PALM - A pattern language for molecular biology

Carsten Helgesen † and Peter R. Sibbald ‡

† Department of Informatics, University of Bergen
Høyteknologisenteret, 5020 Bergen, Norway
E-mail: carstenh@ii.uib.no

‡ EMBL, Data Library,
Meyerhofstrasse 1, Postfach 102209, 6900 Heidelberg, Germany
E-mail: sibbald@embl-heidelberg.de

Abstract

This paper presents a new pattern language, PALM, for describing patterns in molecular biology sequences. The language is intended for representing knowledge about such patterns in a declarative, clear and consise way. It is also shown that its expressive power enables the definition of any regular or context free language, and also higher languages in the Chomsky hierarchy by parameter attachment, variables and procedural attachment. It is also possible to define approximate patterns. The language is rigorously defined, and several examples of its use and expressive power are given.

1 Introduction

Identifying and describing patterns in molecular biological sequences has become a very important research activity. Certain short patterns, called *motifs*, in protein sequences can have biological significance. Citing from the introduction of a series of motif descriptions in Trends in Biochemistry [TIBS 15 - august 1990]:

> The sequences of many proteins contain short, conserved motifs that are involved in recognition and targeting activities, often separate from other functional properties of the molecule in which they occur. These motifs are linear, in the sense that three-dimensional organisation is not required to bring distant segments of the molecule together to make the recognizable unit. The conservation of these motifs varies: some are highly conserved, while others, for example, allow substitution that retain only a certain pattern of charge across the motif.

We suggest that those wishing more background on the subject of protein patterns refer to [Taylor 1992].

There are basically two approaches to abstracting common patterns from a set of related molecular biological sequences. One class of methods tries to summarize the similarities by a *weight matrix* (or *profile*) expressing the frequency with which each amino acid or indel (insertion or deletion) occurs in a multiple alignment of the sequences [Gribskov et al. 1988]. In a related approach each column is summarised by the nearest matching set in a predefined hierarchy of physiochemically determined groups of amino acids [Smith and Smith 1989]. These summaries take the entire alignment into acount, and the similarities are largely represented numerically. To compare a sequence to a profile the two are aligned, using an extended pairwise alignment approach, and a match is found if the score is sufficiently high.

Other approaches try to focus on important subsequences, interleaved with less important areas in the sequence. With these approaches the common pattern is represented as a list of short motifs or subpatterns common to all the related sequences. Some methods allow only exact matching, i.e. any ambiguity in the pattern must be explicitly defined as an alternative, while others allow approximate matching where substitution, insertion or deletion can take place in a matching sequence.

Several computer programs have been implemented which allow the user to define and search for patterns in biological sequences, e.g. SCRUTINEER [Sibbald and Argos 1990], and some unnamed systems [Staden

1988], [Taylor 1989] and [Saurin and Marliere 1987]. Most of them define the pattern in an interactive session, or by filling in parameters in a predefined file format. Thus the pattern definitions are more directed towards machine than towards man, and are often not easy to understand from their description.

Other systems represent patterns in a more direct manner. QUEST [Abarbanel et al. 1984] is a pattern specification language based on regular expression notation, but it does not allow approximate matching. PAMALA [Mehldau 1991] is also based on regular expressions, and it also allows definition of approximate patterns, as well as profiles, within one framework.

Prosite [Bairoch 1991] is a database of short patterns describing biologically important sites in protein sequences. In Prosite a declarative notation is used to describe patterns, but this notation is very limited. It is only capable of expressing a subset of the regular expressions, and has no notion of approximate matching. Still the latest version of Prosite (December 1992) contains more than 800 useful patterns, and some rules which cannot be expressed within its pattern notation.

Representation of patterns in sequences by formal grammars has been studied in [Searls 1990]. He also considers which type of grammar in the Chomsky hierarchy is necessary to describe various patterns. Grammar representation is very powerful, and includes the regular expressions, but expressing patterns using a grammar notation directly can be awkward.

2 The pattern language PALM

PALM is a powerful and flexible pattern language designed for representing knowledge about motifs and patters in molecular biological sequences. There are some similarities between PALM and other languages such as Prosite, SCRUTINEER, QUEST etc. and users familiar with those languages should have little problem coming to grips with PALM.

2.1 Goals

PALM has been designed with several goals in mind:

Expressive power: A large range of pattern should be expressible, both exact and approximate, and both the types and the numbers of allowed errors should be controllable. The valid error types are insertion, deletion and substitution. Ultimately, any pattern should be expressible using the pattern language. However, this goal is not immediately possible to achieve.

Declarative: The language should be *descriptive* rather than *prescriptive*, i.e. it is sufficient to describe what the pattern looks like, and unnecessary to describe how to find it.

Clear, compact: Each language construct should be powerful, short, clear and easy to understand and remember.

Consistent: The language constructs should be defined in such a manner that similar constructs are expressed in similar ways.

Modular: New patterns can be composed by re-using existing patterns.

Flexible: The language should cover both nucleotide- and protein sequence matching.

Link to formal language theory: Language constructs covering some of the standard languages in the Chomsky hierarchy should be identifiable. Having this link explicitly stated allows us to benefit from both theoretical and algorithmic results in formal language theory.

Generalisation of existing language: PALM generalises the PROSITE pattern in order to be able to use the large set of patterns in the PROSITE database.

Some of these design goals are possibly conflicting: To have high expressiveness it would be good to have many basic constructs, which may not give a compact language. Also, expressiveness can conflict with being able to implement an efficient search engine, since a parser capable of handling a pattern generating a language at a high level in the Chomsky hierarchy, can not be implemented easily. Approximate matching for such languages is also not easily implemented.

2.2 Preliminary definitions

We first introduce some notation and definitions, mostly following [Harrison 1978].

A *string* $u = u_1 u_2 \dots u_n$ is a finite length sequence of symbols u_i from a finite alphabet S. The length of a string u, denoted $|u|$, is the number of symbols in u, i.e. $|u_1 u_2 \dots u_n| = n$. The empty string, having length 0, is denoted ϵ. The set of all strings of length ≥ 0 having symbols from S is denoted S^*.

Let $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$ be strings in S^*. Then the *concatenation* of u and v, is defined as $uv = u_1 \dots u_m v_1 \dots v_n$. Let L_1 and L_2 be sets of strings. Then the concatenation of L_1 and L_2, denoted $L_1 L_2$, is defined as the set of all strings $u_1 u_2$ where u_1 is from

L_1 and u_2 is from L_2. Similarly, for a string u and a set of strings L, $u^n = uu\ldots u$, and $L^n = LL\ldots L$ (copied n times) for $n \geq 0$. The *Kleene closure*, L^*, denotes the union of all L^n for all $n \geq 0$. Note also that $L^0 = \{\epsilon\}$ for any set of strings L.

To compare strings u and v for similarity, *distance functions* or *metrics* $\Delta(u, v)$ are defined on S^*. The *Hamming distance* $\Delta_H(u, v)$ between strings u and v with $|u| = |v|$ is the number of positions in which the strings differ. The *string edit distence* or *Levenshtein distance* $\Delta_L(u, v)$ is the minimum number of inserts, deletes or substititions necessary to transform the one into the other [Sankoff and Kruskal 1983], [Waterman 1989]. Thus, the Hamming distance is only defined for pairs of strings of the same length, while the Levenshtein distance is defined for any two strings. Also, Hamming distance is the special case of Levenshtein distance where only substitutions are allowed.

It is often convenient to have better control of allowable errors than provided by the Hamming distance and the Levenshtein distance measures. We will represent the valid error types substitution, deletion and insertion by the codes s, d and i respectively. Let E be a subset of $\{s, d, i\}$, and N a set of non-negative integers. The pair (E, N) is called the *error allowance*, E is the *error type* and N is the *error count*.

Define $\Theta(E, N, u)$ as the set of strings that can be produced from u using in total n operations from E, and $n \in N$. Thus, $v \in \Theta(E, N, u)$ if and only if it is possible to transform u into v using only operations from E, and in total n operations, where $n \in N$. Similarly, if L is a set if strings, $\Theta(E, N, L) = \bigcup_{u \in L} \Theta(E, N, u)$, i.e. it is the set of strings obtainable from any $u \in L$ using n operations from E, and $n \in N$.

For strings u and v, note that $\Delta_H(u, v) = n$ is equivalent to $v \in \Theta(\{s\}, \{n\}, u)$ and $\Delta_L(u, v) = n$ is equivalent to $v \in \Theta(\{s, d, i\}, \{n\}, u)$. Also $\Delta_L(u, v) \leq n$, is equivalent to $v \in \Theta(\{s, d, i\}, [0, n], u)$. It is possible to express much more specific constraints using other values for E and N.

The PALM pattern language is intended for describing sets of sequences with a common characteristic, the *pattern*. Let p be a PALM pattern (to be precisely defined below). The pattern p describes a set of strings $L(p)$ in S^*, and p *matches* a string $u \in S^*$ if and only if $u \in L(p)$. The *length-set* of a pattern p, denoted $||p||$, is the set of all lengths of strings in $L(p)$.

We will mostly use the term string, and let the term *sequence* refer more specifically to a protein or nucleotide sequence.

2.3 PALM pattern notation

We now describe and define each of the PALM language constructs. Each construct is defined separately, with examples of its use. All the 'building blocks' defined are valid PALM patterns themselves. Some of the definitions involve combining (sub)patterns into larger patters. Generally any valid patterns can be combined, and any such combination is again a valid pattern.

Let us in the following define p and p_i as denoting any valid PALM pattern, and u and u_i as the corresponding matching strings.

The notation in PALM extends the notation used in the Prosite motif database [Bairoch 1991], so the Prosite database can be used directly, with one exception, described below.

Alphabets Any alphabet can be used with PALM. However, as the language is specifically aimed at matching biological sequences, we only define the alphabets for nucleotide- and protein sequences.

The 20 different amino acids are represented by a one letter code, according to the IUPAC standard, by the alphabet $S_p = \{A, R, N, D, C, Q, E, F, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$. For nucleotide sequences the alphabets are $S_{DNA} = \{A, T, C, G\}$ and $S_{RNA} = \{A, U, C, G\}$.

Sets of integers Sets of integers can be constructed from single integers or from intervals, and unions are represented using the list-like notation $[i_1, \ldots, i_n]$, where each i_k is either a single integer or interval. Intervals are denoted as follows: $a..b$ are all integers i with $a \leq i \leq b$, $a..$ are all integers $\geq a$ and $..a$ are all integers $\leq a$.

Examples: `[1, 3..5, 9..]` denotes the set of all integers ≥ 1, except 2, 6, 7 and 8; `[..5, 9..]` denotes all integers ≤ 5 and ≥ 9.

Units A *unit* matches *one* character (e.g. amino acid) in a string. The valid units are:

- '**x**' matches any single symbol in the alphabet,
- any single symbol a in the alphabet matches itself,
- '**[as]**' matches any character in the list **as**.
- '**{as}**' matches any character *not* in the list **as**.

Thus, $L(\mathbf{x}) = S$, $L(a) = a$ for any $a \in S$, $L([\mathbf{as}]) = \{\mathbf{as}\}$, and $L(\{\mathbf{as}\}) = S - \{\mathbf{as}\}$, where **as** represent any list of symbols from the alphabet S.

The units `[as]` and `{as}` are also called *character classes*.

Examples: `[LIV]` matches L, I or V; `{LIV}` matches any amino acid but L, I and V.

Chain A *chain* matches a sequence of consecutive (sub)patterns. Each subpattern in a chain is separated from its neighbour by a '-' to improve readability, e.g. `A-L-V`. The pattern 'p_1-p_2' matches u_1u_2, and thus $L(p_1\text{-}p_2) = L(p_1)L(p_2)$.

Examples: `A-L-V` matches the string ALV; `[LIV]-A-{KR}` matches any sequence of 3 amino acids $a_1a_2a_3$ where a_1 is L or I or V, a_2 is A and a_3 is any amino acid except K and R.

Union A *union* matches at least one of several given patterns, and has a similar function for longer patterns as the unit [...] for single symbols. The union pattern '[p_1 ; p_2 ; ... ; p_n]' matches any string matching either of the patterns $p_1, \ldots, p_n$. Thus $L([\ p_1\ ;\ \ldots\ ;\ p_n]) = L(p_1) \cup \ldots \cup L(p_n)$.

Example: `[A-L-V ; [KR]-I-[SE]]` matches the strings ALV, KIS, RIS, KIE and RIE.

Named patterns A pattern p can be given a name M using the "declaration" 'M := p', and later be used as a subpattern by simply referring to its name. Pattern names can be defined locally, or can refer to a stored pattern with unique identifier M in a database of patterns, e.g. Prosite, using the notation `prosite(`M`)`. Any pattern name must be distinct from the alphabet symbols.

If a pattern has several definitions, i.e. is a union, the same name can be assigned to each alternative, e.g. `mot := A-L-V, mot := [KR]-I-[SE]`, which is equivalent to the previous union-example. However, using the union-construct is more terse, and preferable.

Example: `alfa := L-N-T, beta := [KTY]-L-G, gamma := alfa-x-x-beta-x-K`.

A named pattern is allowed to refer to itself, thus giving a recursive definition of a pattern. This option potentially creates patterns matching strings of infinite length, and should be used with care.

Example: `any := [x ; x-any]` matches any string of length ≥ 1.

Repeat A *repeat* is a pattern matching a chain of the same (sub)pattern. Let p be a pattern, and N be a set of non-negative integers. Sets can be constructed as described above.

The repeat pattern '$p(N)$' is equivalent to the union of all the patterns p-p-...-p, repeated n times, where n is any integer in N. Thus $L(p(N)) = \bigcup_{n \in N} L^n(p)$.

A single integer or interval does not need to be enclosed by brackets, e.g `x(5)` and `x(2..3)`. If p is a complex pattern, parentheses must be used to indicate the range of the repeat operation, e.g. `(A-x-G)(6..8)`, otherwise only the closest unit would be repeated. An alternative notation for `p(m..n)` is `p(m,n)`, for compatibility with Prosite.

A repeat where the set N is given can always be written explicitly using the chain and union constructs, and the repeat construction is only added for notational convenience. However, $x(0)$ is the conventional notation for the empty string ϵ.

Examples: `x(2)-(A-x(2))(1..2) = [x-x-A-x-x; x-x-A-x-x-A-x-x]`, and `A(0..2) = [x(0); A; A-A]`.

The number of repeats N in $p(N)$ can also be a *variable*, in which case $p(N)$ matches any string in any set of the form $L^n(p)$ for some $n \geq 0$. For one specific instance of n, N is set to n. This construct can be used to count, and to express constraints stating relationships between the number of "things".

Example: If `N` is a variable, `A(N)-C(N)` matches all strings A^nC^n where $n \geq 0$.

Universal The pattern '*' matches any string of length ≥ 0.

Intersection An *intersection* matches all of several given patterns simultaneously. The intersection pattern '[p_1 & p_2 & ... & p_n]' matches any string matching all the patterns p_i simultaneously. Thus $L([p_1 \ \&\ \ldots\ \&\ p_n]) = L(p_1) \cap \ldots \cap L(p_n)$.

Note that the lenght sets of the patterns p_i must be compatible, i.e. there must be at least one string matching all p_i. Otherwise no simultaneous match is possible.

Examples: `[x-A & T-x]` matches TA; `[x(5) & x(0,4)-A-x(0,4)]` matches any string of length 5 with at least one A; `[x(1,2) & A(3,4)]` matches nothing (the patterns are incompatible), and `[*-p1-* & *-p2-*]` matches any string where both patterns `p1` and `p2` occurs, in any order, and possibly with overlap.

Outlaw An *outlaw* disallows some given patterns from occurring in a match, and has a similar function for longer patterns as the unit {..} for single symbols. Let $m = min(\| p_i \|)$, and $n = max(\| p_i \|)$, i.e. m and n are the shortest and longest possible matching string for any p_i, respectively. The outlaw pattern '{p_1 ; p_2 ; ... ; p_n}' matches any string of length between m

and n, except those matching any of $p_1, \ldots, p_n$, giving $L(\{p_1 \ ; \ \ldots \ ; \ p_n\}) = L(\mathtt{x}(m..n)) - (L(p_1) \cup \ldots \cup L(p_n))$.

Examples: `{A-A ; T-T}` matches any string of length 2, except AA and TT; `{H ; T-A-T}` matches any string of length 1, 2 or 3, except H and TAT.

Catching The string matched by a pattern can be *caught* into a variable, and be made available for subsequent matching or processing. Let p be a pattern, and V be a variable name. By the construct 'p : V' the substring matched by p is copied into the variable V. This feature can be used both for special purpose processing of part a of a string, or for expressing that some parts of a string must be equal.

Example: `x(3):V-x(3):V` matches any string uu where $u \in S^3$.

Parameter attachment A named pattern 'M := p' can have *parameters* attached to its name M. These parameters need not have a value when the pattern is defined, but will get a value when the pattern is used. If parameters to a name are given a value at the time of definition, they will act solely as indices, e.g. `a(1) := A-T, a(2) := S-S`.

Example: If `a(N) := A(N)-T` then `a(1)-a(2)` matches the string ATAAT. If `N` is a variable, `A(N)-T-A(N)` matches the set of all strings of the form A^nTA^n with $n \geq 0$.

Procedural attachment Within a pattern a call can be made to a Boolean procedure in the implementation language. Variables passed to the procedure have to be provided using the catch-mechanism. Calling a procedure F after having processed the pattern p is done using the notation 'p-"F"'. The procedure call "F" syntactically acts as if it were a subpattern, but it matches the empty string ϵ, i.e. consumes no input. The pattern p-"F" matches u if both p matches u and F succeeds with the current variable bindings.

Example: Let `occ(U, S, N)` be a procedure where `U` is a PALM unit, `S` is a string, and `N` is the number of times `U` occurs in `S`. The parameter `N` can either be a variable, in which case it shall be computed, or a set constraining its value. Sets can be constructed as described proviously. The pattern `x(10):L-"occ(A,L,5)"` matches any string of length 10 having 5 A's, while `([AT]-x(8)-[AT]):L-"occ(A,L,..5)"` matches any string of length 10 having at most 5 A's, and starting and ending with A or T.

Mismatching Patterns in bioseqences are often difficult or impossible to describe by patterns which match exactly. In PALM it is possible to allow mismatching, and the types of mismatch as well as the number of mismatches can be specified in a pattern definition.

The valid error types are *substitutions*, *insertions* and *deletions*, referred to by the codes `s`, `i` and `d` respectively, and collectively called *errors*.

Let p be a pattern, E be a subset of {`s`,`d`,`i`} and N a set of non-negative integers. Sets can be constructed as desribed previously, and the subset E is represented as a subsequence of `sdi`, e.g. `di` for delete and insert. The *mismatch* pattern 'p`$mis(`E`,`N`)`' matches the set of string obtainable from any string matching p, using any operations from E, and in total n operations, and n occurs in the set N. Thus, $L(p\mathtt{\$mis}(E,N)) = \Theta(E, N, L(p))$.

Note that parentheses must be used if necessary, as for repeats, to specify the extent of the given error allowance.

Example: `(A-G-T-A)$mis(sdi, 0..2))` matches any string that can be produced from AGTA with 0, 1 or 2 errors of any kind; `(A-G-T-A)$mis(s, 2))` matches any string that can be produced from AGTA by exactly 2 substitutions.

An error allowance can be viewed as a constraint on the allowable mismatch. E.g., `(sdi, 0..2)` corresponds to the equation $x_s + x_d + x_i \leq 2$ where x_s is the number of substitution errors occurring, and so on. Further, `(di, 0..2)` corresponds to the equations $x_s = 0, x_d + x_i \leq 2$.

Reverse complement in DNA When matching DNA-sequences the *base pairing* property is important. For the DNA-alphabet $S_{DNA} = \{A, T, G, C\}$ the base pairing is given by the function $bp(A) = T, bp(T) = A, bp(G) = C, bp(C) = G$. For any string $u = u_1 \ldots u_n$ in S^*_{DNA}, the *reverse complement* is defined as `rc(`u`)` = $bp(u_n)bp(u_{n-1}) \ldots bp(u_1)$.

In PALM the reverse complement operation is usually used in conjunction with a catch-pattern, providing the string through a variable `V`.

Example: `x(5):V-x(7,10)-rc(V)` matches any DNA stem-loop structure with a stem of length 5 and a loop of length between 7 and 10, while the mismatch pattern `x(5):V-x(7,10)-rc(V)$mis(i,0..3)` allows a 'bud' of size at most 3 on one side of the stem.

2.4 How PALM generalises Prosite

The constructs from PALM covered in the Prosite pattern language are the units, chains of units, and re-

peated units for integers and simple intervals. For these the notations are compatible.

In Prosite a pattern is implicitly assumed to describe a match with a *subsequence*. To denote a pattern matching the entire sequence, a Prosite pattern `M` must be extended with variable-length 'fillers' at each end, e.g. `*-M-*`. If either of the fillers are missing, i.e. `M` matches at one end of the string, the symbols '<' and '>' symbols are used to indicate this, i.e. the Prosite pattern `<A-L-F` is equivalent to the PALM pattern `A-L-F-*`, and `<A-L-F>` is equivalent to the PALM pattern `A-L-F`. This notation creates problems when combining smaller patterns into bigger ones, since the fillers are implicit. Consequently we have omitted this notation in PALM, and instead opted for explicit definition of all patterns. This is the only example where Prosite and PALM are not completely compatible.

2.5 Implementation

The PALM language has been implemented as a prototype in Quintus Prolog. The implementation is in many ways similar to the way Definite Clause Grammars (DCG) are implemented in Prolog [Gazdar and Mellish 1989], and the very first steps towards PALM were to use DCGs directly to express patterns. This implementation is not sufficiently efficient, in particular for patterns involving mismatching, and work is underway to improve efficiency.

3 PALM and formal language theory

We have seen that a PALM pattern describes a set of strings, or a *language*. Therefore it is interesting to compare the expressive power of PALM with several languages in classic formal language theory. In this section we will examine the expressive power of PALM by identify constructs capable of expressing languages in the Chomsky hierarchy. Through this subsection we only regard PALM patterns without error allowance. A good introduction to formal language theory is found in [Harrison 1978].

3.1 Regular and context-free languages

A *context free language* is generated by a context-free grammar, which is a grammar $G = (V, \Sigma, P, S)$ where V is the set of all symbols, $\Sigma \subset V$ is the *alphabet* (or the *terminals*), $N = V - \Sigma$ is the set of *variables* (or *nonterminals*), P is a set of *rewrite rules* of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in V^*$, and $S \in N$ is the start symbol.

Context free languages is the second layer in the Chomsky hierarchy. The lowest layer in the Chomsky hierarchy is the *regular languages*, generated by the *regular expressions*, or equivalently by *right-linear grammars*, i.e. grammars having rules $A \rightarrow \alpha$ with $\alpha \in \Sigma^* N$.

Let $P = \{A_i \rightarrow \alpha_i\}$ be the rules of a context free grammar G. Then each variable A_i corresponds to the name and the rule body α_i corresponds to a pattern in the naming-construct 'A_i := α_i' in PALM. Thus, any context-free language, and consequently any regular language, can be expressed by a PALM pattern.

To compare explicitly with regular expression representation of regular languages: The PALM constructs corresponding directly to regular expressions without Kleene closure are the unit, chain and union constructs.

3.2 String Variable Grammars

String Variable grammars (SVG) are introduced in [Searls 1990]. A string variable grammar $G = (V, SV, \Sigma, P, S)$ extends a context free grammar with string variables SV which may bind to any string $u \in \Sigma^*$. Thus a rewrite rule has the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (V \cup SV)^*$. This is very useful for describing patterns in DNA. Searls also has shown that the languages defined by SVG grammars contain the context free languages, and are contained in the indexed languages [Searls 1990].

An SVG rule can also be expressed in PALM using the catch-construct. Any string variable `V` in such a rule corresponds to the construct `*:V`, or `x(N):V`, where in the first case the string length is unbounded, and in the latter case it is bounded by a set `N`. Thus PALM also extends the SVG capabilities by allowing bounds on the lengths of strings matched by string variables.

3.3 Beyond context free languages

The ability in PALM to attach an arbitrary Boolean procedure to a pattern gives the power to recognise any set of strings recognisable by the programming language in which the procedure is written. Thus, calling a Turing-complete programming language enables us to decide membership in any *recursive set*. In particular, it is possible to recognise strings belonging to a context sensitive language.

4 Examples and applications

In this section we apply the PALM formalism to some patterns which have been awkward or impossible to express with other pattern languages. The procedure `occ(U, S, N)` is defined as in subsection 2.3.

Note that a string of length `N` having at least `K` symbols taken from a unit `As` can be expressed as `x(N) & x(0,N-K)-(As-x(0,N-K))(K)`, or equivalently as `x(N):L-"occ(As, L, K..)"`. Similar expressions can be given if at most or exactly `K` occurrences of `As` are allowed. Thus procedural attachment can be used, but is not necessary for this class of pattern. We prefer the second version, though, as it is far easier to read.

Below we assume the following named units are defined: `hydrophobic:=[AFGILMVWY]`, `basic:=[RK]`, `acidic:=[DE]`, and `polar:=[NQST]`.

In the examples cited below the original definitions use the 3-letter standard denotation for amino acids in textual descriptions. In the PALM patterns, however, we always use the one-letter UIPAC notation standard. For convenience we have added the one-letter translation to the original pattern definitions.

4.1 Verbally defined Prosite patterns

The Prosite database contains 4 'rule' entries, i.e. patterns formulated in English since they cannot be expressed in the Prosite pattern notation. This subsection defines all such rules in PALM. The relevant parts of the database entries are reproduced, together with the corresponding PALM pattern.

GLYCOSAMINOGLYCAN rule This Prosite rule is described as:

```
ID   GLYCOSAMINOGLYCAN; RULE.
AC   PS00002;
DE   Glycosaminoglycan attachment site.
PA   S-G-x-G.
RU   Additional rules:
RU   There must be at least two acidic amino acids
RU   acids (Glu/D or Asp/E) from -2 to -4 relative
RU   to the serine.
```

In PALM this is described as:

```
*-x(3):L-x-S-G-x-G-*-"occ([DE], L, 2..)".
```

SULFATATION rule The Prosite rule is described as:

```
ID   SULFATATION; RULE.
AC   PS00003;
DE   Tyrosine/Y sulfatation site.
RU   (1) Glu/D or Asp/E within two residues of the
RU       tyrosine (typically at -1).
RU   (2) At least three acidic residues from -5
RU       to +5.
RU   (3) No more than 1 basic residue and 3 hydro-
RU       phobic from -5 to +5.
RU   (4) At least one Pro/P or Gly/G from -7 to -2
RU       and from +1 to +7 or at least two or three
RU       Asp/D, Ser/S or Asn/N from -7 to +7.
RU   (5) Absence of disulfide-bonded cysteine
RU       residues from -7 to +7.
RU   (6) Absence of N-linked glycans near the
RU       tyrosine.
```

In this pattern the rule part (6) contains the imprecise term 'near the tyrosine'. In the PALM definition we choose to quantify this as between 5 and 10 residues from the tyrosine in either direction. However, this choice is only done for the purpose of this example. An N-linked glycane requires presence of the pattern `N-x-[ST]`.

The Prosite rule translates into PALM as:

```
sulf      := [ x(13)-Y-x(13) & c1 & c2 & c3 & c4 ],

poss_sulf := *-sulf-*,

sulfatation := *-[ sulf & c5 & c6a & c6b ]-*,

c1  := x(11)-x(5):L1-x(11)-"occ([DE], L1, 1..)",

c2  := x(8)-x(11):L2-x(8)-"occ(acidic, L2, 3..)",

c3  := x(8)-x(11):L3-x(8)-
       "occ(basic, L3, 0..1),
        occ(hydrophobic, L3, 0..3)",

c4a := x(6)-x(6):L4-x(2)-x(7):L5-x(6)-
       "occ([PG], L4, 1..), occ([PG], L5, 1..)",

c4b := x(6)-x(15):L6-x(6)-"occ([DSN], L6, 2..)",

c4  := [ c4a ; c4b ],

c5  := x(6)-{C}(15)-x(6),

gly := N-x-[ST],

c6a := { x(0..5)-gly-x(5..10)-Y-x(13) },

c6b := { x(13)-Y-x(5..10)-gly-x(0..5) }.
```

The pattern `sulf` covers the requirements (1)-(4), and `poss_sulf` thus gives a possible sulfatation site, while

the pattern **sulfatation** also ensures (5) by excluding *any* cysteine from occurring within -7 to +7, and (6) by disallowing *any* occurrence of the pattern **gly** := **N-x-[ST]** "near" the tyrosine (see above).

PROKAR-LIPOPROTEIN rule This Prosite rule is described as:

```
ID   PROKAR_LIPOPROTEIN; RULE.
AC   PS00013;
DE   Prokaryotic membrane lipoprotein lipid attach-
DE   ment site.
PA   {DERK}(6)-[LIVMFSTAG](2)-[IVMSTAGQ]-[AGS]-C.
RU   Additional rules:
RU   (1) The cysteine must be between positions 15
RU       and 35 of the sequence in consideration.
RU   (2) There must be at least one charged residue
RU       (Lys/K or Arg/R) in the first seven
RU       residues of the sequence.
```

In PALM this is described as:

```
[ x(4..24)-{DERK}(6)-[LIVMFSTAG](2)-[IVMSTAGQ]-
  [AGS]-C-*  &  x(7):L-*-"occ([KR], L, 1..)" ].
```

NUCLEAR rule This Prosite pattern is described as:

```
ID   NUCLEAR; RULE.
AC   PS00015;
DE   Bipartite nuclear targeting sequence.
RU   (1) Two adjacent basic amino acids (Arg/R or
RU       Lys/K).
RU   (2) A spacer region of any 10 residues.
RU   (3) At least three basic residues (Arg/R or
RU       Lys/K) in the five positions after the
RU       spacer region.
```

In PALM this is described as:

```
*-[RK](2)-x(10)-x(5):L-*-"occ([RK], L, 3..)"
```

4.2 A polymerase pattern

This pattern is described in [Argos 1988]:

1. positions 8 to 10 must be occupied by Asp/D-Thr/T-Asp/D or Asp/D-Asp/D,
2. position 7 can be Gly/G, Met/M, Cys/C, Val/V or Leu/L,
3. position 6 can be Tyr/V, Ala/A, Phe/F, Ser/S, Asn/N, Cys/C, Gly/G, Ile/I, or Met/M,
4. at least 2 of the residues in positions 1 to 5 and in positions 11 to 15 must be hydrophobic,
5. If only two residues in positions 1 to 5 are hydrophobic then there must also be a Ser or a Gly. If only two residues in positions 11 to 15 are hydrophobic then there must also be a Ser.
6. positon 4 cannot contain Lys, Arg, Asp, Glu, Gln or Asn
7. positions 12 and 13 must be hydrophobic.

In this pattern the residues Ala/A, Val/V, Leu/L, Ile/I, Cys/C, Met/M, Phe/F, Tyr/Y, His/H, Trp/W, Pro/P are considered hydrophobic.

Note that the residues considered as hydrophobic in this pattern differs from the previously defined set **hydrophobic**. Therefore the new character class **hydrophob** has been defined within the pattern. This translates into PALM as:

```
[x(3)-{KRDEQN}-x  &  c1]-
[VAFSNCGIM]-[GMCVL]-D-T(0,1)-D-
[x-hydrophob(2)-x(2)  &  c2],

hydrophob := [AVLICMFYHWP],

c1  := [c1a ; c1b],

c2  := [c2a ; c2b],

c1a := x(5):L1-"occ(hydrophob, L1, 3..)",

c1b := x(5):L2-"occ(hydrophob, L2, 2),
                occ([SG], L2, 1..)",

c1a := x(5):L3-"occ(hydrophob, L3, 3..)",

c1b := x(5):L4-"occ(hydrophob, L4, 2),
                occ([S], L4, 1..)".
```

Alternatively, using an **if a then b else c** construct (**a -> b; c.** in Prolog) the constraints **c1** and **c2** can be expressed as:

```
c1 := x(5):L1-"occ(hydrophob, L1, N), N >= 2,
               if N=2 then occ([SG], L1, 1..)",

c2 := x(5):L2-"occ(hydrophob, L2, N), N >= 2,
               if N=2 then occ([S], L2, 1..)".
```

5 Conclusion

In this paper we have defined a new formalism for describing patterns in molecular biological sequences. The pattern language, called PALM, has strong expressive power, and is capable of describing any context free language, and any language generated by a

string variable grammar. By allowing procedural attachment PALM can also recognise any context sensitive language.

PALM is intended for representing knowledge about similarities and patterns in sequences, and we have put high emphasis on designing a clear, consistent, terse and modular notation. The notation in PALM generalises the notation used in Prosite, a database of identified patterns in protein sequences, so that those patterns can be readily used. Patterns from Prosite which are only described in English in the database have been formally defined in PALM.

PALM provides approximate pattern matching by defining the degree to which mismatching may occur. The standard model allowing substitutions, insertions and deletions in the matched string is used, and it is possible to constrain both the types and the number of mismatches allowed for a pattern.

The pattern language is implemented as a prototype in Quintus Prolog. Work is underway to improve the speed of searching with PALM, and also to gaine experience with the expressive power and practical use of the language.

Acknowledgements

CH would like to thank Rein Aasland for enthusiasthic introduction into the area of biocomputing, and thank Peter Sibbald for making two very useful stays at EMBL possible.

References

Abarbanel R.M., Wieneke P.R., Mansfield E., Jaffe D.A., Brutlag D.L. (1984): "Rapid searches for complex patterns in biological molecules.", *Nucleic Acids Res.* 12:263-280.

Argos, P (1988): "A sequence motif in many polymerases.", *Nucl. Acids Res.* 16:9909-9916.

Bairoch, A (1991): "PROSITE: a dictionary of sites and patterns in proteins." *Nucl. Acids Res.* 19:2241-2245.

Bratko, I. (1990): *Prolog programming for artificial intelligence,* Second Edition, Addison-Wesley.

Gazdar, G. and Mellish C. (1989): *Natural language processing in Prolog,* Addison-Wesley.

Gribskov M., McLachlan M., Eisenberg D. (1987): "Profile analysis: detection of distantly related proteins.", *Proc. Natl. Acad. Sci. U.S.A.*84:4355-4358.

Harrison, M.A (1978): *Introduction to formal language theory,* Addison-Wesley.

Mehldau, G. (1991): "A pattern matching system for biosequences", Ph.D. diss., Dept. of Computer Science, University of Arizona.

Sankoff, D., Kruskal, J.B. (1983): *Time warps, string edits and macromolecules: the theory and practice of sequence comparison,* Addison-Wesley, Pub. Co. Reading, Massachusetts.

Saurin W., Marliere P. (1987): "Matching relational patterns in nucleic acid sequences.", *Comput. Appl. Biosci.*3:115-120.

Searls, D.B. (1990): "The computational linguistics of biological sequences", Report CAIT-KSA-9010, UNISYS Center for Advanced Information Technology.

Sibbald, P.R. and Argos, P. (1990): "Scrutineer: a computer program that flexibly seeks and describes motifs and profiles in protein databases", *Comput. Appl. Biosci.* 6(3):279-288.

Smith, R. and Smith, T. (1990): "Automatic Generation of Primary Sequence Patterns from Sets of Related Protein Sequences", *Proc. Natl. Acad. Sci. U.S.A.*87:118-122.

Staden, R. (1988): "Methods to define and locate patterns of motifs in sequences", *Comput. Appl. Biosci.* 4(1):53-60.

Taylor, W.R. (1989): "A template based method of pattern matching in protein sequences", *Prog. Biophys. Molec. Biol.* 54:159-252.

Taylor, W.R. (Ed.) (1992): *Patterns in protein sequence and structure,* Springer-Verlag, Heidelberg.

Waterman, M.S. (Ed.) (1989): *Mathematical methods for DNA sequences,* CRC Press, Boca Raton, Florida.