DOCUMENT RESUME

ED 259 708 IR 011 740

AUTHOR Reeker, Larry H.; And Others
TITLE Artificial Intelligence in ADA: Pattern-Directed Processing. Final Report.
INSTITUTION Air Force Human Resources Lab., Lowry AFB, Colo.
SPONS AGENCY Air Force Human Resources Lab., Brooks AFB, Texas.
REPORT NO AFHRL-TR-85-12
PUB DATE May 85
CONTRACT F49620-82-C-0035
NOTE 46p.
PUB TYPE Reports - Descriptive (141)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Algorithms; *Artificial Intelligence; *Computers; Computer Software; *Programing; *Programing Languages; *Systems Development; Technical Assistance; Technological Advancement
IDENTIFIERS *Ada (Programing Language)

ABSTRACT

To demonstrate to computer programmers that the programming language Ada provides superior facilities for use in artificial intelligence applications, the three papers included in this report investigate the capabilities that exist within Ada for "pattern-directed" programming. The first paper (Larry H. Reeker, Tulane University) is designed to serve as an introduction to pattern-directed programming and to the significance of the two papers that follow. It includes discussions of artificial intelligence programming and the facilities provided by the Ada language, pattern-directed computation, pattern matching, and parsing. The second paper (John Kreuter, Tulane University) describes a project which was part of an overall effort to add useful artificial intelligence tools to Ada through use of pattern-directed string processing of the sort available in the language Post-X (Bailes and Reeker, 1980). The third paper (Kenneth Wauchope, Tulane Unversity) presents a pattern-directed list processing facility for the Ada programming language. Pattern lists for matching against source lists are constructed from a set of SNOBOL4-derived primitives which have been extended to be applicable to arbitrarily complex LISP-like data structures. A list of references completes the document. (JB)

AFHRL-TR-85-12

ARTIFICIAL INTELLIGENCE IN ADA: PATTERN-DIRECTED PROCESSING

Larry H. Reeker
John Kreuter
Kenneth Wauchope

Tulane University
New Orleans, Louisiana 70118

TRAINING SYSTEMS DIVISION
Lowry Air Force Base, Colorado 80230-5000

May 1985
Final Report

AIR FORCE HUMAN RESOURCES LABORATORY

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235-5601

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

HUGH BURNS, Major, USAF
Contract Monitor

JOSEPH Y. YASUTAKE, Technical Director
Training Systems Division

ANTHONY F. BRONZO, JR., Colonel, USAF
Commander

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFHRL-TR-85-12	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Training Systems Division Air Force Human Resources Laboratory 6b. OFFICE SYMBOL (If applicable) AFHRL/ID	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Lowry Air Force Base, Colorado 80230-5000	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Human Resources Laboratory 8b. OFFICE SYMBOL (If applicable) HQ AFHRL	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code) Brooks Air Force Base, Texas 78235-5601	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 61101F; PROJECT NO. ILIR; TASK NO. 20; WORK UNIT NO. 14

11. TITLE (Include Security Classification)
Artificial Intelligence in Ada: Pattern-Directed Processing

12. PERSONAL AUTHOR(S)
Reeker, Larry H.; Kreuter, John; Wauchope, Kenneth

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM _____ TO _____	May 1985	44

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
05	09	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada; artificial intelligence; data structures; linguistics; parsing; pattern matching; programming; programming languages

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

If the programming language Ada is to be widely used in artificial intelligence applications, it will be necessary to demonstrate to programmers that it can provide superior facilities for use in that domain. One means of doing this is to provide facilities for "pattern-directed" programming within Ada. This report includes three papers, of which the first is designed to serve as an introduction to pattern-directed programming and to the significance of the two papers that follow. It includes discussions of artificial intelligence programming and the facilities provided by the Ada language, pattern-directed computation, pattern matching, and parsing. The other two papers deal with the use of Ada for pattern-directed programming. One paper deals with efficient implementation of pattern matching (within Ada), important because pattern matching tends to be inefficient, leading to problems with excessive processing time. Another paper treats extensions of pattern-direction from strings to more general data structures of the sort used in artificial intelligence.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL Nancy A. Perrigo Chief, STINFO Office	22b. TELEPHONE NUMBER (Include Area Code) (512) 536-3877 22c. OFFICE SYMBOL AFHRL/TSR

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

ARTIFICIAL INTELLIGENCE IN ADA:

PATTERN-DIRECTED PROCESSING

by

Larry H. Reeker*
John Kreuter
and
Kenneth Wauchope*

Tulane University†

1984

---

*Current address, Navy Center for Applied Research in Artificial Intelligence, Code 7510, Naval Research Laboratory, Washington, D.C. 20375.

†Work performed at Air Force Human Resources Laboratory, Training Systems Division, Lowry Air Force Base, Colorado, under Contract No. F49620-82-C-0035 from the Air Force Office of Scientific Research to the Southeastern Center for Electrical Engineering Education.

PREFACE

During the summer of 1984, under the auspices of the Summer Faculty Research Program and the Graduate Student Summer Support Program of the Air Force Office of Scientific Research, administered by the Southeastern Center for Electrical Engineering Education, work was undertaken at the Air Force Human Resources Laboratory, Lowry AFB, Colorado, concerning use of the programming language Ada for artificial intelligence programming. Two projects were undertaken, both of which relate to "pattern-directed" programming, by John Kreuter and Kenneth Wauchope, under my direction. I have edited the final reports on these projects and provided them with an introduction, so as to make them intelligible to a larger audience than might otherwise have been the case.

Mr. Wauchope, Mr. Kreuter and I would all like to acknowledge the support of the Air Force Systems Command, Air Force Office of Scientific Research, and the Air Force Human Resources Laboratory (Training Systems Division). At AFHRL, Maj. Hugh Burns deserves special thanks as the person with whom we interacted most closely, and Dr. Roger Pennell, as the person who interfaced with the AFOSR/SCEEE summer program. Col. Crow, Dr. Yasutake and Maj. Baxter were all very cooperative and helpful administratively, as were Mr. Marshall and Maj. Bolz in the computing area, and a number of AFHRL staff members made the visit pleasurable and productive.

L.H.R.

NOTE

Opinions expressed in this report are those of the authors and do not necessarily reflect those of the Air Force.

CONTENTS

INTRODUCTION: AI IN ADA?

Larry H. Reeker*
Tulane University

Abstract

If the programming language Ada is to be widely used in artificial intelligence applications, it will be necessary to demonstrate to programmers that it can provide superior facilities for use in that domain. One means of doing this is to provide facilities for "pattern-directed" programming within Ada. This first paper is designed to serve as an introduction to pattern-directed programming and to the significance of the two papers that follow. It includes discussions of artificial intelligence programming and the facilities provided by the Ada language, pattern-directed computation, pattern matching and parsing. The other two papers deal with the use of Ada for pattern-directed programming. One paper deals with efficient implementation of pattern matching (within Ada), important because pattern matching tends to be inefficient, leading to problems with excessive processing time. Another paper treats extensions of pattern-direction from strings to more general data structures of the sort used in artificial intelligence.

1. THE PROBLEM

The question implicit in the title of this paper might be "Can artificial intelligence be done in Ada". It might also be "*Will* artificial intelligence be done in Ada", which is more to the point, since anything *can* be done in Ada. The purpose of the research reported in the three papers comprising this report is to explore methods of doing artificial intelligence within Ada, using pattern-directed programming. The goal is to show that Ada, appropriately used, can facilitate the programming of artificial intelligence applications.

Ada is the new standard programming language developed for the United States Department of Defense (DoD). It is intended that Ada be used for mission-oriented applications programs within DoD, replacing a variety of languages that have been used previously. Concepts in Ada are based to a large extent on the languages SIMULA and Pascal. Most artificial intelligence (AI) and computational linguistics (CL) research, on the other hand, is done in the language LISP, with some done in Prolog, SNOBOL4, and a variety of other languages. Even within DoD, such research continues to be done in these languages, rather than in Ada. But artificial intelligence research is ultimately applications-oriented, and what we consider to be AI today will be an important part of applications of the future, at all levels from office automation and record keeping to command and control and maintenance-aiding.

If Ada is to be the common DoD language and if various "intelligent" applications are to be interfaced to programs written in Ada, then it would be convenient to be able to program AI and CL applications in Ada. Brian Dallman [1984] has expressed the problem as follows:

> Since Ada recently became the DoD standard computer language, ideally it should be used for all programming applications within DoD. However, there are some applications for which Ada is not currently practical. One of these areas is artificial intelligence. In DoD, the majority of programming for AI applications is done in LISP. Consequently, if

*Current address, Navy Center for Applied Research in Artificial Intelligence, Code 7510, Naval Research Laboratory, Washington, D.C. 20375.

> LISP remains the primary AI language, then Ada's usage and acceptance in a critical new area of software engineering will be severely limited and DoD's effort to establish a common high order language will be hampered.

With these considerations in mind, Dallman suggests the following objective:

> To develop an extension of the Ada language which will provide the capabilities for AI programming applications. This extension can involve possibly only an Ada package or collection of packages.

Two research efforts undertaken in the summer of 1984 toward this objective are reported in the papers that follow in this report. The first, by John Kreuter, looks at methods for implementing efficient pattern-directed computation in Ada. The second, by Kenneth Wauchope, deals with the development of LISP-like list processing and of a language for pattern-directed computation on list structures.

The objective here is not merely to *mimic* LISP in Ada, but to *improve upon* LISP, which has some well-known defects, despite its popularity. We have chosen the pattern-directed paradigm of programming for this purpose. There is today a body of opinion, shared by this author, that says that pattern-directed facilities provide the most effective means for creating complex programs for non-numerical applications. That this opinion is not universally shared could have to do with different individuals' programming styles; but we quote here an opinion that supports our view in this matter [Warren, Pereira and Pereira, 1977]:

> Pattern matching should not be considered an "exotic extra" when designing a programming language. It is the preferable method for specifying operations on structured data, both from the user's and the implementor's point of view. This is especially so where more than one record type is allowed.

The remainder of this paper will concern itself with some of the background issues that will provide a rationale for the work being done and help the reader to understand the Kreuter and Wauchope papers. We shall first look at the programming requirements of artificial intelligence and the facilities provided by Ada at present.

1.1. LANGUAGES FOR ARTIFICIAL INTELLIGENCE PROGRAMMING

Although one could write artificial intelligence programs in any language, certain languages lend themselves to the task. This is largely because they have the data structures that are most natural for the complex information processing necessary in AI built into the language, and because they also feature the operations that are needed to handily manipulate those data structures.

The **linked list** (henceforth, "list") is pervasive in artificial intelligence programming. In early languages, lists were always represented by arrays, and they can still be so represented when it is necessary to use one of the common arithmetic languages, such as FORTRAN*. In other languages, such as Pascal, PL/I and Ada (see §1.2, below), lists are implemented by the provision of a "pointer" datatype. But LISP has long been the most popular AI language because it focuses on lists, providing the needed list-constructing functions and means of selecting the items of a list.

---

*No references are given for the well-known programming languages, as manuals can easily be obtained at bookstores and libraries.

Another datatype that is common in artificial intelligence (particularly in computational linguistics) is the **character string** (henceforth, "string"). A string can be represented as an array of characters (as, for instance, in APL) or as a list of characters. But programmers tend to think about strings in a different way. They tend to think about ***patterns of characters***, and patterns have therefore tended to be emphasized in string processing languages, such as SNOBOL4 (see §2.1, below).

Another data structure that can be represented as a type of list or as a parenthesized string, but that is often conceptualized quite separately from these, is the **tree**, commonly used in games, taxonomies, structural descriptions of strings (parse trees), and the like. Like the string, it is often processed in terms of looking for a pattern. This is particularly apparent in transformational grammars, some examples of which will be seen in Wauchope's paper in this report. Pattern-directed manipulation of trees is not natural in most extant languages, and Wauchope's system is aimed at making it more natural in an extension of Ada.

There have been attempts to generalize structures like trees and lists to directed or undirected linear **graphs**, which may contain cycles (trees are directed acyclic graphs with a single origin or "root node"). These may yet turn out to be useful, and it is suggested that pattern-directed processing will also be useful in processing these generalizations. It is not clear, however, how to treat graphs that are not trees directly, rather than in terms of trees, for pattern matching purposes.

There are factors other than data structures that characterize the languages and environments in which productive AI work is taking place. Richardson [1983] cites the following:

1. Focus on symbol manipulation and list processing
2. Support of representations which change dynamically
3. Support of flexible control by pattern matching rather than procedure calls
4. Supportive programming environment, including
   a. An interactive (interpreted) language
   b. A good editor (program construct oriented, not text oriented)
   c. Debugging facilities (traces, breaks)
   d. Standard systems input/output functions

Of these factors, the first two basically have to do with the processing of lists and strings, which are, by their nature, dynamic entities (their shapes and sizes change throughout processing). Languages which try to do string and list processing with less dynamic entities (e.g. fixed arrays), can quickly be eliminated from contention, unless these entities can be made to ***appear*** dynamic to the programmer. Pattern matching we will address below. We will not directly address the programming environment, except to comment that the types of facilities that we are seeking to provide in Ada can be ***abstracted*** from the language and placed in a "languageless" programming environment (which is not ***really*** languageless, since there is always need for a representation, but is not textually oriented, either). In this case, the underlying programming language is almost irrelevant — it could be LISP, or Ada, or anything else (see [Reeker and Bailes, ***in preparation***]).

Language extensibility, discussed in §1.3, has also been important in AI and CL, since the fields — and therefore, their language support needs — have been evolving

rather quickly. There is no reason to believe that the pace of change is going to slow down in the near future, so one might conclude that extensibility is another need in an AI language.

1.2. PROGRAMMING FACILITIES PROVIDED BY ADA

The Ada* language (see [Honeywell, 1983,1984] and a variety of texts presently on the market and under preparation) was designed for the U. S. Department of Defense, in an attempt to promote language standardization in applications programs and program reliability and maintenance, while maintaining program efficiency. It has a variety of features designed to make it useful in general applications. We shall briefly describe only those that are relevant to the discussion in this report and different from other commonly-used languages, such as Pascal.

An Ada program may contain various types of **program units**, each of which is a **subprogram**, a **package**, a **task**, or a **generic** unit. Each unit contains a **specification** and a **body**. The specification contains information that must be visible to other units, while the body contains implementation details. Units may be compiled separately.

Subprograms consist of the usual procedures and functions, and will not be discussed further. Tasks are units that may be invoked and executed in parallel with other tasks. Generally, in the absence of parallel computation facilities, tasks are executed in an interleaved fashion, but multiprocessing is clearly possible, and it is envisioned that parallel execution will be used commonly as the hardware becomes available. As an example of tasks, consider a multi-player game. Each player could be considered as a task, or as instances of the same task with different parameters (say, different hands in a card game, passed to the instances on invocation).

Packages are usually used to define new datatypes and the operations on them. Portions of the package can be declared **private**, so that details not necessary to the user are "hidden" from the user, thus adding to the apparent (though not necessarily the underlying) simplicity of the program. Both packages and tasks are an outgrowth of SIMULA classes.

Generic subprograms or packages allow the definition of program units that will be applicable to all types of a given class (rather than just a single type). **Derived types** can also be used to the same effect in many instances.

In addition to the usual built-in arithmetic datatypes, Ada provides predefined **character** and **string** datatypes. Strings are vectors (one-dimensional arrays) of characters, indexed by positive integers. The concatenation operator (called **catenation**) is **&**. The built-in string facilities are, however, primitive, and require augmentation to be truly useful. **Access** datatypes (pointers) are used with **record** types to do list processing, much in the manner of Pascal. As with the string processing facilities, the list processing facilities built into the language are clumsy and require extension.

1.3. AI'S NEEDS AND ADA'S FACILITIES

It has often been pointed out that LISP owes much of its success as an AI language to its usefulness as a sort of high-level systems programming language, in which it is

*Ada is a registered trademark of the U. S. Government, Ada Joint Program Office.

possible to construct interpreters for higher-level languages. In other words, it is *extensible*, although it was not designed with extensibility in mind, specifically. The fact that LISP programs are themselves representations of lists facilitates extensibility. (It might be remarked here that the extension languages have tended to share LISP's syntactic shortcomings, partly for this reason.) The applicative* nature of LISP also facilitates extension. Ada has been designed for extensibility — albeit of a limited sort, using packages, generic procedures and tasks. It remains to be seen if this lim[redacted]nsibility will prove as useful as that of LISP.

Ada makes few concessions of a *direct* sort to AI (or to any particular application area), the philosophy being that these facilities will be built upon the basic language. In the Dallman quotation of §1, the package is mentioned. This will be the primary means of adding AI-oriented features, including, but not limited to, string processing and list processing (as described in §1.1). One might envision the creation of the following:

1. String definition and manipulation facilities more flexible than those built into Ada.
2. List processing functions
3. Pattern definition and matching functions for strings and lists
4. Means of manipulating lists returned by the pattern matching functions

A package of string functions has been written by Major R. Bolz [*personal communication*]. In the third paper included in this report, K. Wauchope reports on the provision of list processing facilities and pattern matching functions for lists, while J. Kreuter studies efficiency in string pattern matching methods that could be implemented in Ada. The manipulation of the lists returned by pattern matching functions could be in the manner of Post-X (see §2.3), as Wauchope points out. The exact manner of building in the "actions" of Post-X is a subject for further investigation.

The tasking mechanisms of Ada lead to a number of interesting possibilities. One of them is tentatively explored in the Kreuter paper. It is possible to use "coroutines", which are just a form of task in Ada, to match patterns in a particularly elegant fashion. For the purposes of the type of processing envisioned in our project, the pattern matching would have to provide a structural description of the item matched, as well as an indication of the match. This can be done in much the same manner as SNOBOL4 assignments, making an assignment to each subpart of the pattern. Other possibilities for the use of tasks arise in artificial intelligence in any of the areas where quasi-parallel processes have been used. An example is "word expert" parsing [Rieger and Small, 1979].

2. PATTERN-DIRECTED COMPUTATION

In a pattern-directed computation, the operation that drives the computation is that of finding a pattern in the data and making a change in the data at that point. Pattern directed computation has generally been identified with the processing of character strings. Let us therefore turn to string processing languages to get a feel for this style of programming.

---

*An applicative language works by function application. LISP is an example of an applicative language.

2.1. PATTERN-DIRECTED STRING PROCESSING LANGUAGES

Historically, pattern-directed languages are based on a philosophy built into the **normal algorithms** of Markov [1954] and the **canonical systems** of Post [1943], both of which antedate modern electronic computers. Both Markov algorithms and Post systems provide languages adequate for writing *any* program (i.e. for realizing any algorithm), by encoding the data as strings, if one accepts "Church's Thesis", as most computer scientists and logicians do [Rogers, 1967]. More important, from our point of view (since other programming languages are theoretically adequate in this same way), is the fact that a particular style of programming, which many programmers find particularly cordial, is natural in these languages.

The COMIT programming language of Victor Yngve [1958] was essentially a computer implementation of a version of Markov algorithms (*labeled* Markov algorithms; see [Galler & Perlis, 1970]). In that language, it is assumed that one is operating on a **workspace** containing a sequence of **constituents,** which may be individual characters or character strings. Each step of the program consists of an operation which tries to match a pattern to a portion of the workspace (starting from the left end of the workspace in its matching attempts and working to the right) and effect a change in that portion. As an example, the statement

$$\$1+ABC+\$1+D+\$+E = F+3+1+5$$

would match a single constituent followed by a constituent ABC, followed by another single constituent, followed by a constituent D, followed by any number of constituents, followed by a constituent E, and would replace all of these by a constituent F followed by the first, third, and fifth items matched by the left hand side of the "equation". For example, if the workspace contained

...QRS+ABC+DD+D+EFG+HIJ+E...

at the leftmost place in the workspace where the pattern matched, it would be changed to

...F+DD+QRS+EFG+HIJ...

COMIT had a number of problems as a programming language, but this pattern-directed mode of computation was not one of them, as it turned out to be a natural means of processing character strings in computational linguistics and related fields. It also led to a more successful family of languages, the first of which was called SNOBOL [Farber *et al*, 1964], and the last of which was called SNOBOL4 [Griswold *et al*, 1973].

The original SNOBOL language was similar to COMIT, but with a number of important improvements. The most fundamental of these was the inclusion of variables that could take on the values of strings, rather than the single workspace (COMIT had a construct called "shelves" for storing away portions of the workspace, but SNOBOL's string variables were handier.) SNOBOL also had a more flexible flow of control than COMIT and other improved features.

By far the most popular of the SNOBOL family of languages has been SNOBOL4. The papers of Mr. Kreuter and Mr. Wauchope both mention SNOBOL4 patterns, so we shall discuss them briefly here.

As mentioned above, the COMIT workspace constituents were not necessarily single characters, but could be strings. Each constituent was basically indivisible, so if "AND" was a single constituent, it was not treated as "A", "N", and "D". For the purpose of illustration, however, assume that the constituents in the example above, except for the constituent "ABC" are all single characters. With this assumption, the COMIT statement

$$\$1+ABC+\$1+D+\$+E = F+3+1+5$$

could be written in SNOBOL4 as

LEN(1) . V1 'ABC' LEN(1) . V3 'D' BREAK('E') . V5 'E' = 'F' V3 V1 V5.

In SNOBOL4, all strings are based on single characters; the concept of multicharacter constituents does not exist (in the pattern matching portion of the language, at least).

2.2. SNOBOL4 PATTERNS

Patterns, in SNOBOL4, are data objects, and may be given names in assignment statements. Patterns are constructed out of pattern primitives, including variables and string constants, using pattern operators. They may also contain assignment statements.

2.2.1. Pattern Operators

Concatenation: (blank space) *e.g.* **A B** matches anything matched by pattern **A** followed by anything matched by pattern **B**.

Alternation: (blank)|(blank) *e.g.* **A | B** matches anything matched by pattern **A**, if a match is found. If not it matches anything matched by pattern **B**, if that can be found. If neither is found, it fails.

(**Parentheses** may be used in the conventional way to group items and establish the order of operations.)

2.2.2. Pattern Variables

POS(i) matches a null string after the i-th character. **(POS(0)** is the left end of the string).

RPOS(i) matches a null string *before* the i-th character from the *right*. **(RPOS(0)** is the right end of the string.)

ARB matches an arbitrary string (the shortest one possible within the context of the pattern in which it is included).

REM matches everything to the end of the string.

BREAK(x) matches an arbitrary string up to — but not including — the first occurrence of any character in the string x (*e.g.* **BREAK('abc')** matches a string up to one of the characters a or b or c that does not itself contain any of those characters).

SPAN(x) matches an arbitrary string made up of characters in x (*i.e.* it **BREAK**s at anything not in x).

ANY(x) matches any single character in x.

NOTANY(x) matches any single character not in x.

LEN(n) matches an arbitrary n-character string.

BAL matches an arbitrary string balanced with respect to parentheses.

Other pattern variables include **FAIL**, **FENCE**, **ABORT**, **ARBNO**, **TAB**, **RTAB**, **SUCCEED**, **@**, for explanations of which the reader is referred to the SNOBOL4 manual [Griswold *et al*, 1971].

2.2.3. Other Primitives

Any string (enclosed in single or double quotation marks) may be used as a pattern. It matches exactly itself.

NULL matches the null (zero length) string.

2.2.4. Assignment Operators

Immediate assignment (made to a matching element of the pattern as the pattern match is attempted): (space)$(space).

Conditional assignment (made only if the whole pattern match succeeds): (space).(space) (*e.g.* **X . Y** assigns whatever is matched by pattern **X** to variable **Y** when **X** is part of a successful pattern match).

2.3. POST-X PATTERN MATCHING

The ultimate goal of the work reported in these papers is to make possible the incorporation within Ada of packages that allow pattern matching of the sort defined within the Post-X language [Bailes and Reeker, 1980a,b]. Post-X incorporates pattern definition and matching into an applicative framework. In doing so, the powerful pattern definition facilities of SNOBOL4 have been retained, while other aspects of the utilization of patterns have been improved.

In an applicative framework, the pattern match must retain a value that can be acted upon by other functions. The pattern itself has been generalized to a more powerful object, called the **form**.

A Post-X form consists of a series of alternative **patterns** and related **actions**. Each pattern is very much like a pattern in SNOBOL4. A form may be passed parameters (by value), which are then used in the pattern or action portions of that form.

A pattern determines the structure of the string to which it is matched. The pattern contains a sequence of concatenated elements, which are themselves patterns, primitive patterns (as in SNOBOL4), or strings. The value returned by the pattern is either **false** (if it fails to match) or a *parse tree* designating the structure of the string that corresponds to portions of the pattern. Portions of the parse tree can be accessed by the use of selectors and used in the action portion.

As an example of some of these ideas, consider the definition of a form REPLACE which takes a parameter GRAM (a context free grammar that consists of a sequence of rules, with the nonterminals surrounded by angle brackets).

```
REPLACE GRAM := "<"^BREAK">"^">"
                {$<((REPLACE GRAM)<
                    SELECT_RHS
                    (ALT_LIST<
                    (LHS_FIND $2 <GRAM)))
```

```
        ^$>}
       |NULL{$$};
```

The ^ is a concatenation operator. Post-X allows an alternative postfix representation of function composition, using an explicit postfix operator ., which is sometimes easier to read:

```
REPLACE GRAM := "<"^BREAK">"^">"
        {$<^GRAM
          (<(LHS_FIND $2).
          (<)ALT_LIST.
          SELECT_RHS.
          (<)(REPLACE GRAM)
         ^$>}
        |NULL{$$};
```

The form REPLACE, in whichever form it is written, expects to be passed a grammar, as described above. It utilizes the forms LHS_FIND, ALT_LIST, etc., which must be defined elsewhere in the program. It first finds the leftmost occurrence of a nonterminal in that grammar. In the first alternative, which utilizes the SNOBOL4 function BREAK, the first occurrence of "<" is automatically denoted by **$1**, the item matching BREAK">" (the second item in the pattern; notice that Post-X does not require parentheses around the arguments of built-in functions, in order to lessen the number of parentheses necessary) will be denoted by **$2**, and the ">" following will be denoted by **$3**. These "$ variables" are all available to be used in the action or in other parts of the pattern match.

REPLACE uses the nonterminal found (**$2**) as a parameter to LHS_FIND, which is applied to the grammar GRAM to return the right hand alternatives. Then SELECT_RHS selects an alternative, which is placed in the context of the nonterminal matched by the pattern part of the form. Finally, REPLACE is matched (recursively) to the result. If the first alternative fails, it means that there is no nonterminal. In that case, the second alternative will be matched, and will return the entire string, which will be a string in the grammar generated by GRAM.

Without understanding Post-X completely, it can be seen that pattern matching and function application are the fundamental operations. Furthermore, it is necessary for the pattern match to return a structural description of the string (the grouping of higher-level units in the pattern and the selection of corresponding units of the matched string is not illustrated in the example, but often turns out to be very useful). A portion of Post-X has been implemented as STRIP, and its design rationale has been explained in detail by Paul Bailes [1983].

2.4. PROLOG AS A PATTERN-DIRECTED LANGUAGE

The language Prolog has been chosen as the language of the Japanese "fifth generation" computer initiative. It is a language that is becoming more and more popular in artificial intelligence and computational linguistics. A standard reference is [Clocksin and Mellish, 1981].

A program in Prolog consists of a series of clauses of the logical form

$$A_1 \& A_2 \& \ldots \& A_n \supset C$$

represented in Prolog as "C :- A_1,A_2,...,A_n", and interpretable as "to prove C, prove A_1, then prove A_2, ..., then prove A_n".

The elementary terms, such as "A" above, are predicates and arguments (which may be variables). For instance, "A" might be "BIGGER(x,y)". A problem is solved by a Prolog program by finding an instance of a formula that is true and returning the parameters that instantiate that instance. If the data provided in the program contains pairs of, say people who are bigger than other people, then it would be appropriate to ask whether Paul is bigger than John ("?- BIGGER(Paul,John)") or to find the people bigger than John ("?- BIGGER(x,John)"). The process of attempting to find true instances is the logical operation of **unification**, which can also be viewed as ***pattern matching***. Much of the popularity of Prolog is due to the naturalness of this pattern matching method of programming. In fact the quotation from [Warren *et al*] in §1 is talking specifically about Prolog.

The pattern matching embodied in SNOBOL4 and in Post-X is a more limited form than in Prolog, in that there are control mechanisms other than pattern matching (primarily function composition or application in Post-X, both sequential control and function composition in SNOBOL4). The purer approach of Prolog (although, like the pure applicative control of LISP, often modified in practice) has advantages and disadvantages. We feel that the Post-X framework will be more naturally embedded in the Ada framework, and that if this is carefully done, it can result in an excellent language for AI programming.

Within the LISP community, pattern matching has been recognized as important, but has not generally been viewed as ***fundamental***. Thus Winston and Horn [1981] include a chapter on pattern matching, commenting that

> Although LISP itself has no pattern matching built in, it is easy to write pattern-matching functions in LISP. Hence, we say that LISP is a good implementation language for pattern matchers.

An important experimental language built upon LISP, PLANNER [Hewitt, 1969], partially implemented as MICRO-PLANNER [Sussman *et*al 1971], features pattern-directed procedure invocation. Winston and Horn conclude that many problems remain in pattern matching, including how to deal with more general data structures (the problem that Wauchope [1984] is tackling). They also point out that a matcher which can do partial matches and report on how close they are to a full match would be very useful.

Another language that deserves mention in any discussion of pattern-directed programming is Awk [Aho *et al*, 1979]. Although Awk's patterns are of a restricted sort (for purposes of efficiency), it is very easy to use, and is widely used as a utility within the UNIX system, as well as in file processing programs.

3. PATTERN MATCHING AND PARSING

We have described some design aspects of pattern-directed languages. Of course, designing the language is only half of the task; one must also implement it. We will now discuss a central problem of the implementation of pattern-directed languages — the pattern matching algorithm itself. Because we are interested in implementing facilities along the line of Post-X, we will be interested in passing back a structural description of a string. This is essentially the same thing as parsing a string according to a context free grammar, so we shall next examine context free parsing.

3.1. CONTEXT FREE PARSING

Pattern matching can be considered to be a restricted form of context free parsing. Without going into the details of context free grammars (a good basic reference is [Lewis and Papadimitriou, 1981]; a more advanced one is [Harrison, 1978]), it is possible to see why this is so. A pattern P consisting of a concatenation of patterns $P_1 \ldots P_n$ will match some string S if and only if there exists a decomppsition of S into substrings such that $S = s_1 \ldots s_n$ and such that all of the s_i are substrings matched by the corresponding P_i. If there are alternative patterns $P_1 \mid \ldots \mid P_n$ (using the SNOBOL4 alternation operator "|") then one needs to try matching P_1, then P_2, etc. One might want to return either the first match or all matches. The operations of concatenation and of alternation are also the basic operations of context free grammars. Context is often relevant to the parsing process, but if it is strictly the context of primitive patterns within the overall pattern (as it tends to be in pattern matching), the power of a context free grammar will suffice.

In the case of $P = P_1 \ldots P_n$, an equivalent context free grammar would have the production $P \rightarrow P_1 \ldots P_n$. In the case of $P = P_1 \mid \ldots \mid P_n$, the grammar would have productions $P \rightarrow P_1$, ..., $P \rightarrow P_n$. In either case, finding a successful pattern match is equivalent to recognizing the string S by the corresponding grammar (determining whether or not it can be generated by the grammar).

As explained in section 2.3 above, we desire to return a ***structural representation*** of the string matched — that is, a parse tree that indicates how the match took place. It is then possible to structure the patterns utilized so that this information will be useful. Post-X makes use of the parse information to select out certain portions of the matched string for modification in the action portion.

One of the key issues in the efficiency of parsing, addressed in the Kreuter paper, is the control of **nondeterminism**. A nondeterministic algorithm [Floyd, 1967] is one that has "choices" of various alternative computations at certain points in its operation. These choices can lead to a successful completed computation or may lead to failure. The idea of the nondeterministic algorithm is that if a failure occurs, then another choice can be tried. One could, in fact, try all choices at once if one had sufficient parallel computing capabilities, and this may be possible in the future. At present, we implement nondeterministic algorithms on the machines that we have, which are designed for serial computation. One way of implementing them is to ***backtrack*** when a failure occurs and try another choice. Another is to try to keep around enough information to be able to try all alternative choices in a "pseudo-parallel" manner. These alternatives are best illustrated by looking at some parsing algorithms.

3.1.1. Recursive Descent

Suppose a context-free grammar has a rule of the form

$$X_i \rightarrow Y_1 Y_2$$

where each of the Y_j is either a terminal symbol or a nonterminal. A **recursive descent** parsing algorithm will parse a string

$$s_1 s_2 \ldots s_{n-1} s_n$$

that is suspected of being generated by X_i by calling a routine

$$\text{PARSE}(X_i, s_1 \ldots s_n)$$

which must consider the possibilities that Y_1 matches the empty string and Y_2 matches $s_1...s_n$, that Y_1 matches s_1 and Y_2 matches $s_2...s_n$, etc. To check the first possibility, it calls

$$\text{PARSE}(Y_1,\Lambda^*) \text{ followed by } \text{PARSE}(Y_2,s_1...s_n).$$

To check the second possibility, it calls

$$\text{PARSE}(Y_1,s_1) \text{ followed by } \text{PARSE}(Y_2,s_2...s_n)$$

and so forth. If any of the Y_j is nonterminal, then PARSE(Y_j,x) will make further calls, according to the grammar. If Y_j is a terminal (or Λ), then PARSE(Y_j,x) returns an indication of success if and only if x is also terminal (or Λ) and $Y_j = x$. The alternatives are tried in a nondeterministic fashion either until a successful parse is found or until all successful parses are found, depending on which one wants.

The algorithm can be extended in a straightforward fashion to cases where the right hand side of the production consists of more or less than two symbols, either terminal or nonterminal (or Λ).

As an example of recursive descent parsing, consider the grammar

$$S \rightarrow aTc$$
$$S \rightarrow SU$$
$$S \rightarrow T$$
$$T \rightarrow e$$
$$U \rightarrow d$$

on the input string **aecd**. We start with PARSE(S,aecd), which calls

PARSE(a,Λ);PARSE(Tc,aecd),
PARSE(a,a);PARSE(Tc,ecd), ...,
PARSE(S,Λ);PARSE(U,aecd),
PARSE(S,a);PARSE(U,ecd), ..., and
PARSE(T,aecd).

Notice that PARSE(a,a) succeeds, according to the criterion for success given above, whereas PARSE(T,aecd) will call PARSE(c,aecd), which will not succeed. A real problem occurs with the calls to PARSE(S,x), for any x. This is because of the production $S \rightarrow SU$, which means that PARSE(S,x) will be called again and again recursively, and that the program will therefore be in a loop. Any production of the form

$$X_i \rightarrow X_iW$$

(where W is any string of nonterminals and terminals) will cause this problem. There are various solutions to the left recursion problem, one of which leads to predictive parsing, discussed in the next section.

3.1.2. Predictive Parsing

In order to avoid the problem of left recursion and the infinite loop that it can cause in recursive descent parsing, one can put each production into **Greibach normal form** [Greibach, 1965],where each production is of the form

* The symbol Λ is used for the empty (zero-length) string.

$$X_i \rightarrow a_j X_l \ldots X_n$$

where a_j is a terminal character and the X's are nonterminals (including the case $n = 0$, where no nonterminals are found on the right hand side). Actually, it is always possible to find an equivalent (in the sense that it generates the same language) grammar in Greibach two form — that is, with all productions of one of the forms

$$\text{(i)} \quad X_i \rightarrow a_j X_k X_l$$
$$\text{(ii)} \quad X_i \rightarrow a_j X_k$$
$$\text{(iii)} \quad X_i \rightarrow a_j$$

and possibly

$$\text{(iv)} \quad X_i \rightarrow \Lambda$$

A recursive descent algorithm will then work without getting into a loop. It is also possible to write a simple non-recursive algorithm using a stack (which basically does what the computer would do in implementing the recursion, but does not generally have to push down as much information into the stack, and is therefore marginally more efficient).

This method of parsing, using a stack to keep the information needed to do the recursion, was used early in the history of computational linguistics, by Kuno and Oettinger [1962], and is called **predictive parsing**. It operates smoothly and efficiently in many naturally occurring cases, especially if the strings do not become too long. An informal description of predictive parsing is as follows:

> **1)** The algorithm is initialized by placing S* in the pushdown store and scanning the leftmost terminal symbol of the input string.
>
> **2)** Whenever a character a_j is under scan and X_l is on top of the stack, pick one of the productions with X_l as left hand side and a_j as leftmost character on the right hand side, pop up X_l, and push X_l followed by X_k (so that X_k will be on top), X_k alone, or nothing, depending on whether the production is of form (i), (ii) or (iii) above, respectively, and move on to scan the next character to the right in the input string. (In the case of a Λ right hand side, X_l can be popped up without moving on to scan another character.)
>
> **3)** Accept the string if and only if the end of the string is encountered precisely at the same time that the stack becomes empty. Otherwise, the algorithm fails.

Notice that the formulation uses the "nondeterministic" phrasing "pick one of ...", and the notion of the algorithm "failing" in certain cases where it is still not clear that no parse exists. This means that if the algorithm makes a mistake (picks a production that does not lead to a successful parse), then it can backtrack on failure and make another choice — until no more productions remain to be picked. Any such recognition scheme must backtrack anyway to try all alternatives if *all*, rather than merely *one*, of the parses are to be found. The nondeterminism inherent in the predictive algorithm means that the algorithm will require another stack (for the backtracking) and will tend

*When S is used as a nonterminal symbol in a grammar or as a stack symbol, it will always be used to denote the axiom, or root symbol, of the grammar, as is conventional.

to take exponential time in "bad cases". Some of the pattern matching cases are bad enough that the combinatorial explosion of possibilities slows the process appreciably. The algorithm discussed below (Earley's Algorithm) tends to be faster in these "bad" cases.

As an example of predictive parsing, consider the grammar of §3.1.1, converted to Greibach two form to obtain:

$S \rightarrow aTX$
$S \rightarrow cTX$
$S \rightarrow eV$
$V \rightarrow d$
$V \rightarrow dV$
$X \rightarrow cV$
$T \rightarrow e$

The following table shows the actions on the stack that would result from reading a given symbol in the input for each possible symbol on top of the stack:

Action table for the grammar

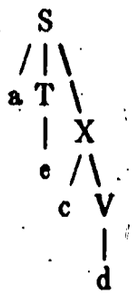
Symbol on Top of Stack	*Next Input Symbol*	*Action*
S	a	pop S, push X, push T
S	e	pop S, push V
S	c	pop S, push X, push T
V	d	pop V
V	d	pop V, push V
X	c	pop X, push V
T	e	pop T

On the input string **aecd**, the algorithm's behavior is as shown in the following table:

Actions on the string "aecd"

Stack	*String to be Read*	*New Stack*
S	.aecd	T X
T X	.ecd	X
X	.cd	V
V	.d	

Now let us consider the generation of a structural description. We start at the top, with an S. Each time that a symbol X_i is popped up and replaced by a right hand side q, we can expand the parse tree portion X_i to $X_i[q]$, always expanding in a leftmost fashion. The parse tree for our example is:



3.1.3. Earley's Algorithm

Earley's parsing algorithm is mentioned in both the Kreuter and Wauchope papers. One of Kreuter's tasks was to implement Earley's algorithm in Ada as a method of pattern matching. The original form of this algorithm is due to Jay Earley [1970], and the form used by Kreuter is based on a modification due to Graham, Harrison and Ruzzo [1976] (see also [Harrison, 1978]). Earley's algorithm is as efficient as any "practical" general context free parsing algorithm known (there are some theoretical results that are marginally more efficient, including [Valiant, 1975]) and does not require that the grammar be converted to any special form. The efficiency of Earley's algorithm is achieved by carrying around possible analyses in parallel, rather than backtracking, as in predictive parsing. The analyses cannot actually be done in parallel, of course, on the serial machines that are standard today; but the algorithm eliminates the repeated generation of information on partial parses that is inherent in the usual backtracking method.*

We will now give an informal description of the modified Earley algorithm as a *recognizer* (the recovery of the parse tree will be discussed later), based on the description of Earley [1970], with modifications:

> The algorithm scans an input string $a_1 \ldots a_n$ from left to right. As each symbol a_i is scanned, a set of "states" S_i is constructed which represents the condition of the recognition process at that point in the scan. *In the modified algorithm, each state set S_i is represented as the column of an array.* Each state in the set represents (1) a production such that the algorithm is currently scanning a portion of the input string which is derived from its right hand side (the portion to the right of the arrow), (2) a *point-of-scan marker* (dot; also called a *cursor*) in that production which shows how much of the production's right hand side has been recognized so far. *In Earley's original formulation, a pointer was also kept to the position in the input string at which the algorithm began to look for an instance of that particular production. This is not necessary when using the array format.*
>
> The algorithm continues as long as any one of three operations is applicable to a production in the array. The operations are mutually exclusive. The **predictor** operation is applicable to a state when there is a nonterminal immediately to the right of the dot. Its effect is to add one new state to S_i for each alternative of that nonterminal. The

*Earley's algorithm is an example of a more general method of reducing exponential processes to polynomial processes. This topic will not be explored here, as it is beyond the scope of this work, and, in fact, has not been systematically developed in the literature. For a discussion of some relevant considerations, see [Tucci, *forthcoming*] and [Pereira and Warren, 1983]. The latter reference also points out that Earley's algorithm is actually a particular case of chart parsing, systematized in [Kay, 1980].

Its effect is to add one new state to S_i for each alternative of that nonterminal. The point-of-scan dot is placed at the beginning of the right hand side of each production added by the predictor, since none of its symbols has yet been scanned. *In the modified algorithm, these are always placed in the array at position (i,i).* Thus the predictor adds to the array at (i,i) all productions which might generate substrings beginning at a_i (but only adds one copy of any production, thus avoiding the danger of infinite looping inherent in recursive descent).

The **scanner** is applicable just in case there is a terminal to the right of the dot in some production in column i. The scanner compares that symbol with a_i, and if they match, it adds the production to column i+1, *in the same row as the original production in column i*, with the dot moved over one position in the production to indicate that that terminal symbol has been scanned. After the scanner is applied to all productions in a column to which it is applicable, the algorithm moves on to the next column.

The third operation, the **completer**, is applicable to a production if its dot is at the end of its right hand side. If the left hand side of the production is "P" and the production is in row i, then the completer adds all productions from *column i* which have P directly to the right of the dot, moving the dot one place to the right (*i.e.* over P). Intuitively, column i is the state set the algorithm was in when it predicted the possibility of the production just completed (the one with left hand side P). Now that P has been successfully found, the completer goes back to all the states in S_i which caused the algorithm to look for a P, and moves the dot over the P in these states to show that it has been successfully scanned.

In the case of rules with Λ right hand sides, some further modifications to each of the processes mentioned need to be made. These will not be detailed here, but may be found in any of the references mentioned above.

Example:

The algorithm described above, using the grammar

$$S \rightarrow aTc$$
$$S \rightarrow SU$$
$$S \rightarrow T$$
$$T \rightarrow e$$
$$U \rightarrow d$$

to recognize the input string

aecd

produces the array below.

Unread input (. indicates point of scan)				
.aecd	.ecd	.ed	.d	.
(1,1) S → .aTc S → .SU S → .T T → .e	(1,2) S → a.Tc	(1,3) S → aT.c	(1,4) S → aTc. S → S.U	(1,5) S → SU.
	(2,2) T → .e	(2,3) T → e.	(2,4)	(2,5)
		(3,3)	(3,4)	(3,5)
			(4,4) U → .d	(4,5) U → d.

Commentary on the Example:

The Array

Since the input string is of length 4, the array will have 5 columns. These are labeled in the example with the input string, with a point-of-scan marker to show how much of the string has been read prior to entering any productions into that column.

Column 1:

The array is initialized by the entry into (1,1) of the productions **S → .aTc**, **S → .SU**, **S → .T**. The predictor then causes the production **T → .e** to be added to (1,1). Since the predictor and completer are not applicable, the scanner is invoked, causing entries to be placed in column 2.

Column 2:

The scanner looks at the next character and finds it to be **a**. Since the only production in column 1 with an **a** at the point of scan is **S → .aTc**, in row 1, the production **S → a.Tc** is entered into (1,2). The predictor is now applicable, causing **T → .e** to be added to (2,2).

Column 3:

The scanner first operates on the production **T → .e**, causing **T → e.** to be entered into (2,3). This latter causes the completer to be invoked. Since it occurs in row 2, the completer searches column 2 and finds one production, **S → a.Tc**, with a T immediately following the point-of-scan marker. This production is therefore moved horizontally to column 3, entered into (1,3) as **S → aT.c**

Column 4:

The scanner now finds a **c** at the point of scan and moves **S → aT.c** into this column as **S → aTc.**. The completer then finds **S → .SU** in column 1 and moves it to (1,4) as **S → S.U**. This causes the predictor to enter **U → .d** into (4,4).

Column 5:

The scanner is used again, producing the production **U → d.** in (4,5). This causes the completer to search column 4 for a production with U at the point of scan, and it finds **S → S.U** and moves it across to (1,5) as **S → SU.** The fact that there is a production in (1,5) with an **S** as left hand side and the fact that the string is now indicated together indicate that a successful parse has been found.

To recover the parse, it merely needs to be noted that the portion of the right hand side of a production in the array to the left of the point of scan has successfully been matched to a portion of the string being parsed that is given by the coordinates of its location in the array. If the production is in the array at (i,j), then that portion of the right hand side has been matched to positions i through j-1 (notice that this means that productions added by the predictor, which have *nothing* to the left of the dot and are at (i,i) have been matched to *nothing*, as we would desire). Completed productions (the ones with the dot at the *end* of the right hand side) constitute a successful parse of a portion of the string, which, in a parse tree, would be dominated by the left hand side of the production. In our example, for instance, **S → SU.** is found in (1,5), so it constitutes a successful parse of positions 1 through 4, that is, of the whole string, while **U → d.** parses only positions 4 through 4 — only the fourth character. The production **S → aTC.** parses positions 1 through 3, while **T → e.** parses position 2. The single characters **a**, **e**, **c**, and **d** cover positions 1,2,3, and 4, respectively, of course.

An algorithm to recover the parse must start with the (1,n) position, where n-1 is the length of the string, then check (1,k) and (k,n) for each k = 1 ... n. For each one found, a recursive call will check in the same manner until everything is reduced to single symbols. The details can be found in [Harrison, 1978] (though the reader should be aware that he numbers his array from zero, rather than one).

3.2. USING EARLEY'S ALGORITHM TO MATCH PATTERNS

Once the parse array has been formed, all parses can be found in time proportional to n^2, where n is the length of the input string, using the algorithm mentioned in the last section. Formation of the array itself takes, in the worst case, time proportional to n^3 because the completer operation potentially has to examine a whole column of the array, which takes some multiple of n operations, for each of some multiple of n^2 entries (see [Harrison, 1978] for a detailed analysis). Storage for a parse array is proportional to n^2 (since it is two-dimensional and each dimension is proportional to n) but can be large if the grammar is large. However, once the results of the pattern match are no longer needed, the storage can be reclaimed. There is also the possibility of storing the array as a list if it is sparse. These factors need to be investigated, as Kreuter is continuing to do (see the second paper of this collection).

In order to understand the emphasis of Kreuter's paper, let us consider briefly how the pattern primitives of SNOBOL4 (§2.2), as used in Post-X (§2.3) would be treated in an appropriately modified Earley's algorithm.

Concatenation and alternation and the grouping thereof by parentheses are reflected in the composition of the grammar rules. Thus

P = Q | ('abc' POS(5)) LEN(12) REM

(where Q is another pattern) would become the grammar

P → Q
P → R LEN(12) REM
R → 'abc' POS(5)

Assignment operators are not used, since the branches of the tree according to the grammar are used in the Post-X action statements. Notice that the parentheses used to group the elements of the pattern have affected the grammar produced, and will

therefore affect the branches of the tree produced upon a successful match (parse) and the selectors used to refer to matched substrings in the action portion of the form (see §2.3).

The pattern primitives can be dealt with when the scanner or predictor are used. They are treated as follows:

POS(i). If POS(i) is found at the point of scan, then the production **POS(i) → A.** can be entered by the predictor in that column if and only if it is column i+1 (indicating that i characters have been scanned).

RPOS(i). Treated similarly, with the count being from the end of the string (this will mean that the string length needs to be obtained before the parse, which is advisable for purposes of efficient storage allocation for the array anyway).

ARB. If ARB is found at the point of scan in column i, then the production **ARB → A.** is entered into position (i,i) in the array, **ARB → s_i.** is entered into (i,i+1) where s_i is the i-th character in the string being matched, etc. to the end of the string. (It is also possible to deal with ARB on a column-by-column basis by entering only the production mentioned in (i,i) and the production **ARB → s_i .ARB** This treatment will produce a right-branching parse tree which can be modified into the desired tree by post-processing.)

REM. When REM is found at the point of scan in column i, then the production **REM → X.** can be entered into (i,n), where X is the remainder of the string and n is the last column (*i.e.* X is $s_i \ldots s_n$).

BREAK(x) is treated like ARB, except that it is necessary to check for the break characters and enter the productions accordingly. That is, **BREAK(x) → y.** is entered into the array only if y does not contain any occurrence of a character in x.

SPAN(x) is treated like BREAK, except that the string y in the description must contain *only* characters found in the string x.

ANY(x) matches only single characters. Thus the production **ANY(x) → y.** is entered only into (i,i+1) — and only if the i-th character is in x.

NOTANY(x) is treated analogously to ANY except that the i-th character must *not* be in x.

LEN(n) is treated like ARB, except that its production is only entered at (i,i+n).

BAL can be done analogously to ARB. If productions are to be entered into (i,i), (i,i+1),... up to the end of the string, then it will be necessary to check for balance in the strings before entering the appropriate productions. Alternatively, the productions **BAL → A. BAL → .'(' BAL ')'** and **BAL → .BAL BAL** can be entered at (i,i), and the parse obtained can be postprocessed to obtain the appropriate tree structure.

Other pattern variables are dealt with analogously. The treatment of NULL and literal strings should be obvious (just the usual treatment in Earley's algorithm).

3.3. ALTERNATIVE PATTERN MATCHING ALGORITHMS

There are various fast string matching algorithms available, but these were not considered in the research because of the requirement for returning a structural description, in order to enable Post-X-like processing. For some specific string matching algorithms and references to the literature, see [Liu and Fleck, 1979]. The reader should be aware that the SNOBOL4 patterns are more powerful than, for instance, regular expressions, which certain algorithms, such as those employed in text editors, match rather

quickly.

4. EXTENDING THE PATTERN-DIRECTED PARADIGM

Given the fact that so much of AI and even CL consists of manipulation of lists and trees, rather than strings, the fact that SNOBOL4 was not considered a list processing language was undoubtedly one reason that it did not become a major AI programming language, despite its excellent pattern-directed string processing facilities (and, not incidentally, free distribution by Bell Laboratories). SNOBOL4 does allow the definition of datatypes, which are basically list structures, but the language for using these is not pattern-directed, and the trace and dump facilities are not developed sufficiently to make them easy to use. (Other reasons for SNOBOL4's failure to capture the "AI market" have to do with control structures [Bailes and Reeker, 1980a]).

The Post-X language (see §2.3), while originally designed for string processing, sought to provide pattern matching on trees in a manner that would be analogous to the string pattern matching facilities of the language. The method of doing this was to extend the use of the SNOBOL4 pattern BAL (see §2.2.2) to allow the specification of the value of the structure within a balanced set of parentheses designating a tree. In addition, some tree functions were added for use within the action portion of the forms dealing with trees (see [Bailes and Reeker, 1980b]). The attempt was only partially successful. Though the specifications were easy to write and easy to read in some cases, they were confusing in others, partially because of confusion among labels within trees and data items on the leaves of trees. Wauchope seeks to remedy this deficiency in the work reported in the third paper of this report, and in continuing work. As Winston and Horn [1981] have said, "Building in these capabilities can be hard. The literature offers little guidance."

5. CONCLUSION

The following papers address in a tentative way two important issues in the provision of pattern-directed string and list processing in Ada, Kreuter's paper deals with alternative algorithms for string pattern matching which will also return parses or structural descriptions of the string, where structural indicators are built into the patterns, in the manner of Post-X. The matching process for such general patterns is time-consuming, so efficiency will be an important consideration. Wauchope's paper makes a further extension of the pattern-directed paradigm — to arbitrary LISP-type data structures. This work should lead to a useful alternative language for artificial intelligence, using Ada, and is being continued.

In consideraing future applications of artificial intelligence, it is important to realize that game playing, language processing, expert systems, and the other sorts of things that we conventionally think of under the umbrella of AI are going to be combined with simulations, numerical programs, large file processing applications, and the like. For these "conglomerate" applications, the languages that have most commonly been used in AI research may not be the most useful. In our view, Ada can provide an excellent environment for artificial intelligence applications of the future because of its flexibility and generality. The problems addressed in this report — provision of appropriate facilities through packages and making those facilities efficient enough that large and complex programs will be feasible within them — are ones that need to be addressed if this potential is to be realized.

PATTERN MATCHING ALGORITHMS IN ADA

John Kreuter
Tulane University

Abstract

To test the usefulness of Ada in artificial intelligence programming, it is desired to develop Ada packages which mirror the capabilities of languages such as LISP or SNOBOL4 which have proven their utility in artificial intelligence. The development of SNOBOL4-like pattern matching packages was undertaken by this author. Viewing pattern matching as an extended parsing problem, Ada packages for pattern matching utilising Earley's efficient parsing algorithm were developed, as well as packages implementing the more traditional backtracking (recursive descent) approach. Since a full Ada implementation was not available at the time, these efforts should be considered as preliminary, but indicate a direction for further research.

1. INTRODUCTION: RESEARCH OBJECTIVES

This project is part of an overall effort to add useful artificial intelligence programming tools to Ada. One such tool is pattern-directed string processing, of the sort available in the language Post-X [Bailes and Reeker, 1980a,b]. This involves the implementation of pattern matching algorithms in Ada which actually return a parse tree of the match of a pattern according to a structured pattern. In other words, pattern matching according to a context-free grammar with primitives like those of SNOBOL4 [Griswold *et al*, 1971] is the goal.

Parsing can be an expensive operation, timewise, to use over and over as a basis of a programming language — especially when the patterns are as general as those in SNOBOL4. The purpose of the research reported here was to consider the particular parsing algorithms that can be implemented using Ada packages. Because a number of considerations are involved, including the basic efficiency of the algorithm and the efficiency of its implementation in the Ada environment, it is advisable to do this in an experimental manner, implementing and testing.

2. DEFINITION OF PATTERNS

Before any pattern matching algorithm could be implemented, a suitable definition — one easily represented in Ada — had to be developed. The packaging facilities of Ada would then allow the pattern representation and pattern building functions to be developed and compiled independently from the pattern matching routines. The packaging facilities allowed by the available compiler are at present incomplete (see the recommendations in §4 of this paper for a discussion of the shortcomings of the current version of the compiler used in this work) but it was possible to demonstrate within them a measure of encapsulation. A fully validated Ada (one that implements the full definition of the language) will enable more extensive use of the package to build hierarchies of libraries of packages, with each library at a given level containing packages useful to the applications at the next higher level. Thus at the bottom level the libraries would contain packages of generally useful abstract data types such as stacks, queues, linked lists, sequences, strings (a more complex variety of string than that built into Ada), matrices, etc., defined in terms of the built-in types provided by Ada. At the next higher level would be packages that could use these lower level defined types. For instance, the pattern matching packages would be defined at this level. At the next

higher level would be packages utilizing complex structures such as patterns — for instance, a compiler could be defined at this level, using the pattern matching algorithms to do the parsing.

The first working definition of a pattern was:

—A **PATTERN** is an (unconstrained) array of **ALTERNATIVE**s.

—An **ALTERNATIVE** is an (unconstrained) array of **BEAD**s. (Bead is used here to correspond to SNOBOL4 terminology [Griswold *et al*, 1971].)

—A **BEAD** is any of
(i) a string;
(ii) a **PATTERN**;
(iii) a primitive function (primitive functions selected corresponded to the most useful SNOBOL4 primitives.)

The available compiler, though it supports unconstrained arrays, does not support size-variant records, so the utility of using unconstrained arrays in a package is limited. The next working definition for a pattern therefore made an Alternative a linked list of Beads. Unfortunately, without generic packages, a linked list could not be conveniently defined outside the pattern package. Thus, although the type Alternative was implemented as a linked list, a Linked List type was never explicitly defined. The working definition for a Pattern then became, in Ada

```
type prim-func is (ARB1, REMAIN1, POS1, SPAN1, ANY1,
                   NOTANY1, BREAK1, TAB1);
type Primitive is record
     Name : prim-func;
     Arg  : string-pointer;
end record;

type Pattern;
type Kinds is (terminal, non-terminal, operation,
               R, L);
-- R and L are used to hold the left and right
-- unmatched substrings

type Bead(Kind : Kinds) is record
    case Kind is
       when non-terminal => Choice : Pattern;
       when terminal => Str  : string-pointer;
       when operation => Op  : Primitive;
       when R => null;
       when L => null;
    end case;
end record;

type alt-pointer;
type Alternates is record
    C    : Bead;
```

```
        next : alt-pointer;
    end record;
    type alt-pointer is access Alternates;

    type pats is array(Positive range <>) of Alternates;
    type Pattern is access pats;
```

Finally a type Sys-Pat was introduced, so that the pattern building functions could distinguish between internally generated subpatterns and actual user-defined patterns. For example, the pattern

$$P = (\text{"a"} + \text{"b"} + \text{"c"})$$

should have a length of 3, but the pattern

$$P = (A + \text{"c"}),$$

where

$$A = (\text{"a"} + \text{"b"}),$$

should have a length of 2. By overloading the pattern building functions, the compiler is forced to choose the proper representation in both cases.

The problem of pattern matching is given a pattern and a target string, to find a substring such that for each set of alternatives a bead can be found which matches the substring starting at the point where the last set of alternatives leaves off. In some schemes the substring may start either flush left, flush right, or anywhere within the target string, depending on a positional indicator passed to the pattern matching algorithm along with the pattern and the target string. Since the patterns used here include the ARB primitive function (which matches any arbitrarily long string of characters) positional indicators have been left out of this initial work. All patterns are matched flush left. However, provision has been made to include positional indicators in future versions.

3. SOME ALGORITHMS FOR PATTERN MATCHING

3.1. BACKTRACKING: RECURSIVE DESCENT

The most intuitive approach to the pattern matching problem is to try every possibility for each pattern element individually. This leads to the "backtracking" method. This method starts by trying each bead, for any given set of alternatives, until a match is found. Then for the next set of alternatives each bead is tried, etc., until all sets of alternatives have been matched. If for any set of alternatives no bead matches, then the algorithm backtracks — that is, the previous set of alternatives is tried again, starting from the bead that just matched. Clearly every possible parse of the string will be found in this fashion, but there are several problems which arise with this method which will be discussed later.

A typical way of implementing the backtracking method, and the way that I chose, is the so called recursive descent parsing algorithm*. As the name implies, recursion is used extensively by this method, especially if the bead being matched is itself a pattern.

[*See §3.1.1 of the first paper in this report. -ed.]

(Recall that a bead may be either a string, a pattern, or a primitive function.) In this case the recursive descent algorithm calls itself (*i.e.* recursion), passing the value of the bead (*i.e.* the pattern) and the cursor position of the string. The positional indicator, if used, would be flush left. In this fashion the algorithm "descends" with recursive calls until the bead being matched is not a pattern. At this point, if the bead is a string it is matched against the target. If the bead is a primitive function, then a string is derived from the function, the target string and the cursor position. This derived string is then matched against the target. If this matching is successful the next set of alternatives at this level of descent is tried. (This may be done by recursion also, by iteration, using a stack, or by coroutines.) After each set of alternatives is matched (or if no match is found), the algorithm returns to the next higher level with the matched substrings (or the null string if no match is found).

3.2. COROUTINE IMPLEMENTATION OF BACKTRACKING

An elegant way of implementing the backtracking aspect of the algorithm — that is, when no match is found, returning to a previous set of alternatives and resuming where the algorithm left off — is through the use of coroutines, which in Ada are *tasks*.* The task starts by examining each bead in the first set of alternatives. For each bead that successfully matches, a new task is started, which examines the remainder of the string and the remaining sets of alternatives. When the last set of alternatives has been examined, the task passes back the matching substrings (or the null string if no match has occurred) and terminates. The parent task then adds its substring to the beginning of each tree on the list which has been passed to it. This new list of trees is then passed back, and the task terminates, etc., until the topmost task completes all possible parse trees. Thus although backtracking takes place (each possibility is considered individually) it occurs with a degree of concurrency dependent on the run-time environment.

Unfortunately, once again the available compiler does not have tasking as one of its features. The process described above can be implemented as a function, but with the loss of concurrency and elegance. Furthermore, as shown in the analysis below, backtracking can be a very costly way of conducting pattern matching. Some of this cost can potentially be absorbed by concurrency, where the system allows, but the implementation and run-time analysis of this must await a validated Ada (so that concurrent tasks can be incorporated into the algorithm). The run-time analysis could then consider both time and resource utilization. As multiprocessors appear this analysis could provide some interesting insights into time consumption versus resource demands.

Two noteworthy problems exist with the pattern matching method outlined above. The first occurs if the pattern itself is left recursive — that is, it has the form $P = P' + A$, where P' is a pattern which can produce P, and A is any pattern (possibly null). The recursive descent algorithm will examine P by first considering its first set of alternatives, *i.e.* P'. This will cause a recursive call, so that P' is considered. Since P' can produce P, eventually the algorithm will recursively consider P, which then causes a recursive call to P', eventually leading to another call to P, etc. without ever having advanced the target string cursor. Thus the recursive descent method goes into an infinite loop if it encounters a left recursive pattern. Fortunately this is not a major problem since it has been shown that any pattern can be generated by a pattern in

*[See §1.2 of the first paper in this report. -ed.]

Greibach Normal Form, which means the pattern has one of the following forms:

$$P = a + P1 + P2 + ... + Pn$$

$$P = a$$

$$P = \text{null}$$

where a is any string, and the P's are all patterns. Clearly a pattern in Greibach Normal Form cannot be left recursive, so if any given pattern is first modified into this form, the recursive descent algorithm will work.* Although this problem of handling left recursion is not major since any pattern can be transformed into Greibach Normal Form, the problem of the time requirements of the backtracking algorithm in the most general cases of patterns remains. Consider as an example the pattern P = ("a" + P + P) or null. Suppose this pattern is to be matched against a string of a's. Clearly the leftmost a will be matched by the "a" of P. All other a's can be matched by either the first (recursive) occurrence of P, or by the second, independent of how any previous or subsequent a's are matched. Thus if the string is L long, combinatorics tells us that the number of possible parses is $1 + 2^L$ — that is, the number of parses is exponential. Since the backtracking algorithm considers each possible parse individually, it will require exponential time to parse such a pattern. So, although the backtracking method may be useful given certain restrictions on the allowable patterns, in the most general case the time constraints become burdensome.

3.3. USE OF EARLEY'S ALGORITHM

Obviously, the way to reduce the time costs of parsing is to not treat each individual possibility by itself, rather to group them into classes. In the above example, for instance, there is no need to consider both of the recursive P's individually since they both reduce to the same tree. Both P's can be considered in parallel, and then the above example will parse in linear time! Even with more complex examples, it can be shown that by developing a scheme to consider similar possibilities in parallel, parsing can be accomplished in polynomial time, a vast improvement over the exponential time required by the backtracking method. One such scheme, which can be implemented without any initial manipulation of the pattern is known as Earley's algorithm.

Earley's algorithm is described in its mathematical details in [Harrison, 1978] where a modified (improved) version is called simply "a good practical algorithm". The main problem I encountered in implementing this algorithm was developing reasonable data structures to represent the rather complex mathematical formulas introduced — patterns must be converted to "dotted rules", a triangular matrix of dotted rules must be created, and the functions "×", "*", and "predict" must be implemented.† Once again

*It should be mentioned here that it is not common for patterns to call themselves recursively. Recursive patterns are, however, a possibility that one might not want to exclude, and are very handy in some instances. In SNOBOL4, they are implemented through the use of "unevaluated expressions", and heuristics are used to prevent infinite loops of recursive calls (which, in implementation, would tend to cause a stack overflow). A good discussion of the use of unevaluated expressions in SNOBOL4 can be found in [Griswold, 1975]; the heuristic mentioned is also discussed in [Griswold, 1984]. In Prolog, there are also problems with left recursion. These can be solved either by automatic transformation of left-recursive clauses or by checking for the occurrence of particular states (see [Enalls *et al*, 1984]). -ed.]

†The "×" operation is used to implement Earley's "scanner" (see §3.1.3 in the first paper of this report),

the effort was hampered by the lack of generic packaging facilities in the currently-available compiler. The development process certainly could have made good use of a lower level of abstract data types which included sets and matrices. As it was, these structures had to be developed concurrently with the rest of the algorithm.

Another problem encountered in implementing Harrison's version of Earley's algorithm is that this version (as most are) is developed for a context-free grammar, not for pattern matching. Although for the most part context-free parsing is analogous to pattern matching, the analogy breaks down when the primitive functions are considered. These primitive functions are in general string and cursor dependent, and so have no context free representation. Since they are at most dependent on the string and cursor, though, it was possible to alter the "predict" function so as to produce simple string derivations of each primitive function as it is encountered during the parse. This increases the time requirements as compared to a simple context-free parsing problem, but the modified algorithm still requires no more than polynomial time.*

As can readily be seen from the above discussion, although Earley's algorithm is faster than the backtracking method, the price is paid in the complexity of algorithm and the space it takes while running. The complexity also may make it more difficult to develop Earley's algorithm to take advantage of a concurrent environment. Now that the algorithms have been developed into working programs, it remains to be studied whether the difficulties of Earley's algorithm outweigh its benefits.

4. CONCLUSIONS

Two working programs in Ada have been produced, utilizing two different methods for pattern matching, but clearly, more work remains to be done. To provide maximum utility to future users and researchers, the programs developed should be rewritten in a fully validated Ada, making use of the packaging facilities as detailed in the DoD specifications for the language. In the specific case of the backtracking algorithm for pattern matching, the rewrite should also include the use of tasks. In this fuller Ada environment, the two methods of pattern matching could better be tested against each other in real time, to provide a comparison of their relative merits in time, space and concurrency.

Two other suggestions regarding Ada have arisen from these efforts: First, Ada makes no provision for treating functions as data types. Such a treatment is especially useful in pattern matching, where it is desirable to associate an action to be taken with a pattern to be matched, as in Post-X. Second, when producing large systems as is often the case in AI programming it would be beneficial to be able to declare subprograms within a package to be external, so as to be able to compile them separately from the rest of the package. Although the separate compilation of the packages themselves is very useful, in complex systems the package itself may grow to a cumbersome point, with each update requiring inordinate amounts of compile time. Facilities for external compilation help to relieve this load.

In this paper, we have discussed methods of implementing within Ada the central facilities for pattern-directed programming with character strings (which could be

---

and both the "X" and "*" operations are used to implement Earley's "completer" in Harrison's version of the algorithm. -ed.]

[See §3.2 of the first paper in this report for a discussion of the modifications needed. -ed.]

extended to other datatypes as suggested in [Wauchope, 1984]). Work is continuing in comparing these methods and testing their efficiency.

PATTERN-DIRECTED LIST PROCESSING IN ADA

Kenneth Wauchope*
Tulane University

Abstract

A pattern-directed list processing facility for the Ada programming language is presented. Pattern lists for matching against source lists are constructed from a set of SNOBOL4-derived primitives which have been extended to be applicable to arbitrarily complex LISP-like data structures. Patterns may also contain user-defined symbols, which can serve as nonterminal symbols of a context-free grammar. Basic list creation and manipulation are made available to the programmer via a package of LISP-like functions and data types. Several examples of possible applications in Artificial Intelligence are explored–focusing on computational linguistics problems such as transformational grammar and parsing–demonstrating the construction of patterns and the use of various operations available for testing and manipulating the values which the matcher returns.

1. INTRODUCTION

The Ada language, with its goal of being the exclusive high level programming system used in the Department of Defense, includes data abstraction facilities that in effect make it possible to extend the language by creating new data types and defining the operators that are to act upon them. For specialized areas of application, a programmer can invoke the appropriate data abstraction (package) and proceed to write code using the new high-level constructs it provides, just as if using a new language specifically designed for that problem domain. One application area of potential interest to Ada users is artificial intelligence–including the field of computational linguistics, which offers such possibilities as natural language interfaces with computers, text understanding and/or information retrieval, natural language programming, and machine translation. Programming tasks in this category are usually undertaken using specialized string- or list-processing languages such as SNOBOL4 or LISP, and extending Ada's ability to process data structures of this sort would greatly facilitate the development of language processing and other AI-related systems in that programming environment.

Pattern matching is a computational paradigm that is particularly appropriate to language processing applications, as a language can generally be described in terms of a series of syntactic patterns and subsequent pattern-directed semantic mappings called a grammar. If an input sequence of terminal symbols successfully matches the grammar's set of patterns (rewrite rules), then it is a legitimate sentence in the language and appropriate further actions (creation of a parse tree, or mapping to a deep structure or meaning representation) can be carried out. Pattern matching can also be used to drive the process in the opposite direction, such as matching certain deep structure kernels and then performing appropriate grammatical transformations on them to yield new surface structure sentences. Many other AI applications also employ production rules that are fired by a successful matching of symbolic state conditions, and so pattern matching can be used to perform such tasks as formula unification, symbolic differential and integral calculus, and similar problems involving sequences of symbols that are to be analyzed for content and structure.

---

*Current address, Navy Center for Applied Research in Artificial Intelligence, Code 7510, Naval Research Laboratory, Washington, D.C. 20375.

The objective of this research was to design and implement a pattern-directed list processing package in Ada to test the hypothesis that such a package would provide a practical and useful facility for artificial intelligence programming. A package of list datatypes and list-manipulation functions were created. List patterns could then be constructed out of SNOBOL-like pattern matching primitives, and the matcher was to return a list of short-term variables corresponding to the values matched (perhaps only partially) by the pattern components. These values should then be capable of being tested, concatenated, or subjected to further matching. Once the matcher was operational, it would be tested in various areas of application (concentrating upon computational linguistics problems) to determine the usefulness of the various matching primitives and the operations upon the returned values.

2. DESIGN OF THE PACKAGE

2.1. LIST PROCESSING

The most widely used AI programming language in the United States is LISP, which represents sequences of symbols (atoms) as binary linked lists. The primary list manipulators are CAR and CDR, which return the first element and remainder of a list, respectively, and CONS, which creates a new list out of a pair of elements (themselves either atoms or lists). Various predicates are also available to test the identity of data items, and more powerful list manipulation functions can be built up out of these simpler ones.

Trees are a natural way of representing the structural composition of sentences in a language, and binary lists can be made to accommodate these structures quite easily. For example, a parse tree for "he is in the garden" can be represented by the binary list

(S(NP(Pro(he))VP(V(is)PP(P(in)NP(Det(the)N(garden))))))),

where the constituents of each phrase marker are to be found as a sublist immediately following it.

The initial task toward creating a list pattern matcher in Ada was to provide means for the creation and manipulation of atoms and lists. This was accomplished by defining "S-Expression" as an abstract data type, with its internal structure (either a list node having left and right child pointers, or atom node having a name field, value field and next pointer) hidden from view so that only the LISP functions exported from the package could be used to operate upon values of the type. S-Expression objects are created by a function "Quote" which converts Ada strings (representing properly balanced S-expressions) into linked-list structures; the function bears little resemblance to the LISP Quote (which suppresses evaluation) since no LISP interpreter is actually involved, but the name was borrowed because of its analogous function. The most useful core LISP functions and predicates, as well as several higher-level ones (such as Member and Append), constitute the remainder of the operations available on the abstract type.

2.2. LIST PATTERN MATCHING

Since patterns would be constructed by the user in the same form as the source lists (i.e. parenthesized strings of symbols), it was decided to convert the patterns themselves into lists (using Quote) and then perform the matching by stepping through each list and mapping corresponding elements onto each other. The matching itself is thus a

list-traversal process and so was implemented in Ada in much the same way as a pattern matcher would be written in LISP itself, using recursive procedures written in the Ada pseudo-LISP.

To adapt SNOBOL-like string matching primitives to the job of list matching, two versions of each primitive were defined: the first class matching list components (i.e. the CARs of each node in a particular sublist, which can be either atoms or lists), and the second matching individual atoms at arbitrary depths of nesting in the tree. Once the pattern matcher became operational, it would then be possible to determine what use might be made of the two classes of primitives in actual applications. The primitives implemented are listed below.

---------- **Class I: List-Component Primitives** ----------

LIT(e): Matches if the next list element of the source is equal to the element e (atom or list). Examples: LIT(hello), LIT(((hi)there)).

LEN(n): Matches a series of n list elements (atoms or lists). Example: LEN(5).

BAL: Matches an arbitrary number of list elements.

ANY(s): Matches if the next list element of the source is a member of the sequence of elements s. Examples: ANY(boy cat dog), ANY(atom1 (list 2) ((list)3)).

NOTANY(s): Matches if the next list element of the source is not a member of the sequence of elements s. Examples: NOTANY(bad worse), NOTANY((real bad)(even(worse))).

BREAK(s): Matches all list elements until one is encountered that is a member of the sequence of elements s. Examples: BREAK(stop), BREAK((go(no)further)).

SPAN(s): Matches list elements until one is encountered that is not a member of the sequence of elements s. Examples: SPAN(ok good), SPAN(yes (fine)).

---------- **Class II: Leaf (Atom) Primitives** ----------

LITL(a): Matches if the next atom in the source is the atom "a". Example: LITL(hello).

LENL(n): Matches the next n atoms in the source. Example: LENL(5).

ARB: Matches an arbitrary number of atoms (possibly none).

ANYL(s): Matches if the next atom in the source is a member of the sequence of atoms s. Example: ANYL(one two three).

NOTANYL(s): Matches if the next atom in the source is not a member of the sequence of atoms s. Example: NOTANYL(bad no).

BREAKL(s): Matches all atoms until one is encountered that is a member of the sequence of atoms s. Example: BREAKL(stop).

SPANL(s): Matches atoms until one is encountered that is not a member of the sequence of atoms s. Example: SPANL(go fine great).

----------**Additional Operators**----------

REM: Matches the entire remainder of the list (possibly empty).

ALT: Attempts to match the first pattern in its argument list followed by the remainder of the original pattern. If the match fails, it tries the next argument, and so on. Example: ALT((SPAN(a)) (SPAN(a b))).

When matching a Class I primitive in a pattern, the matcher steps down the spine of the corresponding source sublist and matches the elements found hanging off of it. If the pattern branches off to the left, the matcher recurses on the CAR of each list and then returns (if successful) to proceed down the spine once again, also recursively. When matching against a Class II primitive, however, the matcher begins a depth-first tree traversal in search of the atoms to match against the primitive. Here the problem arose of how to allow such a search through an arbitrary tree structure while still retaining the recursive nature of the matcher; a leaf-matching might leave an "orphan" of subtree left over with no way to bridge back up to higher unmatched levels of the tree for further matching. The decision was thus made to continually reform the unmatched portion of the source tree back into a single well-formed tree of comparable structure when doing leaf-matching, making further recursion always possible by matching the entire remaining tree at each step. (The backtracking operators BAL, ALT and ARB retain the value of the original tree to return to if necessary.) In essence, then, the source tree is pruned of each successful match and any resulting empty list nodes are condensed out. It is thus possible to freely combine primitives from the two classes in a single pattern, although use of a primitive from Class I must always accurately reflect the structure of the remaining source which it is to match. For example, the source

(a (b (c d),e) f)

will be successfully matched by both patterns

(SPANL(a b c) LITL(d) LITL(e) LITL(f))

and

(SPANL(a b c) ((LIT(d)) LIT(e)) LIT(f)),

where the bracketing in the latter pattern is needed to specify the depth at which each literal must occur.

When a successful match against a primitive is made, the portion of the source tree matched is entered into an output list in the position corresponding to the sequential position of the primitive within the pattern. If a match is not completely successful, the values of the partial matches are returned. This list corresponds to the immediate variable assignments made in the course of a SNOBOL4 string matching, and is available for examination and manipulation until the next pattern matching is undertaken. Both classes of primitive return the portion of the source tree that was traversed in making the match, except that the values returned by Class II primitives are pruned of any superfluous higher level list nodes that may have been traversed in reaching the "fruitful" branches actually matched.

In addition to these literal-matching primitives and operators, patterns may also contain user-defined symbols, which are atoms that have had values associated with them using the "Setq" procedure (like "Quote", a borrowed and somewhat redefined LISP function). For example, a pattern could be constructed as follows:

```
Setq ("DIGIT", "(ANY(0 1 2 3 4 5 6 7 8 9))" );
Setq ("DIGITS", "(ALT( (DIGIT) (DIGIT DIGITS) ))" );
Real_No_Pat: constant string := "(DIGITS LIT(.) DIGITS)".
```

When a user-defined symbol is encountered during matching, its value is substituted for

the symbol in the pattern, and matching then continues.

2.3. ACTIONS

After a pattern match has been performed, the values returned by the matcher are available for subsequent actions such as testing, concatenation into new lists, or further matching. They are accessed from the result list by use of an infix operator "/"(List,n) which selects the n-th member of the list and returns it. Such LISP predicates as Equal, Member, and Nullp are available for testing of these values, and actions taken can be conditional upon their results. Catenation of values is possible using standard LISP Cons or Append, but more convenient is the Ada string concatenator "&" which has been overloaded to append lists as an infix operator (string values can also be &'ed with list values if they represent properly balanced lists). The function List is also useful in correctly structuring the output desired, by forming its argument into a sublist.

An example of pattern matching that illustrates the construction of patterns and the use of several of these action operators is the following (highly simplified) grammatical transformation:

```
Function Pronoun_Subst (Source: S_Expr) return S_Expr is
Success; boolean;
T: S_Expr;
Pattern: constant string :=
   "(LIT<S>[LEN<2>LIT<VP>{BAL LIT<S>(LEN<2>REM)}])";
begin
Match(Pattern,Source,Success,T);
if Success then
   if Equal(T/2,T/6) then return
            T/1 & List(
                T/2 & T/3 & List(
                    T/4 & T/5 & List(
                       "(Pro(he))" & T/7 )));
   else return Source;
end Pronoun_Subst;
```

Pronoun_Subst (Source) for the input

(S(NP(N(John))VP(V(said)S(NP(N(John))VP(V(was)Adj(rich))))))

returns the transformation

(S(NP(N(John))VP(V(said)S(NP(Pro(he))VP(V(was)Adj(rich))))))

3. APPLICATION EXAMPLES

3.1. PARSING

One application of user-defined symbols in patterns is to serve as grammatical rewrite rules, where the symbol represents a nonterminal and its value represents the

right hand side to be expanded to. When doing parsing, the list returned by the matcher amounts to a parse tree of the input source, and an additional input parameter to the matcher can cause nodes of the parse tree to be labeled with the appropriate non-terminals, as well. As currently implemented, non-left-recursive context free grammars can be handled by the matcher, and are parsed by recursive descent. The matching function package contains a procedure "Parse" which takes as inputs a start symbol (the pattern being matched against), source, and label switch, and returns a success/failure boolean and parse tree, as illustrated below:

```
Setq ("S",   "( NP VP )"              );
Setq ("NP",  "( ALT( (N) (Det N) ))"  );
Setq ("N",   "( ANY(ship plane pilot) )" );
Setq ("Det", "( ANY(a the) )"         );
Setq ("VP",  "( ALT( (V) (V NP) ))"   );
Setq ("V",   "( ANY(flew sailed) )"   );
Start_Symbol: constant string := "S";
Source: constant string := "(the pilot flew a plane)";
Label: boolean := true;
Success: boolean;
Tree: S_Expr;
Parse (Start_Symbol, Source, Label, Success, Tree);
```

"True" is returned as the value of Success, and for the list Tree,

(S(NP(Det(the)N(pilot))VP(V(flew)NP(Det(a)N(plane)))))

is returned.

The output of the parser is, clearly, in proper form for further pattern-directed processing such as grammatical transformation, as outlined earlier.

In order to return a parse tree, a matcher must retain the portion of the result that was matched by each non-terminal symbol so as to make it a (possibly labeled) sublist of the tree. One approach to enable this would be to do a "partial match" of the symbol's right hand side against the source, and if successful then match the remainder of the pattern against the remainder of the source and append the results. This approach, however, makes it difficult to backtrack to another alternative (such as ALT, BAL or ARB) in the right hand side if the matching on the remainder fails. In the present work, the matcher avoids this problem by always doing a "complete" match (with backtracking) on the entire pattern, and remembers where each subpattern is to end by the insertion of an end-of-phrase marker into the pattern after each right-hand-side substitution. When it encounters one of these markers during the subsequent matching, it knows to lump the previous results at that level into a separate list, which is stored in a level-indexed array of subresults that are eventually assembled into the final parse tree. When not doing parsing, however, the matcher instead lumps together the values that were matched by each primitive in the pattern, so that these values can be separately accessed from the result list after matching is completed.

3.3. SYMBOLIC DIFFERENTIATION

Another application program written employing pattern matching was a procedure to perform symbolic differentiation of arithmetic expressions. The expression is first parsed using a grammar of operator precedence, such as

```
<S> => <T> + <S> | <T> - <S> | <T>
<T> =>     + <U> |     - <U> | <U>
<U> => <V> * <U> | <V> / <U> | <V>
<V> => <W> ** <V> | <W>
<W> => t | ( <S> ),
```

and if successful an unlabeled parse tree is returned. For example, parsing

(1 * 2 ** - 3 + 4)

returns the tree

((1 * (2 ** (- 3))) + 4),

indicating the correct order of operator application. This tree is then subjected to a series of pattern matches and subsequent actions to generate the derivative, as abbreviated below:

```
Function Deriv (Tree: S_Expr) return S_Expr is
begin
Match( "(LEN(1) ANY(+ - * / **) LEN(1))", Tree, Success, R);
if Success then
   if Eq(R/2, "+") or Eq(R/2, "-") then
      -- D(x +- y) = D(x) +- D(y)
      return Deriv(R/1) & R/2 & Deriv(R/3);

   elsif Eq(R/2, "*") then
      -- D(x * y) = D(x)y + xD(y)
      return List(Deriv(R/1) & "*" & R/3) &
             "+" &
             List(R/1 & "*" & Deriv(R/3));

   else ... etc.

else
Match( "(ANY(- -) LEN(1))", Tree, Success, R); ... etc.
end Deriv;
```

Deriv("(X ** 2 + 5)"), for example, returns the tree

((2 * (X ** (2 - 1) * 1)) + 0),

to which additional pattern-directed processing might then be applied to reduce the tree to (2 * X). Note that a simple rearrangement of the LIT and ANY primitives in the patterns could process a parse in prefix or postfix form.

4. EXTENSIONS AND FUTURE RESEARCH

4.1. ALTERNATIVE ADA IMPLEMENTATIONS

This approach to a pattern matcher for lists was inspired by Post-X [Bailes and Reeker, 1980], an applicative pattern matching language in which patterns have been generalized into a structure called a Form, consisting of an alternating series of patterns and corresponding actions combined into a single data object. In Ada, Form could be realized as a generic package that would be instantiated with a pattern part and an action part, hence serving as a template for new data objects of this type. In the version of Ada that was available for this research, generic packages and procedure-variant generic functions had not yet been implemented, and so more powerful pattern matching structures such as Forms could not be created except as ad-hoc procedures or clauses. The applicative approach to programming used in Post-X and espoused by Backus [1978] and others can also be realized in Ada through the use of generics and abstract types, and so further work on this project using a more complete Ada compiler would lead closer to the Post-X design.

4.2. PATTERN MATCHING IMPLEMENTATION

Pattern matching itself can be considered a form of parsing: for instance, the pattern (BAL LEN(1)) can be represented by the grammar

$$S \rightarrow BAL\ \ LEN1$$
$$BAL \rightarrow t \mid t\ BAL$$
$$LEN1 \rightarrow t$$

and matching a list against the pattern is equivalent to returning a parse of the list in terms of the grammar. In this regard, a backtracking pattern matcher is equivalent to a recursive-descent parser, which is limited in the classes of grammar it can accept and runs in exponential time as well. Kreuter [1984] has implemented a string pattern matcher in Ada using Earley's parsing algorithm, which has a worst case time behavior of N cubed and a more powerful grammar handling capability. Earley's algorithm could certainly be applied to this list pattern matcher as well and thus provide substantial improvements. Alternatively, heuristic methods such as SNOBOL's Quickscan mode could be added to the backtracking design to prune the search space and afford speed-ups.

4.3. SELECTORS

In Post-X, simple pattern matching returns its result in the form of a tree corresponding in structure to the pattern used; values are then accessed by multiple use of a selector operator, e.g. R/3/2 would select the second subtree of the result's third subtree. In the present work, values are instead returned as a linear list. Each approach might be useful in certain applications, and the current matcher could be easily modified to allow the user to select which result mode was desired.

4.4. LEAF MATCHING PRIMITIVES

In writing application programs for this matcher, only small use was made of the leaf-matching primitives. Further research should determine areas where these operators might prove more powerful.

4.5. CONCLUSION

The programming of pattern-directed packages for a variety of datatypes within Ada appears not only feasible, but quite worthwhile. Work is continuing in the areas described above.

REFERENCES

Aho, A. V., B. W. Kernighan and P. J. Weinberger [1979]. Awk — a pattern scanning and processing language, *Software—Practice and Experience*, **9**, 267-279.

Backus, J. [1978]. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Communications of the Assoc. for Comput. Machinery*, **21**, 8, 613-641.

Bailes, P. A. [1983]. The Derivation of An Applicative Programming Language for String Processing, Ph.D. Thesis, Department of Computer Science, University of Queensland.

Bailes, P. A., and L. H. Reeker [1980a]. Post-X: An experiment in language design for string processing, *Australian Computer Science Communications*, **2**, 2, 252-267.

Bailes, P. A., and L. H. Reeker [1980b]. An experimental applicative programming language for linguistics and string processing, *Proceedings, 8th Intl. Conf. on Computational Linguistics* , Tokyo, 520-525.

Bolz, R. [*personal communication*]. A package of string functions in Ada.

Clocksin, W. F., and C. S. Mellish [1981]. *Programming in Prolog*, Springer-Verlag, Berlin.

Dallman, Brian [1984]. AFHRL Program for Artificial Intelligence Applications to Maintenance and Training, *Artificial Intelligence in Maintenance: Proceedings of the Joint Services Workshop*, **TR-84-25**, Air Force Human Resources Laboratory, Training Systems Division, Lowry AFB, Colorado.

Earley, Jay [1970]. An efficient context-free parsing algorithm, *Communications of the Assoc. for Comput. Machinery* , 13, 94-102.

Enalls, R., J. Briggs and D. Brough [1984]. What the naive user wants from Prolog, *Implementations of Prolog* Campbell (ed.), Ellis Horwood, Chichester, England, 376-386.

Farber, D. J., R. E. Griswold and I. P. Polonsky [1964]. SNOBOL, A string manipulation language, *Journal of the Assoc. for Comput. Machinery*, **11**, 1, 21-30.

Floyd, R. W. [1967]. Nondeterministic algorithms, *Journal of the Assoc. for Comput. Machinery*, **14**, 4, 636-644.

Galler, B. A. and Perlis, A. J. [1970]. *A View of Programming Languages*, Addison-Wesley, Reading, Massachusetts.

Graham, S. L., M. A. Harrison and W. L. Ruzzo [1976]. On-line context-free recognition in less than cubic time, *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, 112-120.

Greibach, S. A. [1965]. A new normal form for context-free grammars, *Journal of the Assoc. for Comput. Machinery*, **12**, 1, 42-52.

Griswold, R. E. [1975]. *String and List Processing in SNOBOL4: Techniques and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey.

Griswold, R. E. [1984]. The control of searching and backtracking in string pattern matching, *Implementations of Prolog*, Campbell (ed.), Ellis Horwood, Chichester, England, 50-64.

Griswold, R. E., J. F. Poage and I. P. Polonsky [1971]. *The SNOBOL4 Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey.

Harrison, M. A. [1978]. *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts.

Hewitt, C. E. [1969]. PLANNER: A language for manipulating models and proving

theorems in a robot, *Proceedings of the International Joint Conference on AI*, 295-301.

Honeywell, Inc. [1983]. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A. Produced for the United States Department of Defense by Honeywell Systems and Research Center, Minneapolis, and Alsys, La Celle Saint Cloud, France.

Honeywell, Inc. [1984]. Rationale for the Design of the Ada Programming Language. Produced for the United States Department of Defense by Honeywell Systems and Research Center, Minneapolis, and Alsys, La Celle Saint Cloud, France.

Kay, M. [1980]. Algorithm schemata and data structures in syntactic processing, Technical Report, XEROX Palo Alto Research Center, Palo Alto, California.

Kreuter, John [1984]. Pattern-Matching Algorithms in Ada, Final Report, 1984 USAF-SCEEE Graduate Student Summer Support Program. ***Edited version included as the second paper of this report.***

Kuno, S. and A. G. Oettinger [1962]. Multiple-path syntactic analyzer, *Information Processing 62*, Popplewell (ed.), North-Holland, Amsterdam, 306-311.

Lewis, H. R. and C. H. Papadimitriou [1981]. *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, New Jersey.

Liu, Ken-Chih, and Arthur Fleck [1979]. String pattern matching in polynomial time, *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, 222-225.

Markov, A. A. [1951]. Theory of algorithms, *Trudy Mathematicheskogo Instituta imeni V. A. Steklova*, **38**, 176-189 [in Russian; English Translation, *American Math. Society Trans.*, **2**, 15, 1-14 (1960)].

Pereira, F. C. N., and D. H. D. Warren [1983]. Parsing as deduction, *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, Massachusetts.

Post, E. L. [1943]. Formal reductions of the general combinatorial decision problem, *American Journal of Mathematics*, **65**, 197-215.

Reeker, L. H. and P. A. Bailes [*in preparation*]. A proposal for a graphic programming environment for flexible "languageless" programming.

Rieger, C., and S. Small [1979]. Word expert parsing, *Proceedings, Sixth Intl. Conf. on Artificial Intelligence*, Tokyo, 1979.

Richardson, J. Jeffrey [1983]. Artificial Intelligence: An Analysis of Potential Applications to Training, Performance Measurement and Job Performance Aiding, **TP-83-38**, Air Force Human Resources Laboratory, Training Systems Division, Lowry AFB, Colorado.

Rogers, Hartley, Jr. [1967]. Cambridge, Massachusetts. *The Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

Sussman, G. J., T. Winograd and E. Charniak [1971]. MICROPLANNER Reference Manual, **AI Memo 203A**, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Tucci, Ralph [*forthcoming*]. Analysis and Development. Master's Thesis, Department of Computer Science, Tulane University.

Valiant, L. G. [1975]. General context-free recognition in less than cubic time, *Journal of Computer and Systems Sciences*, **10**, 308-315.

Warren, D. H. D., L. M. Pereira and F. Pereira [1977]. Prolog — the language and its implementation compared with LISP, *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, New York, 109-115.

Wauchope, Kenneth [1984]. Pattern-Directed List Processing in Ada, Final Report, 1984 USAF-SCEEE Graduate Student Summer Support Program. *Edited version included as the third paper of this report.*

Winston, P. H., and B. K. P. Horn [1981]. *LISP*, Addison-Wesley, Reading, Massachusetts.

Yngve, Victor H. [1958]. A programming language for mechanical translation, *Mechanical Translation*, 5, 1, 25-41.

☆U.S. GOVERNMENT PRINTING OFFICE: 1985 559 053 20028