WIS-CS-175-73
University of Wisconsin
Computer Sciences Department
1210 West Dayton Street
Madison, Wisconsin 53706

"FLEXIBLE" PATTERN RECOGNIZERS ARE ALSO CONCEPT FORMERS

by

Leonard Uhr

Technical Report #175

March 1973

Received March 2, 1973

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.

"FLEXIBLE" PATTERN RECOGNIZERS ARE ALSO CONCEPT FORMERS

by

Leonard Uhr

ABSTRACT

The typical pattern recognizer (PR) applies a set of characterizers to an input. Each characterizer implies a set of possible names, and the single most highly implied name is chosen.

The typical concept former (CF) applies a binary test to the input. This test implies either another test to apply, or a name to output.

Pattern recognizers have almost always applied probabilistic (usually two-valued but occasionally multi-valued) characterizers in parallel; whereas concept formers have always applied deterministic two-valued tests, in series.

This paper presents and examines simple computer programs (coded in EASEy-2, a language that is relatively easy to understand) for (1) parallel "pattern recognition" (NAMER), and (2) serial "concept formation" (CONCEIVER).

Finally, these programs are generalized to give a single "flexible" pattern recognizer-concept former (FLEXIBLE PR-CF) that combines the desirable features of parallel-probabilistic and serial-deterministic systems.

"FLEXIBLE" PATTERN RECOGNIZERS ARE ALSO CONCEPT FORMERS

by

Leonard Uhr

INTRODUCTION

Pattern recognizers (PR NAMERS) and concept formers (CF CONCEIVERS) have always appeared to have a lot in common. Yet they have been developed in quite separate lines of research, since each concentrates upon a different problem area (roughly, perception versus cognition), and takes a different approach.

This paper first examines their differences, and then their striking similarities, by presenting, explaining, contrasting, and comparing actual computer programs for each. Then a third "FLEXIBLE" program is presented, one that both recognizes patterns and forms concepts.

The EASEy Programming Language. The programs are coded in *EASEy-2 (an Encoder for Algorithmic Syntactic English that's Easy, version 2--See Uhr, 1973a), an English-language variant of SNOBOL that's designed to be as close as possible to a self-explaining list-processing language.

The programs are bare-bones, kept as short as possible so that we can examine exactly what is happening,

---

*An EASEy-2-to-SNOBOL4 translator has been coded in SNOBOL4 (Uhr, 1973a). So any EASEy program can be translated into SNOBOL and then run on any computer with a SNOBOL compiler.

and make detailed comparisons of small changes, and of their effects. They must be given (or-another story-learn) the particular characterizers needed to recognize the objects expected, or the particular tests needed to identify the desired concept--just as a parser must be given the particular set of rewrite rules that defines the grammar.

Pattern Recognition. Many hundreds of pattern recognition (PR) programs have been written, to handle a wide variety of problems, e.g., letters of the alphabet, spoken words, simple objects, bubble chamber photos, chromosomes, blood cells, and cartoon faces (e.g., Grimsdale, et. al., 1959; Uhr and Vossler, 1963; Andrews, et. al., 1968; Zobrist, 1971; Reddy, 1967; Ledley, 1972. See Uhr, 1973b chapters 1-9). These programs attempt to handle the very difficult problems of recognizing objects when they are distorted in any of the potentially infinite number of unknown and non-linear ways in which real-world objects can vary.

Virtually all pattern recognizers have the following simple basic structure: A set of characterizers is applied to the unknown input pattern, where each characterizer specifies one, or a set of, tests. The characterizer's outcome implies one, or a set of, possible names. The program chooses the single most highly implied name, after combining the outcomes from all the characterizers applied.

Usually characterizers are two-valued: they either succeed or fail. Sometimes they are many-valued, where each outcome implies a different set of possible names. Sometimes characterizers imply transforms that can then be used as inputs for subsequent characterizers (e.g.,

Selfridge and Neisser, 1960). Sometimes the pattern is first "pre-processed," to smooth, eliminate noise, enhance edges, and in other ways regularize. But we will ignore pre-processing, since it is often absent, and in any case can be replaced by additional transforming characterizers.

Most pattern recognizers apply a whole set of characterizers in one parallel look at the input (e.g., Doyle, 1960; Andrews, et. al., 1968); occasionally a system will effect a series of parallel passes (e.g., Uhr and Vossler, 1963); sometimes each characterizer is applied in series (e.g., Unger, 1958, 1959). Usually weights or probabilities are associated with the various implied names, and these are combined and the most highly implied name chosen. But a serial system like Unger's tries to do without weights.

Sometimes characterizers imply other characterizers to apply--giving what I have called "flexibility" (Uhr, 1969, 1973b).

Concept Formation. Concept formers (e.g., Kochen, 1960; Hunt, 1962; Towster, 1969) have been developed chiefly to model psychological research on cognition, where a large number of experiments of the following sort have been run: A sequence of cards is presented to the subject, where each card contains a simple picture, e.g., of a big red solid triangle, or a small blue outlined square. The picture is in fact a collection of dimensions, such as shape (triangle or square), color (red or blue), size (big or small), type (solid or outlined), saturation (bright or hazy). In almost all cases the dimensions are two-valued (e.g., triangle or square), and in all cases the two values are chosen to

be easily discriminable, and the different dimensions are also chosen to be as different and as discriminable as possible.

A typical experiment will use from 3 to 10 dimensions. The problem is to say whether a picture is YES--a member of the concept class--or NO--not a member (e.g., the concept might be [small and square] or [not [small and blue] and triangle]).

All concept formers apply one test at a time, where each test implies two different possible outcomes, on success or failure. Tests in series are "and-ed" together; branching paths are "or-ed." Since concepts are binary a negation can be replaced (albeit awkwardly) by its opposite. The chosen outcome can be either a YES or NO, which is the program's overall decision, or another test to apply to the input picture. This is a strictly serial process; in fact it is the typical form of a discrimination net. It is also similar to the serial net of features used in the "Elementary Perceiver and Memorizer" (EPAM) developed by Feigenbaum (1963) to model human memory.

Pattern Recognition versus Concept Formation. Pattern recognizers apply many characterizers, in parallel, combine their weighted outcomes, and make a probabilistic choice. They look at inputs in two dimensions, with many hundreds, or thousands, of cells, and must be able to assign the same name to variant examples of the same pattern class, where the distortions are complex and non-linear in unknown ways, and the possibilities are potentially infinite. Pattern recognizers typically handle 26 different pattern classes, sometimes only 5, and occasionally as high as 200. Concept formers handle only the two classes, YES and NO.

Concept formers apply one test at a time, in series, where each test specifies a dimension and an expected value, and its outcome (either success or failure) implies either another test to apply, or the name choice to output. They look at inputs with 3 to 10 binary-valued dimensions, where the concept can be any logical combination of these dimensions.

Pattern recognizers must be able to ignore slight distortions and bits of noise; in general, their characterizers (which can be any codable functions, but are often configurations of tests and features) must be good at throwing away, as well as extracting and abstracting, information. Real-world patterns vary and overlap with other patterns in so many unknown ways that it is virtually impossible--and in any case not worth the effort--to design perfect characterizers. Rather, most researchers have opted for a large parallel set of as-good-as-discoverable characterizers, where each is assigned weights that reflect its goodness, and a grand decision is made over many. Thus individual mistakes are expected, with the hope that they will be washed out by the general consensus.

Learning Patterns and Formulating Concepts. This paper examines only the recognition of the pattern or the concept, and not the actual learning of the characterizers and tests that are used. Concept formulators (for a detailed examination see Towster, 1969) generate and add a new test to the end of a path in their discrimination nets, when feedback indicates they have made a wrong choice. The test is generated to correct this error on this input. Pattern recognizers that have been coded to discover new characterizers use similar techniques to generate (except that the input pattern is far larger

than a concept, so that it is harder to find good information-rich regions, and often several tests are compounded into a single characterizer), and weights are used to assess the worth of a newly generated characterizer. (E.g., Uhr and Vossler, 1963; see Levine, 1969.) For a discussion and preliminary examination of recognizers that attempt to discover good characterizers to apply in a parallel-serial structure, see Uhr, 1973b, chapters 16-21. But learning is another story, and beyond the scope of this paper.

A TYPICAL CONCEPT FORMATION PROGRAM

The following computer program (called CONCEIVER) shows how a typical concept former works. CONCEIVER must have a set named *TEST1, TEST2,...TESTN of tests it has built up in formulating the concept (statements M1-MN). Each test specifies (1) a DIMENSION (e.g., shape), (2) the expected VALUE (e.g., triangle) and the names of (3) the next test (YESDO) to apply if the test succeeds, and (4) the next test (NODO) to apply if the test fails. (Statement 3 gets these four parts from the test currently in LOOKFOR.)

The next example is input and stored in LINE by statement 1, and LOOKFOR is initialized (2) to contain the first test, TEST1. The test is applied (4) to the current example. Statements 5 and 6 store the new test in LOOKFOR, and 7 checks for an ANSWER--a YES or a NO (indicated by a star--*), which is output (8). Otherwise it returns to RESPOND (3), to get and apply the next test.

---

*Caps refer to names in the programs, numbers to statement numbers at the right.

Input examples must all be of the form shown by the two Input data cards that follow CONCEIVER's end card. Each dimension must be followed by its value. Spaces must surround every dimension and value, but the dimension-value pairs can be listed in any order.

Figure 1

CONCEIVER: A Simple Program for Concept Formation

```
(CF CONCEIVER Program. Applies TESTs in series until a name is        CF
(reached, and output.
START*  TEST1 = 'SHAPE  SQUARE  TEST3  TEST2 '                       *M1
        TEST2 = 'COLOR RED  A1  TEST4 '                              M2
        A1 = '*YES'                                                  .
(More TESTs must be put, or learned, into Memory.                    .
        TESTN =                                                      MN
(INput the next example (stored in LINE) and put 'TEST1' in
(LOOKFOR
IN      input LINE till ] [-to END]                                  1
        set LOOKFOR = 'TEST1 '                                       2
(From the name stored in LOOKFOR get the next DIMENSION and its
(VALUE
RESPOND from $LOOKFOR get DIMENSION VALUE YESDO NODO                 3
(See if that DIMENSION and VALUE can be found in the example
(in LINE.
        from LINE get # that DIMENSION # that VALUE # [-R1]          4
(Put YESDO if succeed, or NODO if fail, into LOOKFOR.
        LOOKFOR = YESDO [TEST-OR-OUT]                                5
R1      LOOKFOR = NODO                                               6
(If LOOKFOR starts with * this is a choice; output it.
TEST-OR-OUT    at start of $LOOKFOR get '*' ANSWER [-RESPOND]        7
        output LINE ' IS ' ANSWER [IN]                               8
end     [goto start]                                                 -
COLOR  RED  SIZE  BIG  SHAPE  TRIANGLE  TYPE  OUTLINED ]            I1
SHAPE  SQUARE  COLOR  BLUE  TYPE  SOLID  SIZE  BIG ]                 I2
```

*See the Appendix for a brief description of the EASEy language, and Uhr, 1973a for more details.

Note how the tests link, through the YESDO and NODO branches, to form a discrimination net (actually, a tree is almost always used). Thus TEST1 asks that the SHAPE be SQUARE, in which case TEST3 should be tried, but otherwise TEST2 should be tried.

Let's follow through an example. Statement 1 will input the first example, stored in Input card I1, 2 puts the name 'TEST1' into LOOKFOR, and 3 gets DIMENSION = SHAPE, VALUE = SQUARE, YESDO = TEST3, and NODO = TEST2. Statement 4 will fail to find that spefied DIMENSION (that is, 'SHAPE') and that VALUE, so 6 will store TEST2 (the contents of NODO) in LOOKFOR. Since 7 does not find a '*' it returns to RESPOND. TEST2 gives DIMENSION = COLOR and VALUE = RED, which is found, putting A1 (in YESDO) into LOOKFOR, so that 7 now finds the '*' signal that 'YES' be output (8). Thus "not square and red" is a path of tests that leads to the conclusion, "yes" (this is a member of the concept).

Of course many more tests would be used, and longer paths would have to be taken, for interestingly difficult concepts. But notice how CONCEIVER moves serially along a path through its discrimination tree, with each test implying the branch to take to the next test, until a terminal name (signaled by '*') is reached.

CONCEIVER handled negations by using a test for the opposite value (e.g., the negation of COLOR RED is COLOR BLUE). This is possible because all concepts are assumed to be two-valued. But it could easily handle negations directly, by expanding the test to indicate whether it should or shouldn't be found, and if it shouldn't reversing statements 5 and 6 (putting NODO into LOOKFOR if it was found).

A TYPICAL (ALBEIT SIMPLE) PATTERN RECOGNITION PROGRAM

Let's look now at a pattern recognition program (called NAMER) that has been somewhat oversimplified to make it more comparable to a concept former like CONCEIVER. NAMER handles only 1-dimensional patterns (a gross oversimplification, but we will end with a 2-dimensional pattern recognizer-concept former, FLEXIBLE, that both names and conceives). NAMER's characterizers look only for a single test, in DESCRiption (5-6), rather than the typical configuration of tests and other functions. But each successful characterizer implies any number of possible names to output (e.g., M1, M2), each with an associated weight. These names are combined into a MAYBE list (7-10), from which a CHOICE is made (11-14) and output (15).

All characterizers to be applied are put on MEMORY (M1) and then on LOOKFOR (3). NAMER keeps looping through them (4-6, 7-10) until LOOKFOR has been emptied. Then 11-14 get the HIWeighTed CHOICE, and 15 outputs it.

Figure 2

NAMER: A Program for Pattern Recognition

```
(PR NAMER Program. 1-Dimensional inputs. Outputs NAME if                CF    PR
(CHOOSE level is reached.
(CHARacterizers must be put (or learned) in Memory.  E.g.:
START   MEMORY = 'CHAR1 CHAR2 ... CHARN '                                     M1
        CHAR1 = '1111100,0000 I 9 T 7 E 3 ]'                          M1.V    M2
        CHAR2 = '000100 I 5 T 4 E 2 ]'                                M2.V    M3
        CHAR3 = ',0001111 ' E 9 I 1 T 2 ]'                            M3.V    M4
        CHARN =                                                       MN.V    MN
        CHOOSE = 30                                                           M(N+1)
INIT    erase CHOICE                                                          1
(INput one pattern (stored in LINE) and put all CHARacterizers                .
(in LOOKFOR                                                                   .
IN      input LINE till ] [-to END]                                   1       2
        set LOOKFOR = MEMORY                                          2.V     3
RESPOND from LOOKFOR get CHAR = [-OUT2]                               3.V     4
(From the name stored in CHAR get the DESCRiption and IMPLIEDS
        from $CHAR get DESCR IMPLIEDS ]                               3.V     5
        from LINE get that DESCR [-RESPOND]                           4.V     6
(If that DESCRiption was found in LINE, merge IMPLIEDS NAMEs
(into MAYBE
IMPLY   from IMPLIEDS get NAME WT = [-CHOOSE]                                 7
        from MAYBE get # that NAME # TOTAL = [I2]                             8
        WT = TOTAL + WT                                                       9
I2      on MAYBE list NAME WT [IMPLY]                                         10
(CHOICE is the single most highly implied name on MAYBE
CHOOSE  from MAYBE get CHOICE HIWT = [- OUT]                                  11
CH2     from MAYBE get  ORNAME ORWT = [-OUT]                                  12
        is HIWT lessthan ORWT ? yes - HIWT = ORWT [-CH2]              7.V     13
(ORNAME becomes the present CHOICE since its ORWT exceeds HIWT
        CHOICE = ORNAME [CH2]                                                 14
OUT     output 'I CHOOSE ' CHOICE [INIT]                              8.V     15
end     [goto START]                                                          -
001111100,000010000,000110000,000010000,001111100,]                          I1
000111110,000110000,000111100,000110000,000111111,]                          I2
```

Examples. It is too cumbersome to follow through an even faintly realistic example of a pattern recognizer's workings, since several dozen, or hundreds, of the relatively weak characterizers we are using would be needed. But note how an Input card can represent a 2-dimensional matrix (I1 and I2 Input an I and an E, each in a 4 by 9 matrix, with a comma (',') following each row). Both CHAR1 and CHAR2 will succeed on I1, so that MAYBE will contain (NAME WeighT) 'I 14 T 11 E 5 ', and the CHOICE will be 'I'. Only CHAR3 succeeds on I2, so that MAYBE contains 'E 9 I 1 T 2 ' and the CHOICE will be 'E' (but note that CHAR1 almost succeeds).

Note how statement 5 gets the DESCRiption (= '1111100,0000') and IMPLIEDS (= 'I 9 T 7 E 3 ') from CHAR1, after 4 got CHAR1 (CHAR = 'CHAR1 ') from LOOKFOR. Then 6 gets that DESCRiption in LINE (which contains I1).

A SINGLE PROGRAM FOR PATTERN RECOGNITION AND CONCEPT FORMATION

We are now ready to generalize, to a single program (PR-CF FLEXIBLE) that does both pattern recognition and concept formation. But first the reader might want to compare CF CONCEIVER and PR NAMER, by using the statement numbers for CF (where e.g., 2.V means a Variant of statement 2) as given next to the PR statement numbers (in the right margin).

The FLEXIBLE Program Described

FLEXIBLE inputs 2-dimensional patterns LINE by LINE (2), and stores each LINE under its NROW number (3-4). (The 1st LINE must start 'I ' and the rest 'S '

followed by one card that reads 'R]'). It uses CHARacterizers whose DESCRiptions contain a whole set of TESTs, each with a ROW and a COLumn position, and a weight (9). Each TEST is looked for in the ROW and at the COLumn specified (10) and if it is found its WeighT is added to the TOTAL (11).

After all tests have been applied, 12 loops through each threshold BOUND and its associated IMPLIEDS on ALLIMPLIEDS, and 13 sees whether the TOTAL weight of parts got has reached this BOUND level. (Note how this uses a descending sequence of lower BOUNDs, rather than, as in NAMER, a single THRESHold bound--thus giving multi-valued characterizers.) If the BOUND has been reached, 14 gets each NAME and its WeighT and TYPE from IMPLIEDS, and goes to the label stored in TYPE.

If TYPE = N, FLEXIBLE acts just like PR NAMER: 16-18 merge the NAME onto MAYBE.

If TYPE = L, the implied NAME is listed on LOOKFOR, since it is a characterizer, and not an output name. This is the basic feature of what I have called "flexible" pattern recognizers (Uhr, 1969), where characterizers can imply either names to output or characterizers to apply--which gives a parallel-serial process that dynamically tailor-makes the specific set of characterizers it applies to a particular input pattern as a function of what has been uncovered so far about the input.

After all characterizers have been applied, 19-22 CHOOSE the HIWeighTed CHOICE, and 23 outputs it.

Figure 3

FLEXIBLE: A Program for Pattern Recognition and Concept Formation

```
(FLEXIBLE Program for Concept Formation and Focussed                CF      PR      PR-CF
(Pattern Recognition
START   CHAR1  = '1 2 1111 4 2 3 010 3 4 3 0111 3 5 ]
+              '3 0111 4 ]9  N E 12 N I 2 ]5  N I 8 ]
+              'L C23 X ]'
        MEMORY = 'CHAR1 CHAR2 ... CHARN '                                          M1
        CHAR2 = '2 1 00100 3 ]2 N  I 5 N T 4 N E 2 ]                               M2
          :                                                                        M3
        CHARN =                                                                    :
        TEST1 = '1 0 1 1 ]1 L TEST3 1 ]0 L TEST2 X ]'
        CHOOSE = 30                                                                MN
I       erase NROW, CHOICE                                          M(N+1)         M(N+1)
IN      input TYPE LINE till ] [+ to $TYPE. -to END]                  1.V          1
(sense and store the input LINE Row by Row.                           1.V   2.V    2
S       set NROW = NROW + 1
        set $('R'NROW) = LINE [to IN]                                              3
R       set LOOKFOR = MEMORY                                                       4
R1      from LOOKFOR get CHAR = [- OUT2]                              2.V    3     5
        erase TOTAL                                                         4      6
        from $CHAR get DESCR ] IMPLIEDS till end                                   7
(Get each TEST from DESCRiption, with its ROW, COLumn and WeighT.     3.V   5.V    8
R3      from DESCR get ROW, COL, TEST, WT = [- R2]
        at start of $ROW get and call COL symbols LEFT,                            9
+               get that TEST [- R3]                                  4.V   6.V   10
        TOTAL = TOTAL + WT [R3]
(Get each BOUND and IMPLIED and test against TOTAL weight                          11
R2      from ALLIMPLIEDS get BOUND IMPLIEDS ] = [- R1]
        is TOTAL lessthan BOUND ? [+ R2]                                           12
IMPLY   from IMPLIEDS get TYPE NAME WT = [+ $TYPE. - CHOOSE]                       13
L       on LOOKFOR list NAME [IMPLY]                                        7.V    1
N       from MAYBE get # that NAME # TOTAL = [- I1]                   5.V          1
        WT = TOTAL + WT                                                     8.V    1
        on MAYBE list NAME WT [IMPLY]                                       9
CHOOSE  from MAYBE get CHOICE HIWT = [- OUT]                                10
CH2     from MAYBE get ORNAME ORWT = [- OUT]                          7.V   11
        is HIWT lessthan ORWT ? yes - HIWT = ORWT [- CH2]                   12
        CHOICE = ORNAME  CH2                                                1
OUT     output 'I CHOOSE ' CHOICE [I]
end
```

Figure 3

FLEXIBLE: A Program for Pattern Recognition and Concept Formation

```
(FLEXIBLE Program for Concept Formation and Focussed          CF      PR      PR-CF
(Pattern Recognition
START   CHAR1  = '1 2 1111 4 2 3 010 3 4 3 0111 3 5 ]
+               '3 0111 4 ]9  N E 12 N I 2 ]5  N I 8 ]
+               'L C23 X ]'                                                    M1
        MEMORY = 'CHAR1 CHAR2 ... CHARN '                                      M2
        CHAR2 = '2 1 00100 3 ]2 N  I 5 N T 4 N E 2 ]                           M3
          :                                                                    :
        CHARN =
        TEST1 = '1 0 1 1 ]1 L TEST3 1 ]0 L TEST2 X ]'                          MN
        CHOOSE = 30                                          M(N+1)            M(N+1)
I       erase NROW, CHOICE                                          1.V        1
IN      input TYPE LINE till ] [+ to $TYPE. -to END]         1.V    2.V        2
(sense and store the input LINE Row by Row.
S       set NROW = NROW + 1                                                    3
        set $('R'NROW) = LINE [to IN]                                          4
R       set LOOKFOR = MEMORY                                 2.V    3          5
R1      from LOOKFOR get CHAR = [- OUT2]                            4          6
        erase TOTAL                                                            7
        from $CHAR get DESCR ] IMPLIEDS till end             3.V    5.V        8
(Get each TEST from DESCRiption, with its ROW, COLumn and WeighT.
R3      from DESCR get ROW, COL, TEST, WT = [- R2]                             9
        at start of $ROW get and call COL symbols LEFT,      4.V    6.V        10
+               get that TEST [- R3]
        TOTAL = TOTAL + WT [R3]                                                11
(Get each BOUND and IMPLIED and test against TOTAL weight
R2      from ALLIMPLIEDS get BOUND IMPLIEDS ] = [- R1]                         12
        is TOTAL lessthan BOUND ? [+ R2]                                       13
IMPLY   from IMPLIEDS get TYPE NAME WT = [+ $TYPE. - CHOOSE]                7.V 14
L       on LOOKFOR list NAME [IMPLY]                         5.V               15
N       from MAYBE get # that NAME # TOTAL = [- I1]                 8.V        16
        WT = TOTAL + WT                                             9          17
        on MAYBE list NAME WT [IMPLY]                               10         18
    CHOOSE  from MAYBE get CHOICE HIWT = [- OUT]                 7.V 11         19
    CH2     from MAYBE get ORNAME ORWT = [- OUT]                    12         20
            is HIWT lessthan ORWT ? yes - HIWT = ORWT [- CH2]       13         21
            CHOICE = ORNAME  CH2                                    14         22
    OUT     output 'I CHOOSE ' CHOICE [I]                    8.V    15         23
    end                                                      -      -          -
```

followed by one card that reads 'R]'). It uses CHARacterizers whose DESCRiptions contain a whole set of TESTs, each with a ROW and a COLumn position, and a weight (9). Each TEST is looked for in the ROW and at the COLumn specified (10) and if it is found its WeighT is added to the TOTAL (11).

After all tests have been applied, 12 loops through each threshold BOUND and its associated IMPLIEDS on ALLIMPLIEDS, and 13 sees whether the TOTAL weight of parts got has reached this BOUND level. (Note how this uses a descending sequence of lower BOUNDs, rather than, as in NAMER, a single THRESHold bound--thus giving multi-valued characterizers.) If the BOUND has been reached, 14 gets each NAME and its WeighT and TYPE from IMPLIEDS, and goes to the label stored in TYPE.

If TYPE = N, FLEXIBLE acts just like PR NAMER: 16-18 merge the NAME onto MAYBE.

If TYPE = L, the implied NAME is listed on LOOKFOR, since it is a characterizer, and not an output name. This is the basic feature of what I have called "flexible" pattern recognizers (Uhr, 1969), where characterizers can imply either names to output or characterizers to apply--which gives a parallel-serial process that dynamically tailor-makes the specific set of characterizers it applies to a particular input pattern as a function of what has been uncovered so far about the input.

After all characterizers have been applied, 19-22 CHOOSE the HIWeighTed CHOICE, and 23 outputs it.

Note that CHAR1 is a configuration of four parts, and a good bit more powerful than all three characterizers shown for NAMER. It will imply E with a high weight if most parts are found, but I with a high weight if fewer are found. CHAR2 is exactly like CHAR2 of NAMER. (A looseness of position for matching should be allowed, as described in Uhr, 1973, chapters 4, 20.) TEST1 shows the first concept formation characterizer, which acts exactly like TEST1 in CONCEIVER.

A skeptic might argue that this program is a good bit longer than NAMER, and therefore might be expected to handle CONCEIVER's job also. But all the additions serve primarily to improve upon pattern recognition. Thus statements 3 and 4 handle 2-dimensional patterns; while 7, 9, 11 and 13 handle weighted configurational characterizers--and neither of these serves any purpose for concept formation, which is 1-dimensional and deterministic.

Statement 12 gives multi-valued characterizers, which serve to handle the yes versus no branches of a concept former, as a very simple case. But they are also of great value for pattern recognizers, in increasing the power of their characterizers. And the no-branch could be handled in a simpler, and more ad hoc way, by pulling IMPLIEDS and NODO from the characterizer, and branching to NODO on failure (much as was done in CONCEIVER, statements 3 and 6).

Finally, statement 15 puts implied characterizers onto LOOKFOR, and thus makes possible the discrimination net. This is the one contribution from CONCEIVER (5.V). But it is also the key feature of "flexibility" and, in the context of a whole set of parallel characterizers

on the LOOKFOR list, gives pattern recognizers a great deal of new power, and flexibility, in focussing and directing attention.

Pattern Recognition

For pattern recognition-naming, FLEXIBLE behaves much like NAMER, except that it uses more powerful configurational characterizers, with locations, weights and bounds, it applies these to 2-dimensional inputs, and characterizers can imply characterizers to apply as well as names to MAYBE output.

Concept Formation

To handle concept formation, FLEXIBLE must start with a new statement 5:

```
set LOOKFOR = 'TEST1 '                                    5.V
```

to initialize LOOKFOR to contain only the single characterizer, named 'TEST1'. And we must recode the Input examples, to make them compatible with patterns.

Representing and Inputting the Concept. We do not need the richness of a 2-dimensional input to handle a concept, which can fit on a single Input card. Nor do we need to specify the concept by giving the dimension and the value each in its common name, like COLOR RED. Since it is now inconvenient to do so, let's recode the concepts, by assigning a positional number to each dimension, and a 1 to one of its values and a 0 to the other of its values: e.g.,

Shape	1	Color	2	Size	3	Type	4
triangle	0	red	0	big	0	solid	0
square	1	blue	1	small	1	outlined	1

Now we can replace the encoding for CONCEIVER:

SHAPE SQUARE COLOR RED SIZE BIG TYPE OUTLINED] by I1
1 1 2 0 3 0 4 1

Finally we can use the position in the string to indicate the positional numbers, giving:

1001] I1.V

This is instructive also because it makes clear how simple a concept is, in contrast to a pattern, with respect to the number of bits needed to encode it--although it should be remembered that it can be far more complex with respect to the crucial importance of each bit.

Tests are Simple Characterizers. The structure of the "tests" is now a very simple, almost degenerate, form of the structure of pattern characterizers: the DESCRiption contains only one TEST part, with a WeighT of 1 (WT = 1). ALLIMPLIEDS contains a BOUND of 1, with its single IMPLIEDS which is the no branch (NODO), as in:

TEST1 = '1 0 1 1]1 L TEST3 1]0 L TEST2 1]'
TEST2 = '1 1 0 1]1 N YES 31]0 L TEST4 1]'

which are equivalent to TEST1 and TEST2 in CONCEIVER. (E.g., TEST1 looks in ROW = 1, positioning through COLumn = 0, for a 1 (the present encoding of SHAPE = SQUARE). If found, the WeighT of 1 goes into TOTAL, which reaches the first BOUND of 1, so that TEST3 is IMPLIEDS and its TYPE = L puts it onto LOOKFOR.)

"Flexible" Pattern Recognition, and Concept Formation

If we do not impoverish FLEXIBLE's characterizers, to contain only one test in a description, with a weight of 1, and one bound of 1 for the yes branch and a second bound, of 0, for the no branch, we can get a great deal

of power and flexibility. The complex configurational characterizers of a pattern namer can be used, and each can imply not only a whole set of MAYBE names, but also a whole set of characterizers to LOOKFOR. (If this is done with abandon, FLEXIBLE should be improved to merge characterizers onto LOOKFOR, so that each is looked for only once; weights could also be merged, and the most highly implied characterizer looked for next.)

Concept formers can work deterministically only on very small inputs. It has been suggested that they be generalized and used for 2-dimensional patterns; but this would be explosively expensive. We can see how flexible techniques allow us to introduce parallel probabilistic processing, as the input size expands, from 4 to 10 to 100 to 1000 bits.

SUMMARY

The computer programs presented in this paper should make clear exactly what a concept former, and a pattern recognizer-namer, look like, and how they are related. The third program generalizes the first two, so that it both names and conceives. By adding what I call "flexibility" to a recognizer, so that characterizers can imply further characterizers to apply, and by using multi-valued, rather than 2-valued characterizers, we get a program that can also handle the concept former's set of tests, organized into a discrimination net. These tests become very simplified and stylized characterizers--ones that make virtually no use of weights or threshold bounds, specify configurations that contain only one part (the test), and imply only two different things (the yes and the no branch).

By using more of the riches available--configurations of many pieces, with weights, any number of threshold bounds, and any number of implieds, where these are mixtures of possible output names and characterizers to apply--such a "flexible" program can combine many of the advantages of both parallel and serial systems, focussing its attention on aspects of the input as a function of what it has uncovered so far.

REFERENCES

Andrews, D. R., Atrubin, A. J., and Hu, K. The IBM 1975 optical page reader: Part III: Recognition and logic development, IBM J. Res. and Devel., 1968, 12, 364-372.

Doyle, W. Recognition of sloppy, handprinted characters. AFIPS FJCC Proc., 1960, 17, 133-142.

Feigenbaum, E. The simulation of verbal learning behavior. In: Computers and Thought (E. Feigenbaum and J. Feldman, eds.), New York: McGraw-Hill, 1963, 297-309.

Grimsdale, R. L., Sumner, F. H., Tunis, C. J., and Kilburn, T. A system for the automatic recognition of patterns. Proc. Inst. Elect. Engrs., 1959, 106 (pt. B), 210-221.

Hunt, E. B. Concept Formation: An Information Processing Approach. New York: Wiley, 1962.

Kochen, M. Experimental study of "hypothesis-formation" by computer. Trans. 4th Sympos. on Info. Theory (C. Cherry, ed.). London: Butterworth, 1960.

Ledley, R. S. Analysis of cells. IEEE Trans. Comput., 1972, 21, 740-753.

Levine, M. Feature extraction: a survey. Proc. IEEE, 1969, 57, 1391-1407.

Reddy, D. R. Computer recognition of connected speech. J. Acoust. Soc. Amer., 1967, 42, 329-347.

Selfridge, O. G., and Neisser, U. Pattern recognition by machines and men. Scientific American, 1960, 203, 60-68.

Towster, E. Studies in Concept Formation. Unpubl. Ph.D. Diss., Univ. of Wis., 1969.

Uhr, L. A Primer for EASEy (An Encoder for Algorithmic, Syntactic English that's easy). Computer Sciences Dept. Tech. Report, Univ. of Wis., 1973a.

Uhr, L. Flexible pattern recognition. Computer Sciences Dept. Tech. Report 56, Univ. of Wis., 1969.

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

Uhr, L. and Vossler, C. A pattern recognition program that generates, evaluates and adjusts its own operators. In: Computers and Thought (E. Feigenbaum and J. Feldman, eds.). New York: McGraw-Hill, 1963, 251-269.

Unger, S. H. A computer oriented toward spatial problems. Proc. IRE, 1958, 46, 1744-1750.

Unger, S. H. Pattern recognition and detection. Proc. IRE, 1959, 47, 1737-1752.

Zobrist, A. L. The organization of extracted features for pattern recognition. Pattern Recognition, 1971, 3, 23-30.

APPENDIX

A Note on Programs (See Uhr, 1973a for details)

1. Numbering at the right identifies statements, and allows for comparisons between programs. M indicates initializing Memory statements: I indicates cards that are Input by the program. .V indicates a Variant, .1 an additional statement.

2. A program consists of a sequence of statements, an end card, and any data cards for input. (Statements that start with 1) a parenthesis are comments, and are ignored; with 2) a plus continue the previous statement.) Statement labels start at the left; gotos are at the right, within brackets (+ means branch on success; - on failure; otherwise it is an unconditional branch).

3. Strings in capitals are programmer-defined. Strings in underlined lower-case are system commands that must be present (they would be keypunched in caps to run the program). These include input, output, erase, set, list, get, start, call, end, symbols, that, and the inequalities. Other lower-case strings merely serve to help make the program understandable; they could be eliminated.

4. EASEy automatically treats a space following a string as though it were a delimiter; it thus automatically extracts a sequence of strings and treats them as names. The end-bracket] or semi-colon ; also act as a delimiter, but the programmer must specify it. The symbol # is used to stand for any delimiter (a space, ; ,] or #).

5. The symbol $stringI is used to indicate "get the contents of string I, and treat it as a name and get its contents" (as in SNOBOL).

6. Pattern-matching statements work just like SNOBOL statements: there are a) a name, b) a sequence of objects to be found in the named string in the other specified, c) the equal sign (meaning replace), and d) a replacement sequence of objects (b, c, and/or d can be absent). that stringI means "get that particular object"--otherwise a new string is defined as the contents of stringI, which is taken to be a variable name.

7. Functions are underlined, of the form: funct(...) or FUNCT(...). E.g., size(...) is a built-in function that counts the symbols in the string(s) named within parentheses. CHOOSE(LISTA,CLASS) is a programmer-coded function that chooses the most highly weighted object on LISTA that is a member of the designated CLASS.