AN IMPLEMENTATION OF A PROCEDURAL LANGUAGE FOR REPRESENTING TURING MACHINES

By

Charles Bradley Slaten

Bachelor of Science

Arkansas Tech University

Russellville, Arkansas

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1987

AN IMPLEMENTATION OF A PROCEDURAL

LANGUAGE FOR REPRESENTING

TURING MACHINES

Thesis Approved:

R. E. Hedrick

Thesis Adviser

J P Chandler

Donald George

Norman N. Durham

Dean of the Graduate College

PREFACE

In preparation of this study, I would like to express my sincere appreciation to all the people who assisted me. I am especially thankful to my major adviser, Dr. G. E. Hedrick, for his guidance and valuable help.

Appreciation is also extended to the other members of my committee, Dr. George and Dr. Chandler for their advisement in the preparation of this work. I would also like to express my appreciation to Dr. D. D. Fisher who substituted for Dr. George on my oral exam and for all the wisdom he has shared with me.

My deepest appreciation goes to my wife Amanda for her patience and support. I would also like to recognize the contribution made by my daughter Kathleen Elizabeth.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objectives

In 1936 Alan Mathison Turing developed a mathematical model that expresses the ideas of an effective procedure. This model has subsequently been named the Turing Machine after its originator. Although Turing's model may seem simple, it has all of the computing capability of a general-purpose computer. Perhaps the most important concept of this mathematical model is that of the Church-Turing thesis, which states that any function which is computable, can be computed by some Turing machine (or provably equivalent model). The Turing machine is studied also for the class of languages it defines, known as the recursively enumerable sets.

In his text Abstract Machines and Grammars, Savitch presents a high-level language and shows that it can be transformed into a Turing machine which performs the same tasks. This Pascal-like language makes many of the theorems and concepts of Turing machines and computability more easily understandable to persons with a programming, rather than an exclusively formal mathematical, background.

Actually, Savitch defines three languages; TLDelta, TLDelta/S, and PSDelta, each successively more sophisticated. The last of the three, PSDelta, is the high-level language. At this writing, no production compiler exists for any of these three languages [Savitch, 1987]. The primary objectives of this thesis are: (1) to expand upon the definitions of these languages to include subprocedure declarations, (2) to implement compilers which produce a functional Turing machine simulator for them, along with a Turing machine description in standard notation; and (3) to produce compilers for use in teaching automata theory.

PSDelta provides students with a procedural language which is related closely to the Turing machine model and which can be used to solve various problems in automata theory. The output of the compilers provides students with a Turing machine description in standard notation. Therefore, students can solve problems using PSDelta as well as view the resulting Turing machine description. Construction of Turing machines can be extremely time consuming. Since PSDelta enables students to apply top-down structured programming concepts to the construction of Turing machines, students are able to construct several Turing machines within the time constraints of class assignments. Each of the three compilers provides students with a tool which simulates the execution of a Turing machine and enables them to view results which might not be

produced practically by hand.

Preliminaries

According to Hopcroft and Ullman [1979], a Turing machine consists of a finite state control, an input tape which extends into infinity in both directions and is divided into cells, and a tape head which scans one tape cell at a time. Each cell of the input tape is capable of containing exactly one symbol of a finite tape alphabet at a time. All but a finite number of cells contain a blank at any given time. The tape head points to one cell of the tape, can read the symbol at that cell, can overwrite the symbol at that cell, and can move at most one cell in either direction during any unit of time. One of the symbols of the tape alphabet is designated as the blank symbol. One state of the finite-state control is designated the start state. A subset of the set of states is designated as the set of accepting states.

Initially, n cells (for some finite $n \geq 0$) of the input tape contain symbols from an input alphabet such that the input alphabet is a proper subset of the tape alphabet. The remaining cells each hold the blank symbol, which is not an input symbol. The finite-state control is in the start state, and the tape head is positioned at the leftmost nonblank input symbol of the input tape.

The actions of a Turing machine depend upon both the state of the finite state control and the symbol currently

scanned by the tape head. During one unit of time, (1) the finite-state control changes to a state which may or may not be different, (2) the tape head changes the symbol at the tape cell currently being scanned to a new symbol which may or may not be different, and (3) the tape head moves at most one cell to the left, one cell to the right, or remains stationary. A combination of these actions forms a move of a Turing machine. The machine is said to halt when no move is defined for the current state and symbol being scanned [Hopcroft, 1979].

Definitions

> Turing machine. a (simple) Turing machine is a six-tuple $M = (S,\Sigma,\delta,s,B,Y)$ where S is a finite set of states, Σ is a finite set of symbols referred to as the tape alphabet, s is an element of S called the start state, B is an element of called the blank symbol, and Y is a subset of S called the accepting states. The third element, δ, may be any partial function from S x Σ into S x Σ x $\{<-,->,\downarrow\}$ provided that $\delta(q,a)$ is undefined whenever q is in Y. The function δ is called the transition function. If $\delta(p_1,a_1) = (p_2,a_2,->)$, then this is to be interpreted to mean the following. If the finite control of M is in state p_1 and the tape head is scanning symbol a_1, then M will do all of the following in one move: replace a_1 by a_2, change the state of its finite control to p_2, and shift its tape head one square right. If we replace -> by <- or $\downarrow$ respectively, then the tape head instructions would be changed to shift left or to remain stationary respectively [Savitch, 1982].

> Instantaneous description. An instantaneous description or id of Turing machine M is denoted by the ordered pair $(p,\alpha\triangleright\beta)$, where p is in S, $\alpha\beta$ is in Σ^*, and $\triangleright$ is a symbol not in Σ. The intuitive meaning of this id is that

the input tape contains the string $\alpha\beta$ preceded and followed by an infinite string of blanks, the current state of the finite-state control of M is p, and the tape head is positioned at the first symbol of beta.

Halting id. A halting id is an id for which the transition function is undefined [Savitch, 1982].

Tape configuration. $\alpha \triangleright \beta$ is said to be the tape configuration of id $(p, \alpha \triangleright \beta)$, assuming that α does not begin with a string of blanks, and that β does not end with a string of blanks. If α does begin with a string of blanks or β ends with a string of blanks, then the tape configuration is said to be $\gamma \triangleright \mu$, where $\gamma \triangleright \mu$ is $\alpha \triangleright \beta$ with the leading and trailing blanks removed [Savitch, 1982].

Move. If Turing machine M goes from id $(p_1, \alpha_1 \triangleright \beta_1)$ to id $(p_2, {}_2 \triangleright {}_2)$ in one step, M is said to move from id $(p_1, \alpha_1 > \beta_1)$ to id $(p_2, \alpha_2 \triangleright \beta_2)$ and is written $(p_1, \alpha_1 \triangleright \beta_1) \vdash_M (p_2, \alpha_2 \triangleright \beta_2)$ [Savitch, 1982].

Valid output. Turing machine M has valid output for input provided that $(s, \triangleright \xi) \vdash_M (q, \triangleright \beta)$ for some accepting state q.

CHAPTER II

REVIEW OF LITERATURE

History

Alan Turing's machine was actually developed in answer to a challenge. In 1900 David Hilbert presented a list of unsolved mathematical problems at the International Congress of Mathematicians in Paris. Problem twenty-three was "to discover a method for establishing the truth or falsity of any statement in a language of formal logic called predicate calculus." [Hopcroft, 1984]. Specifically, the problem was to determine whether or not an arbitrary function in the first-order calculus which was applied to integers was true. Although Turing was not present at the congress, he became familiar with Hilbert's twenty-third problem through the lectures of M. H. A. Newman.

Kurt Gödel was instrumental in the solution of this problem with his incompleteness theorem of 1931. Gödel proved that no effective procedure could exist within these limitations which could determine the truth or falsity of an arbitrary function. He did this by constructing a formula in the predicate calculus which was applied to integers, but whose definition was such that it could neither be proved

nor disproved. This statement and the formalization of the intuitive idea of an effective procedure is considered by many to be one of the great intellectual achievements of our century [Hopcroft, 1984].

While Turing was developing the solution to Hilbert's problem independently, he faced another problem: how can the concept of method be given a precise definition [Hopcroft, 1984]. By stating that a method is an algorithm, he showed a detailed process by which a method could be developed into a mathematical model. This model would be finitely describable and contain a sequence of discrete instructions which would be carried out mechanically without any creative intervention [Hopcroft, 1984]. The resulting model is called a Turing machine.

The Significance of the Turing Machine Model

Savitch [1982] in the presentation of TLDelta, TLDelta/S, and PSDelta, proves that a partial function is computed by some simple Turing machine if and only if it is computed by some program in each of these three languages. He also presents algorithms which convert TLDelta programs into Turing machines, TLDelta/S programs into TLDelta programs, and PSDelta programs into TLDelta/S programs. Therefore each of these three languages is equivalent to the Turing machine model.

The Turing Machine as a Computer of Functions of Natural Numbers

The Turing machine can be viewed as a computer of functions from the set of natural numbers onto the set of natural numbers. One accepted convention for representing integers is in unary; that is to represent the integer $x \geq 0$ by the string 1^x on the input tape. If a function has multiple arguments, each of these arguments is separated on the input tape by a single 0 [Hopcroft, 1984].

If Turing machine M halts, regardless of whether or not it is in an accepting state, the output of the function is said to be the string of nonblank characters remaining on the tape. If this string is in the form 1^y, then it is said that M computes the function $f(x) = y$. An interesting peculiarity is that Turing machine M may compute one function for one argument, a different function for two arguments, and so on [Hopcroft, 1979].

The Turing Machine as an Acceptor and a Generator of Languages

Turing machines also may be useful as recognizors or acceptors of languages. An acceptor is merely a procedure which is used to define a set [Aho, 1972]. If the Turing machine accepts a string, then the procedure must output the correct result. Otherwise the procedure is not required to output anything.

Hopcroft and Ullman [1979] define the language accepted by Turing machine M as the set of all input strings which cause M to enter a final state. This language is denoted as L(M). The languages which are accepted by at least one Turing machine that halts on all inputs are the recursive sets. In this case the input may or may not be accepted before halting.

Turing machines also may be used to represent procedures which generate the strings of a language as output. It in not necessary for such a procedure to have any input and usually is discussed assuming that there is no input. If the procedure halts, then this language is finite; otherwise it is infinite. A procedure such as this is said to enumerate the language L, where L is exactly those strings which are listed by the procedure. No restrictions are placed upon the order of the strings within the list nor upon the number of times a string may appear in the list, with the exception that each string in L must appear in the list at least once [Savitch, 1982].

There exist languages within the class of recursively enumerable languages, whose membership cannot be determined mechanically [Hopcroft, 1979]. If L(M) is such a language, then there exists a Turing machine M which must fail to accept some input which is not within L(M). If input w is in L(M), then M must eventually halt. If M is still running on some input, then it cannot be determined whether or not M will ultimately accept the input (if the machine runs long

enough) or the machine will run forever.

The Church-Turing Thesis

In the 1930's, Alonzo Church along with two of his premier graduate students from Princeton University, Stephen C. Kleene and J. Barkley Rosser, began to tackle a segment of Hilbert's problem. Church proposed that if any arbitrary mathematical function could be computed under any circumstances, it could be defined by a mathematical model provably equivalent to the Turing machine [Hopcroft, 1984].

Working independently of Church, Turing developed much the same idea, but in a different manner. Turing recognized a technical connection between Hilbert's twenty-third problem and the concept of computable functions. He developed the Turing machine as a simple, but exact model for the process of calculation. Any Turing machine can be expressed as a finite character string, just as all effective procedures are finitely describable [Hopcroft, 1979]. Therefore all possible Turing machines can be listed in alphabetical or numerical order; thus they can be paired one-to-one with the whole numbers [Hopcroft, 1984]. However, the class of functions mapping the nonnegative integers onto $\{0,1\}$ cannot be placed into one-to-one correspondence with the integers [Hopcroft, 1979]. Therefore Turing concluded that some functions are not computable.

Jones [1973] states the Church-Turing thesis as

follows:

> The Turing machine is an accurate formalization of the intuitive concept of 'effective process'. Thus any computation done by a Turing machine is intuitively effective; conversely, any intuitively effective process can be simulated by a Turing machine. In particular, (i) a function is effectively computable if and only if it is Turing computable; (ii) a set or predicate is effectively decidable if and only if it is Turing decidable (recursive); (iii) a set or predicate is effectively enumerable if and only it is recursively enumerable [Jones, 1973].

The Church-Turing thesis does not present itself for formal proof because it deals with a relation between a formally defined system and the intuitive concept of an effective procedure [Jones, 1973]. Cutland [1980] states that this thesis has the status of a claim or belief and must be verified by evidence. Cutland [1980] and Jones [1973] present several informal arguments in favor of the Church-Turing thesis.

Language Representations of Effective Procedures

In addition to the languages presented by Savitch [1982], many representations of effectively computable processes have been language oriented. One example is the lambda calculus developed by Church, Kleene, and Rosser. The Greek letter Lambda, which corresponds to the Roman letter L, was chosen by Church as the name of this formal system to suggest that it is in fact a consistent formal language. The contemporary programming language Lisp, which

is used extensively for list processing in artificial intelligence applications, is modeled on Church's lambda calculus [Hopcroft, 1984].

Martin Davis [1974] also developed languages which are provably equivalent to the Turing machine model. One such language closely resembles the style of FORTRAN. Another of Davis' languages is in essence a language representation of the Turing machine. A program in each of the languages consists of a sequence of instructions from a specified instruction set. The instructions may or may not have labels, but no two instructions can have the same label. Program execution terminates when a branch is made to a label which is not in the program or when the final instruction in the program is not a branch and that instruction is executed.

Machine Construction Tools and Program Generators

Aho, Sethi, and Ullman [1986] present a variety of software-development tools which are used in compiler construction. Two types of these tools have as their basis specific mathematical models and are of particular interest: parser generators and scanner generators. Parser generators, such as Yacc [Johnson, 1975] generally have an input based upon a context-free grammar and generate a push-down transducer as output. Scanner generators, such as Lex [Lesk, 1975] often generate lexical analyzers from an

input based upon regular expressions. A lexical analyzer is basically a finite automaton.

By using automated tools in the construction of complex program components, tasks which consume a large portion of the writing effort can be reduced to one of the easier steps in the development process. Automated development tools can also implement algorithms which are too complex to be carried out by hand effectively. Also it is often easier to produce a correct implementation of a mathematical model using a generator and a description scheme rather than to implement it directly by hand [Aho, 1986].

The two tools Yacc and Lex from the UNIX system are implemented as program generators. Instead of a subroutine, a system command, or a part of the supported features of a compiler, program generators take as input a specification of a task to be performed and produce as output a program which will perform that task. The language in which the output language is written is known as the host language. The host language can be either high or low level, although care should be taken that the generated code is as portable as possible. Both Ratfor and C are used as host languages for Yacc and Lex, however C is used more widely.

Summary

The Turing machine was developed by Alan Turing in the solution of David Hilbert's twenty-third problem: could an arbitrary function in first-order calculus applied to

integers be shown to be true? In his incompleteness theorem of 1931, Kurt Gödel proved that Hilbert's problem could not be solved. Turing's machine, which was a finitely describable mathematical model, was developed as a precise definition of an effective procedure.

The Turing machine may be used to define functions which map the natural numbers to the natural numbers. Turing machines may also be used to accept the strings of a language or to enumerate the strings of a language. There exist languages which are recursively enumerable, but whose membership cannot be determined mechanically.

The Church-Turing thesis developed independently by both Alonzo Church and Alan Turing states that any effective procedure can be defined by a Turing machine. This thesis does not present itself for proof, but is a claim which is backed by substantial evidence.

In addition to the three languages defined by Savitch, other languages have been introduced to represent effective procedures. Church, Kleene, and Rosser developed lambda calculus, a language upon which the programming language Lisp is based. Davis also defined two languages equivalent to the Turing machine model. One of these languages closely resembled FORTRAN, while the other was a language representation of a Turing machine.

Several tools have been developed to implement mathematical models. These tools can be very helpful in reducing the effort of implementing a complex program

component. Lex, a scanner generator, and Yacc, a parser generator are implemented as program generators and produce as their output C source code.

CHAPTER III

SYNTAX AND SEMANTICS OF TLDELTA, TLDELTA/S, AND PSDELTA

Introduction

In his text Abstract Machines and Grammars Savitch [1982] presents three languages, TLDelta, TLDelta/S and PSDelta, which are used to represent the Turing machine model. The first language TLDelta is a variation of the standard notation of the Turing machine model and is a subset of the second language TLDelta/S. TLDelta/S is the language TLDelta expanded to include subprocedures. TLDelta/S is a subset of the third language PSDelta. PSDelta is a high-level language which closely resembles Pascal. Savitch also shows that programs in each of these languages can be translated into equivalent Turing machines.

The Syntax of TLDelta

TLDelta is the language upon which TLDelta/S and PSDelta are based. TLDelta stands for "Turing language with alphabet Δ" [Savitch, 1982]. A TLDelta statement is either a usual statement or an accepting statement. The form of an

accepting statement is

<Label> : ACCEPT. (1)

The form of a usual statement is shown in Figure 1.

```
<Label 1> : IF <Boolean> THEN
              BEGIN
                <Assignment instruction>;
                <Pointer move>;
                GOTO <Label 2>
              END
```

Figure 1. A Usual TLDelta Statement

In this paper, a valid identifier is defined as any finite string of letters, numbers, and underscores which begins with a letter. A valid identifier cannot be a reserved word (Figure 2), although it may contain a substring which is a reserved word. In the original definition, a TLDelta label was any nonempty string of symbols without any TLDelta reserved words or TLDelta symbols. For this paper, however, a TLDelta label is any valid identifier. Each statement must have a unique label. A TLDelta boolean expression has the form

SCAN = a (2)

where "a" is any ASCII character or one of the reserved words BLANK, <YES>, <NO>, or <ANY>. An assignment instruction has the form

SCAN := b (3)

where "b" is any ASCII character or one of the reserved words BLANK, <YES>, <NO>, or <CURRENT>. The pointer moves

consist of the reserved word POINTER followed by one of the symbols ->, <-, or ↓. A TLDelta program consists of the reserved word BEGIN, followed by a sequence of TLDelta statements separated by semicolons, followed by the reserved word END.

ACCEPT	AND	<ANY>	BEGIN
BLANK	<CURRENT>	DO	ELSE
END	F	G	GOTO
IF	IN	<NO>	NOT
OR	POINTER	SCAN	STRING
THEN	WHILE	<YES>	

Figure 2. Reserved Words

:=	=	:
;	(	)
->	<-	↓

Figure 3. Reserved Symbols

The Semantics of TLDelta

A TLDelta program is used to represent a Turing machine. The labels correspond to the states of a Turing machine, SCAN corresponds to the tape symbol currently being scanned by the tape head, the symbol <CURRENT> corresponds to the tape symbol currently being scanned by the tape head, and a pointer move defines the direction which the tape head moves.

A Turing machine $M = (S, \Sigma, \delta, s, B, Y)$ can be obtained from a TLDelta program P in the following manner: Initially S,

Σ, B, and Y are empty. Let S be the set of all labels of P. Let $\Sigma = \Delta \cup \{B\}$, where B represents the blank symbol and is not an element of Δ. Let s be the label of the first statement of P. Let Y be the set of all labels of accepting statements of P.

Although <YES> and <NO> are ordinary symbols which may be in Σ, <ANY> and <CURRENT> are special symbols which are not contained in . In a boolean expression of the form

IF SCAN = <ANY> (4)

<ANY> represents any symbol in Σ. Therefore a boolean expression of this form will always be true. Similarly an assignment instruction of the form

SCAN := <CURRENT> (5)

assigns to SCAN the symbol currently being scanned. Therefore an assignment instruction of this form will not change the value of SCAN.

Define the transition function δ as follows: consider each ordered pair (<Label$_i$>,c) where <Label$_i$> is an element of S and "c" is an element of Δ or B. If <Label$_i$> is the label of some accepting statement in P, then (<Label$_i$>,c) is undefined for all c in Δ. If <Label$_i$> is the label of some usual statement s_j in P in the format of the statement in Figure 4, where <Arrow> is exactly one of the three symbols ->, <-, and $\downarrow$, then δ(<Label$_i$>,c) is determined by one of the cases given below.

Case 1: c = a and <Label 2> is the label of some statement in P.

Case 2: c = B, a = BLANK, and <Label 2> is the label of some statement in P.

Case 3: <Label 2> is not the label of any statement in P.

Case 4: c is a symbol in Δ, but a $\neq$ c.

Case 5: c = B, but a $\neq$ BLANK.

In both cases 1 and 2, δ(<Label$_i$>,c) = (<Label 2>,b,<Arrow>) if b $\neq$ BLANK and <Arrow> is exactly one of the three symbols ->, <-, and $\downarrow$. If b = BLANK then δ(<Label$_i$>,c) = (<Label 2>,B,<Arrow>). In case 3 δ(<Label$_i$>,c) is undefined. In cases 4 and 5, δ(<Label$_i$>,c) = (<Next-label>,c,$\downarrow$), where <Next-label> is the label of the next statement in P. If there is no next statement in P, then (<Label$_i$>,c) is undefined.

```
<Label_i> : IF SCAN = a THEN
              BEGIN
                SCAN := b;
                POINTER <Arrow>;
                GOTO <Label 2>
              END
```

Figure 4. A Usual TLDelta Statement

The Syntax of TLDelta/S

TLDelta/S is the language TLDelta enhanced to provide subroutines. Also, TLDelta/S has a single variable STRING, where TLDelta had none. The structure of the two languages is essentially the same. Each statement is either a usual statement or an accepting statement. The form of a usual

TLDelta/S statement is given in Figure 5. The syntax of TLDelta/S accepting statements is exactly the same as their TLDelta counterparts.

```
<Label 1> : IF <Boolean> THEN
              BEGIN
                <Assignment>;
                GOTO <Label 2>
              END
```

Figure 5. A Usual TLDelta/S Statement

A label in a TLDelta/S program is exactly the same as a TLDelta label. A TLDelta/S boolean expression is defined by Savitch to be of the form

$$\text{STRING} \in \text{A} \tag{6}$$

where "A" was any symbol representing a subprocedure which accepts strings of language A. In this paper a TLDelta/S boolean expression is of the form

$$\text{STRING IN <Language>.} \tag{7}$$

An assignment is of the form

$$\text{STRING} := \text{f(STRING).} \tag{8}$$

The symbol "f" represents a function subprocedure. Any valid identifier is considered an acceptable subprocedure name for a function.

A TLDelta/S schema consists of the reserved word BEGIN, followed by a series of TLDelta/S statements separated by semicolons, followed by the reserved word END. A TLDelta/S program is a triple (P,G,F) such that P is a TLDelta/S schema, G is an assignment of language subprocedures to P,

and F is an assignment of function subprocedures to P [Savitch, 1982]. Although the syntax of subprocedure declarations for a TLDelta/S program was not defined formally by Savitch, one form is presented in Figure 6, where $language_i$ and $function_j$ are any valid TLDdelta/S identifiers and <TLDelta program> is a TLDelta program. Note that in this context G(language) and F(function) are not functions. They are declarations for subprocedures.

$G(language_1)$=<$TLDelta\_program_1$>
$G(language_2)$=<$TLDelta\_program_2$>
.
.
.
$F(function_1)$=<$TLDelta\_program_i$>
$F(function_2)$=<$TLDelta\_program_j$>
.
.
.
<TLDelta/S_schema>

Figure 6. The Form of a TLDelta/S Program

The Semantics of TLDelta/S

A TLDelta/S schema has no meaning until a finite-state language is assigned to each procedure name for a language and a computable partial function is assigned to each procedure name for a function. The first two sections of a TLDelta/S program define these assignments.

The assignment of a function name to a language is defined by the reserved word "G", followed by the name of the language procedure in parenthesis, followed by a TLDelta

program which serves as this procedure. For input ξ, TLDelta programs which serve as acceptors of finite-state languages for TLDelta/S programs should return valid output <YES>ξ if the input string is accepted and <NO>ξ if the input string is not accepted. The assignment of a function name to a function is defined by the reserved word "F", followed by the name of the function procedure in parenthesis followed by a TLDelta program which will compute the partial function.

A TLDelta/S program begins execution at the first label of the TLDelta/S schema. A usual TLDelta/S statement in the form of the statement shown in Figure 7 determines whether or not the contents of STRING is an element of the language A. If STRING is an element of the language A, then the instructions contained in the BEGIN-END block are executed. Otherwise either the following statement is executed or the program abnormally terminates if there is no following statement. An assignment instruction assigns to STRING the result of the application of function f to the contents of STRING. If the function f is undefined for STRING, then the program abnormally terminates. Otherwise the program continues execution at the statement labeled by <Label 2>, or terminates abnormally if no statement labeled by <Label 2> exists. A TLDelta/S accepting statement normally terminates the program when executed.

```
<Label 1> : IF STRING IN A THEN
              BEGIN
                STRING := f(STRING);
                GOTO <Label 2>
              END
```

Figure 7. A Usual TLDelta/S Statement

TLDelta/S program (P,G,F) is said to have valid output ξ for input α provided that the program terminates normally by executing an accepting statement. Initially the contents of STRING are read from the standard input. Upon termination, the variable STRING contains ξ.

The Syntax of PSDelta

Although TLDelta/S is a much nicer language to work with than TLDelta, it is still very cumbersome compared with many modern programming languages. PSDelta is a language based upon TLDelta/S, but has nicer control structures and more variables. The only way to change the flow of control in a TLDelta/S program is through the use of a GOTO instruction. PSDelta provides no GOTO instructions, but uses IF-THEN-ELSE, BEGIN-END, and WHILE-DO as a means to combine simple statements together to get more complicated statements. TLDelta/S has only one variable STRING. A PSDelta program can have any number of variables STRING1, STRING2, STRING3, ... The boolean expressions of PSDelta allow the use of AND, OR, and NOT to form complex expressions, where TLDelta/S allowed only simple boolean

expressions.

A PSDelta variable is represented by STRINGi, where "i" is any base ten numeral with no leading zeros. A PSDelta statement is either an assignment statement (originally called an atomic statement by Savitch, 1982), an IF-THEN-ELSE statement, a WHILE-DO statement or a BEGIN-END block. The two forms of an assignment statement are given in Figure 8. STRINGi, STRINGj, and STRINGk are variables and "f" is a name for a function subprocedure.

```
STRINGi := f(STRINGj)
STRINGi := STRINGjSTRINGk
```

Figure 8. Two Forms of a PSDelta Assignment Statement

An atomic boolean expression has the form

STRINGi IN A (9)

where STRINGi is a variable and "A" is the name of a subprocedure which accepts strings of language A. A boolean expression is either an atomic expression or one or more atomic expressions used in conjunction with some combination of the operators NOT, AND, or OR. Parenthesis also may be used in a boolean expression to impose a precedence upon the operators. NOT is a unary operator and requires one operand, while AND and OR are binary operators requiring two operands.

The form of an IF-THEN-ELSE statement is given in Figure 9. <Boolean> is any boolean expression.

<Statement i> and <Statement j> are any PSDelta statements. The ELSE portion of the IF-THEN-ELSE is required in PSDelta. The form of the WHILE-DO statement is shown in Figure 10. <Boolean> is any boolean expression and <Statement> is any PSDelta statement. The form of the BEGIN-END block is shown in Figure 11. <Statement i> is any PSDelta statement. Through the use of these three constructs, statements may be nested in any way desired.

```
IF <Boolean> THEN
               <Statement i>
             ELSE
               <Statement j>
```

Figure 9. A PSDelta IF-THEN-ELSE Statement

```
WHILE <Boolean> DO
  <Statement>
```

Figure 10. A PSDelta WHILE-DO Statement

```
BEGIN
  <Statement i>;
  <Statement j>;
        .
        .
        .
  <Statement n>
END
```

Figure 11. A PSDelta BEGIN-END Block

There is no distinction between a PSDelta schema and a

PSDelta statement. A PSDelta program is a triple (P,G,F) where P is a PSDelta schema, G is an assignment of language subprocedures to P, and F is an assignment of function subprocedures to P [Savitch, 1982]. Although the form of a PSDelta program was not formally defined by Savitch, one form is presented in Figure 12, where $language_i$ and $function_j$ are any valid TLDdelta/S identifiers and <TLDelta/S program> is a TLDelta/S program. Note that in this context G(language) and F(function) are not functions. They are declarations for subprocedures.

$G(language_1)$=<$TLDelta\_program_1$>
$G(language_2)$=<$TLDelta\_program_2$>
.
.
.
$F(function_1)$=<$TLDelta/S\_program_i$>
$F(function_2)$=<$TLDelta/S\_program_j$>
.
.
.
<PSDelta_schema>

Figure 12. The Form of a PSDelta Program

The Semantics of PSDelta

Like TLDelta/S, a PSDelta schema has no meaning until a finite-state language is assigned to each procedure name for a language and a computable partial function is assigned to each procedure name for a function. The first two sections of a PSDelta program define these assignments.

The assignment of a function name to a language is defined by the reserved word "G", followed by the procedure

name of the language in parenthesis, followed by a TLDelta program which serves as this procedure. The assignment of a function name to a computable partial function is defined by the reserved word "F", followed by the procedure name of the function in parenthesis, followed by a TLDelta/S program which computes the partial function.

An assignment statement of the form

STRINGi := f(STRINGj) (10)

changes the contents of STRINGi to the result of the function named by f applied to the contents of STRINGj. If $j \neq i$, then the contents of STRINGj are not altered. If the function named by f is undefined, the program terminates abnormally. An assignment statement of the form

STRINGi := STRINGjSTRINGk (11)

changes the contents of STRINGi to the contents of STRINGj concatenated with the contents of STRINGk. The contents of STRINGj and STRINGk are not altered unless j=i or k=i.

A boolean expression of the form

STRINGi IN A (12)

is true if the contents of STRINGi is an element of the language A. Otherwise the boolean expression is false. A boolean expression of the form

NOT <Boolean> (13)

where <Boolean> is any boolean expression is true if <Boolean> is false and false if <Boolean> is true. A boolean expression of the form

$<\text{Boolean}_i>$ AND $<\text{Boolean}_j>$ (14)

is true only if both <Boolean$_i$> and <Boolean$_j$> are true; otherwise it is false. A boolean expression of the form

<Boolean$_i$> OR <Boolean$_j$> (15)

is true if at least one of <Boolean$_i$> and <Boolean$_j$> is true; otherwise it is false.

An IF-THEN-ELSE statement of the form given in Figure 9 has the same effect as <Statement i> if <Boolean> is true and the same effect as <Statement j> if <Boolean> is false. A WHILE-DO statement of the form given in Figure 10 has the same effect as executing <statement> again and again as long as <Boolean> is true. A BEGIN-END Block of the form given in Figure 11 has the same effect as executing <Statement i>, <Statement j>, ..., <Statement n> one after the other.

Initially the input for a PSDelta program is read from the standard input and placed into the variable STRING1. All other variables contain the empty string. A PSDelta program begins execution at the first statement of the PSDelta schema and continues execution sequentially until the last statement has been executed. A PSDelta program is said to have output β for a given input provided that upon normal termination of the program STRING1 contains β. A PSDelta program is said to compute the partial function f provided that for input α the program normally terminates with STRING1 containing $f(\alpha)$. Every PSDelta Program defines a unique partial function.

Summary

TLDelta is a variation of the standard notation of the Turing machine model and can be directly converted into a Turing machine. TLDelta is the language upon which TLDelta/S and PSdelta are based and is a subset of both of these languages. TLDelta/S is the language TLDelta enhanced to provide subroutines and a single variable STRING. TLDelta/S is a subset of PSDelta. PSDelta is a procedural language whose form closely resembles Pascal. PSDelta has nicer control stuctures than TLDelta/S and provides any number of variables.

CHAPTER IV

LANGUAGE TRANSLATION METHODS

Introduction

TLDelta is the language upon which TLDelta/S and PSDelta are based. TLDelta can be transformed directly into a Turing machine realization, while TLDelta/S may be transformed into a TLDelta program, and PSDelta may be transformed into a TLDelta/S program.

Translation from TLDelta to a Turing Machine

A TLDelta program P is used to represent a Turing machine M known as the Turing machine realization of P. The labels correspond to the states of a Turing machine, SCAN corresponds to the tape symbol currently being scanned by the tape head, and a pointer move defines the direction which the tape head moves.

A Turing machine $M = (S,\Sigma,\delta,s,B,Y)$ can be obtained from a TLDelta program P in the following manner: Initially S, Σ, B, and Y are empty. Let S be the set of all labels of P. Let Σ be Δ. Let s be the label of the first statement of P. Let Y be the set of all labels of accepting statements of P.

Define the transition function δ as follows: consider each ordered pair (<label$_i$>,c) where <label$_i$> is an element of S and c is an element of Δ or B. If there is no statement in P labeled by <label$_i$>, then δ(<label$_i$>,c) is undefined for all c in Δ. If <label$_i$> is the label of some accepting statement in P, then δ(<label$_i$>,c) is undefined for all c in Δ. If <label$_i$> is the label of some usual statement s_j in P in the format of the statement in Figure 4, where <Arrow> is exactly one of the three symbols ->, <-, and $\downarrow$, then δ(<label$_i$>,c) is determined by one of the five cases given below.

Case 1: c = a and <label 2> is the label of some statement in P.

Case 2: c = B, a = BLANK, and <label 2> is the label of some statement in P.

Case 3: c = <ANY>.

Case 4: <label 2> is not the label of any statement in P.

Case 5: c is a symbol in Δ, but a $\neq$ c.

Case 6: c = B, but a $\neq$ BLANK.

In both cases 1 and 2, δ(<label$_i$>,c) = (<label 2>,b,<Arrow>) if b $\neq$ BLANK, where <Arrow> is exactly one of the three symbols ->, <-, and $\downarrow$. If b = BLANK then δ(<label$_i$>,c) = (<label 2>,B,<Arrow>). In case 4 δ(<label$_i$>,c) is undefined. In cases 5 and 6, δ(<label$_i$>,c) = (<next-label>,c,), where <next-label> is the label of the next statement in P. If there is no next statement in P,

then (<label$_i$>,c) is undefined.

Translation from TLDelta/S to TLDelta

A TLDelta/S program consists of one or more TLDelta languge subprocedures, one or more TLDelta function subprocedures, and a TLDelta/S schema. Both language subprocedures and function subprocedures are implemented as macro expansions in the TLDelta/S statement which calls them. Savitch [1982] states that two programs are input/output equivalent provided that they both compute the same partial function. A function subprocedure P is input/output equivalent to a subprocedure P" such that (1) if P computes the partial function f, then P" also computes f, and (2) for any input, if P" reaches an accepting statement in the computation of P", then P" has a valid output.

Language subprocedures for a TLDelta/S program must be in a certain format. IF A is any finite-state language over Δ, then the desired TLDelta program h_A has the following property: for each ξ in A, $h_A(\xi)$ = <YES>ξ; for each ξ in Δ^* - A, $h_A(\xi)$ = <NO>ξ. TLDelta program h_A may be constructed by the following method. Let M be a deterministic finite-state acceptor which accepts language A. Let $p_0,p_1,\ldots,p_m$ be a list without repetition of all the states of M and let p_0 be the start state. Let $a_0,a_1,\ldots,a_n$ be a list without repetition of all the symbols of M.

Choose the names of the states of M to be TLDelta labels and choose M so that (p_i,a_j) is defined for all p_i and a_j. Let $q_{ij} = \delta(p_i a_j)$, i=0,1,2,...,m and j=0,1,2,...,n. A context-free grammar which generates a TLDelta program for h_A is given in Appendix D.

A computation on input ξ proceeds as follows. The block of code <M code> is executed first. This code simulates M with except for the handling of blanks. Whenever a nonaccepting state is reached, the block <end no?> is executed and whenever an accepting state is reached, the block <end yes?> is executed. These blocks check to see if all of ξ is read. They do this by checking for a blank symbol. If all of ξ is read and an accepting state is reached, then the **GOTO** INA is executed and control passes to the block <yes rewind>. If all of ξ is read and the last p_i was not an accepting state, then control goes from the block <M code> to the block <no rewind>. So after <M code> is executed, all of ξ is read, and control passes to either <yes rewind> or <no rewind> depending on whether or not ξ is in A. Both of these rewind blocks move the pointer to the front of ξ. <yes rewind> puts <yes> in front of ξ. <no rewind> puts <no> in front of ξ. Finally, whichever rewind block is executed, the program ends by a **GOTO** EXITA.

Function subprocedures for TLDelta/S programs may be any TLDelta program. A test is included into the code of each TLDelta program P serving as a function subprocedure to

see if it has a valid output for a given input. This additional code checks to see if the program reaches an accepting statement in its computation on an input. If an accepting statement is reached, then the input tape is checked to determine whether or not it contains a single string of nonblank characters and the tape head is in the correct position.

If f is computed by a TLDelta program P, then a program P" may be constructed such that (1) P" also computes f, (2) For any input α, if P" reaches an accepting statement in the computation of P" on α, then P" has a valid output for the input α. The alphabet Γ for P" will be expanded so that $\Gamma = \Delta \cup \{$<dirty blank>$\}$, where the symbol <dirty blank> is a new symbol that will serve as a pseudo blank. The pseudo blank symbol will serve as a blank symbol, but will mark the portion of the input tape which is scanned. Program P" will simulate P with the exception of the reading and writing of blank symbols. Whenever P would write a blank symbol, P" will write a <dirty blank> symbol. Whenever P" reads a <dirty blank> symbol, P" will simulate the program P reading a true blank symbol. By simulating the execution of P in this manner the input tape may be checked for the correct format. Every cell scanned in the simulation of P must be checked to see that there are not any two symbols of Δ with one or more blank or <dirty blank> symbols between them, and that the tape head is positioned at the first nonblank symbol. Every cell scanned by P" will contain either a

symbol of Δ or the <dirty blank> symbol. Therefore the portion of the input tape to be checked is marked by a blank symbol at each end.

In the construction of P", let <P code> be the program P without the enclosing BEGIN-END. Define program P" to be the program shown in Figure 13 where <new label>, <abort>, and <formcheck> are new labels and <abort> does not label any statement in P". <new P code> is a block of code obtained from <P code> by the algorithm described in Appendix E. <check code> checks that the requirements for a valid output are satisfied. If the requirements are met, then <check code> produces the output that P would produce and transfers control to an accepting statement. Otherwise control is transferred to <abort>.

```
BEGIN
                        <new P code>;
     <new label> : GOTO <abort>;
     <formcheck> : <check code>
END
```

Figure 13 Program P"

A TLDelta/S program (P,G,F) may be changed into an equivalent TLGamma program P'. A TLGamma program is syntactically equivalent to a TLDelta program; however, the alphabet is expanded. First we replace each usual statement of the TLDelta/S program by a block of TLGamma code that has the same effect as the TLDelta/S statement. To obtain P'

from P, replace every usual statement of P by a block of code obtained by the following method. Consider a usual TLDelta/S statement of P as shown in Figure 14. Let BEGIN <h_A code> END be a TLDelta program for the function h_A obtained from G(A). Let BEGIN <f code> END be a program for the partial function F(f) modified as previously described. The usual TLDelta/S statement shown in Figure 14 is replaced by the code shown in Figure 15. The parts of the code are defined as follows:

1. <switch label> and <f label> are new labels.
2. The block <new h_A code> is <h_A code> modified such that every accepting statement <label> : ACCEPT is replaced by the code in Figure 16.
3. <yes/no switch> is the code shown in Figure 17. <correct label> is the label of the next TLDelta/S statement provided there is a next TLDelta/S statement. If this is the last TLDelta/S statement, then <correct label> is a new label which does not label any statement.
4. <new f code> is <f code> modified as follows: every accepting statement <label> : ACCEPT is replaced by the code in figure 18.

```
<label 1> : IF STRING IN A THEN
              BEGIN
                STRING := f(STRING);
                GOTO <label 2>
              END
```

Figure 14 Usual TLDelta/S Statement of P

```
<label 1> : IF SCAN = <ANY> THEN
              BEGIN
                SCAN := <CURRENT>;
                POINTER ↓;
                GOTO <new h_A code>;
              END;
    <new h_A code>;
<switch label> : <yes/no switch>;
<f label> : IF SCAN = <ANY> THEN
              BEGIN
                SCAN := <CURRENT>;
                POINTER ↓;
                GOTO <new f code>;
              END;
    <new f code>
```

Figure 15 TLGamma Code for Usual TLDelta/S Statement

```
<label> : IF SCAN = <ANY> THEN
            BEGIN
              SCAN := <CURRENT>;
              POINTER ↓;
              GOTO <switch label>
            END
```

Figure 16 New h_A Accepting Statement

```
IF SCAN = <yes> THEN
  BEGIN
    SCAN := BLANK;
    POINTER->;
    GOTO <f label>
  END;
IF SCAN = <no> THEN
  BEGIN
    SCAN := BLANK;
    POINTER->;
    GOTO <correct label>
  END
```

Figure 17 <yes/no switch> Code

```
<label> : IF SCAN = <ANY> THEN
            BEGIN
              SCAN := <CURRENT>;
              POINTER ↓;
              GOTO <label 2>
            END
```

Figure 18 New F Code Accepting Statement

To get program P' which is equivalent to TLDelta/S program (P,G,F), replace each usual TLDelta/S statement by the code produced by the method just described. Only the usual statements of the TLDelta/S schema are altered. The accepting statements remain as they are. The labels of <h_A code> and <f code> must have different label names from each other. The resulting program is input/output equivalent to the TLDelta/S program (P,G,F).

Suppose that program (P,G,F) is a TLDelta/S program such that Δ contains at least two symbols and F(f) is a

total function for each subprocedure. Under these conditions, there is no need to check for valid output and the original TLDelta code may be used in place of TLGamma code. In order to change a TLGamma program P, whose Turing machine realization is Turing machine M, into a TLDelta program P', let $a_1,a_2,\ldots,a_n$ be a listing without repetition of the symbols in Γ. Let $p_1,p_2,\ldots,p_m$ be a listing of the states of M. Each symbol and each state may then be coded as strings of the symbols of Δ. This coding may then be used to represent id's of M. For example, if $(p_j,\alpha\triangleright\beta)$ is an id of M, then $\text{code}(p_j,\alpha\triangleright\beta) = \text{code}(\alpha)\text{code}(p_j)\text{code}(\beta)$. Subroutines may then be defined to simulate the execution of M. Define language A to be the set of coded strings such that M is in an accepting state. Define language B to be the set of coded strings which are halting id's of M. Define <initial> such that for all ξ, <initial>(ξ) = code(s,$\triangleright\xi$), where s is the start state of M. Define <next> such that for any id(p,$\alpha\triangleright\beta$) of M, <next>(code(p,$\alpha\triangleright\beta$)) = code(p',$\alpha'\triangleright\beta'$), provided that $(p,\alpha\triangleright\beta) \vdash_M (p',\alpha'\triangleright\beta')$. For any other string ξ, <next>(ξ) = ξ. For any string β in Δ^* and any state p, <decode>(code(p,$\triangleright\beta$)) = β. For any string ξ in Δ^*, which is not of the form code(p,$\triangleright\beta$), <decode>(ξ) = ξ. Because <initial>, <next>, and <decode> are all total functions, a TLDelta program may be constructed which is input/output equivalent to any given TLDelta/S program.

Translation from PSDelta to TLDelta/S

A PSDelta program consists of one or more TLDelta languge subprocedures, one or more TLDelta/S function subprocedures, and a PSDelta schema. PSDelta programs may have more than one string variable. In order to convert a PSDelta program into an input/output equivalent TLDelta/S program, several steps must be taken. First, the PSDelta program must be converted into a PSGamma program with only one string variable. A PSGamma program is syntactically equivalent to a PSDelta program; however, the alphabet is expanded. The alphabet Γ will have one more symbol than Δ. The additional symbol serves as a separator symbol. This will allow several strings from Δ^* to be coded as a single nonblank string in Γ^*. For example, α, β, and γ in Δ^* can be coded as $\#\alpha\#\beta\#\gamma\#$, where # is the separator symbol. Next the PSGamma program will be shown to be equivalent to a PSGamma program which has only one string and in which all Boolean expressions are atomic Boolean expressions. Boolean expressions of this type are essentially the same as those of TLDelta/S programs. Finally the PSGamma constructs, such as WHILE-DO, must be converted into TLGamma/S code by the use branching statement.

A new language subprocedure must be generated each time a different string number is used as the source of the original language procedure. A TLDelta language subprocedure in a PSDelta program expects to have only one

string on the input tape. Therefore additional code must be added to the program to provide for its correct operation in the simulated multiple string environment as follows:

1. At the beginning of the TLDelta code, a section of code must be inserted to place the tape head at the first character of the correct simulated string.
2. Because a language subprocedure does not expect to have any separators in a string, after each statement which shifts the tape, a piece of code must be inserted to check whether the current symbol is the separator symbol. If so, then the TLDelta code should encounter a blank. Therefore a blank must be inserted in this position and the string must be shifted one cell.
3. When the language subprocedure executes an accepting statement, the new code must move the <yes> or <no> from the beginning of the simulated string to the beginning of the entire string.
4. At the end a section of code which removes any extra blanks and repositions the tape head at the beginning of the string should be inserted.

The transformation of function subprocedures in a PSDelta program is similar to that of the language subprocedures. A new function subprocedure must be generated each time a different string number is used as the source of the original function procedure. The additional code necessary for the simulated multiple string environment is as follows:

1. At the beginning of the TLDelta/S function, a piece of code must be inserted to replace the simulated string used as the destination of the function computation with the contents of the simulated string used as the source of the function computation.
2. Next, code must be added which positions the tape head at the first character of the simulated string used as the destination of the function computation.
3. Because the TLDelta subprocedures of the TLDelta/S code do not expect to see any separators, after each statement of a TLDelta subprocedure which shifts the tape, a piece of code must be inserted to check whether the current symbol is the separator symbol. If so, then the TLDelta code should see a blank. Therefore a blank must be inserted in this position and the string must be shifted over one square.
4. Code should be inserted at the end to remove any extra blanks and to reposition the tape head at the beginning of the string.

Compound Boolean expressions of the PSDelta program must be changed into combinations of statements with atomic Boolean expressions. Boolean expressions of the form $STRING_n$ IN A are unchanged. A Boolean expression of the form (NOT $STRING_n$ in A) requires additional code which replaces <yes> returned from a language subprocedure to <no> and replaces <no> returned from a language subprocedure to <yes>. Boolean expressions of the form ($STRING_i$ IN A AND $STRING_j$ IN B) result in the generation of

multiple statements. A statement in the form shown in Figure 19 must be changed into code of the form shown in Figure 20. ANY_LANGUAGE and IDENTITY are two special subprocedures generated in the translation process. ANY_LANGUAGE is a TLDelta language subprocedure which accepts any string. IDENTITY is a TLDelta function subprocedure corresponding to the identity function which maps each string to itself. <label 2>, <label 3>, <label 4>, and <label 5> are new labels. <next statement label> is the label of the next TLDelta/S statement if it exists. Otherwise it is a new label which does not label any statement.

```
<label 1> : IF (STRING_i IN A AND STRING_j IN B) THEN
              <statements_1>
            ELSE
              <statements_2>
```

Figure 19 A PSDelta Statement With a Compound And Boolean Expression

```
<label 1> : IF STRING IN A_i THEN
              BEGIN
                STRING := IDENTITY(STRING);
                GOTO <label 2>
              END;
<label 3> : IF STRING IN ANY_LANGUAGE THEN
                BEGIN
                  STRING := IDENTITY(STRING);
                  GOTO <label 5>
                END;
<label 2> : IF STRING IN B_j THEN
              BEGIN
                <statements_1>;
              END;
<label 4> : IF STRING IN ANY_LANGUAGE THEN
              BEGIN
                STRING := IDENTITY(STRING);
                GOTO <next statement label>
              END;
<label 5> : IF STRING IN ANY_LANGUAGE THEN
              BEGIN
                <statements_2>;
              END;
```

Figure 20 TLDelta/S Code Equivalent to A PSDelta Statement With a Compound And Boolean Expression

Boolean expressions of the form ($STRING_i$ IN A OR $STRING_j$ IN B) result in the generation of multiple statements. A statement in the form shown in Figure 21 must be changed into code of the form shown in Figure 22. ANY_LANGUAGE and IDENTITY are two special subprocedures generated in the translation process. ANY_LANGUAGE is a TLDelta language subprocedure which accepts any string. IDENTITY is a TLDelta function subprocedure corresponding to the identity function which maps each string to itself. <label 2>, <label 3>, <label 4>, and <label 5> are new

labels. <next statement label> is the label of the next TLDelta/S statement if it exists; otherwise it is a new label which does not label any statement.

```
<label 1> : IF (STRING_i IN A OR STRING_j IN B) THEN
              <statements_1>
            ELSE
              <statements_2>
```

Figure 21 A PSDelta Statement With a Compound Or Boolean Expression

```
<label 1> : IF STRING IN A_i THEN
              BEGIN
                STRING := IDENTITY(STRING);
                GOTO <label 2>
              END;
<label 3> : IF STRING IN B_j THEN
                BEGIN
                  STRING := IDENTITY(STRING);
                  GOTO <label 2>
                END;
<label 4> : IF STRING IN ANY_LANGUAGE THEN
              BEGIN
                <statements_2>;
              END;
<label 5> : IF STRING IN ANY_LANGUAGE THEN
              BEGIN
                STRING := IDENTITY(STRING);
                GOTO <next statement label>
              END;
<label 2> : IF STRING IN ANY_LANGUAGE THEN
              BEGIN
                <statements_1>;
              END;
```

Figure 22 TLDelta/S Code Equivalent to A PSDelta Statement With a Compound Or Boolean Expression

In addition to the language subprocedures and function subprocedures, a TLGamma/S schema must also be obtained from the PSDelta program. A recursive algorithm to obtain a TLGamma/S schema T(P) from a given PSGamma program is shown in Appendix F. First the algorithm eleminates all of the BEGIN-END's, IF-THEN-ELSE's, and WHILE-DO's. Next the replaces function calls using specific numbered strings to calls of functions which operate on a simulated string. Assignment statements which concatenate strings together are changed into calls to functions which concatenate the correct simulated strings. The final TLGamma/S program is defined by the following schema:

```
BEGIN
  T(P);
  L : ACCEPT
END
```

where L is a label which does not occur in T(P).

Summary

The languages TLDelta, TLDelta/S and PSDelta may all be used to define Turing machines. A program P written in TLDelta may be directly translated into a Turing machine which is known as the Turing machine realization of P. A TLDelta/S program may be translated into an input/output equivalent TLGamma program whose alphabet is $\Delta \cup \{$<dirty blank>$\}$. By encoding the symbols of Γ and the states of a Turing machine realization M as strings of Δ, a TLDelta program may be obtained from any TLDelta/S program.

Finally a PSDelta program may be translated into an input/output equivalent TLGamma/S program whose alphabet is $\Delta \cup \{\#\}$ where '#' is a separator symbol not found in Δ.

CHAPTER V

IMPLEMENTATION

Introduction

There are several major goals of the implementation of the compilers for TLDelta, TLDelta/S, and PSDelta. First of all it is desired to produce fully operational compilers for each of the languages. Another goal is to produce a Turing machine simulator. Also it is desired that the overall method of implementation follow Savitch's [1982] original discussion of the languages as closely as possible.

The development environment for these compilers is the UNIX system. In particular two tools are used extensively: Lex and Yacc. Lex is used to generate the lexical analyzer for the compilers, and Yacc is used to generate the parsers for the compilers. The C code which Lex and Yacc produces is combined with other necessary functions and UNIX shell programs to form the actual compilers.

Three compilers actually make up the PSDelta compiler. The first compiler translates a PSDelta program into a TLDelta/S program. The next compiler translates a TLDelta/S program into a TLDelta program. The last of the three compilers translates a TLDelta program into a Turing machine representation in standard format and a C program which will

simulate the execution of the resulting Turing machine.

Lexical Analysis and Symbol Table Design

The task of recognizing keywords, installing identifiers into the symbol table, removing comments, and producing listing files is the responsibility of the lexical analyzer. The UNIX development tool Lex is used to produce the lexical analyzer. Because PSDelta programs and TLDelta/S programs both define subprocedures using TLDelta programs, the same lexical analyzer is used for all three languages. This removes the possibility of text from a subprocedure being interpreted differently by the three compilers. Lists of the keywords and symbols recognized by the lexical analyzer are given in Figure 2 and Figure 3.

The symbol table design for the compilers is implemented as a dynamically allocated singly linked list. The symbol table includes a pointer to the name of the identifier, a unique integer associated with each identifier, a flag used to determine whether or not the identifier labels any statement in the program, and a pointer to the next element of the list. Each time an identifier is recognized by the lexical analyzer, the symbol table is searched to see whether the identifier already has been installed. If the identifier already has been installed, a pointer to its entry in the symbol table is returned. Otherwise, the new identifier is installed into

the symbol table and a pointer to its entry is returned. Although the symbol table is maintained as a linear structure, access time other than the original searching of the list is not unreasonable because pointers to the specific entries are used whenever possible. This approach allows direct access to the elements of the symbol table.

Parsing, Intermediate Representation, and Code Generation

The parsers for the three compilers are generated using the UNIX development tool Yacc. Although TLDelta and TLDelta/S are subsets of PSDelta, it is necessary to have a different parser for each of the three languages. The '-d' option of Yacc is used to produce an external header file which contains the definitions of the token values produced by Yacc. This allows the separate modules of the compiler to be compiled separately. Although sometimes there exist obvious optimizations of the code, they are not made in order to remain as close to the original language descriptions as possible.

TLDelta Intermediate Representation and Simulator Generation

The intermediate code which is used for a usual TLDelta statement is a shorthand representation of the same information provided by a TLDelta statement. Each intermediate code instruction contains a pointer to the

symbol table entry for the label of the statement, the symbol to be compared with the current symbol being scanned, the character to be written to the tape, the direction which the tape is to be shifted, and a pointer to the symbol table entry for the next statement . If the statement is an accepting statement, the next state is null. This information is stored in a singly linked list. Because in TLDelta/S and PSDelta multiple TLDelta programs may exist, there is a structure containing certain information about the TLDelta program associated with each TLDelta program. The information contained in this structure includes a pointer to a list of all the labels of the program (states of the machine), a pointer to a list of all the characters of the program, a pointer to the intermediate representation of the transitions of the program, and a pointer to the list of all accepting states of the program.

The Turing machine description is produced directly from the intermedite representation by the method described in Chapter 4. The Turing machine simulator generated by this compiler is table driven. All characters in the input alphabet are placed into a lookup table and given a numeric value based upon their ordering. Three static two-dimensional arrays govern the moves of the machine from one id to the next. The arrays are indexed by the id number of the state and the numeric value assigned to the current character. These three arrays contain the next state, the character to be written to the tape, and the direction to

shift tape. If a transition from a given state is undefined, then the array contains a negative one in the corresponding entry. If a state is a final state, then the array contains a zero in the corresponding entries.

The simulator normally receives its input from the standard input device and writes any output to the standard output device. By using the UNIX operating system, input and output can be redirected to come from and go to several different places. The simulator is capable of single-stepping through its execution. When the single-step option is active, the user is prompted for an input file and may control the execution in various ways. All error messages are written to the standard error file. Therefore any error messages generated will always appear on the terminal regardless of where the output is directed.

TLDelta/S Intermediate Representation

The intermediate code chosen to represent a usual TLDelta/S statement contains essentially the same information as a usual TLDelta/S statement. Each intermediate code instruction contains a pointer to the symbol table entry for the label of the statement, a pointer to the TLDelta code for the language subprocedure, a pointer to the TLDelta code for the function subprocedure, and the label of the next statement. If the statement is an accepting statement, all entries except the label are NULL. This information is stored in a singly linked list. Because

in PSDelta multiple TLDelta/S programs may exist, there is a structure associated with each TLDelta/S program which contains the name of the program and a pointer to the head of the intermediate representation.

The generation of the TLDelta code is exactly as described in Chapter 4. The labels of statements from subprocedures consist of the label of the TLDelta/S statement from which the subprocedure is called concatenated with the label of the statement from the subprocedure. For this reason, statement labels should never be combinations of other statement labels.

PSDelta Intermediate Representation and Code Generation

The intermediate code which is used to represent PSDelta statements is in the form of quadruples. Each intermediate code instruction contains a unique integer label, an opcode, and up to three operands. The opcode is one of the integer representations of IF, FUNCTION, ASSIGNMENT, or GOTO. If the opcode is IF, then the first operand is a pointer to the TLDelta code for a language subprocedure to be executed, the second operand is the number of the string variable to be checked for membership, and the third operand is the id of the statement to be executed if the string variable is an element of the language. If the opcode is FUNCTION, then the first operand is a pointer to the TLDelta/S code for the function to be

computed, the second operand is the number of the string to serve as the source to the function, and the third operand is the number of the string to serve as the destination of the computation. If the opcode is ASSIGNMENT, then the first operand is the number of the left string to be concatenated, the second operand is the number of the right string to be concatenated, and the third operand is the number of the string into which the result will be placed. If the operand is GOTO, then the third operand is the id of the statement to be executed next.

Generation of the resulting TLDelta/S program is by the method described in Chapter 4. Code generation proceeds by first outputing the code for the generated language subprocedure ANY_LANGUAGE. Next all necessary language subprocedures are generated. The code for the generated function IDENTITY is produced next, followed by all necessary function subprocedures and concatenation subprocedures.

Certain routines appear often in the TLDelta/S code produced from a PSDelta program. They include routines which shift the tape over one cell, routines which remove the blanks from one simulated string, and routines which copy one simulated string to another simulated string. Each of these routines is usually quite large but not complicated. The shift routine is the basis for the other two types of routines. In order to shift the tape one cell in either direction, the symbol which is currently being

scanned must be retained in some manner so that it may be written in the next cell. The only way which a Turing machine may retain this information is by the state of the machine. Therefore, the number of states for one of these routines is often quite large, although much of the code is almost identical.

In order to remove the blanks from a given simulated string, the tape head begins at one end of the string and works its way toward the other end. If a blank is encountered the tape is shifted toward the blank in order to eliminate it. If a separator is encountered then all blanks have been removed. In order to copy one string to another, a blank is inserted into the source string to serve as a marker symbol. Then for each character in the source string, the blank is shifted over one character, that character is retained in the state of the machine, the tape head is moved to the destination string where the symbol is written, and the tape head returns to the source where the process is repeated.

Each of these routines is always the same except for the input alphabet of the program and the labels of the statements. Therefore the compiler contains functions which generate these routines for a given input alphabet. The labels of the statements are combinations of the name of the routine used, the numeric value of the character retained, and the label of the originating statement.

Summary

The implementations of these compilers attains three major goals: to produce fully operational compilers for each of the languages, to produce a Turing machine simulator, and to follow Savitch's [1982] original discussion of the languages as closely as possible. The intermediate representations of TLDelta and TLDelta/S closely resemble the information given in their respective languages. The intermediate code for PSDelta is quadruples. Certain routines are produced several times by the PSDelta compiler which vary only in the labels of the statements and possible the input alphabet. Although there exist areas where the code could be optimized, the code is left intact in order to remain as close to the original definitions as possible.

CHAPTER VI

SUMMARY, CONCLUSIONS AND FUTURE WORK

In his text Abstract Machines and Grammars, Savitch [1982] presents a high-level language PSDelta and shows that it can be translated into a Turing machine which performs the same tasks. Compilers have been implemented for this language and for the two languages, TLDelta and TLDelta/S upon which this languge is based. The family of compilers produce a functional Turing machine description in standard notation and a functional simulator of the Turing machine. These compilers are intended for use in teaching automata theory. Therefore the translation methods follow those presented by Savitch as closely as possible without optimization.

> Keep in mind that we are developing a theory about what things can and cannot be done by programs. To do this, it is helpful to know that every PSDelta program can be converted to a TLDelta/S program. However, TLDelta and TLDelta/S are just aids to developing this theory. They are not languages used by any real computers. So we will never implement our PSDelta compiler in the 'real world.' Therefore, there is no need for the algorithms we present to be efficient. Our goal will be to make them correct, easy to prove correct, and easy to understand. Efficiency is not important to our purpose here [Savitch, 1982].

Suggested work in this area includes the implementation of the subprocedures as closed subroutines instead of macro

expansions (open subroutines). This would utilize tape storage better rather than producing an enormous number of identical statements. Additions to the compilers could be made to allow users to associate sections of their own C code with TLDelta, TLDelta/S, and PSDelta statements. These sections of code would be performed when the corresponding statement was executed. This type of enhancement should greatly resemble the actions of Lex and Yacc. Other control structures such as REPEAT-UNTIL, and indexed loops could be added. These would not actually increase the power of the languages, but would strengthen the similarities between PSDelta and other high-level languages such as Pascal. A preprocessor could be developed for the three languages. This would allow file inclusion, global macro substitution, and add the capability of using programs such as Lex routines for the input. This would enable the input to be more legible to the user. Likewise a routine could be used to convert an output from a string of symbols to a more human-readable form.

A SELECTED BIBLIOGRAPHY

Aho, A. V., and J. D. Ullman. The Theory of Parsing, Translation and Compiling, Vol. I: Parsing. Englewood Cliffs, N.J.: Prentice Hall, 1972.

Aho, A. V., Ravi Sethi and J. D. Ullman. Compilers, Principles, Techniques, and Tools. Reading, Massachusetts: Addison-Wesley Publishing Company, 1985.

Cutland, Nigel (ed.). Computability: An Introduction to Recursive Function Theory. New York: Cambridge University Press, 1980.

Davis, Martin. Computability. New York: Courant Institute of Mathematical Sciences, 1974.

Hopcroft, John E. and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, anad Computation. Reading, Massachusetts: Addison-Wesley Publishing Company, 1979.

Hopcroft, John E. "Turing Machines." The Scientific American, Vol. 250 (May 1984), pp. 86-98.

Johnson, S. C. "Yacc - Yet Another Compiler-Compiler." Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).

Johnson, S. C. and M. E. Lesk. "Language Development Tools." The Bell System Technical Journal, Vol. 57, No. 6 (July-August 1978), pp. 2155-2175.

Jones, Neil D. Computability Theory, An Introduction. New York: Academic Press, 1973.

Lesk, M. E. "Lex -- A Lexical Analyzer Generator." Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975).

Savitch, Walter J. Abstract Machines and Grammars. Boston: Little, Brown and Company, 1982.

Savitch, Walter J. Personal Letter. University of California, San Diego. La Jolla, CA, 1987. June 1, 1987.

APPENDIX A

A CONTEXT-FREE GRAMMAR FOR TLDELTA

APPENDIX A

A CONTEXT-FREE GRAMMAR FOR TLDELTA

```
tld_program : BEGIN tld_stmt_seq END

tld_stmt_seq : tld_stmt
             | tld_stmt_seq ; tld_stmt

tld_stmt : tld_usual_stmt
         | accepting_stmt

tld_usual_stmt : stmt_label IF tld_boolean THEN
                   BEGIN
                     tld_assignment_stmt
                     tld_pointer_move
                     tld_goto_stmt
                   END

tld_assignment_stmt : SCAN := SYMBOL ;

tld_goto_stmt : GOTO label

accepting_stmt : stmt_label ACCEPT

stmt_label : label ':'

label : ID

tld_boolean_symbol : tld_symbol
                   | ANY

tld_assignment_symbol : tld_symbol
                      | CURRENT

tld_symbol : SYMBOL
           | YES
           | NO
           | DIRTY_BLANK
           | SEPARATOR

tld_pointer_move : POINTER LEFTARROW
                 | POINTER RIGHTARROW
                 | POINTER DOWNARROW
```

APPENDIX B

A CONTEXT-FREE GRAMMAR FOR TLDELTA/S

APPENDIX B

A CONTEXT-FREE GRAMMAR FOR TLDELTA/S

```
tlds_program : tlds_grammar_seq
               tlds_function_seq
               tlds_schema

tlds_grammar_seq : tlds_grammar
                 | tlds_grammar_seq tlds_grammar

tlds_grammar : G(tlds_language) = tld_program

tlds_function_seq : tlds_function
                  | tlds_function_seq tlds_function

tlds_function : F(tlds_function_name) = tld_program

tlds_schema : BEGIN tlds_stmt_seq END

tlds_stmt_seq : tlds_stmt
              | tlds_stmt_seq ; tlds_stmt

tlds_stmt : tlds_usual_stmt
          | tlds_accepting_stmt

tlds_usual_stmt : stmt_label IF tlds_boolean THEN
                    BEGIN
                      tlds_assignment_stmt
                      tlds_goto_stmt
                    END

tlds_accepting_stmt : stmt_label ACCEPT

tlds_boolean : STRING IN tlds_language

tlds_assignment_stmt : STRING := tlds_function_name(STRING);

tlds_goto_stmt : GOTO label

tlds_language : ID

tlds_function_name : ID

tld_program : BEGIN tld_stmt_seq END
```

```
tld_stmt_seq : tld_stmt
             | tld_stmt_seq ; tld_stmt

tld_stmt : tld_usual_stmt
         | accepting_stmt

tld_usual_stmt : stmt_label IF tld_boolean THEN
                   BEGIN
                     tld_assignment_stmt
                     tld_pointer_move
                     tld_goto_stmt
                   END

tld_assignment_stmt : SCAN := SYMBOL ;

tld_goto_stmt : GOTO label

accepting_stmt : stmt_label ACCEPT

stmt_label : label :

label : ID

tld_boolean_symbol : tld_symbol
                   | ANY

tld_assignment_symbol : tld_symbol
                      | CURRENT

tld_symbol : SYMBOL
           | YES
           | NO
           | DIRTY_BLANK
           | SEPARATOR

tld_pointer_move : POINTER LEFTARROW
                 | POINTER RIGHTARROW
                 | POINTER DOWNARROW
```

APPENDIX C

A CONTEXT-FREE GRAMMAR FOR PSDELTA

APPENDIX C

A CONTEXT-FREE GRAMMAR FOR PSDELTA

```
psd_program : psd_grammar_seq
              psd_function_seq
              psd_schema

psd_grammar_seq : psd_grammar
                | psd_grammar_seq psd_grammar

psd_grammar : G(ID) = tld_program

psd_function_seq : psd_function
                 | psd_function_seq psd_function

psd_function : F(ID) = tlds_program

psd_schema : psd_stmt

psd_stmt : psd_if_stmt
         | psd_begin_block
         | psd_while_stmt
         | psd_assg_stmt

psd_if_stmt : IF psd_boolean THEN
                psd_stmt
              ELSE
                psd_stmt

psd_begin_block : BEGIN
                    psd_stmt_list
                  END

psd_while_stmt : WHILE psd_boolean DO
                     psd_stmt

psd_assg_stmt : VARIABLE := psd_function_name(VARIABLE)
              | VARIABLE := VARIABLE VARIABLE

psd_stmt_list : psd_stmt
              | psd_stmt_list psd_stmt
```

```
psd_boolean : (psd_boolean AND psd_boolean)
            | (psd_boolean OR psd_boolean)
            | (NOT psd_boolean)
            | VARIABLE IN psd_language

psd_language : ID

tlds_program : tlds_grammar_seq
               tlds_function_seq
               tlds_schema

tlds_grammar_seq : tlds_grammar
                 | tlds_grammar_seq tlds_grammar

tlds_grammar : G(tlds_language) = tld_program

tlds_function_seq : tlds_function
                  | tlds_function_seq tlds_function

tlds_function : F(tlds_function_name) = tld_program

tlds_schema : BEGIN tlds_stmt_seq END

tlds_stmt_seq : tlds_stmt
              | tlds_stmt_seq ; tlds_stmt

tlds_stmt : tlds_usual_stmt
          | tlds_accepting_stmt

tlds_usual_stmt : stmt_label IF tlds_boolean THEN
                    BEGIN
                      tlds_assignment_stmt
                      tlds_goto_stmt
                    END

tlds_accepting_stmt : stmt_label ACCEPT

tlds_boolean : STRING IN tlds_language

tlds_assignment_stmt : STRING := tlds_function_name(STRING);

tlds_goto_stmt : GOTO label

tlds_language : ID

tlds_function_name : ID

tld_program : BEGIN tld_stmt_seq END
```

```
tld_stmt_seq : tld_stmt
             | tld_stmt_seq ; tld_stmt

tld_stmt : tld_usual_stmt
         | accepting_stmt

tld_usual_stmt : stmt_label IF tld_boolean THEN
                   BEGIN
                     tld_assignment_stmt
                     tld_pointer_move
                     tld_goto_stmt
                   END

tld_assignment_stmt : SCAN := SYMBOL ;

tld_goto_stmt : GOTO label

accepting_stmt : stmt_label ACCEPT

stmt_label : label :

label : ID

tld_boolean_symbol : tld_symbol
                   | ANY

tld_assignment_symbol : tld_symbol
                      | CURRENT

tld_symbol : SYMBOL
           | YES
           | NO
           | DIRTY_BLANK
           | SEPARATOR

tld_pointer_move : POINTER LEFTARROW
                 | POINTER RIGHTARROW
                 | POINTER DOWNARROW
```

APPENDIX D

A CONTEXT-FREE GRAMMAR FOR h_A PROGRAM

APPENDIX D

A CONTEXT-FREE GRAMMAR FOR h_A PROGRAM

<program> : BEGIN <A code> ; EXITA':' ACCEPT END

<A code> : <M code> <no rewind> ; <yes rewind>

<M code> : <p_0 code> <p_1 code> ... <p_m code>

For all p_i which are not accepting states:
<p_i code : p_i':' <end no?>;<p_i,a_0>;<p_i,a_0>;...;<p_i,a_n>;

For all p_i which are accepting states:
<p_i code> : p_i':' <end yes?>;<p_i,a_0>;<p_i,a_0>;...;<p_i,a_n>;

For all states p_i and all symbols a_j:

```
<p_i,a_j> : IF SCAN = a_j THEN
              BEGIN
                SCAN := a_j;
                POINTER:;
                GOTO q_ij
              END
```

```
<end yes?> : IF SCAN = BLANK THEN
               BEGIN
                 SCAN := BLANK;
                 POINTER↓;
                 GOTO INA
               END
```

<end no?> is just like <end yes?> but with INA replaced by OUTA. Notice that INA will label the start of <yes rewind> anad OUTA will label the start of <no rewind>.

```
<yes rewind> : INA':' <rewind 2>;
               IF SCAN = BLANK THEN
                 BEGIN
                   SCAN := <yes>;
                   POINTER↓;
                   GOTO EXITA
                 END
```

```
<no rewind> : OUTA':' <rewind 1>;
                  IF SCAN = BLANK THEN
                    BEGIN
                      SCAN := <no>;
                      POINTER↓;
                      GOTO EXITA
                    END

For i = 1,2
   <rewind i> : POINTER<-;
                LOOP_i':' IF (NOT SCAN = BLANK) THEN
                            BEGIN
                              POINTER<-;
                              GOTO LOOP_i
                            END;
```

Note: p_0, p_1, ..., p_m, EXITA, INA, OUTA, LOOP1, and LOOP2 must be $m+5$ distinct labels, but this is easy to ensure.

APPENDIX E

ALGORITHM TO OBTAIN <NEW P CODE>

APPENDIX E

ALGORITHM TO OBTAIN <NEW P CODE>

1. Replace each occurrence of
   SCAN := BLANK by SCAN := <dirty blank>

2. Replace each statement of the form

```
<label> : IF SCAN = BLANK THEN
            BEGIN
              .
              .
              .
            END
```

   by

```
<label> : IF (SCAN = BLANK OR SCAN = <dirty blank>) THEN
            BEGIN
              .
              .
              .
            END
```

3. Replace each statement of the form

```
<label> : IF SCAN = <ANY> THEN
            BEGIN
              SCAN := <CURRENT>
              .
              .
            END
```

   by

```
<label> : IF SCAN = BLANK THEN
            BEGIN
              SCAN := <dirty blank>
              .
              .
            END
<labela>: IF SCAN = <ANY> THEN
            BEGIN
              SCAN := <CURRENT>
              .
              .
            END
```

4. Replace each occurrence of ACCEPT by GOTO <formcheck>

APPENDIX F

RECURSIVE ALGORITHM FOR T(P)

APPENDIX F

RECURSIVE ALGORITHM FOR T(P)

T(P) is a TLGamma/S schema obtained from a given PSGamma program. This code is used when transforming a PSDelta program to a TLDelta/S program. This algorithm takes as input a PSGamma program with a single string variable and only atomic Boolean expressions.

1. If P = BEGIN s1;s2;...;sn END then T(P) is
   T(s1);T(s2);...;T(sn).

2. If P = IF boolean THEN s1 ELSE s2 then T(P) is

```
   IF boolean THEN GOTO L1;
   T(s2);
   GOTO L2
L1: <nothing 1>;
   T(s1);
L2: <nothing 2>;
```

where <nothing 1> and <nothing 2> are any TLGamma/S statements that have no effect on the program. For example, each might be a GOTO to the next statement.

3. If P = WHILE boolean DO s, then T(P) is

```
L1: IF boolean THEN GOTO L2;
   GOTO L3;
L2: <nothing 1>;
   T(s);
   GOTO L1;
L3: <nothing 2>
```

where L1, L2, and L3 are new labels and both <nothing 1> and <nothing 2> are TLGamma/S statements that have no effect on the program.

4. If P is $STRING_i$:= f($STRING_j$) then T(P) is
   STRING := f_i_j(STRING)

   where f_i_j is a new function which first copies $STRING_j$ to $STRING_i$ and then computes f using $STRING_i$.

5. If P is $STRING_i$:= $STRING_j$ $STRING_k$ then T(P) is
   STRING := g_j_k_i(STRING)

   where f_j_k_i is a new function which concatenates $STRING_j$ with $STRING_k$ and places the result in $STRING_i$

END OF ALGORITHM

APPENDIX G

A SAMPLE PSDELTA PROGRAM

APPENDIX G

A SAMPLE PSDELTA PROGRAM

The following program is a PSDelta program which will read a string of zeros and ones from the standard input. The program will then replace each occurrence of a zero by a one. The result is written to the standard output.

```
G(ONES) =
BEGIN
ONE:  IF SCAN = 1 THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO ONE
  END;
ONE_BLANK: IF SCAN = BLANK THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER <-;
    GOTO YES_ONES
  END;
ONE_ANY: IF SCAN =<ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER<-;
    GOTO NO_ONES
  END;
YES_ONES: IF SCAN = BLANK THEN
  BEGIN
    SCAN := <YES>;
    POINTER|;
    GOTO ONE_ACCEPT
  END;
YES_ONES_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER<-;
    GOTO YES_ONES
  END;
```

```
REWIND: IF SCAN = 1 THEN
  BEGIN
    SCAN := 1;
    POINTER<-;
    GOTO REWIND
  END;
L3: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO FINISH
  END;
FINISH: ACCEPT
END

BEGIN
TLDS_START: IF STRING IN ANY_LANGUAGE THEN
  BEGIN
    STRING := PROC2(STRING);
    GOTO TLDS_ACCEPT
  END;
TLDS_ACCEPT: ACCEPT
END

BEGIN
  STRING1 := PROC1(STRING1)
END
```

APPENDIX H

A SAMPLE TLDELTA/S PROGRAM SEGMENT

APPENDIX H

A SAMPLE TLDELTA/S PROGRAM SEGMENT

The following program segment is an excerpt of the TLDelta/S code produced by compiling the PSDelta program in Appendix G. Due to the extreme length of the program, only a portion of the function PROC2 is shown. The original PSDelta program contains 112 lines. The resulting TLDelta/S program contains 2980 lines. <SHIFT TAPE> is substituted for the actual block of code which inserts a blank symbol and shifts the tape one cell. <REMOVE BLANKS> is substituted for the actual block of code which removes any embedded blank symbols.

```
F(PROC2_1) =
BEGIN
PROC2_1_1_SEPARATOR: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO PROC2_1_2_SEPARATOR
  END;
PROC2_1_1_NOT_SEPARATOR: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO PROC2_1_1_SEPARATOR
  END;
PROC2_1_2_SEPARATOR: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO PROC2_1_GT_ST_48
  END;
```

```
PROC2_1_2_NOT_SEPARATOR: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L0_1
  END;
PROC2_1_GT_ST_48:
          .
          .
          .
    <SHIFT TAPE>
          .
          .
          .
L0_1: IF SCAN = 0 THEN
  BEGIN
    SCAN := 1;
    POINTER->;
    GOTO L0_1_CHECK_END
  END
L0_1_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L1_1
  END;
L0_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L0_ST_48
  END;
L0_1_NOT_END: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L0_1
  END;
GOTO L0_ST_48:
          .
          .
          .
    <SHIFT TAPE>
          .
          .
          .
L1_1: IF SCAN = 1 THEN
  BEGIN
    SCAN := 1;
    POINTER->;
    GOTO L1_1_CHECK_END
  END;
```

```
L1_1_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L2_1
  END;
L1_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L1_1_ST_48
  END;
L1_1_NOT_END: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L0_1
  END;
L1_1_ST_48:
        .
        .
        .
   <SHIFT TAPE>
        .
        .
        .
L2_1: IF SCAN = BLANK THEN
  BEGIN
    SCAN := BLANK;
    POINTER<-;
    GOTO L2_1_CHECK_END
  END;
L2_1_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO REWIND_1
  END;
L2_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L2_1_ST_48
  END;
L2_1_NOT_END: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO REWIND_1
  END;
```

```
L2_1_ST_48:
          .
          .
          .
    <SHIFT TAPE>
          .
          .
          .
REWIND_1: IF SCAN = 1 THEN
  BEGIN
    SCAN := 1;
    POINTER<-;
    GOTO REWIND_1_CHECK_END
  END;
REWIND_1_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L3_1
  END;
REWIND_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO REWIND_1_ST_48
  END;
REWIND_1_NOT_END: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO REWIND_1
  END;
REWIND_1_ST_48:
              .
              .
              .
        <SHIFT TAPE>
              .
              .
              .
L3_1: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO L3_1_CHECK_END
  END;
L3_1_ANY: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO FINISH_1
  END;
```

```
L3_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO L3_1_ST_48
  END;
L3_1_NOT_END: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO FINISH_1
  END;
L3_1_ST_48:
        .
        .
        .
   <SHIFT TAPE>
        .
        .
        .
FINISH_1: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO PROC2_1_REMOVE_BLANKS
  END;
PROC2_1_REMOVE_BLANKS:
                    .
                    .
                    .
              <REMOVE BLANKS>
                    .
                    .
                    .
PROC2_1_ACCEPT:  ACCEPT
END
```

APPENDIX I

A SAMPLE TLDELTA PROGRAM SEGMENT

APPENDIX I

A SAMPLE TLDELTA PROGRAM SEGMENT

The following program segment is an excerpt of the TLDelta code produced by compiling the TLDelta/S program in Appendix H. Due to the extreme length of the program, only a portion of the function PROC2 is shown. The original TLDelta/S program contains 2980 lines. The resulting TLDelta program contains 4251 lines. <SHIFT TAPE> is substituted for the actual block of code which inserts a blank symbol and shifts the tape one cell.

```
          .
          .
          .
YY_PSD_0_TLDS_START_F_LABEL: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_PROC2_1_1_SEPARATOR
  END;
YY_PSD_0_TLDS_START_PROC2_1_1_SEPARATOR: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO YY_PSD_0_TLDS_START_PROC2_1_2_SEPARATOR
  END;
YY_PSD_0_TLDS_START_PROC2_1_1_NOT_SEPARATOR:
  IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER->;
    GOTO YY_PSD_0_TLDS_START_PROC2_1_1_SEPARATOR
  END;
```

```
YY_PSD_0_TLDS_START_PROC2_1_1_NOT_SEPARATOR_DB:
  IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER->;
    GOTO YY_PSD_0_TLDS_START_PROC2_1_1_SEPARATOR
  END;
YY_PSD_0_TLDS_START_PROC2_1_2_SEPARATOR: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_PROC2_1_GT_ST_48
  END;
YY_PSD_0_TLDS_START_PROC2_1_2_NOT_SEPARATOR:
  IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_PROC2_1_2_NOT_SEPARATOR_DB:
  IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_PROC2_1_GT_ST_48:
          .
          .
          .
    <SHIFT TAPE>
          .
          .
          .
YY_PSD_0_TLDS_START_L0_1: IF SCAN = 0 THEN
  BEGIN
    SCAN := 1;
    POINTER->;
    GOTO YY_PSD_0_TLDS_START_L0_1_CHECK_END
  END;
YY_PSD_0_TLDS_START_L0_1_ANY: IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L1_1
  END;
YY_PSD_0_TLDS_START_L0_1_ANY_DB: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L1_1
  END;
```

```
YY_PSD_0_TLDS_START_L0_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1_ST_48
  END;
YY_PSD_0_TLDS_START_L0_1_NOT_END: IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_L0_1_NOT_END_DB: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_L0_1_ST_48:
        .
        .
        .
    <SHIFT TAPE>
        .
        .
        .
YY_PSD_0_TLDS_START_L1_1: IF SCAN = 1 THEN
  BEGIN
    SCAN := 1;
    POINTER->;
    GOTO YY_PSD_0_TLDS_START_L1_1_CHECK_END
  END;
YY_PSD_0_TLDS_START_L1_1_ANY: IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L2_1
  END;
YY_PSD_0_TLDS_START_L1_1_ANY_DB: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L2_1
  END;
YY_PSD_0_TLDS_START_L1_1_CHECK_END: IF SCAN = <#> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L1_1_ST_48
  END;
```

```
YY_PSD_0_TLDS_START_L1_1_NOT_END: IF SCAN = BLANK THEN
  BEGIN
    SCAN := DIRTY_BLANK;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_L1_NOT_END_DB: IF SCAN = <ANY> THEN
  BEGIN
    SCAN := <CURRENT>;
    POINTER|;
    GOTO YY_PSD_0_TLDS_START_L0_1
  END;
YY_PSD_0_TLDS_START_L1_1_ST_48:
          .
          .
          .
```

VITA

Charles Bradley Slaten

Candidate for the Degree of

Master of Science

Thesis: AN IMPLEMENTATION OF A PROCEDURAL LANGUAGE FOR REPRESENTING TURING MACHINES

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in North Little Rock, Arkansas, February 23, 1963, the son of Mr. and Mrs. Doyle Slaten.

Education: Graduated from Searcy High School, Searcy, Arkansas in May 1981; received Bachelor of Science Degree in Computer Science from Arkansas Tech University, 1985; completed requirements for the Master of Science degree at Oklahoma State University in December, 1987.

Professional Experience: Teaching Assistant, Department of Computing and Information Sciences, Oklahoma State University, 1985-1987.

Professional Organizations: Oklahoma State University Student Chapter of the Association for Computing Machinery, Blue Key National Honor Fraternity.