

STRING PATTERN-MATCHING IN PROLOG

MARCO A. CASANOVA and ANTONIO L. FURTADO*

Rio Scientific Center—IBM Brasil, Estrada da Canoa, 3520. 22.610, Rio de Janeiro, RJ, Brasil

(Received 18 December 1987; in revised form 30 June 1988)

Abstract—A pattern-matching feature for the Prolog language is described. Through the use of patterns, introduced as Prolog predicates, the feature favors the specification of string handling algorithms in a declarative style. A number of convenient pre-defined patterns, adapted from SNOBOL 4, are included. The use of two-level grammars as a paradigm for developing Prolog programs incorporating the pattern-matching feature is also discussed.

Logic programming Prolog Pattern-matching String processing SNOBOL

1. INTRODUCTION

Prolog strings provide a convenient way to represent arbitrary sentences of natural or artificial languages. Unfortunately, most Prolog dialects have a very limited and often low-level set of built-in operations on character strings, such as the substring and concatenation operations. Hence, if an application involves sophisticated string manipulation, then one is almost forced to adopt the strategy of representing strings as lists of characters, since unification cannot “look inside” strings [1]. But this strategy implies that the Prolog programmer must invest some effort in mastering the various techniques for mapping strings into lists, thus diverting his attention from the application in hand. From another perspective, this strategy requires representing a data type T_1 (strings) by another data type T_2 (lists) and expressing the operations of T_1 in terms of those available for T_2 , which conflicts with the current trend towards abstract data types.

This paper then describes a high-level pattern-matching feature that facilitates the specification of string handling algorithms in a declarative style by hiding all details concerning the representation of strings. The paper also includes a number of convenient pre-defined patterns, adapted from SNOBOL 4, and discusses the use of two-level grammars as a paradigm for developing Prolog programs incorporating the pattern-matching feature.

More precisely, the basic idea behind the paper goes as follows. Consider the fundamental problem of determining whether a string S satisfies some property P . The obvious solution in Prolog is to define a predicate p in such a way that S has property P if and only if $p(S)$ is true. Property P may in turn be defined in terms of a set of properties P_1, P_2, \dots, P_n in the sense that a string S satisfies P iff there are substrings S_1, S_2, \dots, S_n of S that satisfy, respectively, properties P_1, P_2, \dots, P_n . Correspondingly, in Prolog, predicate p would have a conditional definition of the form:

$$\begin{aligned} p(S) <- \\ & \text{split}(S, [S_1, S_2, \dots, S_n]) \ \& \\ & p_1(S_1) \ \& \\ & p_2(S_2) \ \& \\ & \dots \\ & p_n(S_n). \end{aligned}$$

where the predicate `split` has the task of splitting S into substrings.

Instead of a predicate like `split`, we introduce however the `match` meta-predicate, leading to a more concise definition of p :

$$\begin{aligned} p(S) <- \\ & \text{match}(S, p_1 \parallel p_2 \parallel \dots \parallel p_n). \end{aligned}$$

*On leave from the Pontificia Universidade Catolica do Rio de Janeiro.

We call the second argument of `match` a *pattern-expression*. The intended meaning of “`match(S, p1 || p2 || ... || pn)`” is “find a split of `S` into substrings `S1, S2, ..., Sn` such that `p1(S1) & p2(S2) & ... & pn(Sn)` is true”. Thus, `match` has, in addition to the task of splitting `S`, the task of adding an extra argument to `pi` and then of calling the modified predicate. Although not indicated, each predicate `pi` may also have other arguments besides `Si`. The backtracking machinery, inherent in Prolog, will try different splits until one succeeds or all of them fail.

A predicate `pi` appearing in a pattern-expression may in turn be structured like `p`, i.e. it may use `match` to apply a pattern-expression invoking still other predicates, and so on, to the point that the program takes the form of a set of mutually recursive pattern-matching predicates. Such programs are appropriately characterized as grammars. It turns out that context-free grammars can be easily mapped into Prolog patterns using `match`. However, more powerful grammars have properties that conflict with this mapping scheme. Fortunately, languages generated by such grammars can also be generated by two-level grammars (also called van Wijngaarden grammars) that, again, have the required properties. The paper then concludes with a discussion about how to use the two-level grammar paradigm to build Prolog programs.

The power gained in Prolog programming by using meta-level constructs has been stressed in [2]. Meta-variables and meta-predicates essentially allow to emulate Second-Order Predicate Calculus features in the First-Order Predicate Calculus formalism that underlies Prolog.

Pattern-matching has been used as a high-level technique for character string manipulation in a number of programming languages, SNOBOL 4 [3] and Icon [4] being perhaps the most successful ones. We borrowed from SNOBOL 4 a number of pre-defined patterns, which provide a flexible control of the pattern-matching process. As a consequence, most SNOBOL 4 programs can now be readily translated into Prolog. In fact, thanks to certain features of Prolog, such as the reversible use of predicates, some pattern expressions become simpler.

The work reported in this paper is closely connected with the use of definite clause grammars (DCGs) (see, for example, [5] for a brief introduction) and, more generally, with the Metamorphosis Grammars of Colmerauer [6]. Implementations of DCGs have used efficient difference list techniques (see [7, Section 2.3] and [8, Section 6.2] for example). It is fair to say that the `match` meta-predicate gives the power and conciseness of DCGs to Prolog programmers that work with interpreters that do not have DCGs built-in.

Generalizations of Prolog by way of grammars were investigated in [9, 1]. The first reference explores how to map definite-clause programs into attribute grammars and vice-versa to prove interesting results about definite-clause programs. The second reference introduces grammatical programs as a generalization of logic programs. This paper follows a much more pragmatic line since it is concerned with Prolog meta-predicates that essentially facilitate coding production rules as Prolog clauses without changing the basic syntax of Prolog.

As for the organization of the paper, section 2 introduces the `match` meta-predicate. Section 3 describes the pre-defined patterns, adapted from SNOBOL 4. Section 4 gives simple examples of pattern-matching using the feature. Section 5 presents the two-level grammar paradigm. Section 6 contains the conclusion. The Appendix lists the prototype implementation of the feature in IBM Prolog, which closely follows the Edinburgh Prolog, the de facto standard [5, page 428]. For concreteness, the examples also use the syntax and facilities of IBM Prolog.

Application-minded readers will notice that pattern-matching permits handling character strings with far less procedural programming effort than required when one is limited to the built-in primitives of most Prolog dialects. On a first reading, we suggest covering Sections 2 and 4 that contain a broad description of the feature and examples of its use. Section 3 goes into some detail, being directed to prospective users or to designers interested in defining their own high-level string handling packages. Section 5 concentrates the more theoretical considerations.

2. THE MATCH META-PREDICATE

The pattern-matching feature is based on a meta-predicate called `match`, which parses or generates strings according to pattern-expressions. We proceed in a top-down fashion to define the syntax and semantics of `match` and of pattern-expressions.

A call to `match` has the form “`match(S, P)`”, where `S` is either a Prolog variable or a Prolog string constant and `P` is a pattern-expression. If `S` is a constant or a variable instantiated with a string constant at the time of the call, then `match(S, P)` will succeed if `P` matches `S`, otherwise it will fail. If the pattern-expression matches the string in more than one way, they will all be tried on successive calls if backtracking is activated. However, if `S` is an uninstantiated variable at the time of the call, `match(S, P)` will succeed and instantiate `S` with a string that `P` matches; upon backtrack the call succeeds again, instantiating `S` with a different string; it will fail when there are no more strings matching `P`.

Therefore, with an uninstantiated variable as first argument, `match` can be used to generate strings that match a pattern-expression. Otherwise, `match` can be used to parse strings according to a pattern-expression. In this paper, we shall be primarily concerned with the latter usage, although occasional remarks will be made on the former.

A *pattern-expression* is an expression of the form “ $P_1 \parallel \dots \parallel P_n$ ”, where $n > 0$ and each *component* P_i is either a Prolog string constant, a Prolog variable or a pattern. If the pattern-expression is used inside a call to `match` intended to generate strings, then a component can also be a limited Prolog variable or a limited pattern.

A *pattern* P_i is in turn either an expression of the form “ $p(t_1, \dots, t_n)$ ” or an expression of the form “ $p(t_1, \dots, t_n) \text{ \# } V$ ”, where p is a Prolog predicate and V is a Prolog variable or a Prolog string constant. We say that the *arity* of P is n and that p is the Prolog predicate *corresponding* to P . Note that, like the nonterminals of a definite clause grammar [5], patterns can have arguments. A pattern will always be associated with a Prolog predicate p that defines a restricted class of strings, as explained in detail later on.

A *limited Prolog variable* is an expression of the form “ $V \ll k$ ”, where V is a Prolog variable and k is a *limit*, that is, either a Prolog integer constant, a Prolog computable expression that evaluates to an integer or a variable that must be instantiated by an integer.

A *limited pattern* is an expression of the form “ $P \ll k$ ”, where P is a pattern and k is a limit as above.

Finally, the reader will note that, by definition, a component of a pattern-expression cannot in turn be a (compound) pattern-expression. This restriction, imposed for the sake of simplicity, excludes for example expressions like $(p_1 \parallel p_2) \text{ \# } X$ and $(p_1 \parallel p_2) \ll X$.

We define the semantics of pattern-expressions through the concept of a *match*.

Thus we say that a string constant *matches* an identical string, whereas a Prolog variable *matches* arbitrary strings.

A pattern P of the form “ $p(t_1, \dots, t_n)$ ” *matches* any string s in the class p defines. A pattern P of the form “ $p(t_1, \dots, t_n) \text{ \# } V$ ”, with V uninstantiated, *matches* any string s in the class p defines and instantiates V with s . However, if V is instantiated to some value s' , then P *matches* s only if, in addition, s is equal to s' . The symbol “ \# ” is analogous to the immediate value assignment operator “ S ” in SNOBOL4; it can also be regarded as a generalization of the “ $\$$ ” type-checking operator of IBM Prolog.

A limited Prolog variable of the form “ $V \ll k$ ” *matches* any string with length less than or equal to the value of k . Likewise, a limited pattern of the form “ $P \ll k$ ” *matches* any string s that matches P and has length less than or equal to the value of k . If k is a computable expression, it is evaluated immediately before matching; if it is a variable, its instantiation may occur at any time previous to matching. The use of limits is an important feature to avoid loops.

Lastly, a pattern-expression of the form “ $P_1 \parallel \dots \parallel P_n$ ” *matches* a string s if s can be split into (possibly null) substrings s_1, \dots, s_n such that P_i matches s_i , for each $i = 1, \dots, n$.

We conclude this section by explaining how to define the Prolog predicate associated with a pattern. Briefly, there are two types of patterns: *context-free* and *context-sensitive*. Intuitively, the implementation of `match` tests whether a pattern P matches a substring T of S by scanning only T itself, if P is context-free, or by examining T , the position of T in S and possibly additional characters in S to the right of T , if P is context-sensitive. The implementation of `match` automatically detects the type of P when it is invoked by comparing the arity, n , of the corresponding Prolog predicate with the arity, m , of P : if $n = m + 1$, then P is context-free and, if $n = m + 2$, then P is context-sensitive. The purpose of the extra arguments of the Prolog predicates will become clear in what follows.

Let P be a pattern either of the form " $p(t_1, \dots, t_n)$ " or of the form " $p(t_1, \dots, t_n) \notin V$ ".

Suppose first that P is context-free. During the processing of a call of the form $\text{match}(S, Q)$, if P is reached, a call of the form " $p(t_1, \dots, t_n, X)$ " is generated, where X represents the substring of S that match currently associates with P . Therefore, the programmer must define p by Prolog clauses whose heads are of the form " $p(u_1, \dots, u_n, s)$ ", where the last argument s is understood as the string being tested for membership (or being generated) in the class denoted by p .

As a very simple example, suppose one wants to use the built-in predicate " $\text{stlen}(V, L)$ " of IBM Prolog, where V is a string and L its length. Clearly stlen does not meet our requirement, since the string is not the last argument. Yet one can define an appropriate predicate that does the same as stlen and meets the requirement:

$$\text{len}(L, V) \leftarrow \text{stlen}(V, L).$$

and in fact this is the definition of len that we use in Section 3.

Therefore, the goal

$$\leftarrow \text{match}('abbavuba', 'ab' \parallel \text{len}(2) \notin W \parallel Z).$$

will succeed and assign 'ba' to W and 'vuba' to Z .

Suppose now that P is a context-sensitive pattern. During the processing of a call of the form $\text{match}(S, Q)$, if P is reached, a call of the form " $p(t_1, \dots, t_n, X, [l, Y])$ " is generated, where X represents the substring of S that match currently associates with P , l the position in S immediately to the left of X and Y is what remains of the substring of S to the right of X . Therefore, the programmer must code Prolog clauses whose heads are of the form " $p(u_1, \dots, u_n, s, [i, t])$ ", where s is understood as the substring being tested for membership in the class denoted by p , i is the current position and t is what remains of the original string to the right of s .

Context-sensitive patterns have thus positional information and the means to perform a "look-ahead" to control the matching process. The present paper puts its emphasis on context-free patterns, whose application is much simpler and also more efficient. Hopefully, the machinery provided by the pre-defined context-sensitive patterns described in Section 3 will be enough for most applications. A general approach to context-sensitivity will be discussed in Section 5.

3. PRE-DEFINED PATTERNS

This section describes a number of useful pre-defined patterns adapted from SNOBOL 4, which remains one of the most widely used languages for text applications. Its successor, Icon [4], inherited its main features within a more modern programming language framework.

The pre-defined patterns are classified as follows:

| | |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <i>context-free:</i> | <code>len</code> , <code>any</code> , <code>notany</code> , <code>bal</code> and <code>arbno</code> ; |
| <i>context-sensitive:</i> | <code>bl</code> , <code>span</code> , <code>break</code> , <code>pos</code> , <code>rpos</code> , <code>tab</code> and <code>rtab</code> . |

They can be freely used in pattern-expressions, except that, in a call of the form $\text{match}(S, P)$, if S is an uninstantiated variable, then P can include only `any`, and `arbno` (noting that the latter must also invoke a pattern prepared to generate strings). Therefore, the pre-defined patterns can be used without restrictions to parse strings, but only `any` and `arbno` can be used to generate strings.

Briefly, their behavior follows the homonymous patterns of SNOBOL [3], with the following exceptions:

- (1) for compatibility with the built-in predicates of IBM Prolog, character positions in strings are numbered starting from zero; as a consequence, -1 means that the current position is just at the left of the string;
- (2) taking advantage of Prolog's characteristics, the argument of `len`, `pos`, `rpos`, `tab` and `rtab` can be a variable or a non-negative integer-valued computable expression;
- (3) pattern `bl` is not pre-defined in SNOBOL 4, but it was introduced for convenience;
- (4) the `arb` pattern of SNOBOL 4 is not included, since its role can be played by named or unnamed (i.e. `"*"`) Prolog variables.

Appendix I shows the implementation of all pre-defined patterns. In an alternative implementation, we embedded their definition in the `match` meta-predicate so as to check the context-sensitive conditions at the time of splitting. This strategy resulted in considerable gains in efficiency, at the expense of clarity and flexibility, because some patterns only have positional conditions and others require a limited amount of look-ahead (for `span`, for example, it suffices to test the first character of the remaining substring).

The rest of this section defines in detail the pre-defined patterns and clarifies with examples the key points. The definition adopts the following conventions:

- `i` will denote a non-negative integer or a Prolog computable expression returning a non-negative integer;
- `V` a Prolog variable;
- `s` a Prolog string; and
- `P` a pattern.

The first patterns are simple and should raise no doubts:

- `bl` matches the longest run of zero or more blanks;
- `bal` matches any nonnull substring which is balanced with respect to parentheses;
- `any(s)` matches any single character appearing in `s`;
- `notany(s)` matches any single character not appearing in `s`;
- `len(i)` matches any substring of the length specified by `i`;
- `len(V)` matches any substring and `V` is instantiated with the length of the matching substring.

For example, the goal

```
<- match('abbavuba', X || any('uv') ☐ Y || Z).
```

yields `X = 'abba'`, `Y = 'v'` and `Z = 'uba'`.

As an example of the use of variables in `'len'`, the goal

```
<- match('abbavuba', 'ab' || len(V) ☐ X || 'ba').
```

yields `V = 4` and `X = 'bavu'` since the string `'abbavuba'` matches the pattern-expression `'ab' || len(V) ☐ X || 'ba'` only if it is split into the substrings `'ab'`, `'bavu'` and `'ba'`.

The pre-defined patterns introduced below are still relatively simple:

- `span(s)` matches the longest nonnull run of characters formed only from characters appearing in `s`;
- `break(s)` matches the longest nonnull run of characters not containing any character appearing in `s`;
- `arbno(P)` matches zero or more consecutive occurrences of substrings matched by `P`. If `P` contains variables, they will be instantiated with the same value in every use of `P` during the processing of `arbno`.

For example, the goal

```
<- match('abbavuba', span('ab') ☐ x || Y),
```

yields `X = 'abba'`, `Y = 'vuba'`, and the goal

```
<- match("abbavuba", break("uv") ☐ X || Y).
```

yields `X = 'abba'`, `Y = 'vuba'`.

As an example of the use of `"arbno"` and `"any"` for generating strings, suppose we want to generate all strings with at most two characters, taken from the set `{'a','b'}`, followed by `'u'`. This can be accomplished by executing:

```
<- match(S, arbno(any('ab')) << 2 || 'u') & write(S) & fail.
```

which initiates `S` with `'u'`, `'au'`, `'bu'`, `'aa'`, `'abu'`, `'bau'`, `'bbu'`, in breadth-first order therefore, and prints these strings.

The last four pre-defined patterns, `pos`, `rpos`, `tab` and `rtab`, depend on the current position being scanned. For `pos` and `tab`, the positions of a string are numbered from the left to the right, starting from 0; thus, “-1” means that the current position is just at the left of the string. However, for `rpos` and `rtab`, the positions of a string are numbered from the right to the left, starting with 0; thus, “-1” means that the current position is just at the right of the string. Their definitions are:

- `pos(i)` matches the null string, if i is equal to the current position; otherwise it fails;
- `pos(V)` matches the null string and V is instantiated to the current position;
- `rpos(i)` matches the null string, if i indicates the position to the right of the current position; otherwise it fails;
- `rpos(V)` matches the null string and V is instantiated with the position to the right of the current position;
- `tab(i)` matches all characters to the right of the current position up to, and including, the position indicated by i , if i indeed indicates a position to the right of the current position; otherwise it fails;
- `tab(V)` matches any substring and V is instantiated to the position where the matching substring ends;
- `rtab(i)` matches all characters to the right of the current position up to, *but excluding*, the position indicated by i , if i indeed indicates a position to the right of the current position; otherwise it fails;
- `rtab(V)` matches any substring and V is instantiated to the position where the matching substring ends.

We close this section by illustrating the use of these patterns. The goal

```
<- match('abbavuba', 'ab' || pos(N) || X).
```

yields $N = 1$ and $X = \text{'bavuba'}$ since the current position is equal to 1 when “`pos(N)`” is reached, that is, after matching ‘`ab`’ with the input string. The situation is depicted below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|------------------------------------------------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | positions counted form the left |
| | | | | | | | | current position when <code>pos(N)</code> is reached |
| | | | | | | | | input string. |
| a | b | b | a | v | u | b | a | |

Now, the goal

```
<- match("abbavuba", X || rpos(5) || Y).
```

yields $X = \text{'ab'}$ and $Y = \text{'bavuba'}$. This follows because the only way the pattern can succeed is when the current position is equal to 6 (counting from the right) when “`rpos(5)`” is reached, as shown below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----------------------------------------------------------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | positions counted from the right |
| | | | | | | | | position indicated by the argument of “ <code>rpos(5)</code> ” |
| | | | | | | | | current position compatible with “ <code>rpos(5)</code> ” |
| | | | | | | | | input string. |
| a | b | b | a | v | u | b | a | |

The following example illustrates the use of `tab` with a variable argument. The goal

```
<- match('abbavuxuba', 'ab' || tab(N) & X || 'u' || Y).
```

yields $N = 4$, $X = \text{'bav'}$ and $Y = \text{'xuba'}$ since the current position is equal to 1 when “`tab(N)`” is reached and ‘`u`’ first occurs at position 5. The situation is depicted below:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|--------------------------------------------------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | positions counted from the left |
| | | | | | | | | | | current position when <code>tab(N)</code> is reached |
| | | | | | | | | | | position before first occurrence of “ <code>u</code> ” |
| | | | | | | | | | | input string with matching substring. |
| a | b | b | a | v | u | x | u | b | a | |

Backtracking to obtain another answer will result in $N = 6$, $X = \text{'bavux'}$ and $Y = \text{'ba'}$ since ‘`u`’

occurs for the second time at position 7. The situation is depicted below:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|------------------------------------------------------|---------------------------------------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | positions counted from the left | |
| | | | | | | | | | | current position when <code>tab(N)</code> is reached | |
| a | b | b | a | v | u | x | u | b | a | position before second occurrence of 'u' | |
| | | | | | | | | | | | input string with matching substring. |

As for `rtab`, the goal

```
<- match('abbavuba', 'ab' || rtab(2) ⚡ X || 'u' || Y).
```

yields `X = "bav"` and `Y = "ba"`. The following figure illustrates this pattern matching process:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|----------------------------------------------------------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | positions counted from the right |
| | | | | | | | | position indicated by the argument of " <code>rtab(2)</code> " |
| a | b | b | a | v | u | b | a | current position when " <code>rtab(2)</code> " is reached |
| | | | | | | | | input string with matching substring. |

4. EXAMPLES

4.1 Isolating words in a sentence

Consider the problem of finding each word in a sentence, where words are separated by the usual punctuation marks or by any number of blanks, and then printing them out.

The Prolog program below solves this problem by using a pattern that reads (recall that "`bl`" spans zero or more blanks): "find the longest substring of `S` terminating just before the first punctuation mark or blank and assign the substring to `W`; then skip over the punctuation mark and any number of blanks that may follow. `R` is what remains of `S` or the null string, if nothing remains".

```
pick_words(S) <-
  match (S, break(',:;?') ⚡ W ||
        any(',:;?') || bl || R) &
  prst(W) & nl &
  R = "" -> true;
pick_words(R).
```

For example, the goal

```
<- pick_words('It was night in white-walled Kaiin, and festival time.').
```

results in each word being displayed.

4.2 Mapping if-then-else expressions into Prolog conditionals

Let `S` be a character string of the form "if `S1` then `S2` else `S3`", where `S1`, `S2` and `S3` are valid IBM Prolog expressions balanced with respect to parentheses. We want to transform `S` into a string `T` of the form "`S1 -> S2; S3`", which is a valid conditional expression in IBM Prolog, and then execute `T`.

The program below solves this problem as follows. The first clause defines an auxiliary pattern "`b`", that allows keywords to be surrounded by any number of blanks.

The second clause first invokes a pattern that assigns to the variables `X`, `Y` and `Z` those substrings that are correctly parenthesized and that are separated, respectively, by the keywords "if", "then" and "else". Then, it creates a computable expression `T` from `X`, `Y` and `Z`, inserting parentheses to ensure correct operator precedence. Next, it calls the utility predicate "`st_to_exp`" to convert `T` into an expression `E`. Finally, it evaluates `E`.

The program goes as follows:

```
b(X,S) <-
  match(S, bl || X || bl).
```

```

cond(S) <-
  match(S, b('if') ||
        bal ϕ X ||
        b('then') ||
        bal ϕ Y ||
        b('else') ||
        bal ϕ Z) &
  T := '(' || X || ')' -> ' ||
        '(' || Y || ')' ; ' ||
        '(' || Z || ')' &
  st_to_exp(T,E) &
  call(E).

```

For example, the total

```

<- cond('if pragma(list, X) & X = 1
        then write(bracket_notation)
        else write(dot_notation)').

```

results in either “bracket_notation” or “dot_notation” being displayed, depending on the pragma options in force.

Checking parentheses is necessary to avoid that the Prolog scanner be confused. The conversion from string into expression or the execution of the expression will fail if any of the strings, S_1 , S_2 or S_3 , even though correctly parenthesized, is not otherwise valid. A more thorough syntactical check would be possible if, instead of “bal”, we used a pattern that fully captured the syntax of Prolog expressions. Patterns such as these could be structured like a grammar (see example 4.4).

4.3 Checking if a word is a palindrome

A palindrome is a word that reads the same forward and backward, such as “madam” and “otto” [3]. The patterns “palindrome(S)” below succeed for strings of size 0 or 1 and, recursively, for strings consisting of a character C followed by a palindrome followed by C.

The definition of “palindrome(S)” is:

```

palindrome(S) <- match(S, * << 1).
palindrome(S) <- match(S, len(1) ϕ C || palindrome || C).

```

For example, the goal

```

<- palindrome('madam').

```

will succeed.

4.4 Accepting restricted arithmetic expressions

A conventional BNF grammar for a restricted class of arithmetic expressions is:

```

<variable> ::= x | y | z
<addop> ::= + | -
<mulop> ::= * | /
<factor> ::= <variable> | '(' <expr> ')'
<term> ::= <factor> | <factor> <mulop> <term>
<expr> ::= <addop> <term> | <term> | <term> <addop> <expr>

```

This grammar can be immediately mapped into a Prolog program that recognizes arithmetic expressions. However, we must note that Prolog’s depth-first-search strategy limits us to right recursion (a “shift” operator to transform left-recursive into right recursive rules is suggested in [5]). The final program, exemplifying the recursive definition of a set of patterns, follows:

```

variable(X) <- match(X,any('xyz')).
addop(X) <- match(X,any('+ -')).
mulop(X) <- match(X,any('* /')).
factor(X) <- variable(X) |
            match(X, '(' || expr || ')').
term(X) <- factor(X) |
           match(X, factor || mulop || term).
expr(X) <- match(X, addop || term) |
         term(X) |
         match(X, term || addop || expr).

```

For example, the following goals

```

<- term('(x+y)/z').
<- expr('-(x+y)/z').

```

with both succeed indicating that the first string is a term and the second is an arithmetic expression.

These examples in fact indicate that the above program can be used to recognize arithmetic expressions, terms, factors, etc.

Going a step further, we recall that IBM Prolog allows adding prefixes to the names of predicates as a strategy to achieve a measure of modularity, producing, in a sense, “variable” predicate names. The notation of variable predicate names means that, besides the usual query “what values satisfy a given predicate?”, we can ask the query “what predicates (from a given set of predicates) are satisfied by a given value?”.

Suppose now that we prefix the predicates in the above program with “g”, understanding “g” as the name of the entire grammar. We have programmed the match meta-predicate to handle a goal like

```

<- isall(S,C,match('x*y', g:C)).

```

where “isall” is a set-forming utility meta-predicate, so that all g-prefixed predicates will be tried against “x*y”. In this example, S will be instantiated with the set (represented as a list, without repetitions) “[expr, term]”.

4.5 Accepting a sentence where subject and verb agree in number

Consider the problem of recognizing the sentences “he sings a song” and “we sing a song”, but not “he sing a song” or “we sings a song”. This section presents two solutions, closely following [10], that also serve to foreshadow the discussion in Section 5.

The first solution begins by saying that grammatical number can be either singular or plural. It does not have a single class “sentence”, but rather two classes: “singular sentence”, and “plural sentence”. Both classes involve a subject and a predicate, which are actually “N subject” and “N predicate” with N consistently replaced by either “singular” or “plural” (i.e. “singular subject” and “singular predicate”, etc.).

```

/* First Solution */
number(N) <- N = 'singular' | N = 'plural'.

sentence(N,X) <-
    number(N) &
    match(X,subject(N) || bl || predicate(N)).

subject(N,X) <-
    number(N) &
    match(X,pronoun(N)).

predicate(N,X) <-
    number(N) &
    match(X,verb(N) || bl || object).

```

```

object('a song').

pronoun('singular','he').

pronoun('plural','we').

verb('singular','sings').

verb('plural','sing').

```

For example, the goals

```

<- sentence(N,'he sings a song').
<- sentence(N,'we sing a song').

```

result in *N* being instantiated with “singular”, for the former, and “plural”, for the latter.

Going back to the program, note that we can completely eliminate all calls to “number” and, hence, its definition, without affecting the correctness of the program. The consistent substitution of the variable *N*, performed by unification, suffices to guarantee that subject and predicate agree in number, provided of course that “pronoun” and “verb”, etc. are correctly coded.

The second solution begins by enumerating the possible pronouns and verbs without distinguishing their grammatical number. The pattern “sentence” provisionally accepts any pronoun-verb-object sequence and then checks if the sentence agrees in number by calling “agrees”, whose failure may invalidate the initial acceptance. Thus, the definition of “sentence” does not explicitly treat the concept of number, leaving it to the definition of “agrees”.

```

/* Second Solution */

pronoun(X) <- X='he' | X='we'.

verb(X) <- X='sing' | X='sings'.

sentence(X) <-
  match(X, pronoun ⋄ P || bl || verb ⋄ V || bl || object) &
  agrees(P,V).

object('a song').

agrees('he','sings').

agrees('we','sing').

```

For example, given:

```

<- sentence('he sings a song').
<- sentence('we sing a song').

```

both goals succeed.

The solutions presented in this section exemplify two techniques which can be used together or separately to verify context-sensitive requirements, such as agreement in number, namely: (a) consistent substitution and (b) predicates that impose further conditions on matching substrings.

5. CODING GRAMMARS AS PROLOG PROGRAMS

We explore in this section how to use grammars as a paradigm for programming pattern-matching algorithms in Prolog using the match meta-predicate. We first discuss the simple case of context-free grammars and then extend the ideas to two-level grammars, which have the same power as type 0 grammars.

Our approach for context-free grammars is to map production rules into Prolog clauses to produce essentially recursive descent parsers. The mapping we describe is similar to that used for definite clause grammars [5]. The related question of compiling attribute grammars into Prolog is

treated in detail in [11]. A comprehensive approach to parsing, translation and compiling using Prolog is described in [7].

5.1 Coding context-free grammars

Recall that a grammar is context-free iff its production rules have the following general format:

$$L \rightarrow R_1 R_2 \dots R_n$$

where L is a nonterminal and R_1, R_2, \dots, R_n are nonterminals or terminals. Such production rules are also called context-free. Also recall that any derivation in a context-free grammar can be visualized as a tree, called a derivation-tree.

Now, going back to the examples of the preceding section, observe that, whenever we wanted to check whether a string S belonged to some class C , we define a predicate p containing a call to the `match` meta-predicate which, in turn, had the task of applying a pattern expression to S . The pattern expression had components p_1, p_2, \dots, p_n , some or all of which are patterns involving further predicate calls. This corresponds to the basic scheme proposed in the Introduction:

$$p(S) \leftarrow \text{match}(S, p_1 \parallel p_2 \parallel \dots \parallel p_n).$$

By exploring the similarities between context-free production rules and clauses with the above format, we immediately obtain a translation procedure that maps a context-free grammar G into a Prolog program g . For simplicity we assume that if α is a string of terminals of G then ' α ' is a valid Prolog string constant. The translation procedure goes as follows:

- (1) map each nonterminal P of G into a unary predicate symbol, that we denote by p by convention;
- (2) map each production rule of the form " $P \rightarrow \alpha$ ", where α is a string of terminals of G , into a unit clause of the form " $P(\alpha)$.", where p is the unary predicate symbol that P maps into;
- (3) map each production rule of the form " $P \rightarrow R$ ", where R is a nonterminal of G , into a clause of the form " $p(S) \leftarrow r(S)$.", where p and r are the unary predicate symbols that P and R map into;
- (4) map each production rule of the form " $P \rightarrow R_1 \dots R_n$ ", with $n > 1$, into a clause of the form " $p(S) \leftarrow \text{match}(S, r_1 \parallel \dots \parallel r_n)$.", where p is the unary predicate symbol that P maps into and r_1, \dots, r_n are the Prolog predicate symbols or string constants that R_1, \dots, R_n map into.

Let H be the start symbol of G . The resulting Prolog program is such that if the goal " $\leftarrow h('s')$." succeeds then the sentence s is in the language of G .

In addition to the translation rules described above, we may perform the following simplifications. Let $\mathbf{P} = \{P \rightarrow \alpha_1, \dots, P \rightarrow \alpha_n\}$ be a set of production rules with the same left-hand side P such that at least one right-hand side α_i is not a single terminal. Just as we may use " $P \rightarrow \alpha_1 \mid \dots \mid \alpha_n$ " as a short-hand notation for the rules in \mathbf{P} , we may also map them all into the single clause " $p(S) \leftarrow \beta_1 \mid \dots \mid \beta_n$ ", where β_i is obtained from α_i as explained above, if α_i is not a terminal string, or β_i is " $S = \alpha_i$ ", if α_i is a terminal string.

Let $\mathbf{P} = \{P \rightarrow a_1, \dots, P \rightarrow a_n\}$ be a set of production rules with the same left-hand side P and whose right-hand sides are single terminals. We may then map all rules in \mathbf{P} into the single clause " $p(S) \leftarrow \text{match}(S, \text{any}('a_1 \dots a_n'))$ ".

Furthermore, we may optimize the final program by symbolic execution. For example, we may replace the pair of clauses

$$\begin{aligned} n('abc'). \\ z(S) \leftarrow \text{match}(S, n \parallel z). \end{aligned}$$

by the single clause:

$$z(S) \leftarrow \text{match}(S, 'abc' \parallel z).$$

In addition to the example in Section 4.4, the following example illustrates the translation procedure. Consider the well-known context-free grammar whose language consists of all sentences of the form $a^n b^n$:

$$\begin{aligned} Z &\rightarrow ab \\ Z &\rightarrow aZb \end{aligned}$$

When applied to this grammar, the translation procedure results in the following Prolog program:

```
z('ab').
z(S) <- match(S,'a' || z || 'b').
```

Thus, for example, the following goal will succeed:

```
<- z('aaabbb'),
```

indicating that the string 'aaabbb' is in the language generated by the grammar.

Naturally, the translation procedure does not guarantee that the Prolog program will always converge, which in fact will not occur if the grammar is left-recursive. However, given a left-recursive context-free grammar G it is always possible to algorithmically find a non-left recursive context-free grammar G' such that G and G' generate the same language [12]. Therefore, one may safely assume that the input grammar is non-left-recursive. However, this result still leaves to the programmer the task of properly ordering the clauses within the Prolog program to avoid loops.

5.2 Coding two-level grammars

The simple strategy described in Section 5.1, when generalized to type 0 grammars, will translate production rules into unrestricted clauses, that is, clauses whose left-hand side contains more than one literal. This follows because type 0 production rules may have a left-hand side with more than one nonterminal. Such strategy was suggested in [6] and has the disadvantage of leading us outside standard Prolog. However, we can replace type 0 grammars by another formalism of equivalent power, namely, *two-level grammars*, for which we sketch a relatively simple translation procedure from production rules into Prolog clauses.

Two-level grammars have been proposed by A. van Wijngaarden as a formalism to describe the entire syntax of ALGOL 68 [13]. An easy-to-read introduction is given in [10]. In Ref. [14] we provide extensive examples, applying two-level grammars to database specification. A generalization of logic programs based on two-level grammars can be found in [1]. Conditions for the existence of parsers have been investigated in several papers; Ref. [15] surveys previous research and proposes a parser based on LL(1) techniques.

A two-level grammar is essentially a finite specification of an infinite set of context-free production rules. As the name indicates, it has two types of context-free rules. The language associated with the grammar is that generated by the rules of the second level from the start symbol. But second-level rules cannot be applied as they are because they contain certain "unresolved" symbols. So, such rules must be converted, in a preliminary step, into ordinary rules by transforming these symbols into ordinary terminals and nonterminals. This preliminary transformation is accomplished by the rules of the first level.

More precisely, a *two-level grammar* is a tuple $G = (M, V, H, T, MR, HR, S)$ where:

- M is a set of *metanotions*;
- V is a finite set of *metaterminals*, disjoint from M ;
- H is a finite set of *hypernotions*, a finite subset of $(M \cup V)^+$;
- T is a finite set of *terminals*;
- MR is a finite set of *metarules*, which are context-free production rules such that (M, V, W, MR) is a context-free grammar, for each $W \in M$;
- HR is a finite set of *hyper-rules*, which are context-free production rules such that the head is a hypernotation and the body is a string of hypernotions or terminals. For legibility, we will enclose each occurrence of a hypernotation within angular brackets;
- S is the *starting symbol*.

The set of *strict rules* of a hyper-rule h is the set of context-free production rules generated from h by consistently replacing each metanotion W occurring in h by a string of metaterminals generated from W by the metarules. The language generated by G is the set of strings of terminals generated from S by the (infinite) set of strict rules associated with the hyper-rules of G .

To illustrate the power of two-level grammars, consider the language consisting of all sentences of the form $a^n b^n c^n$. It is well-known that this language is context-sensitive, but not context-free. In fact, it can be generated by the context-sensitive grammar G_1 whose start symbol is Z and whose rules are:

Rules of grammar G_1 .

$$\begin{aligned} Z &\rightarrow aZBC \\ Z &\rightarrow aBC \\ CB &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc. \end{aligned}$$

The two-level grammar G_2 , whose rules are listed below [16], defines the same language:

Rules of grammar G_2 .

metarules

$$\begin{aligned} m_1. & L \rightarrow a \mid b \mid c. \\ m_2. & T \rightarrow i \mid iT. \end{aligned}$$

hyper-rules

$$\begin{aligned} h_1. & \langle z \rangle \rightarrow \langle aT \rangle \langle bT \rangle \langle cT \rangle. \\ h_2. & \langle LiT \rangle \rightarrow \langle Li \rangle \langle LT \rangle. \\ h_3. & \langle Li \rangle \rightarrow L. \end{aligned}$$

The metanotation L generates the permissible letters a , b and c , whereas the metanotation T generates sequences of one or more i symbols, which provide a device to count the number of occurrences of a , b and c .

As already indicated, hypernotations are represented enclosed in angular brackets. The hyper-rule h_1 contains the start symbol $\langle z \rangle$ and defines an infinite set of context-free rules, obtained by replacing T by each string generated by the metarules from T :

$$\begin{aligned} \langle z \rangle &\rightarrow \langle ai \rangle \langle bi \rangle \langle ci \rangle. \\ \langle z \rangle &\rightarrow \langle aii \rangle \langle bii \rangle \langle cii \rangle. \\ &\dots \end{aligned}$$

Note that the number of occurrences of i is guaranteed to be the same for the sequences of a 's, b 's and c 's simply as a consequence of the uniform substitution performed on the three occurrences of T .

The hyper-rule h_2 just breaks a sequence of more than one i by separating the first i (to be handled by the last hyper-rule) from the rest of the sequence, which is then handled recursively. It defines the following set of context-free rules:

$$\begin{aligned} \langle aii \rangle &\rightarrow \langle ai \rangle \langle ai \rangle. \\ \langle aiii \rangle &\rightarrow \langle ai \rangle \langle aii \rangle. \\ &\dots \\ \langle bii \rangle &\rightarrow \langle bi \rangle \langle bi \rangle. \\ \langle biii \rangle &\rightarrow \langle bi \rangle \langle bii \rangle. \\ &\dots \\ \langle cii \rangle &\rightarrow \langle ci \rangle \langle ci \rangle. \\ \langle ciii \rangle &\rightarrow \langle ci \rangle \langle cii \rangle. \\ &\dots \end{aligned}$$

The hyper-rule h_3 merely yields each a , b and c as a terminal. It results in the following rules:

$$\begin{aligned} \langle ai \rangle &\rightarrow a. \\ \langle bi \rangle &\rightarrow b. \\ \langle ci \rangle &\rightarrow c. \end{aligned}$$

Thus, for example, a derivation of the sentence 'aabbcc' will be:

```

<z> → <aii> <bii> <cii>
      → <ai> <ai> <bii> <cii>
      → a <ai> <bii> <cii>
      → aa <bii> <cii>
      → aa <bi> <bi> <cii>
      → aab <bi> <cii>
      → aabb <cii>
      → aabb <ci> <ci>
      → aabbcc <ci>
      → aabbcc

```

Although two-level grammars have only simple, i.e. context-free rules, one may ask how a grammar with an infinite number of rules can be used in practice. In our terms, one may ask how to obtain a Prolog program that recognizes sentences of the language the grammar generates. The basic idea, in outline, is quite simply to map the metarules into Prolog clauses that produce a fair enumeration of the strings of metaterminals derivable from the metanotions and to map the hyper-rules into Prolog clauses that use such strings to produce, in turn, a fair enumeration of the strict rules that will then parse the input string.

Such a set of Prolog clauses is by no means easy to obtain, for Prolog's depth-first search strategy makes it very difficult to code fair enumerations and constantly leads to a infinite loops. We will illustrate these problems by coding a parser for the two-level grammar G_2 generating sentences of the form $a^n b^n c^n$. As in other examples, we avoid left-recursive rules right from the onset (in fact, the grammar in [16] had left-recursive rules).

Since metarules are context-free, we can in principle use the strategy of Section 5.1 to map metarules into Prolog clauses. However, we must pay attention to the fact that such clauses will be primarily used to enumerate sentences.

For example, Fig. 1 shows the Prolog clauses for the metarules of the grammar G_2 . Note that successive calls to "l" will generate "a", "b" and "c", whereas successive calls to "t" will generate strings of i's of increasing length. Therefore, clauses M_1 and M_2 indeed produce a fair enumeration of the strings of metaterminals derivable from the metanotions of G_2 .

Although clauses M_1 and M_2 produce the desired result, in general, it may be difficult to code fair enumerations in Prolog. Suppose for example that we wished to define instead a fair enumeration of all strings consisting of "a" or "b" followed by an arbitrary number of i's. Then, the following clauses would NOT work since successive calls to "z", upon backtrack, would produce only strings consisting of "a" followed by an arbitrary number of i's:

```

/*      Parser for  $G_2$       */
/* clauses for the metarules */
(M1) l(L) <- match(L, any('abc')).
(M2) t(T) <- T = 'i' | match(T, 'i' || t).
/* clauses for the hyper-rules */
(H1) h('z', S) <- t(T) &
      T1 := 'a' || T &
      T2 := 'b' || T &
      T3 := 'c' || T &
      match(S, h(T1) || h(T2) || h(T3)).
(H2) h(X, S) <- match(X, l & L || 'i') & match(S, L).
(H3) h(X, S) <- match(X, l & L || 'i' || t & T) &
      T1 := L || 'i' &
      T2 := L || T &
      match(S, h(T1) || h(T2)).

```

Fig. 1

```
t(T) <- T='i' | match(T,'i' || t).
z(Z) <- match(Z,'a' || t).
z(Z) <- match(Z,'b' || t).
```

Returning to our original example, Fig. 1 also contains the clauses for the hyper-rules of grammar G_2 . In general, for each hyper-rule h_i of G_2 , we built a clause H_i that progressively generates the set of strict rules associated with h_i by instantiating the metanotions occurring in h_i with strings of metaterminals created by calling the predicates associated with the metanotions. The head of H_i has the form “ $h(u,S)$ ” where

- h is a binary predicate selected arbitrarily;
- if the head of h_i is a terminal string α , u is equal to ‘ α ’, otherwise u is a variable;
- S represents the string being recognized.

The body of clause H_i has three parts. The first part instantiates each metanotion m occurring in the body of h_i by:

- calling the predicate associated with m , if m occurs only in the body of h_i (this is the case of the metanotion T occurring in clause H_1 in Fig. 1);
- calling `match` to test the first argument u against a pattern representing the hypernotation in the head of h_i .

The second part consists of a sequence of assignments to construct strings that represent an instantiation of each hypernotation occurring in the body of h_i . The instantiation is generated by replacing each metanotion by the string obtained in the first part of the body of H_i .

Finally, the last part of H_i is a call to `match` whose pattern stands for the body of h_i , with the instantiations constructed in the second part. Thus, the last part represents the body of a strict rule of h_i .

Therefore, upon backtrack, clause H_i will successively construct the body of the strict rules associated with h_i .

Naturally, we could have translated the left-hand side of the first hyper-rule H_1 without the help of h , but we maintained its use to permit answering questions such as

```
<- match('aaabbbccc', h(X)).
```

which results in the instantiation of X by the start symbol ‘ z ’, indicating that the string is in the language generated by the grammar.

The program shown in Fig. 1 for grammar G_2 works correctly in the sense that, if the goal “ $<- h('z', s)$ ” succeeds, then the string s is in the language of grammar G_2 . However, if s is not in the language, the program does not halt. This problem can be bypassed to some extent by imposing a limit on the pattern t used in the call to `match` in clause M_2 , which incidentally illustrates the usefulness of limits. For example, we could arbitrarily impose a limit of 10 by rewriting clause M_2 as follows:

```
(M2) t(T) <- T='i' | match(T,'i' || t << 10).
```

However, the program in Fig. 1 is very inefficient. We then discuss in the next few paragraphs how to optimize it.

Going back to the grammar, we can obtain a considerable speed-up by eliminating the metanotion l and by expanding and simplifying the hyper-rules. The resulting grammar is:

Rules of grammar G'_2 .

metarule

```
m2. T → i | i T.
```

hyper-rules

```
h1 <z> → <a T> <b T> <c T>.
h21. <a i T> → a <a T>.
h22. <b i T> → b <b T>.
h23. <c i T> → c <c T>.
```

```

(M2) t(T) <- T = 'i' | match(T, 'i' || t).
(H1) h('z', S) <- t(T) &
      T1 := 'a' || T &
      T2 := 'b' || T &
      T3 := 'c' || T &
      match(S, h(T1) || h(T2) || h(T3)).
(H21) h('ai', 'a').
(H22) h('bi', 'b').
(H23) h('ci', 'c').
(H31) h(X, S) <- match(X, 'ai' || t c T) &
      T2 := 'a' || T &
      match(S, 'a' || h(T2)).
(H32) h(X, S) <- match(X, 'bi' || t c T) &
      T2 := 'b' || T &
      match(S, 'b' || h(T2)).
(H33) h(X, S) <- match(X, 'ci' || t c T) &
      T2 := 'c' || T &
      match(S, 'c' || h(T2)).

```

Fig. 2

```

h('z', S) <- match(S, h('a' || T) || h('b' || T) || h('c' || T)).
h('a' || 'i', 'a').
h('b' || 'i', 'b').
h('c' || 'i', 'c').
h('a' || 'i' || T, S) <- match(S, 'a' || h('a' || T)).
h('b' || 'i' || T, S) <- match(S, 'b' || h('b' || T)).
h('c' || 'i' || T, S) <- match(S, 'c' || h('c' || T)).

```

Fig. 3

$$\begin{aligned}
 h_{31}. \langle a i \rangle &\rightarrow a. \\
 h_{32}. \langle b i \rangle &\rightarrow b. \\
 h_{33}. \langle c i \rangle &\rightarrow c.
 \end{aligned}$$

In terms of the program in Fig. 1, these simplifications correspond to symbolically executing the calls to `!` inside the patterns, simplifying the resulting clauses, and then symbolically executing the calls to `h('ai')`, `h('bi')` and `h('ci')`. Figure 2 exhibits the resulting program.

We can optimize the program in Fig. 2 even further by a careful analysis of the clauses along the following lines. Let “`||`” be defined as an infix, right-to-left associative operator (the choice of the operator is immaterial, but convenient at this point). Represent a string “`c1 ... cn`” by a term of the form “`c1 || ... || cn`”. We obtained the program in Fig. 3 from that in Fig. 2 by:

- dropping the computable expressions that build the values of T1, T2 and T3 and replacing each occurrence of T_i by the corresponding right-hand side of the original computable expression;
- replacing X by a term that represents the same hypernotation as the pattern in the first call to `match` in clauses H_{3i} , $i = 1, 2, 3$;
- dropping entirely all calls to `t`.

The first two transformations are correct essentially because, in every call to `h` inside a pattern, the argument will always be instantiated with a string consisting of a, b or c followed by one or more i's. Therefore, we can let the usual Prolog clause selection rule choose which clause on `h` to call. The last transformation is in turn correct because the last six clauses of the program in Fig. 2 generate the sequences of i's bottom-up making the calls to `t` unnecessary.

The terms used as the first argument in the heads of the last six clauses work as expected in the present example. Unfortunately, the term construction rules of Prolog may cause problems in other cases. For instance, consider an attempt to perform the unification below (the equal sign “`=`” denotes the Prolog unification operator):

$$'i' || 'i' || 'i' || 'a' = T || 'i' || 'a'$$

One might expect, on a first thought, that the unification succeeds, with $T = 'i' || 'i'$, but it fails instead.

To avoid such problems we can pass to `match` the task of splitting these terms in all possible ways and then calling the appropriate predicates. To invoke this additional facility of `match`, the terms must be coded as “`<c1 || ... || cn>`”. Although not required by our running example, the program in Fig. 3 can be easily adapted to use the facility.

To summarize, we moved from a program that closely mimicks grammar G_2 to a final program that contains clauses only for the hyper-rules. Consistent substitution therefore is the only feature left from two-level grammars. Indeed, the final program works by first assigning to T a string of i's of the same length as the string of a's in the input; then, this value of T is transmitted by consistent substitution to count the number of b's and c's.

To close the example, we observe that, for the sake of analyzing the use of two-level grammars, we strived to keep the Prolog program as close as we could to the grammar by directly mapping the production rules into Prolog clauses and by avoiding extra programming “machinery”. The most striking case is the use of sequences of i’s for counting. Indeed, by using the pre-defined patterns of Section 3, we can write a single pattern expression that solves the problem far more efficiently (recall that positions start from zero):

```
h(S) <- match(S,
    span('a') || pos(1) ||
    span('b') || pos(2 * I + 1) ||
    span('c') || pos(3 * I + 2)).
```

Besides consistent substitution, two-level grammars also feature “predicates” and “variables” associated with metanotations. A predicate in the context of a two-level grammar is a hypernotation resulting in either the empty symbol, if some condition is satisfied, or in a string from which the hypernotation cannot be eliminated, otherwise. Variables associated with a metanotation T range over the strings generated from T by the metarules and are usually denoted by T1, T2, . . .

To give a simple example, suppose we want to define a language whose sentences are sequences of a’s b’s and c’s with three different lengths. It suffices to replace the first hyper-rule of grammar G_2 by:

```
<z> → <a T1> <b T2> <c T3>
      <where T1 not-equal T2>
      <where T1 not-equal T3>
      <where T2 not-equal T3>
```

where the symbols T1, T2 and T3 are variables associated with the metanotation T and the hypernotations “<where T1 not-equal T2>”, “<where T1 not-equal T3>” and “<where T2 not-equal T3>” play the role of predicates. Consistent substitution is not applicable to T1, T2 and T3 so that they may or may not be replaced by the same sequence of i’s. The production rules associated with the predicates guarantee that the variables will not be replaced by the same sequence. Details about the use of predicates are given in [10].

The mapping of this rule into a Prolog clause is immediate:

```
h('z', S) <-
    match(S, h('a' || T1) || h('b' || T2) || h('c' || T3)) &
    ¬(T1 = T2) &
    ¬(T1 = T3) &
    ¬(T2 = T3).
```

Therefore, the use of predicates and variables in two-level grammars is entirely similar to attaching goals to the bodies of rules in a definite clause grammar [5].

Finally, we remark that we view the two-level grammar paradigm just as a general strategy to obtain Prolog programs that manipulate strings. Therefore, we do not insist on a strict adherence to it. Simpler and more efficient programs may be obtained for specific problems, either directly or by refining the initial “canonical” program derived from a two-level grammar.

Readers interested in further looking into the subject are referred to [10], which contains the grammars for sentences agreeing in number behind the Prolog programs in Section 4.5. They will also notice the analogy between consistent use of grammatical number in sentences and consistent use of data types in programming languages. The relevance of two-level grammars to parametric data type specification has been noted in [17].

6. CONCLUSION

This paper presented a string pattern-matching feature for Prolog that fits with the style of the language. The feature offers a higher-level, declarative notation for describing operations on character strings that compares favorably with the primitives commonly found in Prolog dialects.

Coupled with Prolog's unique characteristics, it led to patterns more powerful than their original counterparts in SNOBOL 4.

The current prototype emulates the use of `append` to implement recursive descent parsers directly from the production rules, as described in [7, Section 2.3]. Further work towards more efficient algorithms may enhance the current implementation of `match`.

REFERENCES

1. Maluszynski J. and Nilsson J. F., A comparison of the logic programming language Prolog with two-level grammars. *Proc. of the First Int. Logic Programming Conf.*, pp. 193-199 (1982).
2. Kowalski R. A., *Logic for Problem Solving*, North-Holland, Amsterdam (1979).
3. Griswold R. E., Poage J. F. and Polonsky I. P., *The SNOBOL4 Programming Language*, Prentice-Hall, Englewood Cliffs, N.J. (1971).
4. Griswold R. E. and Griswold M. T., *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, N.J. (1983).
5. Walker A., McCord M., Sowa J. F. and Wilson W. G., *Knowledge Systems and Prolog*. Addison-Wesley, Reading, Mass. (1987).
6. Colmerauer A., Metamorphosis grammars. In *Natural Language Communication with Computers*. (Edited by Balci L.), pp. 133-189. Springer, New York (1978).
7. Cohen J. and Hickey T. J., Parsing and compiling using Prolog. *ACM Trans. on Programming Languages and Systems* 9(2), 125-163 (April 1987).
8. Clark K. L. and McCabe F. G., *Micro-Prolog: Programming in Logic*. Prentice-Hall Englewood Cliffs, N.J. (1984).
9. Deransart P. and Maluszynski J., Relating logic programs and attribute grammars. *J. Logic Programm.* 119-155 (1985).
10. Peck J. E. L., Two level grammars in action. In *Information Processing 74* (Edited by Rosenfeld J. L.), pp. 317-321. North-Holland, Amsterdam (1974).
11. Arbab B., Compiling circular attribute grammars into Prolog. Technical Report G320-2776, IBM Los Angeles Scientific Center (January 1986).
12. Aho A. V. and Ullman J. D., *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, N.J. (1972).
13. van Wijngaarden A. *et al.* (Eds), Revised report on the algorithmic language ALGOL 68. *Acta Inform.* 5, 1-236 (1975).
14. Furtado A. L., Veloso P. A. S. and Casanova M. A., A grammatical approach to data bases. In *Information Processing 83* (Edited by Mason R. E. A.), pp. 705-710. North-Holland, Amsterdam (1983).
15. Fisher A. J., Practical LL(1)-based parsing of van Wijngaarden grammars. *Acta Inform.* 21, 559-584 (1985).
16. Pagan F. G., *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice-Hall, Englewood Cliffs, N.J. (1981).
17. Gonnert G. H. and Tompa F. W., A constructive approach to the design of algorithms and their data structures. Technical Report CS-80-47, University of Waterloo (1980).

About the Author—MARCO ANTONIO CASANOVA has a B.Sc. in Electronic Engineering from the Army Institute of Engineering, an M.Sc. in Computer Science from the Pontifical Catholic University of Rio de Janeiro and a Ph.D. and an M.A. in Applied Mathematics, both from Harvard University.

Dr Casanova joined the IBM Brasil Scientific Center in November 1982 and is currently conducting research in database theory, database management systems and logic programming. From 1980 to 1982 he was Assistant Professor at the Department of Informatics of the Pontifical Catholic University of Rio de Janeiro, where he was appointed to the position of Graduate Program Coordinator in 1981.

He is author of the book "*The Concurrency Control Problem for Database Systems*" and coauthored the books "*Principles of Distributed Database Management Systems*" and "*Logic Programming*" (both in Portuguese). He has also published several articles in international scientific journals.

About the Author—ANTONIO L. FURTADO has a B.Sc. in Economics from the State University of Rio de Janeiro, an M.Sc. in Business Administration from the Getulio Vargas Foundation, an M.Sc. in Computer Science from the Pontifical Catholic University of Rio de Janeiro and a Ph.D. in Computer Science from the University of Toronto.

Dr Furtado is currently Professor at the Department of Informatics of the Pontifical Catholic University of Rio de Janeiro. From April 1986 to March 1988, he worked at the IBM Brasil Scientific Center as Senior Visiting Researcher. His areas of interest include Data Structures, Databases, Programming Languages and Logic Programming.

He coauthored the book "*Formal Techniques for Data Base Design*" and published several papers in international scientific journals. He participated in various program committees of international conferences, including the IFIP World Computer Congress, 1986, where he chaired the Information Systems area.

APPENDIX I

A Prototype Implementation

```

J := I + K &
match1(Y,R,J) &
(var(S) & stconc(X,Y,S) |
~var(S)).

/* initialization */
initiate() <-
  pragma(list,1) &
  pragma(long,1) &
  dcio('OUTEXP', 'OUTPUT',
    'STACK', 255) &
  dcio('INEXP', 'INPUT',
    'CONSOLE', 255).

<- initiate().

/* special operators */
op("#",r1,56).
op("<<",r1,57).
op("<<",prefix,51).
op(">",suffix,52).

/* the match meta-predicate */
match(S,P) <-
  match1(S,P,-1).

match1(S,P,I) <-
  (var(P) & / | ~(P = *|*)) & / &
  ptype(P,Q,Z,L,T,S) &
  (T == 0 & var(L)) -> S = Q;
  (T == 0 & ~var(L)) -> p_call(Q,S,L);
  T == 1 -> match_f(S,Q,Z,L);
  match_s(S,Q,Z,L,[I,'']).

match1(S,E,I) <-
  ~var(E) &
  E = P || R &
  ptype(P,Q,Z,L,T,X) & / &
  (~var(S) & stconc(X,Y,S) |
  var(S)) &
  ((T == 0 & var(L)) -> X = Q;
  (T == 0 & ~var(L)) -> p_call(Q,X,L);
  T == 1 -> match_f(X,Q,Z,L);
  match_s(X,Q,Z,L,[I,Y])) &
  stlen(X,K) &

J := I + K &
match1(Y,R,J) &
(var(S) & stconc(X,Y,S) |
~var(S)).

/* for context-free patterns */
match_f(X,P,Z,L) <-
  (P =.. [F A] &
  ~(A = [Y] &
  ~var(Y) & Y = <W ) & / |
  ~(P =.. [F A]) &
  F = P & A = nil) & / &
  append(A,[VAR],B) &
  Q =.. [F B] &
  p_call(Q,Z,L,VAR,X).

/* for context-sensitive patterns */
match_s(X,P,Z,L,IY) <-
  (P =.. [F A] & / | F = P & A = nil) &
  append(A,[VAR,IY],B) &
  Q =.. [F B] &
  p_call(Q,Z,L,VAR,X).

p_call(P,Z,N,V,X) <-
  var(N) &
  V = X &
  call(P,Z).

p_call(P,Z,N) <-
  ~var(N) & (var(P) | stringp(P)) &
  NN := N &
  P = Z &
  (stringp(Z) &
  stlen(Z,M) &
  (gt(M,NN) & / & fail |
  le(M,NN)) |
  ~stringp(Z)).

p_call(P,Z,N,V,X) <-
  ~var(N) & ~(var(P) | stringp(P)) &
  NN := N &
  call(P,Z) &
  stlen(V,M) &

```

```

(gt(M,NN) & / & fail |
  !e(M,NN) & V = X).

/* determines the type of components */
/* 0 - constant or variable          */
/* 1 - context-free pattern          */
/* 2 - context-sensitive pattern     */
ptype(P,R,PREF,L,T,X) <-
  (var(P) | stringp(P)) ->
  (T = 0 & P = R);
(P = (Q << L) &
  (var(Q) | stringp(Q))) ->
  (T = 0 & Q = R);
((P = Q & X & / | P = Q) &
  !im(Q,R1,L) &
  pref(R1,R,PREF) &
  (R =.. [F A] & / | F = R & A = nil) &
  (F == break -> T = 2;
  (length(A,M) &
  ~PREF == nil ->
    (axn(PREF:F,N,*) & T := N - M);
  PREF = nil ->
    (axn(F,N,*) -> T := N - M;
    T = 1))))).

lim(P,Q,LIM) <-
  (P =.. ["<<"Q,LIM] & / |
  P = Q).

pref(P,Q,PREF) <-
  (P =.. [CO,PREF,RR] &
  ebcdic(CO,122) & / & Q = RR |
  PREF = nil & Q = P).

/* context-free          */
/* SNOBOL-like patterns */

len(N,X) <-
  ~(var(N) | int(N)) -> V := N;
  V = N &
  stlen(X,V).

any(X,Y) <-
  substring(X,Y,*,1).

notany(X,Y) <-
  stlen(Y,1) &
  ~substring(X,Y,*,1).

bal(X) <- bal(X,0).

ball('',M) <- / & fail.
ball(X,M) <-
  stlen(X,L) &
  (substring(X,'',0,1) ->
    N := M + 1;
  substring(X,'',0,1) ->
    N := M - 1;
  true -> N = M) &
  !t(N,0) -> (/ & fail);
(N == 0 & L == 1) -> true;
(~(N == 0) & L == 1) ->
  (/ & fail);
true ->
  (L1 := L - 1 &
  substring(X,Y,1,L1) &
  ball(Y,N)).

arbno(P,X) <-
  ~var(X) &
  (X == '' |
  match(X,P || arbno(P)) & / |
  match(X,P)).

arbno(P,X) <-
  var(X) &
  (X = '' |
  match(X,P) |
  arbno(P,Z) &
  match(Y,P) &
  ~(Z == '') &
  X := Z || Y).

/* context-sensitive    */
/* SNOBOL-like patterns */

bl(S,[*,Y]) <-
  forall(substring(S,C,*,1),
    C == ' ') &
  ~substring(Y,'',0,1).

span(Z,S,[*,Y]) <-
  stlen(S,L) & gt(L,0) &
  forall(substring(S,C,*,1),
    substring(Z,C,*,1) &

```

```

    ~(substring(Y,D,0,1) &
      substring(Z,D,*,1)).

break(Z,S,[*,Y]) <-
  stlen(S,L) & gt(L,0) &
  forall(substring(S,C,*,1),
    ~substring(Z,C,*,1)) &
  substring(Y,D,0,1) &
  substring(Z,D,*,1).

rpos(I,',[*,Y]) <-
  (~var(I) & ~int(I)) -> J := I;
  J = I.

rpos(I,',[*,Y]) <-
  ((~var(I) & ~int(I)) -> K := I;
  K = I) &
  stlen(Y,L) &
  K := L - 1 .

rtab(N,S,[I,*,*]) <-
  stlen(S,L) &
  (~var(N) & ~int(N)) -> K := N;
  K = N &
  K := I + L.

rtab(N,S,[*,Y]) <-
  stlen(Y,L) &
  ((~var(N) & ~int(N)) -> K := N;
  K = N) &
  K := L - 1 .

/* to handle hyper-rules */

match_f(X,P,G,L) <-
  P =.. [H,ZZ] &
  ~var(ZZ) & ZZ = <Z> &
  hprep(H,<Z>,X,M) &
  Q =.. M &
  call(Q,G).

hprep(H,<Z>,X,[H,<P>,X]) <-
  W =.. [H,<P>,*,*] &
  ax(W,*) &
  hpr1(P,Z).

hpr1(X,Y) <-
  copy(X,CX) & copy(Y,CY) &
  listvar(X,LX) & listvar(Y,LY) &
  listvar(CX,LCX) & listvar(CY,LCY) &
  hpr2(CX,A) & hpr2(CY,B) &
  comp_list(A,B) &
  hpr3(LCX,LCX1) & hpr3(LCY,LCY1) &
  LX = LCX1 & LY = LCY1.

hpr2(X,Y) <-
  (~var(X) & X = A || B) ->
  (hpr2(A,A1) &
  hpr2(B,B1) &
  append(A1,B1,Y));
  Y = [X].

hpr3(nil,nil) <- /.
hpr3([X Y],[A B]) <-
  (listp(X) -> hpr4(X,A);
  A = X) &
  hpr3(Y,B).

hpr4([X],X) <- /.
hpr4([X Y],X || Z) <-
  ~(Y = nil) &
  hpr4(Y,Z).

/* general-purpose utilities */

st_to_exp(Z,X) <-
  (simple_chars(Z) &
  st_to_at(Z,T) & X = T & /() |
  stlen(Z,L) &
  L1 := (L - 1) &
  (substring(Z, '.',L1,1) &
  T = Z & /() |
  T := Z || ' ' .) &
  prst(T,'OUTEXP') & nl('OUTEXP') &
  read(X,'INEXP')).

exp_to_st(X,Y) <-
  (stringp(X) & Y = X & /() |
  atomic(X) & st_to_at(T,X) &
  Y = T & /() |
  writes(X,'OUTEXP') &
  nl('OUTEXP') &
  writes('eof*', 'OUTEXP') &
  nl('OUTEXP') &
  exp_to_st1(' ',Y)).

```

```

exp_to_stl(X,Y) <-
  readli(T,'INEXP') &
  (substring(T,'*eof*',1,5) &
  Y := X & /() |
  Z := X || T &
  exp_to_stl(Z,Y)).

simple_chars(X) <-
  st_to_li(X,L) &
  forall(on(E,L),
    (ebcdic(E,N) & ge(N,129) &
    le(N,169) |
    digit(E))).

isall(X,Y,Z) <-
  (compute(set,Y,Z,[],X) &
  /() | X = []).

forall(X,Y) <-
  ~ (call(X) & ~ call(Y)).

on(X,[X Y]).
on(X,[Y Z]) <- on(X,Z).

append([],X,X).
append([X Y],Z,[X W]) <-
  append(Y,Z,W).

length([],0) <- /().
length([X Y],N) <- length(Y,M) &
  N := M + 1.

listp(X) <- ~var(X) & X = (*.*).
listp(X) <- X == nil.

comp_list(nil,nil).
comp_list([X Y],[A B]) <-
  (var(A) & / | var(X)) ->
  (var(A) &
  ((A = X & comp_list(Y,B)) |
  (comp_list(Y,[P B]) &
  (listp(P) -> A = [X P];
  true -> A = [X [P]])) |
  ~var(A) & var(X) &
  ((X = A & comp_list(Y,B)) |
  (comp_list([P Y],B) &
  (listp(P) -> X = [A P];
  true -> X = [A [P]])) )) )
  true ->
  (X = A & comp_list(Y,B)).

```