelstein, and E. C. Russell.
A formalism for program translation.
Journal of the Association for Computing Machinery 15(2):165-175, April 1968.

116] Larry Snyder.
Lecture notes in semantics.
Unpublished, 1976.

117] R. E. Stearns and P. M. Lewis.
Property grammars and table machines.
Information and Control 14(6):524-549, June 1969.

118] C. Strachey.
Towards a formal semantics.
In T. B. Steel, Jr., editor, Formal Language Description Languages for Computer Programming, 198-220, North-Holland, 1966.

119] Phillip Dale Summers.
Program Construction for Examples.
PhD dissertation, Yale University, 1975. Published as Department of Computer Science Research Report #51.

120] Phillip Dale Summers.
A methodology for LISP program construction from examples.
Conference Record of the Third ACM Symposium on Principles of Programming Languages, 68-76, Association for Computing Machinery, 1976.

121] Gerald Jay Sussman.
A Computer Model of Skill Acquistition.
American Elsevier, 1975.

122] R. D. Tennent.
The denotational semantics of programming languages.
Communications of the Association for Computing Machinery 19(8): 437-453, August 1976.

123] James W. Thatcher.
Characterizing derivation trees of context-free grammars through a generalization of finite automata theory.
Journal of Computer and Systems Science 1(4):317-322, December 1967.

124] James W. Thatcher.
Transformations and translations from the point of view of generalized finite automata theory.
Proceedings of the ACM Symposium on Theory of Computing, 129-142, Association for Computing Machinery, 1969.

125] James W. Thatcher.
Generalized sequential machine maps.
Journal of Computer and Systems Science 4(4):339-367, August 1970.

126] James W. Thatcher.
Tree automata: An informal survey.
In Alfred V. Aho, editor, Currents in the Theory of Computing, 143-172, Prentice-Hall, 1973.

127] James W. Thatcher and J. B. Wright.
Generalized finite automata theory with an application to a decision problem of second order logic.
Mathematical Systems Theory 2(1):57-81, March 1968.

128] Jeffrey D. Ullman.
Applications of language theory to compiler design.
In Alfred V. Aho, editor, Currents in the Theory of Computing, 173-218, Prentice-Hall, 1973.

129] Jeffrey D. Ullman.
The complexity of code generation.
In J. F. Traub, editor, Algorithms and Complexity: New Directions and Recent Results, Academic Press, 53-70, 1976.

130] Leslie G. Valiant.
General context-free recognition in less that cubic time.
Journal of Computer and Systems Science 10(2):308-315, April 1975.

131] Gordon J. VanderBrug and Jack Minter.
State-space, problem-reduction and theorem proving: Some relationships.
Communications of the Association for Computing Machinery 18(2): 107-115, February 1975.

132] Daniel A. Walters.
Deterministic context sensitive languages.
Information and Control 17(1), part I:14-40, part II:41-61, August 1970.

133] Peter Wegner.
The Vienna definition language.
Computing Surveys, 4(1):5-63, March 1972.

93] Howard H. Metcalfe.
A parametrized compiler based on mechanical linguistics.
In Richard Goodman, editor, Annual Review in Automatic Programming, volume 4, 125-165, MacMillan, 1964.

94] J. Mezei and J. B. Wright.
Algebraic automata and context free sets.
Information and Control 11(1):3-29, July 1967.

95] Perry L. Miller.
Automatic creation of a code generator from a machine description.
Massachusetts Institute of Technology, Project MAC Technical Report #TR-85, 1971.

96] W. F. Ogden and W. C. Rounds.
Compositions of n tree transducers.
In Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, 198-206, Association for Computing Machinery, 1972.

97] Alan J. Perlis.
Automatic programming.
Quarterly of Applied Mathematics, 30(1):85-90, April 1972.

98] C. Raymond Perrault.
Intercalation theorems for tree transducer languages.
Proceedings of the Seventh Annual ACM Symposium on Theory of Computing, 126-136, Association for Computing Machinery, 1975.

99] Stanley Roy Petrick.
A Recognition Procedure for Transformational Grammars.
PhD dissertation, Massachusetts Institute of Technology, 1965.

100] Stanley Roy Petrick.
On the use of syntax-based translators for symbolic and algebraic manipulation.
In Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, 224-237, Association for Computing Machinery, 1971.

101] Hartley Rogers, Jr.
Theory of Recursive Functions and Effective Computability.
McGraw-Hill, 1967.

102] Gene F. Rose.
An extension of Algol-like languages.
Communications of the Association for Computing Machinery 7(2): 52-61, February 1964.

103] Daniel J. Rosenkrantz.
Programmed grammars and classes of formal languages.
Journal of the Association for Computing Machinery 16(1):107-131, January 1969.

104] Barry K. Rosen.
Tree-manipulating systems and Church-Rosser theorems.
Journal of the Association for Computing Machinery 20(1):160-187, January 1973. Also in Proceedings of the Second Annual ACM Symposium on Theory of Computing, 117-127, Association for Computing Machinery, 1970.

105] William C. Rounds.
Context-free grammars on trees.
Proceedings of the ACM Symposium on Theory of Computing, 143-148, Association for Computing Machinery, 1969.

106] William C. Rounds.
Tree-oriented proofs of some theorems on context-free and indexed languages.
Proceedings of the Second Annual ACM Symposium on Theory of Computing, 109-116, Association for Computing Machinery, 1970.

107] William C. Rounds.
Mappings and grammars on trees.
Mathematical Systems Theory 4(3):257-287, September 1970.

108] Gregory R. Ruth.
Automatic design of data processing systems.
Laboratory for Computer Science, Massachusetts Institute of Technology Technical Memo #70, 1976.

109] Naomi Sager and Ralph Grishman.
The restriction language for computer grammars of natural language.
Communications of the Association for Computing Machinery 18(7): 390-400, July 1975.

110] Roger Schank.
Conceptual Information Processing.
North-Holland, 1975.

111] C. P. Schnorr.
Transformational classes of grammars.
Information and Control 14(3):252-277, March 1969.

112] Peter Paul Schreiber.
Tree-transducers and syntax-connected transductions.
In G Goos and J. Hartmanis, editors, Automata Theory and Formal Languages, Lecture Notes in Computer Science, Volume 33, 202-208, Springer-Verlag, 1975.

113] Dana Scott.
Mathematical concepts in programming language semantics.
In Proceedings of the Spring Joint Computer Conference, 225-234, American Federation of Information Processing Societies, 1972.

134] John Dryer Wick.
Automatic Generation of Assemblers.
Phd dissertation, Yale University, 1975. Published as Department of Computer Science Research Report #50.

135] Terry Winograd.
Breaking the complexity barrier again.
In Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting: Programming Language--Information Retrieval, 1973. Published in SIGPLAN Notices, 10(1):13-30, January 1975.

136] W. A. Woods.
Transition network grammars for natural language analysis.
Communications of the Association for Computing Machinery 13(10):591-606, October 1970.

137] Bernard P. Zeigler.
Towards a formal model of modeling and simulation: Structure preserving morphisms.
Journal of the Association for Computing Machinery 19(4):742-764, October 1972.

INDEX

3-satisfiability 37, 114, 123

==> 16, 33, 60, 83, 129

Aho, Alfred V 44, 60, 112, 122, 180
Algorithm 141
Alphabet 14
  ranked 25
Ambiguous 18, 162, 165
  bounded 162
Applicable 31, 83, 176
Applications 167-169, 185, 200, 204
Applying a parse 17, 131
Arbitrary 70
Associate 128
Associated 30
  derivation 66
Association
  integer 129
Automatic code generation 42, 50, 185, 190, 198, 200, 206
Automatic programming 2, 39, 45, 167, 200-203, 205, 207
  practical 122, 156, 166, 200-201, 204, 207

B-fst 75, 82, 84
  counter 176-177, 180
  deterministic 163
  extended 103
  linear 172, 177
  vector 181
Baker, Brenda S 128, 131
Balzer, Robert M 40, 55, 201
Biermann, Alan W 42
Bottom-up 17, 34, 75, 136
Bounded
  ambiguity 162

Cartesian product 10
Closure 15
Code generation 170, 179-180, 184, 206
CODER 177
Common parse 136-139, 141, 150
Complete 36
Complexity 34, 121, 156, 162, 201, 204
  inverting deterministic b-fst . . 163
  inverting deterministic t-fst . . 118, 161
  inverting GSDTS 156, 161
  inverting linear nondeleting t-fst 88-89
  inverting linear t-fst 99
  inverting nonlinear t-fst 109
  inverting regular GSDTS 164
  inverting regular t-fst 113, 121
  inverting SDTS 92
  inverting t-fst 111, 121, 166
  of parsing 38
  parsing 88, 92, 110
  parsing P-grammar 156
  vector addition system 182
Composition 10
Concatenation 14-15
Condition A 136
Condition B 136
Contained in 10
Context free 18
  parsing 21
Context sensitive 18
  hardest language 37
  recognition 37
Counter 176, 181
Cycle 12, 18
Cycle-free 18, 112

Dag 12, 170, 206
Decompiling 55, 168, 200, 207
Defined over 10
Deletion 95, 99
Derivation 17
  associated 66
  tree 29
Derived grammar 135
Derives 33
  directly 16
Deterministic 63, 76, 126
Directed edges 11
Directed graph 11, 183
  acyclic 12
Domain 10
  grammar 62
Dotted rule 22, 140

Earley, Jay 22, 162
  algorithm 22, 38, 134, 138, 140, 142
Engelfriet, Joost 59, 75
English 40
Enter 12
Equivalence 47, 200
Equivalence relation 11
Example 41
Exponential 5
Expression
  language 15
  regular 15
  tree 26
EXPTIME 35
Extensions
  of model 169, 201, 205

Father 24
Feldman, Jerome A 44
Fixed 70
Flowchart 186, 188, 197
Frontier 27
Function 11
Further study 201, 205

Grammar 15, 30
  ambiguous 111
  context free 134
  derived context free 135
  domain 62
  P-grammar 128
  source 65, 128
  target 69, 79
Green, C Cordell 40, 42, 201
Gries, David 44
GSDTS 60, 64-66, 71, 121-123, 128-130, 144, 156, 161-163, 168, 190, 193-195, 197-198, 200
  nondeleting 67
  regular 122, 164
Gsm mapping 19, 58-59

Hard 36
Heidorn, George E 40
History string 140
Homomorphism 19

ICHECK 152-153, 156-157, 159-160
Indegree 12
Infix 13
Input-output-form semantics 55
Instruction Set Processor 42, 44, 186
Interior

  node . . . 24
Intersection . . . 10
Intractable . . . 121, 201
Inverse . . . 11, 48
Inversion . . . 54, 72, 85, 87, 90, 92, 99, 102, 108, 110, 112, 118, 121-123, 127, 129, 161, 163, 166

Inverting
  nonlinear t-fst . . . 108
Irons, Edgar T . . . 44
Item . . . 22, 140

Knapsack . . . 37, 182
Knuth, Donald E . . . 44

Labeling . . . 11
Language . . . 14
  generated by a grammar . . . 16
  target . . . 80, 101
LCHECK . . . 150-151, 153, 157, 159-160
Leaf . . . 24
Leave . . . 12
Leftmost . . . 18
Length . . . 14
Level . . . 24
Lewis, P M . . . 44, 60
Linear . . . 63, 76, 84
Linear bounded automata . . . 21
List of potential items . . . 140

Manna, Zohar . . . 41, 201
Many-many . . . 11
Many-one . . . 11
Mapping . . . 10, 57
Matches . . . 31, 83
Member of . . . 10
MIT . . . 40
Model . . . 45, 49

Natural language . . . 40
Nodes . . . 11
Nondeleting . . . 63, 67, 76, 84, 95
Nondeterminism . . . 20, 35, 126
Nonlinear . . . 100, 127
Nonterminal . . . 16
NP . . . 5, 35, 112
  complete . . . 5, 121, 123, 182, 201
  hard . . . 113, 182

One-many . . . 11
One-one . . . 11
Order . . . 34
Ordered . . . 25

Ordered pair . . . 10
Outdegree . . . 12, 26

P-grammar . . . 128-129, 134, 136, 156, 162
  parameters of . . . 157
  parsing . . . 132, 141
Parse . . . 84
  common . . . 136
  tree . . . 29
PARSECHECK . . . 141, 150, 155, 157, 159
Parser-transformer . . . 82, 85, 102, 106
  linear . . . 84
  nondeleting . . . 84
  regular . . . 84, 107-108
  simple . . . 84
Parsing . . . 17, 131-132
  P-grammar . . . 135, 137-138, 141
Path . . . 12
Polynomial . . . 4
Potential items
  list of . . . 140
PPARSE . . . 141, 143, 150, 153, 157-159
Practical . . . 4, 204
Predicate calculus . . . 41
Prefix . . . 13, 194
Production . . . 16
Program-form semantics . . . 56-57
PSPACE . . . 5, 35
  complete . . . 5, 37, 118, 121
  hard . . . 118
PTIME . . . 35

Random access machine . . . 34
Range . . . 10
Ranked . . . 25
Reachable . . . 12
Recursive . . . 57, 200
Reducible . . . 36
  polynomial . . . 36
Reflexive . . . 10
Regular . . . 15, 18, 84, 112
  expression . . . 15
Relation . . . 10
Restrictions
  of problem . . . 169, 185, 188-190, 201, 205
Root . . . 12, 24
Rosenkrantz, D J . . . 44
Rounds, William C . . . 59

SDTS . . . 67, 69, 71, 90, 92-93, 131
Semantics . . . 43, 45, 50, 55, 192, 198, 201-202
  axiomatic . . . 44
  denotational . . . 44

input-output-form 55
interpretive 44
operational 44, 187
program-form 56-57
translational 43
Sentential form 16
P-grammar 129
Set 10
Simple 84
Solvable 34-35
Son 24
Source 128
grammar 65
production 65, 68
Start symbol 16
State-frontier 28, 81
Stearns, R E 44, 60
String 14
mapping 61
null 14
Subgraph 12
Subtree 24
Sum 15
Summers, Phillip Dale 41
Sussman, Gerald Jay 41, 55
Symmetric 11

T-fst 59-60, 71, 167-168
deleting 96
deterministic 63, 89, 112, 118, 122, 161, 166
linear 63, 166
linear and nondeleting 71-72, 76, 85, 88-89, 93
nondeleting 63, 95
nonlinear 100, 103, 106, 108
regular 112-113
Target 128
grammar 69, 79, 130
language 80, 101
machine 172
production 65, 68, 130
Terminal 16
node 24
Thatcher, James W 59, 75
Theorem proving 56
Top-down 17, 34, 60, 136
Tractable 4
Transition 20
Transitive 11
Translate 65, 128, 140, 194
Translation 43, 65
Tree 12, 24, 170
expressions 26

mapping 61
over a ranked alphabet 26
parse 29, 60
program 170
rewriting system 30, 60, 75, 103, 176
variable 31, 60
Triples 170
Tuple 10
Turing machine 19, 186-187

Ullman, Jeffrey D 44, 60, 112, 122, 180
Undecidable 35
Union 10
Universal 22
Unrestricted language 18

Vector addition system 12, 181
Vertices 11
Vienna Definition Language 44, 186

Waldinger, Richard J 41
Worthwhile 6
Wright, J B 59