Palingol: A Declarative Programming Language To Describe Nucleic Acids' Secondary Structures And To Scan Sequence Databases.

Bernard Billoud*, Milutin Kontic and Alain Viari

Atelier de Bio-Informatique URA CNRS 448 - Institut Curie, 26, rue d'Ulm 75005 Paris. France.

(*) to whom correspondence should be addressed.

ABSTRACT

At the DNA/RNA level, biological signals are defined by a combination of spatial structures and sequence motifs. Until now, few attempts had been made in writing general purpose search programs that take into account both sequence and structure criteria. Indeed, the most successful structure scanning programs are usually dedicated to particular structures and are written using general purpose programming languages through a complex and time consuming process where the biological problem of defining the structure and the computer engineering problem of looking for it are intimately intertwined. In this paper, we describe a general representation of structures, suitable for database scanning, together with a programming language, Palingol, designed to manipulate it. Palingol has specific data types, corresponding to structural elements - basically helices - that can be arranged in any way to form a complex structure. As a consequence of the declarative approach used in Palingol, the user should only focus on "what to search for" while the language engine takes care of "how to look for it". Therefore, it becomes simpler to write a scanning program and the structural constraints that define the required structure are more clearly identified.

INTRODUCTION

In the course of a sequence analysis project, much attention is paid to searching for functional regions of DNA or RNA molecules. Up to now, most of the known biological signals were defined at the primary structure level by one or more sequence patterns. Numerous algorithms and software packages were thus designed for identifying such signals in sequence data bases, the most sophisticated and complete one probably being the ANREP system (1) and the most well known, the FindPattern utility included in the GCG Package (2). For several biological processes, however, true signals are actually defined by a combination of spatial structure and sequence motifs (3-6). This is especially true of RNA molecules for which compensatory base change studies and/or experimental evidences have clearly shown that a secondary or tertiary structure can be a stronger constraint than the primary sequence itself (7-10). There is therefore a growing demand for general purpose search programs that take into account both sequence and structure patterns (11). It should be pointed out that we are concerned here with the specific problem of searching for known 'structure patterns' within a sequence database and neither finding the optimal folding of a sequence (12, 13) nor learning a common fold of a set of sequences (14), which are different problems. The main difficulty of our specific goal is in establishing a general representation of 'structure patterns' suitable for database scanning and, until now, few attempts have been made in this direction. The tree representation initiated by Shapiro (15) is unfortunately limited to 'pure' secondary structures. Therefore, it cannot capture tertiary structural interactions such as non-pairwise interactions or pseudoknots and cannot deal with complex user requirements such as mixtures of structure and sequence patterns elements. Until recently, the most accomplished efforts to devise structure scanning programs were generalizations of pure sequence searching algorithms. Some elements of this approach, presented by Saurin and Marlière (16) were further generalized by Sibbald et al. (17), but the most complete program to date is probably RNAMOT (18, 19). This approach may be called 'descriptive' since the search is based on the association of specific sequence and structure descriptors that act as 'patterns'. The advantage of this approach is that all the sequence and structural constraints are clearly identified independantly from the procedural details of the search itself. As mentioned in Gautheret et al. (18), its main drawback appears when complicated correlations between sequence and/or structure elements must be introduced (or when some global scoring must be performed) such as "if length of helix H3 is greater than 4, then single-strand #2 may have a length of zero". As the authors explain: "The complexity and diversity of these constraints will require much greater flexibility in the coding of pattern descriptors". The consequence of this situation is that, at the present time, the most sophisticated and successful structure scanning programs are very specific and usually written from scratch using a high level - general purpose programming language like C. This is what we call the 'programming' approach in this paper. Examples of these specific programs are tRNAscan (20) for tRNAs, d'Aubenton's program (21) for *E. coli* rho-independent transcription terminators and CITRON (22) for Group I catalytic introns. Writing such a program is often a complex and time-consuming task because it involves two very different skills corresponding to two distinct tasks. The first one is to define "what to search for" and is the true biological question while the second one is to define "how to look for it" and is a computer engineering problem. It is therefore important to set up a system in which these two tasks are clearly separated.

The main idea behind the development of the Palingol language described in this paper is to offer the capabilities of both the descriptive and programming approaches. It should therefore:

-a- be general enough to act as a real programming language;

-b- be specialized in the treatment of secondary structures, so that the specification of common constraints does not require a lot of code;

-c- emphasize the descriptive aspect of programming rather than its procedural details.

With this aim, Palingol is dedicated to RNA structure handling and has some specific data types, corresponding to structural elements that the user can manipulate directly. More importantly, Palingol is a declarative - as opposed to procedural - language. This means that the user needs only to focus on the description of the structure to search for, while the language engine takes care of the searching process itself.

DESCRIPTION OF SECONDARY STRUCTURES

Overview

An overview of the whole process, starting with the graphical description of a secondary structure and leading to the list of all sequences in a sequence data bank that are able to fold to this structure, is given on figure 1. At the beginning of the process, the user should describe the structure as a list of helices and a list of constraints between them. There are actually two kinds of constraints: local constraints, that act on each individual helix specifying its length, the size of the loop, the presence of particular primary sequence patterns etc. and global constraints that act between helices, specifying their relative location or any kind of cross-conditions and correlation between properties of different helices. These latter constraints can involve features that participate in tertiary rather than secondary interactions. More generally, both local and global constraints can actually act on the helices themselves or on any single stranded region anywhere on the sequence. Once all the local and global constraints are identified and written down in natural language, they should then be translated in the Palingol syntax giving rise to a Palingol program. The rest of the analysis proceeds in two main steps: the search for elementary helices and the Palingol interpretation/search.

In the first step, the sequence database is scanned by an external program (which we generically call HelixSearch) which builds, for each sequence, an ordered list of all helices found on it. We shall see later what 'ordered' means. It is important to note that, at this step, the Palingol program is not yet involved, HelixSearch just makes a list of helices without checking any constraint between them. In fact, for efficiency reasons HelixSearch may already perform some simple checking to avoid generating too long a list of helices, but this is not strictly necessary. The only important point is to ensure that HelixSearch produces <u>all</u> helices that could be involved in the structure, it does not matter if it produces more of them since Palingol will take care of checking the proper constraints. We chose to clearly separate the helix search from the constraint checking process for two main reasons: first HelixSearch can be any user supplied program which might involve some thermodynamical model restricting the set of valid helices or any kind of user preferences; second the set of helices produced by HelixSearch can be stored in a file so that modifying the constraints in the Palingol program does not require the helices to be recomputed. Of course the Palingol package comes with a default HelixSearch program called Palamou. In its present version, Palamou allows for non-canonical base pairings but not for bulges; the loop size can be as large as required.

The second step, which constitutes the main subject of this paper, is performed by the Palingol interpreter and engine. The interpreter reads the user's program written in Palingol and builds an evaluation tree for all the constraints. Then the engine runs through the list of helices, trying to find all subsets of helices which match the required constraints. This is done by a Branch-and-Bound procedure which is described below.

Description of Secondary Structures in Palingol

The elementary objects manipulated by Palingol are the helices computed by HelixSearch. Each helix is described within Palingol by three physical elements, respectively called: 'head', 'tail' and 'loop', where 'head' and 'tail' represent the two paired regions and 'loop' represents the region in between (that may itself contain other helices or parts of helices). The start and end positions of each of these three elements on the sequence are respectively referred to as 'start' and 'end'. For instance 'start head' refers to the index on the sequence of the beginning of the head. Together with these positions and the complete sequence, the presence and position of bulges -if allowed by HelixSearch- should also be attached to the helix description. From this knowledge, other helix properties, such as the energy according to a given thermodynamical model, can be further computed within Palingol itself.

We call 'local' the constraints acting on one elementary helix alone. For instance, the following requirement "the length of the head should be less than 5" is a local constraint. As we shall see later, Palingol allows a great variety of constraint specifications about the size and position of the physical elements, the presence of particular symbols or patterns within the helix itself or in single stranded regions positioned relative to it.

A real secondary structure is actually described by the association of several elementary helices. More precisely, it is described by a set of elementary helices (each of them with local constraints) and a set of constraints between them. These latter constraints are referred to as 'global'. For instance, the structure described in figure 2-a' may be represented by two elementary helices, together with the specification that "helix 2 is fully embedded in helix 1".

In order to identify without ambiguity each elementary helix composing a complete structure, we should order them. By convention, in Palingol this ordering is based on the value of 'start head' (this defines a total order on the sets of helices). Note that if two elementary helices do start at exactly the same position then their order is arbitrary and does not matter in Palingol.

It should be pointed out that this representation of structures is general and not restricted to 'planar' structures (that is structures than can be drawn on a plane without crossing lines). This is illustrated on figure 2, in the case of a 'planar' structures, a 'non planar' structure (a pseudo-knot) and an 'alternate' structure (which correspond to two different planar structures.)

The global constraints specify the relative locations of the elementary helices. The full embedding of 2 in 1 in figure 2-a' may be described by requiring that "start head of 2 is greater than end head of 1 " <and> "end tail of 2 is smaller than start tail of 1", In the same manner, the crossing of 2 and 1 in figure 2-b may be described by requiring that "start head of 2 is greater than end head of 1 " <and> "start tail of 2 is greater than end tail of 1".

Global constraints may also be used to specify ranges of distances between the various elementary helices. For instance in figure 2-b we may require that the distance between "start head of 2" and "end head of 1" lies between 2 and 5 bases. Finally, global constraints may be used to specify a great variety of dependencies between the elements of a structure in conditional statements like "if there is at least 2Gs in head of 1 then there must be at most 3As in tail of 2" or "the last three bases of head of 1 must be complementary to three bases anywhere in loop of 2".

In summary, the complete description of a secondary structure can be reduced to a sorted list of elementary helices together with several local and global constraints. As shown in the previous examples, these constraints may be computationally very sophisticated yet simple to describe in natural language.

THE PALINGOL LANGUAGE

General Principles

From the user's point of view, a constraint (or logic) programming language (like Prolog) mostly differs from a procedural language (like Fortran or C) in the fact that with constraint programming, the user should only specify what he or she wants to do, whereas he or she should also specify how to do it in a procedural language. In order to work properly, a logic programming language must have its own general purpose search procedure built into the language, called the language "engine". Palingol is a constraint programming language whose data types and search engine have been particularly adapted for secondary structures. The engine is based on a classical Branch and Bound mechanism whose main outlines will be described next.

The Branch-and-Bound

Starting with the list of all helices provided by the HelixSearch program, Palingol's engine first tries to build a list of candidates by evaluating local constraints on each individual helix. Since this list is ordered, this is performed in the following way: (let us suppose, as an example, that we are looking for a structure with three helices) Palingol first looks through the list for a candidate as the first helix, then for the second helix candidate, starting in the list after the first, then for the third, starting after the second etc. Once the third helix candidate has been found, the sub-list of three candidates is then checked against the global constraints. Then the process continues with another third candidate starting just after the previous one. This goes on until the list is exhausted for possible third candidates and the process recurses to the next second candidate and, when the list is exhausted for possible second candidates, to the next first candidates. At this step, we have done an exhaustive exploration of the search space. But this exploration can be efficiently bounded by a simple consideration. Keeping in mind that we are looking for local structures it appears efficient to stop the search for the n^{th} candidate as soon as it is too far away from the $(n-1)^{th}$. In a tRNA for instance, if one assumes a maximum intron size of 120 nucleotides, then looking at a potential TΨC arm candidate located 150 bases downstream from the current anticodon arm is totally useless. These additional bounding constraints are called "span constraints" in Palingol. Although, strictly speaking, they are not required in a Palingol program, they are very important, since they can speed up the search process by several orders of magnitude. Thus we shall describe them more precisely later.

Language Syntax

We will not present all technical details of the language syntax but rather outline its general aspects. The syntax of Palingol is strongly inspired by functional programming languages (like LISP) and is based on parenthesized expressions of the following form:

(operator argument argument ...)

where operator stands for one of the built-in operators of the language and argument is either:

- a constant
- a named variable
- another parenthesized expression (without limitation on the nesting level).

The number of arguments in an expression depends upon the operator. The 'value of an expression' is the result of the operator’s action on the arguments and what we call the 'type of an expression', is actually the type of its results. In addition to the traditionnal types (boolean, numerical, string), Palingol makes use of a new and specific type called 'physical'. It is intended to represent all the basic information pertaining to an elementary helix that has been computed by the "Helix Search" program. The user can access to these data by using three physical constants: *head*, *tail* and *loop*. In addition, the user can gain access to the complete sequence being processed through the physical constant *fullsequence*.

Because of the practical importance of these physical elements, we should emphasize now some of the operators that compute various values by acting on them. Two numerical operators *start* and *end* take one physical argument and return the position of the physical element in the sequence. For instance: *(start head)* is a numerical expression, whose value is the position of the first symbol belonging to the head. A string operator *sequence* takes one

physical argument and returns the string of characters composing the physical element. For instance, *(sequence tail)* is a string expression whose value is the sub-sequence of the tail. Palingol provides a lot of other operators covering a wide range of traditional and more specific operations. Classical boolean and numerical operations (*and*, *or*, *add*, *sub*, *div* etc.) are of course present along with most classic string operations (substring extraction, string searching, etc.). Some biologically specific string operators have been included (complementation, inversion etc.). Finally Palingol also provides still more specific operators like pattern matching with IUPAC codes, or consensus matrix computations. It should be noted that a constraint is always expressed in Palingol as a boolean expression: the constraint is said to be satisfied if the value of the corresponding expression is true. A set of constraints is therefore expressed as logically connected boolean expressions. Since *and* is the most usual connector, it is considered to be implicit when the boolean expressions are simply juxtaposed. Thus writing *(expression1)*
(expression2)
(expression3)
is equivalent to writing: *(and (and (and (expression1) (expression2)) (expression3)))*. In a similar way, the parser of Palingol is smart enough to add some missing operators when the context is unambiguous. As mentioned above, the search engine in Palingol tries to satisfy the constraints which have been specified by the user. These constraints are expressed as boolean expressions. When evaluating an expression composed of several sub-expressions, the engine stops as soon as it is sure that the final result will be true or false regardless of the values of the sub-expressions which have not yet been evaluated. This process of stopping evaluation is called pruning. By default the Palingol engine does pruning on *and* and *or* expressions. Because of this, it is important to specify the order of evaluation of the expressions: this is from left to right at a given parenthesis level and from deeper to upper parenthesis levels. For instance in *(and (and (expression1) (expression2)) (expression3))*, expression1 is evaluated first, then expression2 (if needed), and finally expression3 (if needed). By using the implicit *and* which has been described above, this just means that the expressions are evaluated from top to bottom. Note that this has an important consequence to the optimization of Palingol programs: if several anded expressions can be evaluated in several different orders, then it is more efficient to place the 'strongest' condition first (the strongest condition is the one which is most often false), since the evaluation will stop sooner.

Palingol allows the user to store intermediate results into variables (whose names start with a $ sign). Two operators are provided to set and get variables: *(set $variable value)* and *(get $variable)*. Variables don't need to be declared, their type is automatically determined when set and automatically cast when get. *(set $variable value)* and *(get $variable)* are treated as boolean expressions which always have the value true (they are 'side effect' expressions). This allows these expressions to be inserted anywhere within an implicit *and* structure without stopping the evaluation.

Although this may appear quite contradictory with constraint programming, Palingol also provides control structures, namely the *if..then..else* and the *while* structure. It should be pointed out that they are not used to control the engine process itself, but mostly to simplify the writing of constraints.

Structure of a program

The overall structure of a Palingol program can be decomposed into three principal parts:

- the prologue sections (optional)
- the main section
- the epilogue sections (optional)

The main section is the only required one. It contains the description of the structure and is composed of three parts corresponding to various constraint levels: the *helix* part(s), the *span* part and the *cross* part. This overall layout is given on figure 3-a. The *helix* part(s) describe local constraints for each helix in the structure. The *helix* sections are ordered as indicated above. There must be as many distinct *helix* sections as there are distinct helices in the required structure. Each *helix* section is composed of one boolean expression (usually containing several boolean expressions implicitly anded). Once a sub-list of n candidates (where n is the number *helix* sections) has been found by the engine, this list is submitted to the global constraints described in the *cross* section. Again, this section is composed of one boolean expression (usually containing several boolean expressions implicitly anded). Note that there is no specific 'print result section', this is done within the *cross* section, as a side effect of the *print* operator. Finally, the optional *span* section has been added to speed up the overall searching process. As previously mentioned, it describes the maximum distance allowed between two consecutive helices. More precisely, the *span* section takes the special form given in figure 3-b, where the i^{th} line specifies the maximal distance allowed between the i^{th} and $(i+1)^{th}$ helix. This distance is computed as the algebraic difference between 'start head' of the $(i+1)^{th}$ helix and %keyword of the i^{th} helix, where keyword can be any of *start_head*, *end_head*, *start_tail* or *end_tail*.

The optional prologue and epilogue sections are paired, each prologue section having its epilogue counterpart, namely the *start*/*end* sections and the *before*/*after* section. The *start* (prologue) and *end* (epilogue) sections each contain one boolean expression (the value is ignored) which is evaluated once at program startup and ending respectively. This is useful for setting up variables (like counters) or for printing some general information. The *before* (prologue) and *after* (epilogue) sections each contain one boolean expression which is evaluated once for each new sequence in the sequence database (i.e. for each new list of helices provided by HelixSearch). Only the value of the *before* boolean expression is meaningful; if it is false then the main section is not evaluated. This may be used to skip over specific entries in the data bank, by checking that a specific pattern is present in the sequence, for instance.

Like most the compilers and interpreters, the lexical and grammatical analyzer of Palingol, requires pre-processing. This is done by a small program called Galopin whose main responsibility is to rewrite the user supplied Palingol program in a suitable form for the analyzer, mostly filtering purely syntactic errors. This preprocessor also provides additional features like file inclusion directives and macros. This allows the user to build libraries of program pieces that can be reused.

RESULTS

We present the use of Palingol with two example programs. Each of them searches for a known functional secondary structure defined by well-characterized secondary structure elements and several sequence patterns. The first one concerns the Iron Responsive Elements (IREs) and the second, transfer RNAs. Efficient procedures have been published for identifying them in databases: IREsearch (11) for IREs and tRNAscan (20) for tRNAs. It is essential to note that our purpose here is not to challenge these programs, but rather to try to imitate them as much as possible in order to demonstrate that the constraints expressed by the biologist and written by the programmer could be expressed just as well in the Palingol formalism. IREs were chosen as a very simple example to illustrate Palingol's formalism, whereas tRNAscan was chosen as a good example of 'real-sized' application featuring complicated constraints. For this latter case particularly, it should be kept in mind that the algorithm was designed by an experienced C programmer, accustomed to procedural descriptions. As matter of illustration and comparison with these previous works, we tried to follow, as much as possible, the original 'programming logic' as it was published even if in several cases, slight modifications would lead to simpler, more efficient or 'Palingol friendlier' programs. Finally, note that since our tests were performed on DNA databases, sequences will contain "T" instead of "U", even if the considered structures are defined on RNAs.

Iron Responsive Elements

IREs are involved in regulation of some mRNA's translation or stabilization by IRP (23). The definition of an IRE (3, 24) is quite simple and is described on figure 4. It is basically made of one helix bearing a bulgy C and the loop sequence is always CAGTGH. This bulgy helix can be depicted as a pair of two nested helices. The usual description of IREs uses this latter definition and the two helices are traditionally named "Top" and "Bottom". In our case, our HelixSearch program (Palamou) does not allow bulges in helices, and we will thus adopt this latter definition. Moreover, instead of generating all possible helices, we instructed Palamou to search for two sets of helices, one for the top helices (minimum length 4, maximum length 5 and loop size strictly equal to 6) and the other for the bottom helices (minimum length 3, no maximum length and loop size equal to 17 or 18). We have now to deal with the Palingol description of the constraints defining the IRE structure. The corresponding program is given in figure 5 and is detailed here. The *start* section is executed once at program startup. It just initializes the base pairing matrix which will be used later to compute the helix scores. Then comes the main section, comprising in this case two *helix* sections. The first *helix* section describes the bottom helix since, by the convention described above, its head comes first. The first constraint requires that it have a loop of 17 or 18 bases and the second constraint requires that its helix score be at least 7. The score is computed by the *scorebp* operator, using the matrix which has been previously initialized with *bpcompile*. The second *helix* section describes the top helix. As above, a loop size of 6 is in this case a sufficient condition to select a potential top helix. The next two lines check the presence of the IRE signal at the correct position. This is done by extracting (using the *sstr* operator) the sequence just after the bulgy C, i.e. 6 nucleotides upstream of the start of loop, and of 12 bases long. This extracted string is stored in the variable $zone for clarity and the IRE pattern defined by Dandekar, "CNNNNNCAGTGH" is searched for in $zone with 0 mismatch allowed. Note that in this particular case the searched zone has the same length as the pattern, nevertheless we have used a general *patsearch* operator. At this point, we have two helices, each satisfying its local constraints. We now have to specify their respective arrangement and potential cross-constraints. As mentioned, the *span* section is used to speedup the engine by limiting the distance between the first and second helices. In this case, the head of the second helix cannot start downstream of the fourth base after the end of head of the first one (a maximum two bases for the bulge and possibly 1 non paired base before the top helix). Finally, the *cross* section checks the global constraints and prints the results. Variables are used here for clarity. The first constraint is always verified since *set* is a pure side effect operator that always returns *true*. It sets the variable *$bul* to the length of the bulge (that is the loop size of the bottom helix minus 16). Similarly, the *$mis* variable is set to 1 if the bases next to the bulge are mismatched and 0 otherwise (i.e. 5 minus the length of the top helix). Using these two variables, the actual *cross* constraints become very simple:

- Starting from the end of head#1 and adding the bulge size, we should come to the start of head#2.
- Starting from the end of tail#2 and adding the possible mismatched pair we should come to the start of tail#1.

Finally, a set of two helices satisfying all of the above constraints is identified as an IRE and the only purpose of the next lines is to print the result. Again, we use variables to make the program easier to read. $pos is just the position of the start of head #1, and $size is the total size of the detected IRE. The printing instructions cause output of the sequence name, the position and the sequence of the IRE, respectively. This ends the *cross* section, and the main program section.

We tested the IRE searching program on four different sets of vertebrate sequences of length greater or equal to 23 nucleotides (minimum size of an IRE), extracted from the EMBL release 43 by using the Hovergen database (25): (i) known IRE of ferritin related sequences, i.e. sequences having "ferritin" as a keyword, (ii) 5' non coding regions, (iii) 3' non coding regions and (iv) introns. The results and scanning times are summarized in table 1, which displays the number of positive sequences meeting increasingly stringent requirements.

(i) *Ferritin related set*: Amongst the 58 sequences in EMBL 43 which contain "ferritin" as a keyword, the primary IRE motif, CNNNNNCAGTGH, was found in 51 sequences. 22 of them are able to form a correct top helix, and could be therefore considered as true IREs (table 2). However only 19 of them meet all the criteria given in the IRE.p program (table 2-a). The three false negatives are displayed on table 2-b. HSAFL12 and MMFERLSUB were rejected because they bear a bulge and a mismatch in the bottom helix, a feature which was not allowed in our description. The third sequence, MMFERLA, is actually too short at its 5' end to form a sufficiently long bottom helix.

(ii) *5' non coding* This set is composed of 17642 sequences totaling 6,971,415 nucleotides. Our IRE program found 34 IRE sequences in this set, including those originally characterized by Dandekar (11), which were not known to be involved in the iron homeostasis system. In particular, eALAS IRE was found correctly (HSALASR).

(iii) *3' non coding*. This set is composed of 20171 sequences totaling 12,010,172 nucleotides.

(iv) *introns* This set is composed of 21288 sequences totaling 9,129,148 nucleotides. No functional IRE are expected to be found in introns. Thus, all of the "IREs" found here should be false positives. As expected, although this set contains the largest number of sequences, it has the smallest number of detected IREs: 16 positive answers (two of them being alternative foldings at the same position). Strikingly, 0,7% of the sequences matching the consensus motif are finally identified as IREs, while this ratio is 1,9% for 5' and 1,4% for 3' non coding regions. Similarly, the ratio of sequences identified as IRE to those with only the TOP helix (37%) is also the smallest of all (others: 47% to 86%). These results reflect the fact that these patterns and TOP helices occur here only by chance and not as part of a functional IRE.

Bacterial tRNAs

tRNAs are more complex structures, comprising 4 stems arranged in a well defined manner, i.e. the well-known cloverleaf structure. tRNAscan, the algorithm devised by Fichant et al. (20) involves 7 sequential steps which are summarized in figure 6. It also features a sophisticated scoring scheme. At each step, examination of a structural element can lead to (i) increasing the score, (ii) continuing the algorithm without increasing the score, or (iii) rejecting the sequence fragment. The final score has to be 5 or above. Our purpose here was not to challenge the Fichant's procedure, but to illustrate Palingol programming logic with this example. To simplify, we considered in a first time, only intronless prokaryotic sequences. Each step was translated into a Palingol constraint. Other parameters (for instance the distance between stems), not fully detailed in the original paper, were derived from the tRNAscan C source code, kindly provided by the authors. For clarity in the resulting Palingol program, we took care to write constraints in an order similar to that of tRNAscan, instead of trying to optimize the computing process for the Palingol engine. For example, to fit Fichant's algorithm, step 7 ("T in 5' from anticodon") was examined only if the score was 4 or more after the five first steps were fulfilled.

We used both Palingol and tRNAscan with the *B.Subtilis* database release 6 (26). Both found 100% of known tRNAs, with no false positives. In this particular case, we could conserve the same 100% retrieving without false positive even after modifying three constraints:

(1) 3/4 invariant (instead of 2/4) bases in TΨC signal.

(2) T-Ψ-C arm can be only 3 base pairs long if it contains only GC pairings

(3) a T residue is necessary in 5' of the anticodon, instead of just increasing the score.

Moreover, examining D arm is not necessary, as removing this constraint did not change results at all. Note that modifying these constraints in the C source code would not be an easy task, whereas it is much simpler in the Palingol description, even if the user did not write the original program. In the particular case of the removal of the D arm constraints, this modification is achieved by simply (i) removing the whole *helix* section depicting the D arm (ii) changing in the *cross* section, the numbering of helices 3 and 4 to 2 and 3 respectively and (iii) removing the corresponding span instruction. From this point of view, Palingol could be also considered as a practical tool for constraint exploration e.g. for adapting general constraints to a specific organism, as in this example.

We undertook a second experiment on a set of prokaryotic sequences in EMBL release 43. The comparison between tRNAscan and the Palingol program (trna.p) is given in table 3. As stated above, our Palingol description was intended to imitate tRNAscan. Indeed, most of the 678 annotated tRNAs are found by both programs and both display few false positive (37 for tRNAscan and 30 for trna.p with 27 common to both). Moreover, closer examination of these "false positive" sequences show that most of them are actually true tRNAs, although not annotated as such in EMBL. There are, however, slight differences between the two results in terms of false negatives: 21 true tRNAs are missed by tRNAscan but found by trna.p, whereas 19 are missed by trna.p and found by tRNAscan. These discrepancies reflect differences between the definitions of helices used in the two programs.

A more complete (and complex) tRNA program, taking introns into account, gave similar results: 665 tRNA sequences were found by tRNAscan and 684 by Palingol where 644 are common to both.

CONCLUSION

To quote Dandekar (27), the purpose of Palingol could be described as having to "search the hairpin in the haystack", i.e. build sophisticated sieves to screen a set of hairpins, looking for those that meet a specific set of criteria. This is done by using a real programming language which allows - as opposed to the descriptive approach - arbitrarily complex constraints to be expressed, involving sequence patterns, 'pure' secondary structure description but also tertiary structure elements (like pseudoknots or other non-nested hairpins) or even 'equilibrium' structures (like bacterial attenuators). Of course, there is still room for improvement in this approach. The language itself still needs to grow, by including new operators. It should however be pointed out that this enrichment should be directed by the usage; that is by programming new structures. Up to now, we successfully applied Palingol in our laboratory in our studies of prokaryotic genomes - particularly on the *E. coli* and the *B. subtilis* chromosome. This involved writing Palingol descriptions to search for: bacterial tRNAs, terminators of the transcription and tRNA synthetases regulatory elements (28). From this point of view, Palingol proved to be a convenient tool for constraint exploration, to investigate quickly which constraints in the description of a structure are really important. This is done mostly by a trial and error approach which consists of modifying the constraints and searching the whole database for the number of true/false positives and negatives. Once a convenient description has been found, it can be 'frozen' and subsequently used as a specific recognition program. We have already included Palingol in the co-operative computer environment, dedicated to the study of the *B. subtilis* genome, which is developed in our laboratory (29). Finally, during the course of this work, we noticed several times that the constraints described in papers were ambiguous and we had either to resort to the original sources or to set up our own definitions. The fact that Palingol requires all the constraints to be clearly specified in a formalized way thus appears to us an important benefit, since it could provide the starting point for the unambiguous transmission of this information in the biological community.

Palingol has been implemented in C using the lex and yacc unix tools. Full documentation, sources and binaries for SUN and Silicon Graphics workstations are available on anonymous ftp at: ftp.radium.jussieu.fr /pub/palabi.

ACKNOWLEDGMENTS

This work was supported by the Ministère de l'Enseignement Supérieur et de la Recherche (MESR). We wish to thank Laurent Duret for his help in selecting NCRs using the Hovergen database, Antoine Danchin for initiating the work with helpful discussions, Kyle Weinandy, Chris Burge and Boris Barbour for proof-reading the manuscript.

REFERENCES

1. Mehldau, G. and Myers, G. (1993) *Comput. Appl. Biosci.*, **9** 299-314
2. Devereux, J., Haeberli, P., Smithies, O. (1984) *Nucleic Acids Res.,* **12**, 385-395
3. Hentze, M.W., Caughman, S.W., Casey, J.L., Koeller, D.M., Rouault, T.A., Harford, J.B. and Klausner, R (1988) *Gene*, **72**, 201-208
4. Grundy, F. and Henkin, T. (1993) *Cell,* **74**, 475-482
5. Spedding, G., Gluick, T. C. and Draper, D. E. (1993) *J. Mol. Biol.,* **229**, 609-622

6. Cotmore, S.F. and Tettersall, P. (1994) *EMBO J.,* **13**, 4145-4152
7. Woese, C.R., Gutell, R., Gupta, R. and Noller, H.F. (1983) *Microbiol. Rev.,* **47**, 621-669
8. Michel, F. and Westhof, E. (1990) *J. Mol. Biol.,* **216**, 585-610
9. Gutell, R.R. (1993) *Curr. Opin. Struct. Biol.,* **3**, 313-322
10. Pleij, C.W.A. (1994) *Curr. Opin. Struct. Biol.,* **4**, 337-344
11. Dandekar, T. and Hentze, M.W. (1991) *EMBO J.,* **10**, 1903-1909
12. Gouy, M. (1987) In Bishop, M.J. and Rawlings, C.J. (eds), *Nucleic acid and protein sequence analysis*. I press, Oxford, pp. 259-284
13. Zuker, M. (1994) In Griffin, A.M. and Griffin, H.G. (eds), *Computer analysis of sequence data*, Pa Totowa, N.J., pp. 267-294
14. Eddy, S.R. and Durbin, R. (1994) *Nucleic Acids Res.,* **22**, 2079-2088
15. Shapiro, B.A. and Zhang, K. (1990) *Comput. Appl. Biosci.,* **6**, 309-318
16. Saurin, W. and Marlière, P. (1987) *Comput. Appl. Biosci.,* **3**, 115-120
17. Sibbald, P.R., Sommerfeldt, H. and Argos, P. (1992) *Comput. Appl. Biosci.,* **8**, 45-48
18. Gautheret, D., Major, F. and Cedergren, R.J. (1990) *Comput. Appl. Biosci.,* **6**, 325-331
19. Laferrière, A., Gautheret, D. and Cedergren, R. (1994) *Comput. Appl. Biosci.,* **10**, 211-212
20. Fichant, G.A. and Burks, C. (1991) *J. Mol. Biol.,* **220**, 659-671
21. d'Aubenton Carafa, Y., Brody, E. and Thermes, C. (1990) *J. Mol. Biol.,* **216**, 835-858
22. Lisacek, F., Diaz, Y. and Michel, F. (1994) *J. Mol. Biol.,* **235**, 1206 - 1217
23. Melefors, O. and Hentze, M.W. (1993) *BioEssays,* **15**, 85-90
24. Leibold, E.A., Laudano, A. and Yu, Y. (1990) *Nucleic Acids Res.,* **18**, 1819-1824
25. Duret, L., Mouchiroud, D. and Gouy, M. (1994) *Nucleic AcidsRes.,* **22**, 2360-2365
26. Moszer, I., Glaser, P. and Danchin, A. (1995) *Microbiology,* **141**, 261-268
27. Dandekar, T. (1995) *Trends Genet.,* **11**, 45-50
28. Grundy, F. and Henkin,T. (1994) *J. Mol. Biol.,* **235**, 798-804
29. Médigue C., Moszer, I., Viari, A. and Danchin, A. (1995) *Gene,* **165**, GC37-GC51