MATHEMATICAL AND LOGICAL SEQUENCE IDENTIFICATION

Ashish Tiwari

May 7, 1996

1 Introduction

This is a documentation for the prototype expert system built by the author as part of the one semester long project for the course CSE 539, "Expert Systems" offered in the spring of 1996. This article essentially contains all information regarding the design and implementation of the *Sequence Expert* prototype. This report is accompanied by a users guide which has details on how to run the system.

1.1 The Problem

The problem is predicting the next term of a partially specified sequence. The user shall input the first few terms of a mathematical sequence. The expert system will first try to understand the pattern and using the found pattern it would predict the next term. The domains from which the user can choose terms are the following : The domain of integers and the domain of strings over a finite alphabet. This report also discusses the extention of the prototype to include other domains like the rational numbers. It must be noted that there are quite different kinds of rules for these different domains. Whereas we have some arithmetic operators predefined over numbers that implicitly store the *pattern* in the sequence of numbers, for strings its identifying the *changes* in the strings.

1.2 Outline of the Report

The next section consists of a brief introduction to the terms in the domain. This is to familiarize the user with the terminology the experts in the field might use. Section 3 discusses some related expert systems. The discussion there includes a brief overview of the various *computer algebra* systems that are currently in use. If the prototype is extended to a final system, then probably it will best fit into one of these systems.

The problem of determination of next term of a sequence can be solved elegantly by first separating the different domains from which the values in the sequence come from. (ie range of the function f). These 3 major categories are : Integers, Rationals and strings over finite alphabet. The mode of reasoning employed for these 3 cases is substantially different. In section 4 we start of with the details of the implementation of the prototype. First there is a discussion on the methods used to identify *integer* sequences. All relevant details are provided there along with suitable examples. The subsection on explanation gives details about the explanation facility provided by the prototype system for integer sequences. Section 5 contains a short note on how to include the domain of rational sequences in the prototype. This is followed by the second main part of the report in section 6, which is *string* sequences. Details similar to the ones provided for integer sequences are given here for string sequences. Sections 4 and 6 form the heart of this report.

Section 7 has some more implementation details. Section 8 deals with the extention of the prototype to a final system. The report ends with a brief summary.

2 The Domain

Definition 1 *A sequence is a function whose domain is the set of positive numbers.*

We may denote the function by f. Then its value at n is $f(n)$. The sequence f is the set

$$\{(n, f(n)) : n = 1, 2, 3, \ldots\};$$

that is, the set of all pairs $(n, f(n))$, with n a positive integer.

2.1 Example

The set $\{(n, 1/n) : n = 1, 2, 3, \ldots\}$ is a sequence whose value at n is $1/n$.

2.2 Notation

Since the domain of a sequence is always the same (the set of positive integers), it is customary to shorten the notation and just write $\{f(n)\}$ instead of $\{(n, f(n))\}$. If the function f is not known or it is not specified explicitly, the sequence is also represented by giving the values of $f(1), f(2), f(3), \ldots, f(k)$ for some finite k. Usually, $k < 10$.

The particular problem at hand is given the values of the function f at some initial values, say at $n = 1, 2, 3, 4, 5$, we have to find the function f and use it to predict the next term of the sequence, which in this case is $f(6)$.

2.3 Sources of Expertise

Textbooks of Mathematics that introduce the concepts of recursion and principle of inductive thinking contain a good range of such problems, with explanations. For this one semester term project the knowledge engineer was the domain expert too. Some books on Disrete Mathematics will provide some added knowledge.

3 Related Systems

Discussion on related expert systems is included in this section. For a more complete discussion, see [2] Most of the large-scale production expert systems that can be considered close to the full-production system for this prototype come under what is now known as *Computer Algebra Systems.* Computer algebra is that part of computer science which designs, analyzes, implements and applies algebraic algorithms. A few examples of currently available computer algebra systems are ALTRAN, CAMAL, FORMAC, MACSYMA, MATHLAB, muMATH, REDUCE, SAC-2, SCRATCHPAD and SYMBAL. Different systems allow command oriented, interactive access. Others require batchprocessing. Some combine both facilities.

Going briefly over some general aspects of design and implementation, one major influential point is that of garbage collection in these systems. This is because intermediate expressions can be highly redundant.

Many of the earlier systems on computer algebra were programmed in FORTRAN. Due to the demand of dynamic storage allocation, recursion and list processing to effectively automatize mathematics, Standard Lisp has been proposed. It has been effectively used in describing and implementing REDUCE-2. But, well known other systems like MACSYMA and SCRATCHPAD have their own LISP base.

General purpose systems, ones that were designed to serve unknown users, include MATHLAB-68, SYMBAL, FORMAC and SYCOPHANTE. MATHLAB-68 [3] is an online system. It is capable of performing differentiation, polynomial factorization, indefinite integration, direct and inverse Laplace transforms and the solution of differential equations with constant (symbolic) coefficients. MATHLAB-68 was written in LISP. The most recent computed expression is thought of as occupying a construct named "workspace". Everything computed is preserved in a stack called "history of workspace". The general purpose simplification facilities are mixed with those for special purpose seminumeric RATional SIMPlification in the same system, both transformations having access to the same data.

Both MACSYMA [5] and SCRATCHPAD [4] use more than one representation for expressions and have more than one approach to simplification. Implementation of project MAC's SYmbolic MAnipulator, written in MACLISP, began in July 1969. MACSYMA extends the capabilities of automated algebraic manipulation systems in several areas, including limit calculations, symbolic integration, solution of equations, canonical simplification, user-level pattern matching, user-specified expression manipulation and programming and bookkeeping assistance. MACSYMA makes extensive use of the power of its rational function subsystem.

In contrast to MACSYMA the SCRATCHPAD system [4] has never known a wide user's community; in fact, this interactive system, is not used outside IBM. It was originally implemented in LISP. An extensible language approach was adopted in the design of SCRATCHPAD's user language. The initial user language, the base language, contains a set of basic syntactic constructs, described by notations resembling those in conventional mathematics. These basic constructs may be extended by an individual user. In fact SCRATCHPAD is a very high level procedural language.

4 Integer Sequences

Integer sequences are the most commonly seen sequences. The main idea of the approach used in identifying the integer sequences is that we can perform certain operations on the terms of the given sequence and form new integer sequences such that once the next term of the new sequence is known, the next term of the original sequence can be determined fairly easily.

For integer sequences, its the addition, subtraction and multiplication operators that play the major role in fixing up the function f. So, in order to discover the function f, we need to perform various operations on the integers that are the first few given terms of the sequence. For example, consider the sequence $3, 7, 11, 15, \ldots$. The way this sequence is understood is by taking the difference between adjacent terms of the sequence. Doing this we get another sequence namely, $4, 4, 4, \ldots$. This sequence can be trivially identified as the constant sequence. But the new sequence thus constructed may not always be so simple. So essentially we have formed another problem from the original problem. We can recursively solve the problem for the new sequence. Consider another example, $2, 2, 4, 16, 256, \ldots$. This time construct a new sequence by dividing a term by its predecessor. We get $1, 2, 4, 8$. Applying the same principle to this sequence yields $2, 2, 2$. Again this can be easily identified as the constant sequence.

In case simple transformations like the ones discussed above fail, we go into more complicated methods.

Algorithm Identify-Integer-Sequence(S)

1. If the sequence can be identified to be a sub-sequence of a sequence in the database of integer sequences, then its trivially solved.

2. If the sequence is seen to be a sub-sequence of some sequence in its *context*, then it is solved.

3. If the sequence has all of its terms equal, then it is a *constant sequence.*

4. Method 1 : Construct a new sequence S_1 with the difference of adjacent terms in the sequence S as its terms. Put sequence S in the *context* of sequence S_1. Recursively call *Identify-Integer-Sequence(S_1)*. In case the sequence S_1 is successfully identified and its next term is found to be s_1, then the next term of sequence S is s_1 plus the last term of S.

In case the process of identification of sequence S_1 fails, continue with the next method.

5. Method 2 : If each term in the given sequence divides the next term, construct a new sequence S_1 with the quotient of adjacent terms in the sequence S as its terms. Put sequence S in the *context* of sequence S_1. Recursively call *Identify-Integer-Sequence(S_1)*. In case the sequence S_1 is successfully identified and its next term is found to be s_1, then the next term of sequence S is s_1 multiplied by the last term of S. In case the process of identification of sequence S_1 fails, continue with the next method.

6. Method 3 : If each term in the given sequence divides the previous term, construct a new sequence S_1 with the quotient of adjacent terms (term divided by the next term) in the sequence S as its terms. Put sequence S in the *context* of sequence S_1. Recursively call *Identify-Integer-Sequence(S_1)*. In case the sequence S_1 is successfully identified and its next term is found to be s_1, then the next term of sequence S is the last term of S divided by s_1. In case the process of identification of sequence S_1 fails, continue with the next method.

7. Method 4 : If the number of terms in S is substantially large, (which in this case is 5), then divide the sequence S into 2 sub-sequences S_1 and S_2, where S_1 contains all the *odd* terms of sequence S and S_2 contains all the *even* terms. Again recursively identify the next terms of the sequences S_1 and S_2 after having put S in their context. If both S_1 and S_2 are identified, then S is trivially identified. If either one fails, then this method fails. Move on to the next method.

8. Method 5 : Get a formula (atmost fourth degree polynomial in n) that *best* describes the last given term in the sequence S as a function of n. If that formula explains all the other terms of the sequence too, then it can be used to generate the next term of sequence S.

9. If all the above methods fail, then fail. In case any of the sequences is identified, its result is propogated back to the original sequence at the top level.

The methods are tried in the order in which they are enumerated above. The choice for the specific order was based on experience and knowledge of the domain.

It is also clear that the prototype can be easily expanded to include more methods by adding them down in the above ordering. That way the system continues to do what it does presently and also more depending on the power of the new methods.

One good method for future consideration would be improving the last method for identifying the integer sequences. Currently it does not do much but it can be easily improved. The main idea is to generate a function that has the minimum number of non-zero coefficients. Mathematically, this problem involves solving some system of linear equations (in 5 variables for polynomials of degree 4) with the above mentioned constraint. This was not done in this project because of limited amount of time.

Other methods for identification of integer sequences are not difficult to envisage. One such method would be to convert the integer sequence into a string solution by mapping 0 to a, 1 to b, 2 to c and so on. Subsequently the methods for identifying the string sequences can be used on this new sequence. Finally, the results can be converted back.

Any new method added over the existing structure would actually enhance the power of the system many-folds because of the inherent recursive way the methods are used. For example, method 1 may be used on the original sequence S to form a new sequence S_1. Method 4 may then be applied to S_1 to form S_2. S_2 may be finally solved by some other method (which can be the new method also!).

As an example of the application of the above procedure, consider the following sample sequence,

$$1, 2, 1, 2, 2, 4, 3, 6, 5, 10, 8, 16, \ldots$$

Method 1, method 2 and method 3 fail on this. Applying method 4, we get two sub-sequences,

$$S_1 = 1, 1, 2, 3, 5, 8, \ldots$$

$$S_2 = 2, 2, 4, 6, 10, 16, \ldots$$

Now, we can apply method 1 on the sequence S_1 to get,

$$S_3 = 0, 1, 1, 2, 3, \ldots$$

which is seen to be a sub-sequence of the sequence S_1 in the *context*. Hence, the next term of sequence S_3 is 5. So, the next term of S_1 is $8 + 5$ which is 13. Similar reasoning leads to an answer of 26 for the second sequence S_2. This solves the original sequence S.

4.1 Explanation for Integer Sequences

The method used for generating explanations is based on the ideas in [6] and [7]. The part of the program that identifies a particular sequence is responsible for generating explanations for the solution of that sequence. All explanations are template based. Since the method used for identifying integer sequences is recursive in nature, so we get different explanations for each level.

For example, consider the sequence $S = (2, 3, 5, 9, 17)$. Applying method 1 to this sequence we create the new sequence $S_1 = (1, 2, 4, 8)$. Applying method 1 again on S_1 gives $S_2 = (1, 2, 4)$. Now step 1 of the above algorithm identifies the sequence S_2 to be a sub-sequence of sequence S_1 which happens to be in the context of sequence S_2. So, next term of S_2 is 8 because 8 occurs after $(1, 2, 4)$ in S_1. Therefore next term of S_1 is $8 + 8 = 16$. Therefore the next term of S is $17 + 16 = 33$.

The explanation generated by the program is similar to what has been described above and is shown below!

```
************ SEQUENCE IDENTIFICATION EXPERT ************
Input the first few terms of the sequence : 2 3 5 9 17
Input sequence is (2 3 5 9 17)

***** 33 is the next term
At level 3: Sequence (1 2 4) is observed to  be a sub-sequence
of the sequence at level 2. This is a recursively defined
sequence. Since the term after (1 2 4) in sequence
(1 2 4 8) is 8, therefore the next term is 8.
At level 2: Sequence (1 2 4 8) has the property that
each element can be obtained from the previous one
by adding numbers from the sequence (1 2 4). Since
the next term of sequence (1 2 4) is 8, therefore the
next term of sequence (1 2 4 8) is 16.
```

```
At level 1: Sequence (2 3 5 9 17) has the property
that each element can be obtained from the previous
one by adding numbers from the sequence (1 2 4 8).
Since the next term of sequence (1 2 4 8) is 16, therefore
the next term of sequence (2 3 5 9 17) is 33.
```

Each sub-sequence created by any method is considered to be at a level one below the level of the original sequence. Each level generates explanation for why the next term for that level is the one displayed.

4.2 Where the Expert fails!

The program fails to identify sequences in which the function f describing the nth term of the sequence, is not a simple closed form mathematical function. Getting general methods for these sequences is difficult. Examples of such sequences are

1. $2, 3, 5, 7, 11, \ldots$ where $f(n) = n$th prime number.

2. $3, 1, 4, 1, 5, 9, \ldots$ where $f(n) = n$th digit in the decimal expansion of π.

Clearly, there are no methods to identify sequences like these!

The program also fails for integer super-exponential sequences. For example, consider $S = (2, 6, 30, 260)$. These are the first four terms of the sequence $f(n) = n^n + n$. The program fails to identify these kind of sequences. But as mentioned above, general methods for solving these sequences can be added.

Here are a few more examples where the prototype fails.

1. $1, 2, 2, 3, 3, 3, 4, \ldots$ - This is the sequence where the number i is repeated i times.

2. $1, 2, 3, 2, 1, 2, 3, \ldots$ - This sequence is formed by repeating the sequence $1, 2, 3, 2$. The system goes into very deep searching and identifies the sequence incorrectly. The interesting fact to note is that for the sequence $1, 2, 3, 2, 1, 2, 3, 2, 1, \ldots$ - the prototype correctly solves this one and gives the correct answer 2.

5 Rational Sequences

Rational sequences, ones in which the range of the function f is the set of all rational numbers, are more complicated. Essentially, rational numbers are numbers of the form p/q where p and q are integers and $q \neq 0$. One way to look for the function f given the first few rational numbers is to look at the sequence formed by taking the numerators of the given rationals separately and looking at the denominators separately. If a function f_1 can be identified for the numerator sequence and f_2 can be identified for the denominator sequence, then we can predict the next term of the rational sequence. This idea will take care of simple rational sequences like $1, 1/2, 1/3, 1/4, \ldots$.

Second idea is generalizing methods for integer sequences to rational sequences by considering operations on rationals rather than operations on integers.

The current version of the *Sequence Epert* does not consider rational sequences. But with the kinds of methods that are available for integers, a fairly strong implementation for rational sequences can be obtained by just converting the given rational sequence into a integer sequence and then using the integer-sequence identifier to solve the problem. For example, a sequence like $1, 1/2, 1/3, 1/4, \ldots$ can be *linearized* to give an integer sequence $1, 1, 1, 2, 1, 3, 1, 4, \ldots$. This sequence can be easily identified by the integer part of the prototype system.

6 String Sequences

String sequences involve string operators, like concatenation and substring matching. So the basic question here is how to get the next term from the current string by only applying these operators. For example, in the sequence $aa, aba, abba, abbba, \ldots$ its very clear that the next term if got by inserting an b in the middle of previous string. But not all string sequences can be described by such simple operations. In many cases the alphabet has some implicit ordering in them. Consider the example, $a, b, d, g, \ldots$. In this case, the sequence can more easily be understood by transforming it to a integer sequence by replacing a by 1, b by 2 and so on. Sequences like $a, ab, abc, abcd \ldots$ can be explained by a combination of both the above ideas.

Algorithm Identify-String-Sequence(S)

1. Method 1 : Take the last two terms of the string sequence S and find the *edit distance* between them. Store the *transformations* T corresponding to the calculated edit distance. Check to see if the transformations T also describe the previous terms of sequence S. If yes, then the next term is just T applied to the last term of S. If not, then continue with the next method.

2. Method 2 : Calculate the *edit distance* between each of the adjacent terms of sequence S. Let D denote the sequence of the distances (integer sequence) and T be the sequence of transformations.

   - Case 1 If D is a constant sequence containing 1, then *find the next transformation* and if successful then apply the new transformation to the last term of S. If not successful then try the next method.
   - Case 2 If D is a constant sequence containing 2, then break up T into T_1 and T_2 where T_1 is a sequence of all first transformations and T_2 is a collection of all second transformations. *Find* the next transformation for T_1 and T_2. If successful in both then apply both the transformations to the last term in S to get the next term. If not then continue with the next method.
   - Case 3 If D is anything else, then continue with the next method.

3. Method 3 : Convert each string in the sequence S to an integer. This is done by considering a to represent 1, b 2, c 3, and so on, in *base* 27. The resulting integer sequence is solved using methods for integer sequences. If the integer methods are successful then the solution integer is converted back to a string. If the integer methods fail, then this method fails. Since there are no more methods to try, the procedure fails for sequence S.

6.1 Edit Distance

Suppose s and t are arbitrary strings over the english alphabet. One *change* on string s is defined to be either an *insertion* of a character at any position in s, or a *deletion* of an existing character in s or a *replacement* of a character in s by some other character at the same *location*. For example, an *insertion* of character x at position 1 in the string *abcde* gives the new string *axbcde*. *Deletion* at position 2 (of b) on the same string gives *acde*. *Replacement* of a at position 1 in *abcde* by b gives *bbcde*.

The *edit distance* between two strings s and t is defined as the minimum number of *changes* needed to be performed on s so that it gets converted to t. For example, let s be "*ababa*" and t be "*babab*". Then the edit distance between s and t is 2. The corresponding two changes (also referred to as *transformations* are "deletion at position 1 (of a) in s" followed by "insertion of b at position 4". Similarly, the edit distance between strings "*aba*" and "*aca*" is 1. The corresponding transformation is "replacement at position 2 of b by c".

The edit distance between two strings can be found using an algorithm based on *dynamic programming.* For details on the dynamic programming paradigm, refer to [8]. Essentially, if s and t are the two given strings with lengths l_s and l_t respectively, then the algorithm maintains a two-dimensional array *distance*, where $dist[i][j]$ = the edit distance between the string $s[1,\ldots,i]$ and the string $t[1,\ldots,j]$. Note that $s[1,\ldots,i]$ denotes the sub-string of s from character 1 to the ith character of s. Clearly $dist[i][j]$ has the required edit distance. Also, we have,

$dist[i][j] = \text{minimum}(\text{dist}[i][j-1]+1,\ \text{dist}[i-1][j]+1,\ \text{dist}[i-1][j-1])$ if $s[i] = t[j]$

$= \text{minimum}(\text{dist}[i][j-1]+1,\ \text{dist}[i-1][j]+1,\ \text{dist}[i-1][j-1]+1)$ if $s[i] \neq t[j]$

The array *distance* can be easily calculated using the above formula incrementally. Also, the corresponding transformations can be saved.

6.2 The Transformation Sequence

The problem of identifying a string sequence now reduces to finding the next transformation following a given sequence of transformations. For convenience let (insert, *ch*, *n*) denote the transformation "insertion of the character *ch* at position *n*". So for example, given the sequence

$$(\text{insert}, a, 1), (\text{insert}, b, 2), (\text{insert}, c, 3)$$

the next transformation would be

$$(\text{insert}, d, 4)$$

The technique used to identify the next transformation is to break up the problem into 3 sub-problems. The first one being identifying the next *operation*, the second is identifying the next *character* and the third one is identifying the next *position*.

6.2.1 Next Operation

There is a database of possibilities for the sequence of operations since there are not too many different ways in which operations can be arranged. A few of the facts in the database are the following sequences,

1. (insertion, insertion, insertion, insertion, ...)
2. (deletion, deletion, deletion, deletion, ...)
3. (replacement, replacement, replacement, replacement, ...)
4. (insertion, deletion, insertion, deletion , insertion, deletion...)
5. (insertion, modification, insertion, modification, ...)

6.2.2 Next Character

Again since there are not many different choices here, there is a database maintained containing all facts about the next character. Typical facts that are stored in the knowledge base for predicting the next character are

1. (a, b, c, d, e, f, g, h, ..., z)
2. (z, y, x, w, v, u, t, s, ..., a)
3. (a, c, e, g, i, k, m, ...)
4. (b, d, f, h, j, l, n, ...)
5. Constant character sequences.

Although currently all constant character sequences too are put in as facts in the knowledge base, the prototype can be made more fast if these are hard-coded into the program.

6.2.3 Next Position

The method used to infer the next position is the same as the method used above for finding the next character and next transformation. The reason is again the same. Some typical facts here are,

1. (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2. (1, 3, 5, 7, 9, 11, 13)
3. (2, 4, 6, 8, 10, 12, 14)
4. (1, 2, 1, 2, 1, 2, 1, 2)
5. Constant number sequences.

Again here too it would have been better if the integer sequence program were used to get the next position, but since only simple sequences arise in practice here, a database was maintained. The remarks about improving performance by hard-coding some of these facts carry over.

6.2.4 Example

As an example of a string sequence consider,

$$S = a, ab, aab, aabb, aaabb, \ldots$$

The sequence formed by the edit distances between adjacent terms of S is,

$$D = 1, 1, 1, 1, \ldots$$

which is a constant sequence containing 1. The corresponding transformation sequence is,

$$T = (\text{insert } b \text{ at position } 1), (\text{insert } a \text{ at position } 0),$$

$$(\text{insert } b \text{ at position } 2), (\text{insert } a \text{ at position } 0)$$

Therefore, the next transformation is (insert b at position 3).

6.3 Explanation for String Sequences

The explanation generated for string sequences follows the same general idea. Once the rule that generates the next string from the previous string is found, it is printed out based on the templates stored.

For example, consider the sequence $S = (az, by, cx)$. The first method for strings fails here because not the same rule determines the next term in the sequence. Applying method 2, we get the following transformation sequence,

$$T = (\text{replace } a \text{ by } b \text{ at position } 0,\ \text{replace } z \text{ by } y \text{ at position } 1),$$

$$= (\text{replace } b \text{ by } c \text{ at position } 0, \text{replace } y \text{ by } x \text{ at position } 1)$$

and the distance sequence is $D = (2, 2)$, since the edit distance between alternate terms is always 2. Therefore, the next transformation will be found out to be, (using the above mentioned procedure),

$$(\text{replace } c \text{ by } d \text{ at position } 0, \text{replace } x \text{ by } w \text{ at position } 1)$$

This gives dw as the next term. The explanation produced for this example looks like this,

```
LIPS> (run)
************ SEQUENCE IDENTIFICATION EXPERT ************
Input the first few terms of the sequence : az by cx
Input sequence is (az by cx)
-------- Explanation --------
The way the sequence is built up is
1. Change to b at position 1 and Change to y at position 2
2. Change to c at position 1 and Change to x at position 2
Therefore, next term can be obtained by
3. Changing to d at position 1 and Changing to w at position 2

outsequence (az by cx) dw
Identified as a string sequence at level 1
True
```

The sentence at position 1 in the explanation explains how the second term in the input sequence can be obtained from the first term. Similarly, the second line explains how to get the third term from the second term. The last statement explains how to get the *next* term of the input sequence from the last term.

When the system uses method 3 to find the next term of the string sequence it doesnot produce any explanations apart from saying that the strings were converted into appropriate integers and the final integer solution and its corresponding string are shown. For example consider the sequence $S = (a, b, d, g, k)$ which is actually an integer sequence $S_i = (1, 2, 4, 7, 11)$. The explanation for this sequence looks like,

```
(CLIPS> run)
************ SEQUENCE IDENTIFICATION EXPERT ************
Input the first few terms of the sequence : a b d g k
Input sequence is (a b d g k)

The sequence (a b d g k) can be viewed as the integer
sequence (1 2 4 7 11). Since the next term in the integer
sequence is 16, the corresponding string is p
True
```

Needless to say, when the method fails, no explanations are produced!

6.4 Complications and Expansion of the Prototype

Actually the categorization into the 3 groups, integer, rational and string sequences, is just theoretical, as seen in the last example in the previous subsection. A string sequence too can be posed as an integer sequence problem. For example consider the sequence $1, 12, 123, 1234\ldots$ This is more of the string type than the integer type. Also, not all sequences can be explained by such simple rules. Consider the following : $jan, feb, mar, \ldots$.

One main reason why the program may fail is because in method 2, we fail if the edit distance sequence is not a constant 1 or a constant 2 sequence. Consider for example, $S = (a, bb, ccc)$. The edit distance sequence would be $D = (2, 3)$. So, we fail. Whereas we can infer that the next distance value would be 4. The problem is how to infer the next transformation, which in this case would be three replacements of c by d and one insertion of d.

In order to overcome some of these cases, one really good method would be to use other *string distance measures*, apart from just edit distance. For example, we could define another distance measure similar to edit distance but excluding the *replacement* operator. There are other distance measures too in the literature.

7 Some Details

The prototype expert system has been developed using the CLIPS environment. Therefore its a rule-based system. The essence of the reasoning structure is search on a state space. The given input sequence undergoes various tests and then a possible path is further explored. Since the problem is being reduced to newer subproblems of the same type that return answer to the parent sequence, we differentiate between sequences that are yet to be solved and that have been solved. The former are represented by facts like (insequence n ...) and the latter by (outsequence n ...). The n in the facts denotes the correspondence between the insequence and outsequences. It has been referred to as *level* in the previous sections.

The control is passed from one method to the other by asserting appropriate facts. Each method is applied only if proper facts are present in the knowledge base (*precondition*). The method *retracts* the facts that form its pre-conditions and asserts appropriate *post-condition* facts. Depending on the success or failure of the method, different facts are asserted in the database. These asserted facts then form the pre-condition for other rules.

For string sequences, calculation of edit distance too is done by asserting and retracting facts. The 2-dimensional array required for the algorithm is stored as facts in the knowledge base. Once the edit distance is found, all these facts are retracted.

8 Expandability

As has been mentioned previously, the system has been built in a style that allows easy addition of new rules and methods. The power gained by the system by addition of one new method is not linear, as pointed out earlier. The prototype can be very easily extended to a full-scale production system. It can possibly be included as a sub-module in the other general purpose computer algebra systems.

New facts can be added as CLIPS *facts*. The *deffacts* constructs allows one to include as many facts in the knowledge base. The addition of more facts will *not* interfere with the other parts of the program.

The system can be augmented bu adding *learning* capabilities. Learning facts can be done in a straight forward way. Learning new methods (ie CLIPS *rules*) will be more difficult.

9 Summary

A fairly complete description of the *Sequence Expert* prototype system that finds the next term of any given integer or string sequence was presented in this report. The prototype developed is powerful enough to detect fairly complicated integer and string sequences. The accompanying *users guide* contains a list of testbench problems which the system successfully solves.

References

[1] Giarratano and Riley, *Expert Systems - Principles and Programming*, PWS-Kent, Boston, 1994.

[2] van Hulzen and Calmet, J, *Computer Algebra Systems* Computer Algebra Symbolic and Algebraic Computation, Second Edition, Ed. Buchberger, Collins and Loos, Springer-Verlag, 1983.

[3] Engelman, Carl, *The Legacy of Mathlab 68*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Ed. Petrick, March 23-25, 1971, Los Angeles, California.

[4] Griesmer and Jenks, *SCRATCHPAD/1 - An Interactive Facility for Symbolic Mathematics*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Ed. Petrick, March 23-25, 1971, Los Angeles, California.

[5] Martin and Fateman, *The MACSYMA System*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Ed. Petrick, March 23-25, 1971, Los Angeles, California.

[6] Startout et al, *Design for Explainale Expert Systems*, IEEE Expert, June 1991.

[7] Tanner and Keunke, *The roles of the task structure and domain functional models*, IEEE Expert, June 1991.

[8] Cormen, Leiserson and Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, 1990.