WIS-CS-74-233
COMPUTER SCIENCES DEPARTMENT
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received September 13, 1974

EASEy: An English-Like Programming Language for Artificial Intelligence and Complex Information Processing

by

Leonard Uhr

Computer Sciences Technical Report # 233

December 1974

EASEy: An English-Like Programming Language for Artificial Intelligence and Complex Information Processing

ABSTRACT

EASEy (an Encoder for Algorithmic Syntactic English that's easy) is a programming language for list-processing and pattern-matching systems of the sort typically used for artificial intelligence and complex information processing. EASEy is designed primarily with ease of reading in mind, so that the reader can examine the actual program itself, rather than rely upon the inevitably vague verbal descriptions of programs. It is an English-like version of a subset of SNOBOL4 [3], with several extensions that enhance SNOBOL's list-processing capabilities.

This paper gives several examples of EASEy programs, for pattern recognition, deductive problem-solving, learning, and information processing, comparing them with equivalent programs in LISP and SNOBOL4, and with verbal algorithms. It also includes the primer for EASEy.

EASEy: An English-Like Programming Language for Artificial Intelligence and Complex Information Processing[1]

by

Leonard Uhr
University of Wisconsin

INTRODUCTION

EASEy is an English-like programming language that was developed in order to describe artificial intelligence systems by actually presenting the computer program, rather than by giving a typically (and inevitably) vague description. This lets the reader see exactly what is happening, and allows the author to make variations, comparisons and extensions in a concise and precise way.

EASEy is a general-purpose programming language. It exists as an EASEy-to-SNOBOL4 translator (coded in SNOBOL4), and therefore programs coded in EASEy can be executed on any computer (e.g., IBM 360 and 370 series, CDC 6400 and 6600, Univac 1108 and 1110) on which SNOBOL runs. Actually executing EASEy programs is awkward and expensive, since they must first be translated into SNOBOL, and error messages will then refer to the SNOBOL program. But the possibility of execution allows programs to be debugged and tested, so that they can be used for precise and complete communication to a reader.

[1]This research has been partially supported by grants from the National Institute of Mental Health (MH-12266), the National Science Foundation (GJ-36312), NASA (NGR-50-002-160) and the University of Wisconsin Graduate School.

This paper contains as its appendix the Primer for EASEy-2 previously issued as UW Computer Science Department Technical Report 178.

In order that such communication be effective, the EASEy program must be relatively easy and painless to read, by somebody not familiar with the language. EASEy certainly fails to some extent (the reader can judge from the several sample EASEy programs that follow), but it comes closer to this goal than does any other language known to the author. And it appears to be a language that can be learned, as needed, in a relatively painless way: by trying to read programs and, when necessary, referring to the EASEy Primer for help.

Criteria for Programming Languages:

Coding, Communicating, Computing

We might judge a programming language in terms of its suitability for a) coding programs, b) reading and understanding other people's programs, c) execution on a computer. We can further consider how powerful, efficient, and general the language is, and how easy it is for a user (either coder or reader) to learn. EASEy is designed to be easy to read and understand. Further, it is designed to be learnable with a minimum of effort. It is English-like, and therefore builds on the already-known mother tongue. It allows for short example programs that are of interest in themselves, so that the reader is motivated to understand these programs, and can understand a good bit about them at first reading. It therefore allows the reader to learn by immersing himself in examples of the language, much as we learn a foreign language in a foreign country--by speaking and listening, even when we don't quite understand.

EASEy reads like a stilted, awkward, succinct English. A number of alternative ways of expressing the same statement allow the user to make the code more English-like. This makes EASEy programs easier to read, but it may also make them more cumbersome to write. Short and more succinct constructs are made available to the user who wants to use EASEy to code and execute programs, and not just to communicate. But several things need to be done to make EASEy more useful for that purpose: An EASEy compiler that by-passes SNOBOL and gives good error messages should be coded. And the variety of

ways of expressing a statement should be made greater, so that the system is flexible enough to allow a user to start coding without worrying too much about precise formats, and expect the system to help correct him. This means that coding should be done interactively, so that the system can both teach the user, and adapt to his modes of expression. Without extensions of this sort, it seems likely that a serious programmer would find it just as convenient to learn and code in SNOBOL, rather than use EASEy.

Communicating of the Precise and Complete Level of Actual Programs

But the primary purpose of EASEy is to communicate with readers, and here it would appear to be rather successful--and without any loss in generality, since EASEy is a general purpose programming language. Programs coded in EASEy are still hard to follow and understand. But I would assert that most of the barriers of the peculiar form of the programming language have now been stripped away, and the remaining problems follow from the difficulties of the content--what the program does, and the mechanisms that are needed. And these content issues are just what we want to communicate about, when we write papers and books on artificial intelligence and information processing systems. We will be able to begin building upon one another's systems only when we can see and understand just what they do, and what mechanisms they use. We are also forced to make our code as simple and elegant as possible if we must justify every statement to a reader.

EASEy and SNOBOL as List-Processing Language Alternatives to LISP for Artificial Intelligence Research

We will now compare EASEy programs with verbal algorithms, to show how much more precise and complete is the program, but with little sacrifice in readability. We will also compare EASEy programs for simple 1) deductive problem-solving, 2) question-answering, and 3) pattern recognition with equivalent programs in LISP and SNOBOL4.

The common wisdom in Artificial Intelligence circles has been that LISP, and LISP-based systems like Conniver and QLISP are essential for list-processing problems of the sort found in Artificial Intelligence (see e.g. Bobrow and Raphael, [1]). SNOBOL, however, also allows for very convenient and simple handling of list structures, and also gives great power for pattern matching (which is just beginning to be realized in the new LISP-based higher-level systems like Planner [4], (Conniver [9], QLISP [8] and Fuzzy [5].

EASEy uses the full-blown SNOBOL pattern match capability, and enhances SNOBOL's already convenient and powerful list-processing capabilities. It also shares with SNOBOL relatively powerful and convenient arithmetic and function capabilities.

EASEy and SNOBOL therefore seem to this author to be worthy languages for artificial intelligence and other complex information processing programs that work with patterned data organized in list structures or other forms of graphs. SNOBOL may be preferable for coding and executing; EASEy for reading and communication. SNOBOL, and therefore EASEy, is often available in conveniently usable form at installations where LISP is not. LISP is chiefly used at centers for Artificial Intelligence that have dedicated (and heavily

funded) PDP-10 systems. A large LISP program can therefore take over the entire system, without the user having to pay, or be aware of, the real costs. So people have the impression that LISP systems can be larger than SNOBOL systems, and cheaper to run. But I know of no evidence one way or the other on this matter.

The crucial point is that artificial intelligence research can be done quite adequately in languages other than LISP or the other so-called "list-processing" languages, and that SNOBOL and EASEy are interesting, widely available candidates. They have the advantages of powerful pattern-matching (which is only available in the even-larger-than-LISP extensions to LISP), and EASEy has the advantage of at least some steps toward readability.

EASEy-2 Compared with Verbal Algorithms and LISP Programs

Let's look now at several simple examples of EASEy programs, comparing them with verbal algorithms, and with the same programs coded in LISP. (See Uhr [10] [11] [12] for additional and less toy-like examples of EASEy programs.)

Problem-Solving, Using Nilsson's "Breadth-First" Algorithm

Nils Nilsson's book on heuristic "Problem-Solving Methods for Artificial Intelligence" [7] presents a number of verbal algorithms to show the reader how a deductive problem-solver can go about finding a path to the goal posed it (e.g. a theorem to be proved, a win position in a game, a solution state in a puzzle) from a starting position (e.g. the axioms, the initial game board or puzzle state). These are probably the clearest, simplest and most precise statements of artificial intelligence algorithms in the literature, so it is especially instructive to compare them with actual computer programs.

The following is the flow-chart for a "breadth-first" algorithm for trees, exactly as given in Nilsson, Figure 3-1, page 46.

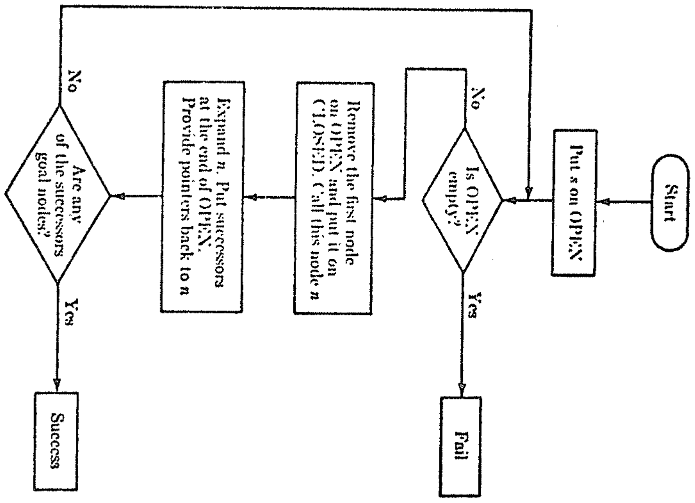
Start
Put s on OPEN
Is OPEN empty?
Yes
Fail
No
Remove the first node on OPEN and put it on CLOSED. Call this node n
Expand n. Put successors at the end of OPEN. Provide pointers back to n
Are any of the successors goal nodes?
Yes
Success
No

FIG. 3-1 *Flow chart of a breadth-first search algorithm for trees.*

The following is Nilsson's verbal statement of the algorithm, and an equivalent EASEy-2 program (algorithm sections are placed next to their equivalent program statements, which are numbered at the right):

Breadth-First Algorithm (Nilsson, p. 45)	An Equivalent Program in EASEy	SOLVE-A
(1) Put the start node on a list called OPEN.	(1) list OPEN = START	1
(2) If OPEN is empty, exit with failure; otherwise continue.	(2) is OPEN sameas NULL? [+to FAILURE]	2
(3) Remove the first node on OPEN and put it on a list called CLOSED; call this node n .	(3) from OPEN get the next NODE erase	3
	on CLOSED list the NODE	4
	set N = NODE	5
(4) Expand node n , generating all its successors.	(4) set SUCCESSORS = $ N	6
If there are no successors, go immediately to (2).	is SUCCESSORS sameas NULL? [+to (2)]	7
Put the successors at the end of OPEN and provide pointers from these successors back to n .	on OPEN set SUCCESSORS	8
	on POINTERS list SUCCESSORS : N]	9
(5) If any of the successors are goal nodes, Exit with the solution obtained by tracing back through the pointers;	(5) from SUCCESSORS get that GOAL [-to (2)]	10
	5B output N	11
otherwise go to (2)	from POINTERS get that N, XX till : and next N till [+to 5B]	12
	end	

NOTE That the algorithm does not specify how to get the start or the goal nodes, handle more than one problem, or erase and initialize lists. It does not handle the case where the START node is also the GOAL. No use is made of CLOSED.

An algorithm of this sort will inevitably be less detailed or vaguer in certain places than in others. For example, phrases like "provide pointers," "exit with the solution obtained by tracing back through the pointers," or "expand node N" can be interpreted and implemented in a variety of different ways. They depend upon the reader's understanding of what is to be done based upon some additional outside knowledge of the meanings of the words used, and the processes being handled (as given in Nilsson's discussions and examples of the algorithm). The EASEy program makes these precise (I think doing what Nilsson intended, but that's a matter of interpretation)--but without cluttering or lengthening the program. On the contrary, the EASEy statements 6, 9, and 11-12 are shorter and, at least in ways, clearer than the comparable sections of the algorithm.

Note that the algorithm (and the equivalent EASEy program) is not complete. It does not specify how to get the START or the GOAL nodes, or what to do upon "failure." Nor can it handle more than one problem, doing the necessary bookkeeping to erase and initialize lists, and looping back to input and process subsequent problems.

The following additions to the EASEy program will do these things:

```
SOLVE     input START GOAL [- to END]                      Before 1
          erase CLOSED, SUCCESSORS [+ to 5B - to SOLVE]    Before 1  12. Variant
FAILURE   output 'NO PATH FOUND' [to SOLVE]                          13
```

Note how easy it is to complete the program; more important, how easy it is to see where these additions are made, and to compare variant programs.

Finally, the program makes obvious several peculiarities, and possible improvements, that are not clearly visible from the original algorithm. For example, the CLOSED list is never used, so that we might just as well eliminate statement 4. More fundamental, given the EASEy program to simplify and improve upon, we can recode, getting the following appreciably simpler yet more powerful programs:

(Program SOLVE-B. Simplified breadth-first program (Nilsson, p. 45. Inputs problems, outputs solutions, handles a whole sequence of problems. Note that this handles the case where the start node is also the goal)

```
                                                                A     B
NEXT   input OPEN GOAL [- end]                                 1.A    1
BUD    from OPEN get a NODE erase [-to  FAIL]                  3.A    2
       is NODE sameas GOAL? [ +to  PATH]                      10.A    3
       on POINTERS set $NODE : NODE                            4.A    4
       on OPEN set $NODE [ goto  BUD]                          8.A    5
PATH   output the NODE                                        11.A    6
       from POINTERS get that NODE, XX : NODE]                12.A    7
+                                [+to PATH -to NEXT]
FAIL   output 'NO SOLUTION FOUND' [NEXT]                              8
end
```

An Equivalent SNOBOL4 Program

The following shows how similar (but far less readable) is the equivalent program in SNOBOL4

```
NEXT     INPUT BREAK(' ') . START ' ' BREAK(' ') . GOAL :F(END)              1
NEXT     OPEN BREAK(' ') . NODE ' ' = :F(FAILURE)                            2
         EQUALS (NODE, GOAL) :S(PATH)                                        3
         POINTERS = POINTERS $NODE ':' NODE ']'                              4
         OPEN = OPEN $NODE :(BUD)                                            5
PATH     OUTPUT = NODE                                                       6
         POINTERS NODE BREAK(':') ':' BREAK(']').  NODE :S(PATH)F(NEXT)      7
END
```

The next major variant of a problem-solver is one that makes a "depth-first" search, going deep into the network that may lead from start to goal nodes, rather than advancing shallowly on all fronts (see Nilsson, p. 50 for a verbal algorithm of a slight variant). It is interesting to note how easily this variant can be presented, in a way to make crystal clear exactly what is being changed, as follows:

```
    at start of OPEN set $NODE [ goto BUD]                        5.Variant
```

Equivalent LISP Programs

The following LISP[12] program is an equivalent breadth-first problem-solver (without any input).

```
(CSETQ PATH
(LAMBDA  (START GOAL)
   (PROG <(OPEN (LIST START)) NODE>
    NEXT    <COND [OPEN <SETQ NODE (CAR OPEN)>]
                      [T <GO FAILURE>]>
           <SETQ OPEN (CDR OPEN)>
           <COND [<EQUAL NODE GOAL><GO PATH>]>
           <MAPC (EXPAND NODE)(LAMBDA (A) (PUT A 'BACK NODE))>
           <SETQ OPEN (APPEND OPEN (EXPAND NODE))>
           <GO LOOP>
    PATH <PRINT NODE>
           <COND [<SETQ NODE (GET NODE 'BACK)><GO PATH>]>)))
```

The following LISP program does depth-first problem solving. It is coded recursively, because that is the way a good LISP programmer would code it.

```
(CSETQ PATH
(LAMBDA (NODE DEPTH)
          (COND [<OR (WINNER NODE)
                          (AND (NOT (GT DEPTH MAX-DEPTH))
                               (MAP-PATH (EXPAND NODE)))>
                     <PRINT NODE>])))
(CSETQ MAP-PATH
(LAMBDA (NODES)
          (COND [<NULL NODES>NIL]
                [<PATH (CAR NODES) (ADD1 DEPTH)>]
                [T <MAP-PATH (CDR NODES)>])))
```

A Recursive EASEy Program for Breadth-First Problem-Solving

The following is a recursive EASEy program. It shows how conveniently recursion can be handled. But the program is far more difficult to read and understand.

```
          DEFINE: ONWARD(TOBUDS)                    .1    1
NEXT      input OPEN GOAL [- end]                   1     2
          from OPEN get NODE = [- FAIL]             2     3
BUD       ONWARD(OPEN)  [- FAIL]                    .1    4
          output NODE 'Q.E.D.' [NEXT]               6     5
```

```
ONWARD    from TOBUDS get TOBUD = [- freturn]       2.V   6
          is TOBUD sameas GOAL? [+ PATH]            3.V   7
          ONWARD($TOBUD)  [+ PATH - ONWARD]         .1    8

PATH      output TOBUD [+ return]                   6-7   9
FAIL      output 'NO SOLUTION FOUND' [NEXT]         8     10
end
```

A Program for Simple Information Retrieval-Question Answering-Learning

The following is a very simple example of a stylized information retrieval-question answering program that learns. It assumes that questions are of the form Subject-Relation, and that replies contain the object (or objects) listed in memory for that relation, under that subject. It "learns" in the very simple sense that when an input card is of the type "LEARN" it stores that object under that relation under that subject. (Note that a much more sophisticated system is needed for less stylized questions. But this program is of interest here because it shows how easily very simple kinds of learning, in which lists must grow without bounds and information can be inserted anywhere in a list, can be handled, and how appropriate EASEy is for information processing programs.)

```
(Retrieves, and Learns, information about related objects)
      RELATIONS = IS ABOVE' |'IS TO THE EA5 of' |'GROWS' ...       Memory
INPUT       input TYPE INFO till] [- to end ]                           1
            COPYRELS = RELATIONS                                        2
            from INFO get SUBJECT that RELATIONS OBJECT
                                                                        3
REPLY       from $SUBJECT get that INFO REPLIES] [- FAIL]               4
            output REPLIES [INPUT]                                      5
FAIL        output 'NO REPLY' [INPUT]                                   6
LEARN       from $SUBJECT get INFO = INFO OBJECT [INPUT]                7
            on $SUBJECT list INFO OBJECT] [INPUT]                       8
end
LEARN WISCONSIN IS  TOTHE EAST OF IOWA ]
LEARN WISCONSIN GROWS CORN PEAS ]
REPLY WISCONSIN IS ABOVE WHAT? ]
```

A Simple Pattern Recognition Program

The EASEy primer begins with an example pattern recognition program that the reader is asked to try to follow in order to get the gist of EASEy, and then use, along with the primer's explanations, for a more precise understanding. The following is the LISP version of that program:

Summary Discussion

To sum up: A program, whether in EASEy, LISP, SNOBOL or Fortran, is more precise and complete than is a verbal algorithm. If programs are simple enough, they can also be read, understood and compared. We are therefore in a far better position to examine them, and make improvements, whether simplifications, generalizations, or combinations.

EASEy handles list searching cleanly and succinctly, without as much peculiar jargon as LISP, and without the to-the-non-mathematicia unfamiliar parenthesised functional notation. EASEy handles pattern matching, and list manipulation and restructuring, a good bit more cleanly and conveniently than does LISP. EASEy arithmetic and input-output also seem more convenient, and more understandable to the non-programmer.

The English-like look of EASEy can be a powerful incentive luring the non-programmer into the details of the program, since he can almost immediately grasp the general drift of what is going on. We can present and describe graded sequences of programs that are of interest in themselves, to seduce him to an understanding of the EASEy programming language, and therefore of programs and computers, as well as of the particular program. The longer forms of EASEy make for smoother reading, the shorter forms make for less cumbersome coding.

EASEy shares with SNOBOL the extremely powerful pattern-matching capabilities that are so important for pattern recognition and language processing. And pattern-matching, along with the capability for indirection (that is, for treating a string as a name that points to its contents), make extremely convenient the writing of

```
[DEFINE'(
?A LISP VERSION OF EXAMPLE PROGRAM IN EASEY PRIMER
?THIS CODE IS NOT VERY LISP-LIKE, SINCE IT INTENTIONALLY
?MIRRORS THE EASY CODE.
[PROGRAM-A (LAMBDA NIL (PROG NIL
?INIT
  <SETQ CHAR1 '( (<(Ø 1 1 1) 2> <(1 Ø Ø Ø) 9> <(1 1 1 1) 24>),
                 ( <B 6> <F 9>) ) >
  <SETQ CHAR2 '( (<(Ø Ø 1 1 1 1 1 1 1) 3> <(Ø Ø Ø Ø Ø Ø Ø Ø) 18>)
                  ( <B 5> <E 9>) ) >
SENSE
  <SETQ LOOKFOR '(CHAR1, CHAR2)>
  <SETQ MAYBE NIL>
?IN
(COND [(NULL(SETQ PATTERN(READ))), (RETURN'GOOD-BYE)] )
RESPOND
(COND [(NULL LOOKFOR), (GO OUT)] )
  <SETQ CHAR (CAR LOOKFOR)>
  <SETQ LOOKFOR (CDR LOOKFOR)>
  <SETQ DESCR (CAR(EVAL CHAR))>
  <SETQ IMPLIEDS (CADR(EVAL CHAR))>
R1
(COND [(NULL DESCR), (GO IMPLY)] )
  <SETQ HUNK (CAAR DESCR)>
  <SETQ LOCATION (CADAR DESCR)>
  <SETQ DESCR (CDR DESCR)>
(COND [(MATCH (NTH PATTERN (ADD1 LOCATION)) HUNK), (GO R1)] )
  <GO RESPOND>
IMPLY
(COND [(NULL IMPLIEDS), (GO RESPOND)] )
  <SETQ NAME (CAAR IMPLIEDS)>
  <SETQ WT (CADAR IMPLIEDS)>
  <SETQ IMPLIEDS (CDR IMPLIEDS)>
(COND [(NULL(SETQ X(ASSOC NAME MAYBE))),
        (DO <SETQ MAYBE (CONS(CONS NAME WT)MAYBE>
             <GO IMPLY>) ] )
  <RPLACD X (+(CDR X)WT) >
?TEST
(COND [(LESSP (CDR X) 8), (GO IMPLY)] )
OUT
  <PRIN1 PATTERN> <PRIN1 " IS A "> <PRINT NAME>
  <GO SENSE>
       ] ?END OF PROGRAM-A
?
?'MATCH' DETERMINES IF HUNK MATCHES PATTERN.
[MATCH (LAMBDA (PATTERN, HUNK)
(COND [(NULL HUNK), T]
      [(EQUAL(CAR HUNK)(CAR PATTERN)), (MATCH(CDR PATTERN)(CDR HUNK))
      [T, F] )   ] ?END OF MATCH
?
?[NTH L N] IS 1108 LIB. ROUTINE WHICH RETURNS THE
?NTH FINAL SEGMENT OF L
     ]  ?END OF DEFINE

EVAL:
(PROGRAM-A)
```

```
 INPUT:  (Ø Ø Ø 1 1 1 1 1 1 1 Ø Ø Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 1 1 1 1)
OUTPUT:  (Ø Ø Ø 1 1 1 1 1 1 1 Ø Ø Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 Ø 1 1 1 1 1) IS A F
```

code that searches through and manipulates list structures and other kinds of graphs. Extended delimiter capabilities further enhance EASEy as a list processor. Whereas LISP is limited to the single binary tree list structures (from which other kinds of structures must be built) EASEy and SNOBOL allow the user to compose and use any kind of structure he desires - with very little trouble.

A Primer for EASEy-2, An _E_ncoder for _A_lgorithmic _S_yntactic _E_nglish that's EASE_y_.

CONTENTS

Overview of the EASEy-2 Programming Language

I. A Simple EASEy-2 Program (A)

II. EASEy-2 Constructs Described

Introduction: Combining Objects and Names onto Strings, Lists, Graphs and Arrays

A. Basic Statement Types for Manipulation of List Structures
B. Types of Objects Used
C. Functions
D. Flow of Control
E. Flexible Constructs

III. Summary of EASEy-2 Constructs

IV. Appendix

A. A Detailed Description of Program A
B. The Relationship between EASEy and SNOBOL
C. Variant and Short Forms of EASEy-2, for Easier Coding

OVERVIEW OF THE EASEy-2 PROGRAMMING LANGUAGE

The following gives a program in, and then explains, an English-like programming language called EASEy-2* (an Encoder for Algorithmic Syntactic English that's easy-Version 2). EASEy is modelled after pattern matching languages like SNOBOL (Farber, Griswold, and Polonsky, [2]; Griswold, Poage, and Polonsky, [3]); and Comit (Yngve, [15]). It is, essentially, a simplified English-like version of SNOBOL, with constructs added to make list processing more convenient, as in LISP (Weissman, [14]) and IPL (Newell *et al*, [6]). At present it exists in the form of a SNOBOL4 program that translates an EASEy program into an equivalent SNOBOL4 program that can then be executed by a SNOBOL4 translator.

EASEy is designed primarily for easy *reading*, to be understood by someone who knows nothing about programming. It also can tolerate a number of alternate constructs, to give flexibility in coding. EASEy programs are stilted and occasionally awkward. But they should give the reader at least a general idea of what the system is doing, along with the opportunity to study the actual code, when desired, until it is understood. Most of the difficulties in reading will result from the logical structure of the program's processes, rather than the peculiarities of the program's language--that is, from content and not form.

*EASEy-1 (see Uhr, [11], [12] is a proper subset of EASEy-2 (except that parentheses around gotos must be changed to brackets), and will run under the EASEy-2 translator. EASEy-2 is more powerful, more flexible, and more understandable.

A concise explanation of EASEy follows the example program in the primer. But the reader should first try to read the program without the primer.

Here are the essentials: EASEy allows the user to name lists, and then manipulate them. EASEy defines a list by assigning a string of objects as the contents of a name (e.g.: *list* TODO = LAYERS CHARS, or set X = X + 1). Objects are got from lists (e.g.: *from* TODO *get* ...) and added to lists (e.g.: on MAYBE *list* NAME WEIGHT).

"Goto" a label is indicated at the right of a statement, in brackets. Comment cards start with '(' and continuation cards start with '+ '.

Most other conventions are quite natural, except for the very confusing construct that means "the contents of the contents of this name", which can be indicated by $name (or, alternately, what's *under* name). E.g.:

Code	Meaning	Result
set R = R + 1	Add 1 to the contents of R	R contains 1
set $('L.' R) = R' *0011'	Assign '1*0011' as the contents of ('L. ' R)	L.1 contains 1*0011

List structures and graphs can now be handled by storing a string of names, getting a name, and looking at the string it points to, using the $name construct.

The user is given a number of options as to constructs. Thus there are long forms, more suitable for casual reading, and short, more succinct forms that are more easily read and coded by an experienced programmer. E.g., the following two statements are equivalent:

```
on  TO-DO list the next TEST, and its WEIGHT.
    TO-DO list TEST WEIGHT
```

Introduction to the Primer

EASEy-2 is a list processing, pattern-matching language that uses simple English formats designed to be easy to understand.

An EASEy program is a sequence of statements that construct and rearrange lists of information, find items on these lists, compute transformations on these items, rearrange information within and between lists, and input and output information. Statements are executed from top to bottom except when GOTO's indicate otherwise. A GOTO may be conditional on the success or failure of the statement's search for a pattern, or test for an inequality.

The following program will introduce the reader to EASEy, giving him a feeling for the language. Then EASEy's basic constructs and variants will be described. Finally EASEy's constructs will be summarized, and compared to SNOBOL.

I. A Simple EASEy-2 Program

```
                                                                          A
(Program A.  An example pattern recognizer.                               C1*
(Positioned n-tuples imply weighted names.                                C2
(Initializes CHARacterizers, LOOKFOR.  Inputs PATTERN.                    C3
†INIT     †set CHAR1 = '0111 2 1000 9 1111 24 ]B 6 F 9 '                   M1*
               CHAR2 = '001111111 3 00000000 18 ]S 5 E 9 '                M2
                     :                                                    :
           set CHARN = . . .                                              MN
SENSE      set LOOKFOR = 'CHAR1  CHAR2 . . . CHARN '                      1
           erase MAYBE.                                                   2
IN         input the PATTERN till '/' [-to end]                           3
(Gets each CHARacterizer's DESCRiption and IMPLIEDS)                      C4
RESPOND    from LOOKFOR get the next CHAR.  erase.  [- FAIL]              4
           from $CHAR get DESCR till  ]  and IMPLIEDS till the end        5
(All HUNKS must be found for the CHARacterizer to succeed.)               C5
R1         from the DESCR, get HUNK and its LOCATION. = [-to IMPLY]       6
           at the start of PATTERN, get and call LOCATION symbols
+              LEFT, and get that HUNK.  [+ R1.  - RESPOND]               7
(Merges IMPLIED NAMEs onto MAYBE.                                         C6
IMPLY      from the IMPLIEDS, get the next NAME and its WT.
+              erase.  [-to RESPOND]                                      8
           from MAYBE, get # that NAME # and its SUM.  replace by
+              NAME and SUM + WT  [+to TEST]                              9
           on MAYBE list the NAME and its WT [goto IMPLY]                 10
(OUTPUTs the first multiply-implied NAME whose SUM of WeighTs exceeds 30, C7
TEST       is the SUM + WT greater than 30? [-to IMPLY]
OUT        yes - output the PATTERN ' IS A ' NAME [SENSE]
FAIL       output 'I FAILED TO DECIDE'. [SENSE]                           13
(The end card, and 3 patterns to be read in on data cards follow.)        C8
end            [goto INIT]                                                14
0001111111000010101010101011000/  (first two hunks of CHAR1               I1
                                   will succeed, third fails)
000111111100010101010101011111/   (CHAR1 succeeds)                        I2
0000011111110000100000000/        (CHAR2 succeeds)                        I3
```

*A number in the right margin refers to a statement in Program A that illustrates the construct being discussed. C = Comment, I = data program Inputs, M = Memory initialization. The appendix describes program A.

†Lower case letters indicate system words (those not underlined are optional), capital letters indicate program names. To run a program, lower case words may be keypunched in caps; underlined words must be.

II. EASEy-2 Constructs Described

Introduction: Combining Objects and Names onto Strings, Lists, Graphs and Arrays

This introduction briefly examines some key issues.

1. Manipulating Structured Sets of Objects

EASEy uses a number of constructs designed to handle sets of objects or names that have been put together into strings, lists, graphs and arrays. This allows for convenient building and manipulating of such things as perceptual and cognitive "chunks," natural language phrases and sentences, sensory patterns in several dimensions, cognitive networks for models and maps, and other types of compounds. It is therefore a very convenient language for problems in artificial intelligence, pattern recognition, natural language processing, and modelling of cognitive processes.

Objects are set onto strings, or listed onto lists. Objects, strings, or lists are also combined into arrays, trees, list-structures, and graphs. An object can be treated as a name, and that name can be used to get its contents (what it names, or points to). Thus any object, name, string, list, or array can be given a name, and then accessed through that name.

2. Constructing and Using Structures of Lists of Names

EASEy is designed to make it as easy as possible to handle structures of lists - by getting an object from a list, using that object as the name of another list, getting an object from that list, and so on.

As a simple example, Program A sets up a CHARacterizer as a
list of a DESCRiption (HUNK LOCATION pairs) followed by a] and
then IMPLIEDS (NAME WeighT pairs)(see statements M1, M2). The M1,M2
CHARacterizer names (CHAR1, CHAR2, etc.) are then set onto the
list named LOOKFOR (1).* Each CHARacterizer is got from LOOKFOR 1
(4), then its DESCRiption and IMPLIEDS got (5), and then each HUNK 4,5
and LOCATION got from the DESCRiption (6). Such a procedure can 6
continue to any depth.

3. Matching Patterns of Objects

EASEy uses all the SNOBOL pattern match techniques (see Griswold et al, 1968, for details). Essentially, a set of objects is to be looked for in a named string. If these objects are found, as specified, then, optionally, they are deleted and, optionally, any specified replacements are made.

The pattern match of the objects looked for starts at the left of the
named string and moves to the right (as in statements 4, 5, 7). 4,5,7
Essentially, the first possible assignment of an object is made, then the next object is assigned, and so on. Whenever it is impossible to make an assignment, the matcher moves back to the last object assigned, unassigns it, and gets the next possible assignment. This procedure continues until all objects have been assigned (which will be the left-most possible assignment), or the pattern match has failed.

4. Using Delimiters to Get Names from Lists

Delimiters are used to allow convenient access of names. EASEy
assumes that a name to be assigned will be followed by one space,
and it handles spaces automatically (as in statements 4 and 8). The 4,8

*Numbers in the text and right margin refer to statements in program A. See the previous footnote for Program A.

programmer can specify several other delimiters, including], :
and ; (as in 5). A general delimiter, #, can be specified to mean 5
any of the delimiters (including #). Since EASEy handles the nec- 9
essary details, this allows for quite convenient, and clean and readable, code for handling lists.

5. Using Names

Once a name has been got (as the name CHAR is got from
LOOKFOR in 4), its contents can be looked at by using the dollar- 4
sign ($) construct, which looks at the string whose name is preceded
by the $. (An alternate way of saying this in EASEy is 'what/s under'
- that is, $CHAR is equivalent to what/s under CHAR.) Thus in 5, 5
$CHAR means "get the contents of CHAR (which, the first time state-
ment 4 has pulled CHAR from LOOKFOR, will be CHAR1), and look at 4
its contents."

This kind of "indirect addressing" makes list processing very easy and convenient, especially when used with the delimiters introduced above. But the reader should not feel uneasy if he finds these topics confusing. They are; but the detailed examination that follows, with references to Program A, and some practice, should clear things up.

A. Basic Statement Types for Manipulation of List Structures

1. Lists are initialized and added to:
   a. Names can be assigned to strings of objects:

```
            set (name) = (objects) [general form]            M1,1*
   E.g.:    set C1 = '00111'  [example of code]
            set LOOKFOR = C1  ' '  C1  ' '  C1  ' '
```

*See footnote for program A.

assigns '00111' as the contents of C1, and then assigns '00111 00111 00111 ' as the contents of LOOKFOR.

(Note that set is optional (see M2). M2

b. Objects can be added to the end of a named string:

```
         on (name) set (objects)
E.g.:    on COUNTRIES set COUNTRY  ]  AREA  ]
```

adds the contents of COUNTRY followed by], the contents of AREA,], to the end of COUNTRIES.

c. Objects can be added to the start of a named string:

```
         at start of (name) set (objects)
E.g.:    at start of DESCRIPTORS set DESCRIPTOR ' ' WT ' '
```

d. Objects can be formed into linear lists:

```
         list (name) (objects)
E.g.:    list LOOKFOR = C1  C2  C3
```

creates a list consisting of the contents of C1, C2 and C3, each followed by one space, and names this list LOOKFOR.

e. Objects can be listed at the end of a named list:

```
         on (name) list (objects)
E.g.:    on IMPLIED list NAME WT                              10
```

List is much like set, except that it automatically puts a delimiter (one space) after each object listed, unless that object is a literal string (those enclosed in quotes), or is itself a single-symbol delimiter (#,], :, ;, or ' ').

f. Objects can be listed at the start of a named list:

```
         at start of (name) list (objects)
E.g.:    at start of DESCRIPTORS list DESCRIPTOR WT
```

2. Information is got, erased, and replaced in lists:
   a. Objects can be got from a named list:
      from (name) get (objects)
      E.g.: from SENTENCE get WORD 4,5,6
      will assign the name WORD to the first string on SENTENCE, ending with the space delimiter (but without changing SENTENCE).
   b. Objects can be got and erased from a list by extending the get command:
      from (name) get (objects) erase
      E.g.: from LOOKFOR get CHAR erase 4,8
   c. The objects can be replaced by other objects:
      from (name) get (objects) replace by (objects)
      E.g.: from LOOKFOR get CHAR WT replace by TRANS 9

An equal sign (=) can be used instead of 'erase' or 'replace by'. 6

   d. All contents can be erased from named lists:
      erase (names)
      E.g.: erase R C MAYBE 2
3. Information is input and output:
   a. One card of data can be input and names assigned to its contents:
      input (objects)
      E.g.: input TYPE till] PHRASE till '. ' LINE till ' ' 3
   b. Lists can be printed out:
      output (objects)
      E.g.: output LOOKFOR ' = ' $LOOKFOR 12

B. Types of Objects Used

An object is a string of symbols followed by one or more spaces. Such a string is often a name whose contents are some other string of objects to which it points. Several different kinds of strings are used, as follows:

1. Names: A name is an alphanumeric string that points to (names) some contents. 1,4

2. Literals: When a string is in quotes (either single (') or double (") it is a literal ikon that signifies itself.
E.g.: from SENTENCE get 'AND' 3
means that the thing in quotes-- 'AND' should be found in SENTENCE.

3. Specified Objects: that string will look for the contents of the string.
E.g.: set PHRASE = 'THE TABLE'
from TEXT get that PHRASE 9
will see whether 'THE TABLE' (the contents that has been assigned to PHRASE) is in TEXT, whereas:
from TEXT get WORD
assigns the name WORD to the first string that ends with a space in TEXT.

4. Indirect and Compound Names: $string will treat the contents named by that string as a name, and look in the string it names. Parentheses can be used to compound together a sequence of several literals and named strings.
E.g.: set R = 1
set $('ROW.' R) = '1001100' 5
will set Row.1 to contain 1001100 (since R contains 1).

5. Using Delimiters to get Names from Lists: A name is broken out of a string of symbols (to which another name has pointed) by finding a delimiter, and then using the entire substring up
to that delimiter as the name. Note how statement 1 puts a 1
space after each name of a characterizer ('CHAR1 CHAR2 ...
CHARN '). This allows statement 4 to get each CHARacterizer
from LOOKFOR - because built into EASEy are procedures that 4
look for the space delimiter, when it is asked to assign a name
(as by 'get the next CHAR'). Then erase eliminates the space 4
delimiter, and the entire string up to it. (This string, which is itself a name, e.g., CHAR1, has now been assigned as the contents of CHAR).

In addition to the space, EASEy uses the end-bracket ( ]),
semi-colon (;), and colon (:) as delimiters. These must be spec-
ified (as in statement 5). The programmer can use other symbols 5
for delimiters, but they must be enclosed in quotes. Finally, a general delimiter, the pound (#) can be used, which will match
any particular delimiter, including itself (see statement 9). 9

When the # delimiter is used, the delimiter is returned to the list if the name it delimits is returned (e.g., a # preceding
NAME is returned to MAYBE in 9), and the first name on a list 9
will be got whether it actually has a delimiter preceding it or not
(9), and the last name will be similarly treated with respect to 9
its end bound - that is, the bounds of the list are treated like delimiters.

6. Variable Names: A string of symbols that comes after get is treated as a name to be assigned some contents. It will be assigned the string in the named list up to the next space delimiter, unless it is followed by a specified object (that NAME), a

literal object, or a specified delimiter ( ], :, ; or #) in which case it is assigned the string up to that object. Till end will assign the rest of the list, till its end, to the variable name.

E.g.: From SENTENCE get MODIFIER, NOUN till ' IS ' 4,5,9
+ OBJECT till end

7. Matching from the Start of the List: at start of insists that the match begin at the very start of the list.

E.g.: at start of SENTENCE get 'THE' 7
looks for 'THE' only at the very start of the SENTENCE.

8. Specifying the Length of a String: call length symbols (name) will get a string exactly length symbols long, and assign the string following the word symbols as its name.

E.g.: from PATTERN get and call N + 6 symbols PIECE 7
will assign PIECE as the name of the first N + 6 symbols in PATTERN.

C. Functions

1. Arithmetic is handled in the conventional way. Parentheses are not needed if ordinary precedence of operators is desired. + = add, - = subtract, * = multiply, / = divide, ** = exponentiate.

E.g.: set WEIGHT = WEIGHT + 100 / WEIGHT 9,11

2. Tests for inequalities are of the form: is (Object1) test (Object2)? The tests are a) numeric: greaterthan, lessthan, or equalto and b) string-matching: sameas.

E.g.: is SUM lessthan THRESHOLD? 11

3. The built-in function size((object)) will count the symbols in the object (if it is a literal) or the list named (if the object is a name).

The function random((number)) will get a random number between

1 and the number specified.

4. The user can define his own functions by saying define: followed by the function, with its arguments as they are named within the function. When the function name is then used in a statement, the program goes to the statement with that name as its label, executes the function, and exits using return and freturn (for failure) in gotos.

E.g.: DEFINE: ABS(EXPRESSION)
(The code for the ABSolute value function must be written by the programmer, e.g.:

```
ABS            at start of EXPRESSION get '-' = [return]
```

D. Flow of Control.

1. A label can be used to name a statement. All labels must start in column 1. No two statements can have the same label. 1,3

2. Statements are tied together by gotos at the right of the statement which name labels at the extreme left of the statement to be gone to. Unconditional gotos are of the form: [goto (label)]. 10⇒8
Gotos conditional on the success or failure of the statement (either a pattern match or a test) are specified by [+to (label)] and 9⇒11
[-to (label)]. Alternately, parentheses, and succeedto +, or sto, 11⇒8
and failto, -, or fto, can be used.

3. A program statement too long for one line can be continued by starting the next card with a plus, space ('+ ') in column 1. 7

4. A comment card must start with a left parenthesis ('(') in column 1. C1

5. A program must be followed by a card that has end in its first three columns. Optionally, a goto can be given, to specify the 13
first statement to be executed; otherwise execution starts with the first statement in the program.

6. An EASEy program is a) a sequence of statements (each card can contain up to 72 columns; the last 8 columns are reserved for identification), b) an end card, and c) cards with data that will be input (all 80 columns can be used). 11

E. Flexible Constructs

1. A number of words and punctuation marks are ignored, so that they can be used as filler by the programmer, to make statements easier to read. (The programmer cannot use these words to name lists.) These include the words (when between two spaces) at, and, into, it, its, next, no, of,the, till, yes, and the punctuation marks (only when followed by a space) . , - and , .

2. Several spacing variants are allowed: a) One or more spaces must bound all names and objects, except b) No spaces are needed in gotos, or around arithmetic operators when within parentheses.

3. A number of alternate forms are possible, as shown in the summary which follows. These need not be examined when reading code. But they allow a user to write code using contructs that he finds congenial. And they allow for code that is a good bit more compact (by sacrificing mnemonics, which are a help in reading but can become cumbersome in writing).

III. Summary of EASEy-2 Constructs

A. Basic Statement Types for Manipulation of List Structures:

1. Build lists:

a.	Assign strings:	set	name	=	objects
b.	Add: (at end)	on	name	set	objects
c.	Add: (at start)	at start of	name	set	objects
d.	Assign lists:	list	name	=	objects
e.	List: (at end)	on	name	list	objects
f.	List: (at start)	at start of	name	list	objects

2. Get, erase, replace:

a.	Get:	from	name	get	objects
b.	Get and erase:	from	name	get	objects erase
c.	Get and replace:	from	name	get	objects1 replace by objects2
d.	Erase:	erase	names		

3. Input and output:

a.	Input:	input	objects	(inputs one data card)
b.	Output:	output	objects	

B. Types of Objects Used

1. Names: alphanumeric strings
2. Literals: strings surrounded by quotes
3. Specified objects: that name specifies the contents of the name
4. Indirect and compound names: $name, $(name literal...)
5. Delimiters: for breaking out names from lists
   - a. one space (handled automatically)
   - b.] (or) : (or) ; (must be specified)
   - c. # (matches any delimiter or bound)
6. Variable names: to be assigned contents up to either
   - a. if a literal or specified object or delimiter follows, that object;
   - b. if end or till end follows, the end of the list
   - c. otherwise the next delimiter space
7. To match from the start: at start of name get objects
8. To specify length: from name get and call length symbols name

C. Functions:

1. Arithmetic: +, -, *, /, **. E.g.: RESULT = A + B - C * D / E ** F
2. Inequalities: is number1 ineq number 2?
   (where ineq is greaterthan, lessthan, equalto)
   is object1 sameas object2? (objects must match exactly)
3. Built-in:
   - a. size(objects) (counts symbols)
   - b. random(number)
4. User defined: define: FUNCTION(Arguments)

D. Flow of Control:

1. Labels start statements at the left, in column 1.
2. Gotos at the right in brackets name labelled statements to be branched to:
   - a. Always: [goto label]
   - b. On success: [+to label]
   - c. On failure: [-to label]
3. Continuation cards start '+ ' (or) '. '
4. Comment cards start '(' (or) '*'
5. end starts the card that ends the program
6. Program structure:
   - a) Program (72 cols);
   - b) end card;
   - c) data (80 cols).

E. Flexible Constructs:

1. Filler words that are ignored: 'at', 'and', 'into', 'it', 'its', 'next', 'no', 'of', 'the', 'till', 'yes', '.', ',', '-'.
2. One or more spaces must bound names and objects, except gotos and arithmetic operators in parentheses. '=' can replace 'erase' or 'replace by'.
3. Equivalent alternate forms:
   - a. start ≡ <
   - b. that ≡ >
   - c. $ ≡ what/s under
   - d. is number1 ineq number2 ≡ ineq(number1, number2);
   - e. erase name ≡ name =
   - f. greaterthan ≡ greater than ≡ GT
   - g. lessthan ≡ less than ≡ LT
   - h. sameas ≡ same as ≡ ident
   - i. +to ≡ succeedto ≡ sto ≡ yesto ≡ +
   - j. -to ≡ failto ≡ fto ≡ noto ≡ -
   - k. to ≡ goto ≡ (nothing)
   - l. end ≡ ## ≡ % ≡]]
   - m. get ... erase ≡ pluck ≡ pl ≡ take
   - n. = ≡ replace by (or) erase
   - o. from ≡ in ≡ on
   - p. get ≡ find
   - q. both get and find are actually optional (could be written get or find, and not used in the punched program)
   - r. call ≡ @
   - s. symbols ≡ @

F. Forbidden Words that the programmer cannot use:

1. filler words (see E.1. above)
2. system words (list, set, on, from, get, erase, replace, by, input, output, end, that, start, call, symbols).
3. inequalities and built-in functions (see C.2. and C.3. above)
4. (within the goto brackets only) the gotos (see D.2. above)

IV. Appendix

A. A Detailed Description of Program A

Program A is a fairly typical, albeit simple, pattern recognizer.

Statements M1 through MN *INITialize the program's memory, setting each CHARacterizerI to contain a DESCRiption (5*) of the specified HUNKs to be looked for, and their exact LOCATIONs (6-7), in the unknown pattern, and the IMPLIEDS (5) NAMEs and their WeighTs (8) to be merged into the MAYBE list (8-10) if the entire DESCRiption is found.

Statement 1 sets the LOOKFOR list to contain the names of all the CHARacterizersI in memory, 2 erases the MAYBE list, to initialize it, and 3 inputs the unknown pattern from the next data card in memory, up till the first '/'. 4 gets and erases the next CHARacterizer from LOOKFOR, failing to 13, to indicate failure when no more are left. 5 gets the DESCRiption and IMPLIEDS from the string stored in the characterizer name stored in CHAR. 6 gets and erases each HUNK and its LOCATION from the DESCRiption, failing to IMPLY when no more are left. 7 gets LOCATION symbols, from the start of the PATTERN, and tries to get that HUNK at that point (if it succeeds it goes to R1, to get the next HUNK; if it fails, it gives up on this characterizer and goes to RESPOND, to get the next characterizer).

Statement 6 fails to statement 8 if all HUNKs have been found in their specified LOCATIONs - that is, if the characterizer has succeeded. 8 gets and erases each next NAME and its WeighT from IMPLIEDS. 9 sees if that NAME and its SUM of weights is already on MAYBE and, if it is, replaces it by SUM + WeighT (to add the new WeighT of the new implication of this name into the grand SUM), and goes to TEST whether to choose this name. If not, 10 lists the new NAME and its WeighT on

*Caps refer to program constructs. Numbers refer to statement numbers.

MAYBE. 11 TESTs whether the new SUM + WeighT is greaterthan 30 (a pre-set level for choosing) and, if it is, 12 outputs that the PATTERN ' IS A ' NAME (which contains the chosen name). 13 is the end card that shows the program has ended, and I1- I3 are three examples of simple unknown patterns that might be input to the program.

B. The Relationship Between EASEy and SNOBOL

Essentially, EASEy is a variant of a simple subset of SNOBOL4. Enough SNOBOL4 constructs have been taken to make a general purpose programming language. These include the basic pattern-matching and pattern-manipulation constructs that make SNOBOL so powerful as a language processor, and also constructs that handle arithmetic expressions, inequalities, and programmer-defined functions.

These SNOBOL4 constructs have been changed, to make them more understandable to a reader who does not know SNOBOL or, for that matter, has not been exposed to programming languages or computers. Since English is our common tongue, EASEy is chiefly in English, but with a few pieces of jargon for constructs that are too awkward when expressed in English. (E.g., "What/s under"-- which serves for indirect pointing -- can more succinctly be expressed by "$".)

The use of EASEy as a list-processing as well as a pattern-matching language was emphasized and enhanced by the addition of several constructs that set up, access and manipulate lists of objects in a convenient way. (E.g., "list (name1) = (name2) (name3)" will put name2, 1 space, name3, 1 space as the contents of the string named name1.) Then "from (name1) get (object1)" will look for a space, and assign the string up to that space (that is, name1) as the contents of object1. The additional delimiters ( ], :, ; and #) give further power.

EASEy also uses mnemonics to make its statements more understandable to the untrained reader. E.g., the SNOBOL statement:

(name1) (name2) =

is equivalent to the EASEy statement:

from (name1) get (name2) erase

Finally, EASEy allows some flexibility in the way the same statement can be coded. A number of alternate synonymous constructs are allowed. (E.g., either "erase" or "=" can be used; from (name1) get is equivalent to (name1) get or on (name1) find.) And a number of filler words that are ignored by the system are allowed, to improve readability. (E.g., "and", "its", "the".)

To summarize: EASEy takes a simple subset of SNOBOL constructs, tries to make them understandable to the non-programmer, adds some list-processing constructs, and accepts a number of alternate ways of saying the same thing.

These changes are designed to make programs easier to read, so that we can begin to communicate about complex programs at the concrete level of the programs themselves. The logic of the program itself will often remain difficult. But EASEy allows the reader to confront the real program difficulties, as though through a relatively clear glass of the programming language, rather than have to worry about the peculiarities of the programming language.

A Quick Comparison of EASEy and SNOBOL4 Constructs that Differ

EASEy	SNOBOL4 (Equivalent statement)
A. Statements for Pattern Manipulation:	
set A = B C	A = B C
list A = B '(' C	A = B ' ' '(' C ' '
on A set B C	A = A B C
on A list B C	A = A B ' ' C ' '
at start of A set B C	A = B C A
at start of A list B C	A = B ' ' C ' ' A
from A get B C	A BREAK(' ') . B ' ' BREAK(' ') . C
from A get B C erase	A BREAK(' ') . B ' ' BREAK(' ') . C
from A get that B	A B
from A get # that B # C	A '(' B ')' BREAK(' ') . C ' '
+ = B C D	+ = '(' B ')' C ' ' D ' '
B. Names and Other Objects Used in Patterns:	
1. Variable names:	
from A get B	A BREAK(' ') . B
2. Defined names:	
from A get that B	A B
3. Fixed-length variable names:	
from A get and call N symbols B	A LEN(N) . B
C. Other Constructs: GOTOs, Comment Cards	
1. GOTOs:	
[+to LABELA. -to LABELB]	:S(LABELA)F(LABELB)

2. Comment cards start with '(' (or '*') rather than '*'.

C. <u>VARIANT AND SHORT FORMS OF EASEy-2, FOR EASIER CODING</u>

EASEy-2 is designed primarily as a tool for presenting programmed models, so the crucial thing has been to make it as easy as possible to <u>read.</u> The primer emphasizes what is pretty much the standard form of EASEy, the form that I have used when coding programs, because in my judgment it comes as close as possible to being self-explanatory.

But the complete EASEy system includes additional variant forms, including short symbols that can be used to replace some of the mnemonically self-explanatory constructs, like "<u>start</u>" and "<u>call</u>". These are of special use and importance for <u>writing</u> code in EASEy. The variant forms give the coder a certain amount of flexibility, and naturalness, in saying things the ways that come most easily to him. And the short forms allow for more compact code.

The first way EASEy can be varied is by the elimination of the filler words (like "and" and "from") and the constructs that are not necessary (those without underlines, like "set" or "goto"), keeping only those constructs that are indicated as necessary, by underlining. These we have already seen.

The second way is by using any of the synonymous constructs that are summarized in section III.E of the EASEy-2 primer, above.

Note in particular that this allows the programmer to use forms that are quite short and succinct. For example,

```
     at the start of LISTA get and call N + 3 symbols OBJECTA,
+          and the REST till the end erase.
```

can be replaced by the equivalent statement:

```
          < LISTA pluck @ N + 3 @ OBJECTA REST %
```

Such flexibility might well make EASEy an alternative to SNOBOL or LISP worth considering by somebody who has access to, and money to spend for translation to, a SNOBOL system.

References

1. Bobrow, D. J. and Raphael, B., New programming languages for AI research, Presented at 3d IJCAI, Palo Alto, 1973. (To appear in Computing Reviews.)

2. Farber, D. J., R. W. Griswold, and I. P. Polonsky, SNOBOL, a string manipulation language, J. Assoc. Comput. Mach., 1964. 11, 21-30.

3. Griswold, R. W., J. F. Poage and I. P. Polonsky, The SNOBOL4 Programming Language (2d Ed.) Englewood-Cliffs, N.J.: Prentice-Hall, 1971.

4. Hewitt, C., Procedural embedding of knowledge in PLANNER, Proc. IJCAI 2, 1971, 167-182.

5. Le Faivre, R., Fuzzy: A programming language for fuzzy problem-solving, Computer Science Dept. Tech. Rept. 202, Univ. of Wisconsin, 1974. (To appear, in modified form, in L. A. Zadeh et al., eds. Fuzzy Sets, New York: Academic Press, 1975.)

6. Newell, A. et al., Information Processing Language V Manual. The RAND Corporation Tech. Report P-1897, Santa Monica, California, 1960.

7. Nilsson, N., Problem-Solving Methods in Artificial Intelligence, New York: McGraw-Hill, 1971.

8. Reboh, R. and E. Sacerdoti, A preliminary QLISP manual. Artificial Intelligence Center Tech. Note 81, Stanford Research Institute, 1973.

9. Sussman, G. J. and McDermott, D. V., Why conniving is better than planning, Proc. AFIPS FJCC, 1972., 41, 1171-1180.

10. Uhr, L., Describing, Using "recognition cones". Proc. 1st Int. Joint Conf. on Pattern Recognition, Washington, 1973.

11. Uhr, L., Layered "recognition cone" networks that pre-process, classify and describe. IEEE Trans Computers, 1972, 21, 758-768.

12. Uhr, L., Pattern Recognition, Learning and Thought. Englewood-Cliffs, N.J.: Prentice-Hall, 1973.

13. Uhr, L., The description of scenes over time and space, Proc. AFIPS NCC, 1973, 42, 509-517.

14. Weissman, C., et al., LISP 1.5 Primer, Belmont, Calif.: Dickenson, 1967.

15. Yngve, V. H., et al., An introduction to COMIT Programming. Cambridge, Mass.: MIT Press, 1961.