SPRINT a direct approach to list processing languages

by CHARLES A. KAPPS
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

Most current list processing languages such as LISP and IPL-V operate in an indirect manner, i.e., during execution of a program written in these languages, the basic operations do not deal directly with data but rather with addresses which point to the data, making it awkward to perform operations at the input syntax level.

Other languages, such as L6, give the programmer a closer relationship with data on which his program operates, but are highly machine oriented languages, and therefore are really effective on only a small number of machines.

The aims of SPRINT are to give the programmer direct access to both data and program in a way that is as nearly machine independent as is possible. In particular, SPRINT is completely unstratified. It has divorced itself from any machine oriented addressing schemes, number schemes, and word formats. It allows the programmer direct access to a data scheme which is as general as possible from the human point of view, while keeping in mind some degree of practicality.

In order to achieve these goals some sacrifice had to be made with respect to machine efficiency. SPRINT is intended to reflect the thought processes of the programmer, not the logic of some machine currently in existence which will be obsolete in a few years. In a way SPRINT was modeled as the machine language for a futuristic machine, one where the hardware was designed to employ a more humanly, logical structure than those of today. The result is that SPRINT would operate much more efficiently if mechanized on machines which have more advanced organizations than those of today.

Basic properties of SPRINT

The "word," being the basic unit of information, consists of a variable length string of alphanumeric characters. (In order that some semblance of efficiency be maintained, there is an upper bound on the length of a word. In the case of present mechanization, this bound is 179 characters, which seems adequately long for most purposes.) SPRINT words have no meaning in themselves, and take on meaning only when interpreted by the programs which use them. This is an important feature of SPRINT, since it leads to non-stratification of the language.

Arithmetic information in SPRINT is denoted by any word composed solely of numeric characters, and represents the corresponding decimal integer.

For purposes of operating the execution cycle, the programmer must mark each SPRINT word with a class mark indicating whether it is to be treated as an instruction or as data when it is encountered by the instruction decoder. This is analogous to the use of the operator "QUOTE" in LISP.

Several SPRINT words (about 35) have been set aside to name "basic instructions." Basic instructions are operations which have been programmed in machine language, or which would be in the hardware of an actual machine. Figure 1 gives a list of some of the more important basic instructions of SPRINT.

Linearly ordered sets of SPRINT words form "lists." Lists may contain either programs or data, but since SPRINT is unstratified, program lists and data lists are indistinguishable except in context. Lists have names, which consist of a single SPRINT word, and SPRINT programs refer to the lists by means of their alphabetic names. Lists are stored in an associative type of memory area, and are located when sought, by means of their names. It is important that these names are preserved, since this allows SPRINT programs to call for lists which are named
<table>
<thead>
<tr>
<th>MNEMONIC MEANING</th>
<th>ARGUMENTS REMOVED FROM THE OPERAND STACK</th>
<th>RESULTS RETURNED TO THE OPERAND STACK</th>
<th>OTHER ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDT</td>
<td>W</td>
<td>---------------------------------------</td>
<td>An input card is read and stored as a list named &quot;W&quot;.</td>
</tr>
<tr>
<td>WDT</td>
<td>W</td>
<td>---------------------------------------</td>
<td>The list named &quot;W&quot; is printed out.</td>
</tr>
<tr>
<td>ADD</td>
<td>A, B</td>
<td>A + B</td>
<td>------------------</td>
</tr>
<tr>
<td>SUB</td>
<td>A, B</td>
<td>B - A</td>
<td>------------------</td>
</tr>
<tr>
<td>MPY</td>
<td>A, B</td>
<td>A × B</td>
<td>------------------</td>
</tr>
<tr>
<td>DIV</td>
<td>A, B</td>
<td>B mod A and B/A</td>
<td>------------------</td>
</tr>
<tr>
<td>REV</td>
<td>W, V</td>
<td>V and W</td>
<td>------------------</td>
</tr>
<tr>
<td>REP</td>
<td>W</td>
<td>W and W</td>
<td>------------------</td>
</tr>
<tr>
<td>DEL</td>
<td>W</td>
<td>---------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>CONCAT</td>
<td>W, V</td>
<td>Concatenation of W and V</td>
<td>------------------</td>
</tr>
<tr>
<td>DCN</td>
<td>A, W</td>
<td>First A characters of W and the rest of W.</td>
<td>------------------</td>
</tr>
<tr>
<td>ENG</td>
<td>W</td>
<td>The entire list named &quot;W&quot;</td>
<td>The entire list named by &quot;W&quot; is pushed into the instruction stack.</td>
</tr>
<tr>
<td>CLL</td>
<td>W</td>
<td>---------------------------------------</td>
<td>&quot;V&quot; is stored as a one-word list named &quot;W&quot;.</td>
</tr>
<tr>
<td>TRA</td>
<td>W, V</td>
<td>---------------------------------------</td>
<td>&quot;V&quot; through &quot;V_A&quot; are stored as an A word list named W.</td>
</tr>
<tr>
<td>STO</td>
<td>A, W, V_1...V_A</td>
<td>---------------------------------------</td>
<td>If W names a list in memory the next instruction is skipped, otherwise the next instruction is executed and the one following is skipped.</td>
</tr>
<tr>
<td>FND</td>
<td>W</td>
<td>W</td>
<td>If W belongs to O, the next instruction is skipped, otherwise the next instruction is executed and the one following is skipped.</td>
</tr>
<tr>
<td>TZE</td>
<td>W</td>
<td>W</td>
<td>A and B represent arbitrary numeric words; W and V represent arbitrary words.</td>
</tr>
</tbody>
</table>

Figure 1 – A partial enumeration of the basic instruction of SPRINT
by SPRINT words which are "computed" during execution. Basic instructions are available in SPRINT which allow any word of any list or any character of any word to be modified in any Turing computable fashion.

The execution of SPRINT programs takes place in an area consisting of two push down stacks, designated as the instruction stack and operand stack. Basic instructions in SPRINT do not have "addresses," but refer to the top several levels of the operand stack for their arguments and results. In addition, some instructions affect the instruction stack and the associative memory area. Figure 2 shows the basic instruction cycle of SPRINT.

It can be seen that Figure 2 is actually a flow chart for a suffix language processor; consequently, simple SPRINT programs will appear in suffix form.

For example, to evaluate the algebraic expression:

$$(((A + B) C + D) (E - 10))$$

one could execute the following program list:

A B ADD C MPY D ADD E ⊕10 SUB MPY

Here "⊕" indicates a data class mark, and instruction class marks are assumed elsewhere. "ADD," "SUB," and "MPY" are the SPRINT mnemonics for add, subtract, and multiply respectively, and "A," "B," "C," "D," and "E" are assumed to be either the names of data lists which contain a single numeric word in the data class, or SPRINT programs for computing a single numeric word.

Complex data structures within SPRINT

One of the main advantages of having an associative memory is that names can be created by com-

---

**Figure 2 - Basic instruction cycle**

From the collection of the Computer History Museum (www.computerhistory.org)
putation during the execution program and can be used to store and retrieve information for that program. An example of how this is useful is the case of indexed arrays such as matrices.

The typical method for handling matrices in many programming languages is to use a symbol of the form $A(5, 3)$ to represent an element of a matrix. Since "A", "("", ",", ")", and numeric digits are all legal characters in the SPRINT alphabet, the concatenation $A(5, 3)$ is a legal SPRINT word, and thus should be used as the name for a data list of one numeric word representing the matrix element $A(5, 3)$. As concatenation (symbolized by "CON") and decimal arithmetic are basic operations in SPRINT, the above scheme for storing the elements of a matrix can be programmed quite easily. For example, the SPRINT program list

\[
\begin{align*}
\text{I} & \text{ J} + \text{ I} \text{ A} \text{ CON} \text{ CON} \\
\text{ CON} & \text{ CON}
\end{align*}
\]

would produce the "name" of the element $A_{ij}$ for any numeric values of "I" and "J," i.e. if "I" and "J" were the names of the list containing the single numeric data words "5" and "3" respectively, the above program list would output the SPRINT word "A(5, 3)." (Note that the reverse order of concatenation is due to the suffix nature of SPRINT.)

It should be noted here that the above scheme for naming the elements of a matrix is completely independent of any knowledge concerning the size of the matrix. This gives us a completely dynamic storage allocation procedure for matrices which never requires the use of any array size declarations.

The above techniques employed with matrix notation can obviously be generalized to operate with arrays having any number of indices (the only limitation being the maximum length of a SPRINT word). A further generalization is to construct a data scheme where the number of indices itself is a variable. An example of this is canonical tree addressing. Figure 3 shows an example of a simple tree with the addresses of each node given, which could be used in a SPRINT program exactly as shown, i.e., "A(2, 1, 2, 1)" would name a list containing the data to be associated with the lowest node on the tree in Figure 2.

**Mechanization of the SPRINT language**

The SPRINT language has been mechanized on two machines, the IBM 7094 and the IBM 7040. The 7040 version, being the latter, is somewhat improved over the 7094 version. Both operate in the interpretive mode and use a combination hash address linked list search for the associative memory. The push down stacks are mechanized by linear arrays of computer memory backed up by reels of magnetic tape or sequentially accessed disk memory, thus giving them practically unbounded length. Both versions of the interpreter contain roughly 3000 machine instructions and operate reasonably efficiently considering that a binary machine is not well suited for the mechanization of SPRINT. As an example, six minutes were required to compute the factorials up to 93 factorial (in full precision!). This required more than 4400 recursions and multiplications, many of which involved numbers with decimal precisions greater than 100 digits.

Although quite usable, the mechanization of the SPRINT interpreter is still in the experimental stage, and improvements and additions are continually being made.

**A programming example**

As an example of what SPRINT programming looks like and how it operates, the tradition for list processing languages was followed and a program for computing Ackermann's function was written. This program, however, differs from the classical Ackermann function routine in several aspects.

Firstly, Ackermann's function was defined not in its usual form but rather in terms of an infinite class of *primitive recursive functions*, i.e.

Each of the expressions on the right is a primitive recursive function, and in fact consists of an operation which is a simple repetition of the previous operation. The first three of these operations can of course be programmed directly in SPRINT without "loops." The rest of course requires looping or recursive subroutines.
The result of programming Ackermann's function in this manner was a tremendous gain in efficiency. This program was able to compute values of Ackermann's function in a few seconds which upon calculation would appear to take millions of years using programs such as that in the IPL-V manual.5

At first glance, it would seem that a program of this sort, although efficient, could not be general since an infinite number of subroutines would be required to describe the infinite class of primitive recursive functions. Nevertheless, this program is completely general (ignoring such limitations as the fact that the SPRINT machine has to fit in the physical universe), since for any given computation only a finite number of the primitive recursive functions need be defined. This program is written to write as many subroutines as it needs to compute Ackermann's function for the arguments given.

This program then indicates many of the powerful features of SPRINT in that it is easily able to perform the following operations:

1. Test whether or not a given subroutine exists in its memory.
2. Perform computation for writing subroutines.
3. Execute subroutines after they have been written (or modified).
4. Produce an expanding hierarchy of information within its internal structure.

The program for computing Ackermann's function is shown in Figure 4. This is a listing of the input cards to the program; it is not directly in SPRINT but in a second level language. The input program which reads in these cards and compiles them into SPRINT is actually written in SPRINT, thus again illustrating the non-stratification of SPRINT. There are several meta-linguistic characters which have the following interpretation at "load time."

* introduces the name of a new list.
* introduces the name of a new list.
+
and-
introduce a word of data
/
introduces comments.
$ marks the end of input.

The absence of a special character indicates an instruction word.

Blanks separate words.

Figure 5 illustrates a sample of output from the program. Note that A(3, 75) would not only take a long time to compute using the IPL-V routine, but is also too large a number to be handled in most IPL-V mechanizations. Of course, the values of Ackermann's function become so large so quickly, that even the increased power of this routine gains little once the higher values of arguments are reached.

SUMMARY

SPRINT is a programming language which is being designed to be an experimental tool for studying problems in the theory of programming, and for testing algorithmic concepts. Since mechanizations of SPRINT on present machines must operate interpretively, in order that the language be completely unstratified, sacrifices are necessary with regard to the overall efficiency of the system. Much of the lack of efficiency could be overcome, however, if SPRINT were mechanized on a computer which had hardware facilities built in for performing some of the more complicated functions of SPRINT. Some work is currently being done in this area. In fact some machines presently have built in push down stacks, and decimal as well as binary arithmetic circuits, and studies are being made on hardware associative memories. Thus part of the work of the project for which SPRINT was developed was concerned with the design of a machine which would have SPRINT as its machine language.

Experience has shown that SPRINT is quite easy to learn and use. Persons of various backgrounds ranging from high school students to Ph.D. candidates have been able to use SPRINT with little difficulty. A recent example concerns a student who developed an operating syntax directed recognizer in about two weeks work, using SPRINT. This program was written "from scratch" without prior knowledge of SPRINT or the employed algorithms.

An example of a language which is semantically similar to SPRINT is the GIPSY system developed by W. H. Burge.7 SPRINT, however, is a much more highly developed language, especially with respect to the use of SPRINT words to form complex data structures. This is due to the fact that SPRINT language is handled directly throughout execution.

ACKNOWLEDGMENTS

I would like to give thanks to Dr. Harry J. Gray, Dr. John W. Carr, III, Mr. Alvin Vivatson and Mr.
SPRINT INTERPRETER OUTPUT

READ IN ARGUMENTS
   +x  KDT

COMPUTE ACKERMANN'S FUNCTION.
   +2  x  +3  ADD  REV  +0  CON
   DO  +3  SUB

CREATE OUTPUT FORMAT, AND PRINT
   +1=  +x  BNG  +,  REV  +AI  CON
   CON  CON  CON  CON  +Y  TRA  +Y
   WDT  +  CLL

*DO  / ROUTINE FOR CALLING OR WRITING ROUTINE
   FND  GEN  CLL

*GEN  / ROUTINE FOR STARTING THE PROGRAM WRITING
   REP  +NAME  TRA  MAKE

*MAKE  / ROUTINE FOR THE ACTUAL WRITING OF THE NECESSARY PROGRAMS
   +1  DCN  DEL  REP  +1  SUB  +GO
   BNG  CON  +NEXT  TRA  +GO  BNG  CON
   +THIS  TRA  +GUZ  BNG  +P  +THIS  BNG
   CON  +GTZ  BNG  +THIS  BNG  +3  STO
   +NEXT  BNG  REP  +THIS  BNG  +GHD  BNG
   +P  +THIS  BNG  CON  +8  STO  FND
   MAKE  DONE

*DONE  DEL  NAME  CLL  / AFTER ROUTINES ARE WRITTEN, CALL THEM

/ DATA FOR MAKE
   *GO  0  *GOZ  OZ  *GTZ  TZE
   *GHD  *TEMP  TRA  REP  TEMP  +1  SUB

/ NON RECURSIVE OPERATORS
   *00  REV  DEL  +1  ADD  *01  ADD
   *02  MPY  *OZ  DEL  DEL  +1  $
William Slemmer who contributed much to the development of SPRINT.

The work on SPRINT has been supported by the following:

Contract numbers
Rome Air Development Center, Research and Technology Center, Air Force Systems Command, Griffiss Air Force Base, New York, Contract No. AF 30(602)-2994, Project 5581, Task 558102.


REFERENCES
1 S GORN
Processors for Infinite Code of the Shann-Fano Type
Proceeding of the symposium on mathematical theory of automata
Polytechnic Institute of Brooklyn April 1962

2 H F GRAY C KAPPS ET AL
Interaction of computer language and machine design
Technical report No RADC TR 64 511 May 1965

3 C KAPPS
Basic programmer's manual for SPRINT I
University of Pennsylvania
The Moore School of Electrical Engineering
Available upon request from the author December 1964

4 J MCCARTHY ET AL
LISP 1.5 programmers manual
MIT Press Cambridge Massachusetts 1962

5 A NEWELL Ed
Information proceeding language V manual
Prentice Hall
Englewood Cliffs New Jersey 1961

6 K C KNOWLTON
A programmers description of L6
Communications of the ACM
Vol 9 no 8 pp 661-625 August 1966

7 W H BURGE
Interpretation stacks and evaluation
Introduction to system programming
Syntax-checking and parsing of context-free languages by pushdown-store automata

by VICTOR SCHNEIDER

Computer Science Center, University of Maryland
College Park, Maryland

INTRODUCTION

By way of two heuristic examples, this paper demonstrates an algorithm that constructs computer-realisable machines for syntax checking and parsing of computer programs constructed from context-free grammars. The algorithm followed constructs a pushdown-store, one-way automaton that corresponds to the context-free version of the grammar being considered.* This automaton has the property of accepting a string of symbols on its input tape if and only if that string belongs to the language of its grammar. In addition, it signals the source of error in an unacceptable string on its input tape as a result of the correspondence between states of the automaton and rules of the grammar.

If the automaton designed from some grammar is initially nondeterministic in operation, a further algorithm can be applied for constructing a deterministic version of the automaton. Those cases for which a deterministic version of the automaton cannot be constructed by this algorithm are conditions for the impossibility of checking the grammaticality of its language in single, one-way scans. These cases further correspond to a sufficient condition for the language to be ambiguous.

Notation

In this paper, a pushdown-store automaton (PDS automaton) consists of:

1. A pushdown store, or tape which is written on from right to left and read from left to right, and an associated alphabet.
2. An input tape that is read from left to right in one scan, and an associated distinct alphabet.
3. A set of states, including an initial state \( S_0 \) and a nonempty set of final states.
4. A next-state relation \( M \), to be defined.
5. A terminating criterion, to be defined.

The alphabet of symbols read on the input tape is represented by (possibly subscripted) lower-case letters, and the push-down-store alphabet is denoted by (possibly subscripted) upper-case letters.

Associated with the next-state relation \( M \) are the tape instructions \((x,y)\), with \( x,y \in \{0,1\} \).

For \((x,y)=(1,0)\), the reading head of the input tape is moved over one space.

For \((x,y)=(0,1)\), the leftmost symbol of the pushdown store is erased, and, either a new symbol is written in its place, or the read-write head moves one space to the right. The conditions that determine whether or not a symbol is written on the PDS tape will be discussed presently.

Boundary symbols are assumed to exist on the input and pushdown-store tapes to prevent the respective tape heads from running off the ends of the tapes.

As an example of how the next-state relation will be defined, suppose that \( a_1 \) is a symbol of the input tape, and \( A_2, A_3 \) are symbols of the pushdown store. \( S_1 \) and \( S_2 \) are states of the automaton in this example. The relation

\[
M(a_1, A_2, S_1) = S_2, A_3, (0,1)
\]

means that, for this example, when the currently read input tape symbol is \( a_1 \), the currently read PDS tape symbol is \( A_2 \), and the automaton is in state \( S_1 \), the following occurs:

The automaton transfers into state \( S_2 \), the symbol \( A_2 \) is erased from the pushdown store, \( A_3 \) is written in place of \( A_2 \), and the reading head on the input
tape continues to scan symbol $a_i$.

The relation

$$M(\emptyset, S_0) = S_f, (0, 0)$$

(1)

means that no more symbols remain to be read on the input tape, the pushdown-store tape is blank, and the machine is in state $S_f$, one of a set of final states. The automaton remains in state $S_f$, the notation $(O, O)$ indicating a halt in operation. Relation (1) is the termination criterion adopted in this paper. When this criterion is satisfied by a computation on some input tape, the tape is said to be accepted by the automaton, and its string is a valid string of the language.

As is shown elsewhere,* it is possible to reduce any context-free grammar to a weakly equivalent, normal-form grammar** whose rules are of the following three types:

$$A_i : = A_i A_{i2}$$  

(2,a)

$$A_j : = A_j a_j$$  

(2,b)

$$A_k : = a_k$$  

(2,c)

In the rules of (2), capitol letters represent the nonterminal symbols of the grammar; i.e., those symbols $A_e$ for which there is at least one rule of the forms (2,a), (2,b), or (2,c), with $A_e$ on the left-hand side. The remaining symbols in lower case are terminal symbols; i.e., symbols that appear in some string of the grammar's language. Note that the alphabet of terminal symbols is disjoint from the alphabet of nonterminal symbols, because the condition defining nonterminals does not apply to terminal symbols.

*Reference 1, Chapter I.

**In many cases of interest, the normal-form grammar is strongly equivalent to the original grammar in the sense that the original parsing of a string of some language can be reconstructed directly from the parsing assigned to that string by the normal form grammar.

As is shown elsewhere,† the nodes $A_{i1}$ and $A_{i2}$ of the partial tree diagram represented in (5) are adjacent in the sense that, before $A_i$ links $A_{i1}$ and $A_{i2}$ in constructing the tree diagram, $A_{i1}$ is the nearest node to the left of $A_{i2}$ that is both unconnected to a node on its right and represents the left-hand side of a rule of the grammar. This property is naturally related to the convention of reading the topmost symbol of a pushdown store.

In constructing a pushdown-store automaton from the normal form of some grammar, three cases of transitions arise, corresponding to the three types of rules in normal form.†

Case I. Rules of the form (2,c), $U_j : = U_j$.

Next-State Relation (a): $M(U_j, S_0) = U_j, (1,0)$

The transition corresponding to relation (a) above can be pictured as a transition in the state diagram of the PDS automaton:

$$\text{Transition (a): } S_0 \rightarrow U_j, u_j/(1,0)$$

In (a), $u_j$ is read from the input tape. A transition occurs from initial state $S_0$ to state $U_j$.

Next-State Relation (b): $M(U_j, S_0) = S_f, U_j, (1,0)$

The transition corresponding to relation (b) can next be given in terms of the state diagram of the PDS auto-

†Subcases (a) and (b) will be explained in what follows.
maton:

**Transition (b):**

\[ S_0 \quad \text{u}_j U_j (1,0) \]

In (b), \( u_j \) is read from the input tape. Symbol \( U \) is written on top of the pushdown-store tape. A transition occurs to initial state \( S_0 \).

Transitions (a) and (b) of Case I correspond to adding a node and one branch to the partly constructed tree diagram over a string of some language, as shown in example (3) above.

**Case II. Rules of the form (2,b), \( U_j : = V_j V_{j'} \).**

**Next-State Relation (a):**

\[ M(v_j, V_j) = U_j, (1,0) \]

The state-diagram transition corresponding to relation (a) is given by

**Transition (a):**

\[ V_j \quad v_j (1,0) \]

In (a), \( v_j \) is read from the input tape. A transition occurs from state \( V_j \) to state \( V_j \).

**Next-State Relation (b):**

\[ M(W_j, W_j) = S_j, U_j, (0,1) \]

The state-diagram transition corresponding to relation (b) is given by

**Transition (b):**

\[ W_j \quad S_j \]

In (b), symbol \( W_j \) is read from the top of the pushdown store (leftmost symbol on the PDS tape) and erased. A transition occurs from state \( W_j \) to state \( U_j \).

**Transitions (a) and (b) of Case II correspond to adding a node and two branches to the partly constructed tree diagram over a string of some language, as shown in example (4) above.**

**Case III. Rules of the form (2,b), \( U_j : = W_j W_{j'} \).**

**Next-State Relation (a):**

\[ M(W_j, W_{j'}) = U_j, (0,1) \]

The state-diagram transition corresponding to relation (a) is given by

**Transition (a):**

\[ W_j \quad U_j \quad W_j (O,1) \]

In (a), symbol \( W_j \) is read from the top of the pushdown store (leftmost symbol on the PDS tape) and erased. A transition occurs from state \( W_j \) to state \( U_j \).

**Next-State Relation (b):**

\[ M(W_j, W_{j'}) = S_j, U_j, (0,1) \]

The state-diagram transition corresponding to relation (b) is given by

**Transition (b):**

\[ W_j \quad S_j \]

In (b), symbol \( W_j \) is read from the top of the pushdown store and erased. Symbol \( U_j \) is written in place of \( W_j \) on the pushdown store. A transition occurs from state \( W_j \) to initial state \( S_j \).

Transitions (a) and (b) of Case III correspond to adding a node and two branches to the partly constructed tree diagram over a string of some language, as shown in example (5) above.

We now consider the circumstances under which subcases (a) and (b) are valid transitions of a parsing automaton. For cases I, II, and III above, if \( U_j \) appears on the right-hand side of a rule of the forms (6)

\[ A_e : := U_j b_{a_2} \quad \text{or} \quad A_f : := A_1 U_j \]  

(6)

transitions (a) appear in the constructed automaton. If \( U_j \) appears on the right-hand side of a rule of the form (7),

\[ A_e : := U_j A_{e_2} \]  

(7)

transitions (b) appear in the constructed automaton. In general, for the transition corresponding to a particular rule, both transition (a) and (b) may occur. The following examples will illustrate the construction of PDS automata directly from normal-form grammars, and will indicate how deterministic versions of nondeterministic PDS automata are constructed.

**Examples in the construction of deterministic DPS automata**

**Example 1. A Grammar of Griffiths and Petrick.**

\[ S : := A B \quad B : := b c \]

\[ A : := a \quad B : := B d \]

\[ A : := A B b \]
The grammar of Example 1 is rewritten in normal form by introducing additional nonterminal symbols that are subscripted so that each nonterminal introduced is distinguishable from all other nonterminals in the grammar:

**Grammar 1.1. Example 1 Rewritten in Normal Form**

\[
\begin{align*}
S & : := AB \\
X_1 & : := AB \\
B & : := X_2c \\
A & : := a \\
X_3 & : := B \\
B & : := Bd \\
A & : := X_1b
\end{align*}
\]

By means of the correspondence between rules and automata states introduced earlier, Grammar 1.1 is translated into the PDS automaton of Figure 1.1.

**Figure 1.1.** - The acceptor for the language of example 1.1

The acceptor of Figure 1.1 as shown is not deterministic in operation. This is because two possible transitions may occur from state B when symbol A is read on the PDS tape. In addition, a transition is defined for state B when symbol d appears on the machine input tape. To construct a deterministic version of Figure 1.1, we must construct a table listing a single transition for each possible pair of symbols on the input tape and pushdown store that can appear when the machine is in state B. We do so as follows:

(a) Consider a transition from state B to state S.

In order that the input tape string be accepted, an A must appear on the left end of the PDS tape string, and symbol \( \phi \) (the boundary marker, indicating the end of the input string) on the input tape. The pair of symbols \( \phi A \) can be read in state B. For no other pair of symbols can a transition occur from B to S, since S will then be reading an undefined pair of symbols.

(b) Consider a transition on the loop from state B back to state B. This transition can occur whenever symbol d appears on the input tape and either A or % (the boundary marker of the PDS tape) is read on the left end of the PDS tape string. Since no other transition between states of the automaton involves reading symbol d on the input string, whenever symbol d is read on the input tape in state B, a transition occurs along the loop leading back to state B.

(c) Consider a transition from state B to state \( X_1 \).

To make this transition, an A must be read from the PDS tape. In addition, only one transition (in this example) is defined from the state following B on this path, namely from \( X_1 \).

Thus, whenever A appears on the PDS tape and symbol b is read from the input tape in state B, a transition occurs to \( X_1 \).

Cases (a), (b), and (c) represent all possibilities that appear in this example. Hence, all transitions from state B are uniquely defined in terms of pairs of symbols read from the input and PDS tapes, and we can construct the deterministic version of Figure 1.1, as shown in Figure 1.2.

**Figure 1.2.** - The deterministic version of figure 1.1

In Example 1, all transitions possible between states of the PDS acceptor are listed. When the acceptor is performing a computation in which some symbol read from the input or PDS tape is not defined for any transition from the current state of the acceptor, the input string is not part of the language accepted by the machine. It happens that this property of undefined transitions is true in general for machines constructed by the algorithm illustrated above. Because of the correspondence between states of the PDS acceptor and rules of its grammar, undefined machine transitions during some computation of the machine automatically signal the source of errors in the input string (or computer program) being processed. By our convention, the computation that accepts some string of a language must terminate in a state corresponding to the initial symbol of its grammar. In Example 1, the initial symbol is S, and the final state of the constructed machine is state S.

As a second example, consider the following grammar of Eickel et al.:

**Example 2. An ALGOL Sub-Grammar of Eickel.**

\[
\begin{align*}
Z & : := VeZ \\
Z & : := V \\
V & : := Ue \\
Z & : := e \\
U & : := adb
\end{align*}
\]

In the grammar above, there is a rule of the form

\[
Z : := V
\]

with both Z and V nonterminal symbols. Such a rule
cannot appear in a grammar converted into a PDS acceptor using the algorithm of this paper. Since a weakly equivalent grammar can always be constructed so that rules like (8) are absent, we construct Grammar 2.1, weakly equivalent to Example 2 and lacking rule (8) above:

Grammar 2.1. A Grammar Weakly Equivalent to
\[
Z : = \text{VcZ} \\
Z : = \text{Ue} \\
Z : = e \\
V : = \text{Ue} \\
U : = \text{adb}
\]

![Figure 2.1](image)

Figure 2.1. The PDS acceptor constructed from grammar 2.1

The acceptor constructed from the normal-form version of Grammar 2.1 is shown in Figure 2.1 above. Because of the choice of this grammar, two transitions from state U are possible for the same input-tape symbol. Figure 2.2 indicates the transition from state U, and the transitions which can follow. As can be seen in Figure 2.2, the sequence of symbols ec appearing on the input tape unambiguously causes a transfer to state S0 from U by way of state (Z, V).

![Figure 2.2](image)

Figure 2.2. Transitions following state U of the acceptor in figure 2.1

The sequence of input tape symbols ed, with e read on the input tape in state U and d read on the input tape in the combined state (Z, V) unambiguously causes a transfer from (Z, V) to Z. Using these transitions of the states following state U, we can immediately construct the deterministic version of Figure 2.1, as shown in Figure 2.3.

![Figure 2.3](image)

Figure 2.3. — The deterministic version of the acceptor in figure 2.1, incorporating the transitions of figure 2.2

Automatic construction of tree diagrams

As can be verified readily, the string of symbols (8)
\[
s = abcbbcd
\]
belongs to the language of Example 1. Using the criteria of Section III of this paper, we can indicate the correspondence between the record of a computation of the machine in Figure 1.2 given string (8) above, and the construction of the appropriate tree diagram over that string. Figure 3 below shows the sequence of state transitions made by the machine of Figure 1.2 when accepting string (8) on its input tape. The lines drawn between state symbols and input tape symbols in the figure can be produced automatically, since lines only connect states with symbols read by states or with immediately preceding states of the computation.* Circles are drawn around PDS symbols in Figure 3 to distinguish them from states. PDS tape symbols recognized during the computation are located where a state would be written in the diagram. Thus, initial state S0 does not appear as a symbol on the constructed tree diagram.

ACKNOWLEDGMENT

The author wishes to acknowledge many valuable discussions on the subject of the paper with Drs. A. A. Grau, G. K. Krulee, and E. Shamir, all of Northwestern University.

REFERENCES

1 V SCHNEIDER

* A more detailed account of this algorithm can be found in reference 1.
2 R W FLOYD
The syntax of programming languages—a survey
Comm ACM 6 451 1963

3 T V GRIFFITHS S R PETRICK
On the relative efficiencies of context-free grammar recognizers
Comm ACM 8 289 1965

4 J EICKEL M PAUL F L BAUER K SAMELSON
A syntax controlled generator of formal language processors
Comm ACM 6 451 1964

5 S GINSBURG S GREIBACH M HARRISON
Stack automata and compiling
System Development Corp Document No TM-738/021/00
Santa Monica 1965

6 G K KRULEE D M LANDI
Ambiguity and delay in finite encodings
Dept of Industrial Engineering and Management Science
Research Memorandum Northwestern University
Evanston Illinois 1965

7 N CHOMSKY
Formal properties of grammar
in Luce R Bush R and Galanter E Eds

8 Y BAR-HILLEL M PERLES E SHAMIR
On formal properties of simple phrase structure grammars
Zeitschrift fuer Phonetik, Sprachwissenschaft und Kommunikationsforschung 14 143 1961

Figure 3.—A tree diagram record of the computation performed by the acceptor of figure 1.2 on string (8)