RNAMotif Users' Manual

(Version 3.0.7, April, 2010)

Thomas C. Macke and David A. Case

Department of Molecular Biology
The Scripps Research Institute
La Jolla, CA 92037

The *rnamotif* program [1] searches a database for RNA sequences that match a "motif" describing secondary structure interactions. A match means that the given sequence is capable of adopting the given secondary structure, but is not intended to be predictive. Matches can be ranked by applying scoring rules that may provide finer distinctions than just matching to a profile. The *rnamotif* program is a (significant) extension of earlier programs *rnamot* and *rnabob* [2-4]. The nearest-neighbor energies used in the scoring section are based on refs. [5] and [6]. The code and most of this Users' Manual was written by Tom Macke; Dave Case also contributed to this manual.

The basic literature references are:

[1] T. Macke, D. Ecker, R. Gutell, D. Gautheret, D.A. Case and R. Sampath. RNAMotif -- A new RNA secondary structure definition and discovery algorithm. *Nucl. Acids. Res.* **29,** 4724-4735 (2001).

[2] D. Gautheret, F. Major and R. Cedergren. Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA. *CABIOS* **6,** 325-311 (1990).

[3] A. Laferrière, D. Gautheret, and R. Cedergren. An RNA pattern matching program with enhanced performance and portability. *CABIOS* **10,** 211-212 (1994).

[4] S.R. Eddy. RNABOB: a program to search for RNA secondary structure motifs in sequence databases. Unpublished.

[5] M.J. Serra, D.H. Turner, and S.M. Freier. Predicting thermodynamic properties of RNA. *Meth. Enzymol.* **259,** 243-261 (1995).

[6] D.H. Mathews, J. Sabina, M. Zuker, and D.H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *J. Mol. Biol.* **288,** 911-940 (1999).

We thank Dave Ecker and Ranga Sampath for encouragement, and ISIS Pharmaceuticals for financial support; Robin Gutell and Daniel Gautheret for suggestions; Vickie Tsui for serving as a beta tester, and for many helpful comments and suggestions.

---

1. Short overview

1.1. Introduction

The art and science of drawing connections between sequence and structure tends to be much different for RNA than for proteins. There are several reasons for this. First, the presence of a twenty-letter "alphabet" for protein sequences means that frequency counts or simple pattern matching can often be a useful strategy for identifying secondary structure or searching for similarities among sequences; these strategies are much less useful with the four-letter alphabet of nucleic acids. Second, there are many more proteins than RNA fragments whose structure has been been determined, so that much more is known about sequence-structure relations. Even at the secondary structure level, it is most often the case that RNA hydrogen bond (base-pairing) patterns have to be inferred from analysis of sequence variation (and co-variation) among related members of some family, rather then being directly visible in crystal or NMR structures, which is often the case in proteins.

Nevertheless, interest in identifying structural similarities in RNA from sequence is strong, and there are an increasing number of cases where enough is known to make at least secondary structure identification a plausible goal. The approach taken here involves encoding secondary structure patterns into a "profile" or "motif", and searching through genome databases to find sequences that have the capability of adopting the given secondary structure pattern. These patterns can as simple as arrangements of Watson-Crick paired "stems" (interspersed with single-stranded regions whose interactions are not specified). They can also be more complex, through use of extensions to the canonical Watson-Crick pairing rules, imposition of particular sequence restrictions (such as asking for a "uucg" tetraloop), or through specification of higher-order interactions involving base triples or quadruples.

It is important to remember that sequences matching a particular profile only have the *possibility* of forming the given structure; *rnamotif* is not a structure prediction method. It can fish sequences out of databases that can be fed to "folding" algorithms such as *mfold* [1] or *pknots* [2] to see if the proposed structure is low in energy (based on empirical base-pairing energies) relative to alternative folds; this can often be useful to further prune profile matches. The "scoring section" of *rnamotif* provides an interface to describe other models for ranking hits that can be based on empirical knowledge or just intuition. Fundamentally, *rnamotif* is a tool for investigating hypotheses: it provides a precise language for defining secondary structure motifs, a search engine to find matching sequences, and framework for ranking the returned sequences by a variety of empirical or energetic criteria.

The use of profiles to describe non-trivial sequence patterns has been an important component of

---

1. M. Zuker, “Calculating nucleic acid secondary structure,” Curr. Opin. Struct. Biol. **10,** 303-310 (2000).

2. E. Rivas and S. Eddy, “A Dynamic Programming Algorithm for RNA Structure Prediction Including Pseudoknots,” J. Mol. Biol. **285,** 2053-2068 (1999).

3. C. Dodge, R. Schneider, and C. Sander, “The HSSP database of protein structure-sequence alignments and family profiles.,” Nucl. Acids Res. **26,** 313-315 (1998).

4. E. Sonnhammer, S. Eddy, and R. Durbin, “Pfam: multiple sequence alignments and HMM-profiles of protein domains.,” Nucl. Acids Res. **26,** 320-322 (1998).

5. A. Panchenko, A. Marchler-Bauer, and S. Bryant, “Combination of Threading Potentials and Sequence Profiles Improves Fold Recognition,” J. Mol. Biol. **296,** 1319-1332 (2000).

amino acid sequence analysis for some time [3-7]. The analogue for RNA was pioneered by Gautheret *et al.* [8,9], whose *rnamot* program was an inspiration for the development of *rnamotif*. The current code extends Gautheret's original work by providing for profiles that specify pseudoknots, and base triplexes and quadruplexes, by improvements in the pruning and presentation of results, and by the incorporation of "scoring rules" that go beyond the simple "yes/no" results from profile screening. The next two chapters give a tutorial introduction to the program; this is followed by a fairly detailed description of the algorithms used, and a final, reference chapter summarizing in tables the input expected by the code.

1.2. Quick overview on installing and running the program

1.2.1. Installation

(a) Edit the "config.h" file in this directory to suit your local environment. This may require no change on many machines, but you may need to locate the C-compiler, lex or yacc, and set compiler flags.

(b) Now "make" will make the executables, leaving them in the src directory. There is no "install" target, but (after you run the test suite) you can manually move "rnamotif", "rmfmt", "rm2ct" and "rmprune" to someplace in your PATH if you wish.

(c) Set the environment variable EFNDATA to point to the efndata subdirectory under this one. Then "make test" will run some test calculations and report results. (Warning: running the tests may take some time.)

More details are given in the README file in the distribution. Please send comments and questions to Tom Macke <*macke@scripps.edu*>.

1.2.2. rnamotif

rnamotif is run from the command line. It takes two arguments: a descriptor that specifies the secondary structure to look for, and a file of fastn data to search for sequences that can adopt that secondary structure. The output is written to stdout. The command

```
rnamotif -descr test.descr test.fastn
```

---

6. M. Pellegrini, E.M. Marcotte, and T.O. Yeates, "Assigning protein functions by comparative genome analysis: Protein phylogenetic profiles," Proc. Natl. Acad. Sci. USA **96,** 4285-4288 (1999).

7. L. Rychlewski, L. Jaroszewski, and A. Godzik, "Comparison of sequence profiles. Strategies for structural predictions using sequence information," Prot. Sci. **9,** 232-241 (2000).

8. D. Gautheret, F. Major, and R. Cedergren, "Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA.," CABIOS **6,** 325-331 (1990).

9. A. Laferrière, D. Gautheret, and R. Cedergren, "An RNA pattern matching program with enhanced performance and portability," CABIOS **10,** 211-212 (1994).

searches the fastn file 'test.fastn' for sequences that can fold up into the secondary structures specified in the descriptor file *test.descr*. If no fastn file is given, rnamotif uses *stdin*. Note that in the above example, the files have suffixes .descr and .fastn; however, these suffixes are only a convention, as they can have any names.

The full form of the rnamotif command is shown below:

```
rnamotif [ options ] descr [ seq-file ... ]

options:
    -c         Compile only, no search
    -d         Dump internal data structures
    -h         Dump the strucure hierarchy
    -s         Show builtin variables
    -v         Print Version Infomation
    -O<n>       Optimize search with minimum length of n
    -Dvar=expr     Set the value of var to expr
    -Idir          Add include source directory, dir
    -xdfname file-namePreprocessor output file
    -help          Print this message

descr: Use one:
    -descr descr-file May have includes; use cmd-line defs
    -xdescr xdescr-fileMay not have includes; ignore cmd-line defs
```

1.2.3. rmprune

Many descriptors, such as a variable length stem that contains a variable length loop, have several solutions that differ only in that the stem has had bases on one or the other or both ends "unzipped". *rmprune* takes the output of *rnamotif* and removes unzipped solutions, leaving only that solution with the longest possible stem. As an example consider this descriptor:

```
descr
    h5( minlen=4, maxlen=6 ) ss( minlen=4, maxlen=6 ) h3
```

This descriptor specifies a simple hairpin where the stem must be 4-6 base pairs and the loop 4-6 bases. When rnamotif searches the sequence

```
aaaaaaccccttttttt
```

it finds eleven possible hairpins:

```
 1  1 14 aaaa   aacccc   tttt
 2  1 15 aaaaa  acccc   ttttt
 3  1 16 aaaaa  acccct  ttttt
 4  1 16 aaaaaa cccc   tttttt
 5  2 13 aaaa   acccc    tttt
 6  2 14 aaaa   acccct   tttt
 7  2 14 aaaaa  cccc    ttttt
```

```
 8  2 15 aaaaa  ccccc   ttttt
 9  3 12 aaaa   cccc     tttt
10  3 13 aaaa   ccccc    tttt
11  3 14 aaaa   cccctt   tttt
```

Many of these hairpins are merely subsets of the base pairs of the others. *rmprune* removes these "unzipped" subsets, leaving five unique pairings:

```
 1  1 14 aaaa   aacccc   tttt
 2  1 15 aaaaa  acccc   ttttt    contains 5
 4  1 16 aaaaaa cccc   tttttt    contains 3,6,7,8
 8  2 15 aaaaa  ccccc   ttttt    contains 10
11  3 14 aaaa   cccctt   tttt
```

rmprune is run from the command line and takes an optional argument, a file containing the output of an rnamotif search. If this file is not given, it reads from stdin, allowing it to be placed in a pipeline following rnamotif:

```
rnamotif -descr hp.descr test.fastn | rmprune
```

1.2.4. rmfmt

rmfmt is used to format rnamotif output. Two formats are supported. The first is a viewing option that removes the fastn definition line from the rnamotif output. At the same time sequences that correspond to long fields in the descriptor (ie > 20 bases) are condensed, showing only the 1st 3 and last 3 bases surrounding the total length of the string in parens. For example the string 'acgtacgtacgtacgtacgtacgt' (len=24) would be replaced by 'acg...(24)...cgt'. The strings are also aligned so that strings that represent ss, h5, p5, t1, t3, q1 and q3 elements are left justified and those that correspond to h3, p3, t2, q2 and q4 fields are right justified. The name of each solution is also simplified, using only the LOCUS name from the fastn definition line. Finally, if at all possible the rnamotif solutions are sorted into descending order based on the value of the score column (#2) placing the best (or worst) solutions at the top. This format option is specified by the -l option.

The second format option is to convert the rnamotif output into a fastn format alignment. Justification is as in -l, but the fill characater is the dash (-). Fields are separated by a vertical bar (|). Aligned output is not sorted and the fastn definition lines are retained. This format option is specified by the -a option. The full rmfmt command is show below:

```
rmfmt [ -a ] [ -l ] [ -smax N ] [ -td dir ] [ rm-output-file ]
```

The *-smax* option sets an upper bound on the size of the file of the output to sort, as this can be impractical for very large output. The default sort limit is 30MB. The *-td* option allows the user to specify the temp directory that rmfmt places its work files in. These files are needed because the alignment can not be computed until all solutions have been seen. The default temp directory is system dependent and is either /tmp or /var/tmp.

The *-a* flag creates a "fasta" format alignment, and the output is not sorted. The flags *-a* and *-l* can be used together in which case the file is formatted with dashes (unsorted) and the names of the sequences are reduced to the last name in the bar separated name section of the fastn definition line.

1.2.5. rm2ct

This program converts the rnamotif output to a ct-format structure file, which can used by various programs to create a graphical representation of the sequence and its 2d structure. The rm2ct command takes a single optional argument, the name of the file containing the rnamotif output to convert. The output is written to stdout. If no argument is given, rm2ct reads from stdin, allowing it to stand in a pipe:

```
rnamotif -descr hp.descr test.fastn | rm2ct
```

The ct-format can only represent duplexes, for this reason rm2ct exits with an error message if its input contains triples or quads. However, all duplexes, including parallel duplexes and pseudoknots are converted, although it is likely that many ct-format plotting programs can not handle such structures.

2. Constructing descriptors

Perhaps the simplest way to see how *rnamotif* works is to examine some simple applications. In this chapter, we present several increasingly complex descriptors, showing how they are constructed, and what sort of output one should expect. The following chapter deals with the optional scoring section, where matches to motifs can be ranked according to a variety of criteria.

2.1. Hairpins.

2.1.1. The UUCG-loop.

Our first example will be to use `rnamotif` to find sequences that contain at least one subsequence that could fold up into a UUCG-loop, a hairpin in which the single stranded loop that connects the two strands of the helix has the sequence "UUCG". These loops occur frequently in RNA and have unusual thermal stability due to hydrogen bonding between the X of the loop and the phosphate backbone. Here is an `rnamotif` descriptor that finds UUCG-loops that have a helical stem of 4 to 6 base pairs.

```
descr
    h5( minlen=4, maxlen=6 ) ss( seq="^UUCG$" ) h3
```

While this descriptor is not the simplest possible `rnamotif` descriptor, it is typical of real descriptors that find some simple motif — in this case, a hairpin with both length and sequence constraints.

Every `rnamotif` descriptor consists of one to four sections in the following order: parms, descr, sites and score. Only the descr section is required and is introduced by the reserved word `descr`. The motif we wish to find is described — thus the term descriptor — by the second line of the example, the one beginning `h5` and ending `h3`. In order to understand exactly what this descriptor means, how it works, and why `rnamotif` was designed this way, it will be useful to discuss how one might go about specifying RNA secondary structures in general as well as how it applies to this particular UUCG-loop.

Our goal is to describe the set of sequences that can fold up into UUCG-loops. Obviously this description must include both the sequence "UUCG" and the requirement that the two sequences that immediately precede and follow it be able to base pair into a standard Watson/Crick (W/C) helix. While the first requirement — the sequence must contain the string "UUCG" is obvious, the base pairing requirement is not. The difficulty is that while Watson/Crick rules mean that only one of the two sequences that precede and follow the "UUCG" is independent, which ever of these two sequences we choose as the independent sequence can be any sequence at all. Thus these two sequences will have to be defined functionally, as in *Some Sequence* and *Its W/C Reverse Complement*. Symbolically our descriptor would be

```
descr
    Some Sequence  "UUCG"  Its W/C Reverse Complement
```

If we let `h5` and `h3` stand for *Some Sequence* and *Its W/C Reverse Complement*, respectively, then our potential UUCG-loop descriptor becomes

```
descr
    h5                                  "UUCG"      h3
```

which resembles the original descriptor introduced at the start of this section. Note that in this descriptor, the literal sequence `"UUCG"` is an implied single stranded region. While this descriptor (it is a descriptor, just not an `rnamotif` descriptor) works, it returns *all* UUCG-loops, not just those with a stem of 4 to 6 base pairs, because it says to find any sequence that contains a `"UUCG"` surrounded by a Watson/Crick helix of *any* length.

What we need is some way to attach the desired length constraints to either the `h5` and/or `h3` parts of this descriptor. In `rnamotif`, you do this by placing these additional constraints in a parameter list, a list of values surrounded by parentheses that immediately follows the descriptor element you wish to constrain. In this case we wish only UUCG-loops with a stem of 4 ot 6 base pairs, so we attach a parameter list to the `h5` containing the numbers 4 and 6. And although it is not strictly necessary, for reasons that will become clear later on in this tutorial we separate the two numbers with a comma. Here is the modified descriptor

```
descr
    h5(        4,        6 )         "UUCG"     h3
```

This type of parameter list scheme is adequate for this example but it is unsatisfactory for more complicated descriptors. Each `rnamotif` structure element has several properties, all of which have default values. One of the design goals of `rnamotif` was that if the default value is acceptable, it does not need to be explicitly set. The default values for W/C helix lengths are 3 to 30, and you only have to explicitly set the one(s) you wish to differ from these default values. For example, had we wished to find UUCG-loops that had a stem length of 3 to 6 base pairs, we would have only had to set the maximum helix length to 6. But now we cannot decide if this 6, as in `h5( 6 )`, means the minimum length or the maximum length! And in fact it is even worse, since an `h5` element has four integer valued parameters — the minimum length, the maximum length, the number of mispairs allowed in the helix and the number of mismatches to any sequence constraint. The way `rnamotif` removes this ambiguity is to use keyword parameters where the parameter's name is explicitly associated with its value. Adding keywords to our evolving descriptor gives

```
descr
    h5( minlen=4, maxlen=6 )         "UUCG"     h3
```

which differs from the real `rnamotif` only in how it treats the single stranded `UUCG`.

Once again, the scheme of using a literal sequence to represent a single stranded region works in this example, but there are places where it can be made to work only with difficulty. How, for example, would one indicate that the literal may have one or more mismatches, or that it is merely a run of any base, where the length of the run could even be zero? `rnamotif` solves these problems by introducing a new structural element, `ss` which represents a single stranded sequence. Any sequence constraints on this element are just placed in the optional parameter list. Making this last change to the descriptor we are working on, produces

```
descr
    h5( minlen=4, maxlen=6 ) ss( seq="UUCG" )   h3
```

which is, finally, a real `rnamotif` descriptor differing only from the one shown at the beginning of this section, in that it lacks the two characters `^` and `$` in the literal sequence `"UUCG"`.

The meanings of (and need for) these special characters are due to the fact that sequence constraints in `rnamotif` are “unanchored”. This means that the constraint `seq="UUCG"` matches not just `UUCG`, but *any* occurence of `UUCG` in a longer string, eg `aUUCG`, `UUCGu` or `aUUCGu`. When you wish `rnamotif` to limit its matches to only those sequences that begin with `UUCG`, you precede the `UUCG` with a `^`, a “meta-character” which matches the “beginning of the string”. Similarly to match only those strings ending with `UUCG`, you follow the `UUCG` with a `$`, another meta-character which matches the “end of the string”. Using both the `^` and the `$` will limit the matches to strings that exactly match `UUCG`, which is what we want. Putting it all together we have

```
descr
    h5( minlen=4, maxlen=6 ) ss( seq="^UUCG$" ) h3
```

which is the `rnamotif` descriptor shown at the beginning of this section. When this descriptor is applied to the RNA part of Genbank 111.0 (15-Apr-1999) it finds 445 hits, the first 10 of which are shown below.

```
AEURR16S   0.000 0 1035 12 tccc   ttcg   ggga
AGIRRDX    0.000 0  180 14 ggagc  ttcg  gctcc
AGIRRDX    0.000 0  181 12 gagc   ttcg   gctc
AHNRRDX    0.000 0  181 12 gagc   ttcg   gctc
ANNRRO     0.000 0   39 12 cccc   ttcg   gggg
AOSRR18S   0.000 0  175 12 gccc   ttcg   gggc
APBRR16SD  0.000 1 1321 12 tacc   ttcg   ggta
APSRRE     0.000 0  227 12 cacc   ttcg   ggtg
AQF16SRRN  0.000 1   39 12 agcg   ttcg   cgct
AQF16SRRN  0.000 1   40 14 cagcg  ttcg  cgctg
```

Why, if we looked for `UUCG`, does the output contain only `ttcg`, and why are they all lower case? First, `rnamotif` considers `u` and `t` to be the same, and as `t` is used to represent `u` in most databases, it translates any `u`’s in sequence constraints, as well as in the databases that are being searched to `t`’s. Second, any upper case literal sequences and database sequences are converted to their lower case equivalents as well.

By default rnamotif searches both the sequences you enter and their w/c complements. We did this because if you’re searching genomic DNA the motifs can be on either strand. Of course, if you wish to search only the sequences you’ve entered you can turn this feature off by adding the line

```
chk_both_strs = 0;
```

to the parms section.

So far so good. But there can be confusion as to how to number the hits on the complementary strand. Hits on the input (sense?) strand are numbered from the 5’-most base. So we decided that we should number hits on the complemenary strand also from their 5’-most base, and indicate the this hit is on the compliementary strand by putting a 1 in the third column:

Thus (for a different example):

```
    gi|94681052|ref|NM_017623.4|    0.000 0      41   35 c tg c
ggcgggt cggttaggctggtt gttcgcc g cg
```

with the 0 in column 3 is on the input (sense) strand while

```
    gi|94681052|ref|NM_017623.4|    0.000 1    4634   39 t gg g
gagggag atccttgtggggatggga cttcttt a tc
```

with the 1 in column 3 is on the (implied) complementary strand; in the original sequence the motif begins at postion 4634 and moves *right to left*. (When we were designing rnamotif we looked hard at this double stranded numbering problem and decided that this was the least worst case.)

2.1.2. Sequences and Regular Expressions.

Based on our discussion of the UUCG-loop, we suspect that we could convert the UUCG-loop descriptor into a GNRA-loop descriptor simply by changing the `"^UUCG$" to "^GNRA$"`, and in fact this is so. The descriptor shown below finds all GNRA tetra-loops with a 4 to 6 base pair stem.

```
descr
    h5( minlen=4, maxlen=6 ) ss( seq="^gnra$" ) h3
```

This descriptor shows an important `rnamotif` feature — in character strings representing sequences, the letters represent not just the individual bases, with `u` → `t`, but also the IUPAC codes, which in this example, `n` and `r` have their usual meanings of any base or "purine" (`a` or `g`) respectively. The complete list of IUPAC codes is shown in Chapter 5.

The character strings that `rnamotif` uses to represent literal sequences are actually a special type of pattern called regular expressions, (or to be precise, simple regular expressions without alternation.) A regular expression (RE) is a sequence of ordinary characters and/or meta-characters. An ordinary character matches itself. The RE `a` matches any string that contains at least one `a`. Similarly the RE `ag` matches any string that contains an `ag`, and as we've already seen, the RE `uucg` matches any string containing `uucg`, or actually `ttcg` as `rnamotif` translates all `u`'s to `t`'s in both literal strings and in the sequences of the databases being searched. This is of course useful, but what gives RE's their power is the set of meta-characters and how they interact with the ordinary characters. The `rnamotif` meta-characters, along with their meanings are shown in the folllowing table.

Meta	Meaning/Action
`.`	Match any single character.
`^`	Match the beginning of the string.
`$`	Match the end of the string.
`[`*ab...*`]`	Match any single character between the brackets.
`[^`*ab...*`]`	Match any single character that is *not* between the brackets.
`*`	Match *zero* of more instances of the *preceding* RE.
`\{`*n*`\}`	Match a run of *n* instances of the *preceding* RE.
`\{`*n,m*`\}`	Match a run of *n* to *m* instances of the *preceding* RE.

These meta-characters allow `rnamotif` descriptors to specify all sorts of useful string patterns. For example, here's a tetra-loop descriptor that accepts any four bases in the loop, but imposes constraints on the two bases adjacent to the loop.

```
descr
```

```
h5( seq="g$" ) ss( len=4 ) h3( seq="^yy" )
```

This descriptor contains two semianchored sequences. In the `h5` element, the sequence `"g$"` forces this strand of the helix, whatever its length, to *end* with a `g`. The sequence `"^yy"` in the corresponding `h3` element forces that helical strand to *begin* with two pyrimidines, as `y` is the IUPAC code for `c` or `u`. Note that the two sequence constraints have different lengths, which is acceptable as long as the two lengths are consistent. In this example, the 5' constraint requires a helix of at least one base pair, while the 3' constraint requires a helix that has at least two base pairs. These two constraints are consistent as they can both be satisified with a helix of two or more base pairs. This indirectly specified minimum length of two overrides the default W/C helix minimum length of 3, but the maximum length of this helix is unchanged from 30. Had the sequence lengths inferred from the `seq` constraints been inconsistent with each other or with other explicit length information, `rnamotif` would report this inconsistency to the user, and then exit without performing the search. This descriptor also introduces one more `rnamotif` feature, the `len` keyword. This keyword is a convenient shortcut for use when the minimum and maximum length (`minlen`, `maxlen`) of a structural element are the same.

The next descriptors show how to include runs of specified bases in a sequence constraint. The first descriptor uses the `*` (or Kleene closure) operator to specify an arbitrary run of specified characters. From the meta-character table, we see that the star operator (`*`) means accept *zero* or more instances of the RE that precedes it. Thus `g*` matches ε (the empty sequence of *zero* `g`'s), `g`, `gg`, `ggg`, etc. Similarly `y*`, `b*` and `n*` match runs (including the zero length run) of pyrimidines, any base but `a`, and any base at all, respectively.

The next descriptor finds tetraloops where the 5' helical stem is composed solely of pyrimidines

```
descr
      h5( seq="^y*$" ) ss( len=4 ) h3
```

The obvious question about this descriptor is that if the RE `"^y*$"` matches any run of pyrimidines, including the run containing *zero* pyridines, does this descriptor actually return "degenerate" tetraloops consisting solely of four bases? The answer is no. The reason is that the implied length of the `^y*` is $[0, \infty)$, which is no constraint at all, and therefore does not change the W/C helix length values from their defaults of [3,30]. In general, any sequence constraint that contains the `*` operator, or any sequence without a `*` that is not anchored at both ends either uses the default maximum length or the maximum length supplied by either the `maxlen` or `len` parameters.

Next we consider sequence constraints that require either a run of n or n to m instances of the preceding RE. This is done by following the character (or character class, eg `y`) with either `\{`n`\}` for a run of n instances or `\{`n`,`m`\}` for a run of n to m instances. These operators are not as useful as they seem, in part because they imply more than what they actually do. Consider finding our usual tetra-loops with stems of length 4 to 6 that contain runs of exactly 2-4 pyrimidines. Accordingly we try this next descriptor

```
descr
      h5( minlen=4, maxlen=6, seq="y\{2,4\}" ) ss( len=4 ) h3
```

which returns (among others) this result

```
AEURR16S 0.000 1 204 16 tcttcc atgc ggaaga
```

The 5' side of this helix contains a run of six pyrimidines, which, while not the desired result, is what the descriptor specifies. The problem is that while we wanted runs of *exactly* 2-4 pyrimidines, the \{,\} operator accepts any run that has a length 2-4 even if that run is contained in longer run. If we surround the original sequence constraint with the purine character, `r`, we get a new sequence constraint, `seq="ry\{2,4\}r"`, which does find only sequences containing helices with runs of 2-4 pyrimidines, but only when those helices begin and/or end with a purine. It no longer finds helices that begin and/or end with a pyrimidine even when those sequences have runs of the correct length. If we attempt to fix this RE by using `r*` for `r`, we're right back where we started because `r*` matches runs of zero purines, once again allowing runs of pyrimidines longer than 4. In fact, there is no way to do this with regular expressions because they are simply not powerful enough to count. However, the `rnamotif` score section can count and we will return to this (and other counting problems) below, in a section which contains a descriptor that does find all tetra-loops that contain only runs of 2-4 pyrimidines in the helical stem.

We conclude this section on sequence constraints and regular expressions with a discussion about the use of character classes. There are two kinds: ordinary character classes which specify a set of acceptable characters, and negated character classes which specify a set of unacceptable characters. A character class is two or more characters enclosed in brackets. A negated character class is two or more characters enclosed in brackets, where the first character that follows the left bracket is a circumflex (^). For example, `[ag]` is a character class that accepts either an `a` or a `g` and `[^a]` is a negated character class that accepts anything but an `a`. Functionally, they act like the IUPAC characters `r` and `b`. They are most useful to specify the less common IUPAC codes like `k` and `m` which stand for the character classes `[gt] and [ac]`. Thus the GNRA loop desriptor could have been written

```
descr
      h5( minlen=4, maxlen=6 ) ss( seq="^g[acgt][ag]a$" ) h3
```

which uses the two character classes `[acgt]` for `n` and `[ag]` for `r`.

2.1.3. Changing the base pair rules.

In all of the examples up to this point, the helices have contained only strict Watson/Crick pairs: {`"a:u"`, `"c:g"`, `"g:c"`, `"u:a"`}. However, there are many motifs that contain other base pairs, and `rnamotif` allows the user to specify which pairs (or triples or quads) are acceptable in the various helical elements of the descriptor. Two levels of control are provided. By adding a statement in the optional parms section, the user can change the default pairing rule for each of the four types of helix — Watson/Crick, parallel duplex, triplex and quadruplex. In addition the user can set the `pair` parameter of any particular helical element which will set the pairing rule for this helix. The default base pairs or "pairsets" for each type of `rnamotif` helical element are shown in Chapter 5.

A pairset is a comma separated list of special strings enclosed in curly brackets. Each string in a pairset contains 2-4 instances of `a`, `c`, `g`, `t` (or `u`) separated by colons. The order of the strings is irrelevant and multiple copies of the same string are ignored. The letters represent the bases on the first, second (and third or even fourth) strand that make an acceptable base pair (or triple or quad). For example, the strings `"a:u"`, `"a:u:u"` and `"g:g:g:g"` specify a standard A:U basepair, an A:U:U triple, where the second U is the Hoogsteen base from the third strand, and a four-stranded G-quartet, respectively. The strand directions are ↑:↓, ↑:↓:↑, and ↑:↓:↑:↓ respectively. All elements of pairset must have the same number of bases. For example, a pairset may not contain both pairs and triples.

`rnamotif` provides several pairset operations, three of which are discussed here. A complete list of pairset operations and their actions is shown in score section. Pairsets can be assigned (`=`), have

pairs added to them (`+=`) and have pairs removed (`-=`). The next table shows the effect of these operations on the default pairset for duplexes (both Watson/Crick and parallel). Note that the removal operation in Line 3 of this table has no effect, as it is not possible to remove items from a set unless they are already members of it.

	Operation	New Pairset
1	`pair={"g:u","u:g"}`	`{"g:u","u:g"}`
2	`pair+={"g:u","u:g"}`	`{"a:u","c:g","g:c","u:a","g:u","u:g"}`
3	`pair-={"g:u","u:g"}`	`{"a:u","c:g","g:c","u:a"}`
4	`pair-={"a:u"}`	`{"c:g","g:c","u:a"}`

The most common non-standard base pair is the G:U or "wobble" pair. In order to search for UUCG-loops that permit G:U pairing in the stem, all we do to the descriptor from §1.1 is change the pairing rule via the `pair` parameter to the `h5` (or `h3`) structural element. Since we wish to add both the G:U and U:G pairs, we use "increment" operator `+=`.

```
descr
      h5( minlen=4,maxlen=6,pair+={"g:u","u:g"} ) ss( seq="^UUCG$" ) h3
```

In fact, G:U pairs are so common in RNA sequences, that `rnamotif` provides a predefined pairset called `gu` which is equal to `{"g:u","u:g"}`. Instead of using the literal pairset `{"g:u","u:g"}` as the value to be added to the `pair` parameter, we could have used the shorter `gu` as in

```
descr
      h5( minlen=4,maxlen=6,pair+=gu ) ss( seq="^UUCG$" ) h3
```

Finally, we could have used the parms section to change the default duplex rule, which is useful for descriptors that contain more than one duplex.

```
parms
      wc += gu;

descr
      h5( minlen=4, maxlen=6 ) ss( seq="^UUCG$" ) h3
```

2.1.4. Mispairs and Mismatches.

In this section we discuss the ways in which users write descriptors that contain imperfections, ie mispairs in helices and mismatches to sequence constraints. Every `rnamotif` structure element has a `mismatch` parameter, which sets the largest number of mismatches that are allowed to a sequence constraint. For example, the element

```
ss( seq="^uucg$", mismatch=1 )
```

would accept any sequence that matched at least three of the four characters `"uucg"`. The default value of the `mismatch` parameter is zero. Note that while this descriptor accepts mismatches, it does not require them. In addition to sequence mismatches, helical elements, may also specify the

maximum number of mispairs that will be accepted in this helix. The structural element

```
h5( minlen=4, maxlen=6, mispair=1 )
```

defines a stem of 4-6 base pairs that may include one mispair. The mispair must be an internal base pair unless the user has also included the `ends` parameter to inform `rnamotif` that mispair(s) may be on one or both ends of the helix. If we add `ends='mm'` to the previous element giving

```
h5( minlen=4, maxlen=6, mispair=1, ends='mm' )
```

then this helix would allow (but not require) a mispair at any point including the distal or proximal base pair. A complete list of `ends` values is shown in the table below.

Value	Meaning
´mm´	Allow a mispair on either the proximal* or distal† ends.
´mp´	Allow a mispair on the distal end.
´pm´	Allow a mispair on the proximal end.
´pp´	Allow only internal mispairs. (Default.)
* The proximal end is the end closest to the loop. † The distal end is the end farthest from the loop.	

Note that the four values `'mm'`, `'mp'`, `'pm'` and `'pp'` are surrounded by single quotes (`'`) instead of the double quotes (`"`) that surround literal sequences. This is necessary because sequences and strings are different. For example, in the *sequence* `"gnra"`, the `n` and the `r` are the IUPAC codes for any base and `a` or `g`. In addition, any upper case letters in a literal sequence are converted to lower case, and any `u`'s are converted to `t`'s. However, the literal *string* `'gnra'` is just the four characters, `gnra`. Characters in strings are case sensitive, are never IUPAC codes and `u`'s represent themselves. The explanation for having both strings and sequences will be be covered in detail in the score section, but for now it is important to realize that the assignment `ends='mp'` informs `rnamotif` to accept a mismatch on the distal end of a helix, while the assignment `ends="mp"`, which is stored internally as `ends='[ac]p'` doesn't make sense and `rnamotif` marks it as an error and aborts the search.

We close this section on imperfect matches by noting that specifying the allowable number of mispairs becomes increasingly inadequate as the difference between the minimum and maximum allowable helix lengths becomes larger. The following descriptor specifies a hairpin whose stem length can vary from 8 to 16, with up to four mispairs

```
descr
      h5( minlen=8, maxlen=16, mispair=4 ) ss( len=4 ) h3
```

The problem with this desriptor is that we had to set the `mispair` value high enough to deal with maximum possible helix length, with the undesired result that many shorter helices will have too many mispairs. What we wanted is for the number of mispairs to be proportional to the length of the stem, and `rnamotif` allows us to do this with the `pairfrac` parameter. This parameter takes a floating point number between 0 and 1 and represents the minimum fraction of the stem that must be paired for the helix to be accepted. Returning to our example, we find that 4/16 is .25, so we replace the `mis-pair=4` parameter with `pairfrac=0.75`

```
descr
      h5( minlen=8, maxlen=16, pairfrac=0.75 ) ss( len=4 ) h3
```

which accepts up to 2 mispairs for stems with lengths 8 to 11, up to 3 mispairs for stems of length 12 to 15, and up to 4 mispairs for a stem length of 16.

2.2. Properly Nested Hairpins.

This section shows how to use `rnamotif` to search for larger motifs that are composed entirely of properly nested hairpins. A properly nested hairpin is a hairpin that is either completely enclosed in another hairpin or a hairpin that is not enclosed at all. Thus all single hairpins are properly nested hairpins as are any two or more consecutive hairpins. A set of two or more hairpins is inproperly nested if none of the hairpins in that set is properly nested. Improperly nested haipins are called pseudoknots and are discussed in §3.

The basic rule for creating an `rnamotif` descriptor that describes a particular motif is to begin at the 5' end of that motif and reading from 5' to 3' writing `h5`, `ss` and `h3` each time one encounters the 5' strand of a helix, a single stranded region or the 3' end of a helix. If any of these substructures have length, sequence or other constraints, write these constraints in a parameter list that immediately follows the substructure which is to be constrained. Thus every hairpin has this skeleton — `h5 ss h3`. Two consecutive hairpins would read `h5 ss h3 ss h5 ss h3` and a hairpin with an internal loop — nested hairpins — uses this template `h5 ss h5 ss h3 ss h3`. If all the hairpins in this motif are are properly nested, the corresponding `h5` and `h3` elements are automatically paired, using the rule that reading from 5' to 3', the current `h3` element is paired with the closest unpaired `h5` element. In fact, the validity of this pairing rule is another definition for properly nested hairpins.

2.2.1. Consecutive Hairpins.

Suppose, now, that instead of looking for hairpins in isolation we wish to find GNRA-loops that are followed within 10 bases by a UUCG-loop. From the previous section we know that two consecutive hairpins will use the template `h5 ss h3 ss h5 ss h3`, so we immediately write down this descriptor.

```
#     gnra loop followed by uucg loop.

descr
      h5
            ss( seq="^gnra$" )
      h3
      ss( minlen=0, maxlen=10 )
      h5
            ss( seq="^uucg$" )
      h3
```

Searching the RNA part of Genbank 111.0 (15-Apr-1999), `namotif` finds 541 instances of this motif, of which the first 10 are shown next.

```
AEURR16S   0.000 0 1018 28 cccta   gaga   taggg nttt       ccc     ttcg     ggg
AEURR16S   0.000 0 1018 29 cccta   gaga   taggg ntt        tccc    ttcg    ggga
```

```
AEURR16S   0.000 0 1019 27 ccta    gaga    tagg gnttt      ccc     ttcg     ggg
AEURR16S   0.000 0 1019 28 ccta    gaga    tagg gntt       tccc    ttcg    ggga
AEURR16S   0.000 0 1020 26 cta     gaga     tag ggnttt     ccc     ttcg     ggg
AEURR16S   0.000 0 1020 27 cta     gaga     tag ggntt      tccc    ttcg    ggga
AHCSSRRNA  0.000 0  646 32 cacc    gcaa    ggtg agcactgctc tgg     ttcg     cca
ANCRRDAA   0.000 0  953 27 acca    gaga    tggt ttctt      ctc     ttcg     gag
ANCRRDAA   0.000 0  954 26 cca     gaga     tgg tttctt     ctc     ttcg     gag
AZORRDC    0.000 0  978 26 tga     gaga     tca gggagt     tcc     ttcg     gga
```

In addition to showing how to describe two consecutive hairpins, the previous descriptor also shows some new `rnamotif` features. The first line which begins with a # is a "comment" and is ignored by `rnamotif`. Comments, which may be placed anywhere in a descriptor and not just on lines by themselves, always begin with a # and tell `rnamotif` to ignore the rest of the line. A descriptor may extend over more than one line. In fact, `rnamotif` is "free-format" meaning that the descriptor may be spread over as many lines (including blank lines) as are convenient, and that the various elements may be indented to improved readability, although `rnamotif` ignores indenting, treating any run of spaces, tabs and newlines as a single space.

Two final comments about this descriptor: Had we wished to find three or more consecutive hairpins we simply would have added additional instances of the `h5 ss h3` skeleton, along with constraints following the UUCG-loop in this descriptor. And if any of the hairpins were adjacent, ie, with no single stranded sequence between them, no intervening `ss( len=0 )` would have been needed.

2.2.2. Bulges and Internal Loops.

By now, the reader has seen how to create several kinds of helices (with varying lengths, sequence constraints, mispairs, and even nonstandard base pairs), but no helices with bulges. The reason for this curious lack is simple — `rnamotif` does not permit bulges in helices. Instead, `rnamotif` requires that any helix that contains one or more bulges be subdivided into two or more helices such that the bulge(s) become single stranded sections that now connect nested helices.

The rationale for this restriction is that pattern-based descriptors, while powerful, suffer from two serious shortcomings. One of these, their inability to count; a second is that they have difficulty dealing with context. The `rnamotif` score section removes these two limitations, but in order to do so, `rnamotif` requires that every position in every element of the descriptor have a unique name, and it is this unique name requirement that is incompatible with helices containing bulges.

To understand why this is so, consider a helix that contains a single *optional* bulge of one base. If the bulge is absent, the length of the helix is, for example, 5 and if the bulge is present 6. If we number the bases from the 5' end as 1, 2, ..., the numbers up to the bulge don't change, but the numbers after the bulge are increased by 1 when the bulge is present over when it is absent. Because the bulge is optional, base number after the bulge are ambiguous. Is base number 5 the last base or the second from last base? There is no way to tell. In fact this problem is not limited to helices with bulges but is present in every helix that has a variable length. The solution here is to allow the user to number the bases of any `rnamotif` element both from the 5' end as $1, 2, \cdots$ and from the 3' end as $\$, \$-1, \cdots$.

Unfortunately, this scheme would break down if a helix were allowed to have two or more optional bases. Bases from the 5' end to the first bulge base can be referenced as $1, 2, \cdots$ and those from the 3' end to the last bulge base as $\$, \$-1, \cdots$, but those base numbers between the bulge bases are now ambiguous. And there are problems with bulges. In general the user would not only like to

specify their size(s), but their location(s), including which bulges are allowed to oppose each other. Taken all together, satisfying these requirements would greatly increase the complexity of the helical descriptor elements, so `rnamotif` just makes the user break the helix into subhelices to accomodate bulges. Optional bulges are repesented by `ss` elements with a `minlen=0`.

2.2.2.1. UUCG-loop with a helical bulge.

In this section we return once again to the UUCG-loop. However, this time we would like UUCG-loops that not only contain stems of 4-6 basepairs but may include an optional bulge base either 2-3 bases from the 5' end and/or 2-3 bases from the 3' end of the 5' strand of the helix. Based on the rule discussed in the previous section for bulge bases in helices, we know that the original helix must be divided into two nexted helices with the bulge represented as an `ss` between the two `h5` elements. Here is the descriptor.

```
descr
      h5( minlen=2, maxlen=3 )
      ss( minlen=0, maxlen=1 )      # optional bulge base!
      h5( minlen=2, maxlen=3 )
            ss( seq="^uucg$" )
      h3
      h3
```

A search of the RNA part of Genbank 111.0 (15-Apr-1999) find 769 instances of this motif, the first 10 of which are shown below.

```
ACNRRNAJ   0.000 0  205 13 gg  a cc  ttcg  gg  cc
AEORR16SA  0.000 0  205 13 gg  a cc  ttcg  gg  cc
AEORRDA    0.000 0  169 13 gg  a cc  ttcg  gg  cc
AEURR16S   0.000 0 1035 12 tc  . cc  ttcg  gg  ga
AGIRRDX    0.000 0  179 15 gg  g agc ttcg gct  cc
AGIRRDX    0.000 0  180 14 gg  . agc ttcg gct  cc
AGIRRDX    0.000 0  180 14 gga . gc  ttcg  gc tcc
AGIRRDX    0.000 0  181 12 ga  . gc  ttcg  gc  tc
AGTRRNA    0.000 0  159 13 gg  a tc  ttcg  ga  cc
AHNRRDX    0.000 0  181 12 ga  . gc  ttcg  gc  tc
```

The bulge base is shown in the third column of bases where a dot (`.`) is used to indicate the bulge is absent. The output of this descriptor shows both an advantage and a disadvantage of requiring the user to break helices to accomodate bulges. On the plus side, the bulge is clearly delimited in its own column, thus providing a partial alignment. The minus is that original helix now takes 5 columns instead of the more functionally correct 2.

2.2.2.2. Internal Loop: Iron Response Element.

In this example we consider the "Iron Response Element" or IRE. An IRE is a helical stem that contains an internal loop. The loop consists of three bases on the 5' side opposed by a single base on the 3' side. In some IRE's, the internal loop degenerates into a single 5' bulge base. In all cases the 5' side of the internal loop ends with a `c`. Since `rnamotif` does not permit helices to contain bulges, the long helix must be divided into two shorter helices connected by single stranded regions representing the internal loop. The 5' `ss` element needs a length range of 1 to 3 and the 3' `ss` a range of 0 to 1.

A simplified IRE descriptor is shown below.

```
parms
      wc += gu;

descr
      ss( len=3 )
      h5( len=3 )
            ss( minlen=1, maxlen=3, seq="c$" )
            h5( len=5 )
                  ss( len=6, seq="^cagug" )
            h3
            ss( minlen=0, maxlen=1 )
      h3
      ss( len=3 )
```

Although this descriptor does find all sequences that could be IRE's, it also finds sequences that are not IREs. This is because an IRE has either an internal loop of 3:1 bases *or* a single 5' bulge, but this descriptor accepts any sequences that contain any of the six possible internal loops that have 1 to 3 5' bases opposing 0 to 1 3' bases (1×0, 2×0, 3×0, 1×1, 2×1, 3×1). As such it illustrates the second major limitation of pattern based descriptors — there is no easy way to use the part of the solution that has already been found as context to limit what will be an acceptable solution for the part still to be found. Specifically, what we need is some way to tell `rnamotif` that if it has found a partial solution that contains a 3 base bulge on the 5' side of the helix, then it should accept only a single base bulge on the 3' side of this helix. Similarly, if it is has found a single base 5' bulge, then it must only accept a 0 length bulge on the 3' side. All other matches, while they do indeed match the descriptor, are not IRE's and must be discarded. The solution to this problem is provided by the `rnamotif`'s score section, and the complete IRE descriptor is provided and discussed there.

2.2.2.3. Using consecutive helices to change the pairing rules.

In addition to requiring users to break helices with bulges into two or more consecutive helices, `rnamotif` also requires that a helical element in which the pairing rule is not the same for every position in the helix be broken into two or more consecutive helices, each with its own pairing rule. Once again, the reason was to remove possible ambiguity as to which bases are covered by which rule because (as was shown in §2.2.1,) the number of the interior bases in helices of varying lengths is inherently ambiguous. To see how this works, let us return to the GNRA-loops with stems of 4-6 base pairs. Since the G and the A that open and close the loop form an unusual G:A pair, we now wish to search for possible GNRA-loops where the last base pairs of the stem are G:A. To do this we will need to break the original stem of 4-6 base pairs into two helices — the outer of 3-5 (standard) base pairs and the inner of length 1 that uses only G:A pairs. Here is the descriptor.

```
descr
    h5( minlen=3, maxlen=5 )
    h5( len=1, pair={ "g:a" } )
        ss( seq="^gnra$" )
    h3
    h3
```

Our usual search over the RNA part of Genbank 111.0 (15-Apr-1999) finds 124 instances of this motif with the first 10 shown below.

```
AGCRR18S   0.000 0 1706 12 ggg   g ggaa a   ccc
BLHRR24S   0.000 0  676 12 tgc   g gtaa a   gca
BLO18SRRNA 0.000 0  725 12 tgg   g ggga a   cca
BODRR24SA  0.000 0  667 12 tgc   g gtaa a   gca
CABRR16S   0.000 0  633 12 acc   g ggga a   ggt
CFX16RNA01 0.000 1  199 12 att   g ggga a   aat
CFXRRDA    0.000 1  350 12 att   g ggga a   aat
CHORREB    0.000 0  301 12 tga   g gtga a   tca
CLORR16SK  0.000 0  808 12 agg   g ggga a   cct
CRORRZAAC  0.000 0  444 12 aac   g gaaa a   gtt
```

2.2.3. tRNA.

We close this section on properly nested hairpins with a descriptor for tRNA. A tRNA molecule contains 3 consecutive hairpins, called the X, anticodon and Z loops, enclosed by another helix. A short single stranded region called the acceptor stem follows the 3’ end of the outer helix.

```
parms
      wc += gu;

descr
      h5( minlen=6, maxlen=7 )
            ss( len=2 )
            h5( minlen=3, maxlen=4 )
                  ss( minlen=4, maxlen=11 )
            h3
            ss( len=1 )
            h5( minlen=4, maxlen=5 )
                  ss( len=7 )
            h3
            ss( minlen=4, maxlen=21 )
            h5( minlen=4, maxlen=5 )
                  ss( len=7 )
            h3
      h3
      ss( len=4 )
```

2.3. Pseudoknots.

Pseudoknots are a class of structures composed of improperly nested W/C helices. The mininal pseudoknot consists of two such helices where the 5’ strand of the second helix begins between the 5’ and 3’ strands of the first helix. Originally considered unlikely, pseudoknots have been found to be involved in a number of RNA functions, such as read through and frameshifting.

In the section, **Properly Nested Hairpins** (§2), we pointed out that improperly nested hairpins are precisely those in which the corresponding 5' and 3' strands of each helix can not be automatically paired using the "current unpaired `h3` element goes with the closest unpaired `h5` element" rule. For this reason, pseudoknots require some way to explicitly link the corresponding `h5` and `h3` elements, and in `rnamotif` this is done by "tagging" the corresponding helical elements by setting the `tag` parameter of these elements. A descriptor for a mininal (two helix) pseudoknot is shown below.

```
parms
    wc += gu;

descr
    h5( tag='h1', minlen=4, maxlen=6 )
            ss( seq="^gnra$" )
        h5( tag='h2', minlen=4, maxlen=6 )
            ss( minlen=3, maxlen=10 )
        h3( tag='h1' )
            ss( seq="^uucg$" )
        h3( tag='h2' )
```

The two helices, as shown by the values of their `tag` parameters, are called `h1` and `h2`. Structure element tags can be up to 255 characters in length and can be any convenient string of characters including spaces, but may not include newlines. Because `tag` values are strings and not sequences, they should be enclosed in single quotes (`'`) instead of the double quotes (`"`) used for literal sequences. A search of GBRNA v. 111.0 (15-Apr-1999) finds only six sequences that are consistant with the secondary structure of this descriptor and they are shown below.

```
CLORR16SB 0.000 1 982 31 gcag   gtaa ggtt  cttcgcg      ttgc ttcg  aatt
HUMAALU20 0.000 0 367 35 tagaa  gaga aggtg gagtgcc     tttta ttcg tattt
OCERRDA   0.000 1 988 31 gtag   gtaa ggtt  cttcgcg      ttgc ttcg  aatt
TRBRR24S  0.000 0 322 37 gggggg gaga ggca  aagcgctcc  ttcttc ttcg  tgtt
TRBRR24S  0.000 0 323 36 ggggg  gaga ggca  aagcgctcct  tcttc ttcg  tgtt
TRBRR24S  0.000 0 327 32 ggagag gcaa agcg  ctcc       ttcttc ttcg  tgtt
```

2.4. Other Structural Elements.

In addition to structures composed of Watson/Crick helices and single stranded regions, `rnamotif` descriptors can also specify three other secondary structures. These are parallel duplexes, triple helices (triplexes) and 4-stranded helices (quadruplexes). These new structures may be freely intermixed among themselves or with the previously described W/C duplexes including pseudoknots and single stranded regions with one exception — all parallel helices, triplexes and 4-plexes must be properly nested. What this means is that any helical structure that begins within a parallel duplex, triplex or 4-plex must be completed before the next structural descriptor of that non-W/C helix.

2.4.1. Parallel Helices.

A parallel duplex is represented by the `rnamotif` descriptor elements `p5` and `p3` which represent the 5' and 3' strands of the parallel duplex. These two structures have the same parameters as the W/C elements `h5` and `h3` except that the default base pairing is held in the `rnamotif` variable `ph`.

```
descr
    p5( minlen=4, maxlen=6 ) ss( seq="^gnra$" ) p3
```

2.4.2. Triplexes.

`rnamotif` uses the three symbols `t1`, `t2` and `t3` in that order to represent the three strands of a triplex. `t1` is the 5'-most strand and is followed by `t2` and `t3` (in that order) representing the antiparallel strand and the 3' parallel strand. The default base pair rule for triplexes consists of a single triple — A:U:U — and is held in the predefined `rnamotif` variable `tr`.

```
descr
    t1( tag='tr1', minlen=4, maxlen=7 )
        ss( minlen=3, maxlen=10 )
    t2( tag='tr1' )
        ss( minlen=3, maxlen=10 )
    t3( tag='tr1' )
```

2.4.3. 4-Stranded helices (Quadruplexes).

Similary, `rnamotif` uses the four symbols `q1`, `q2`, `q3` and `q4` in that order to represent the four strands of a quadruplex. `q1` is the 5'-most strand and is followed by `q2`, `q3` and `q4` representing the first antiparallel strand the second parallel strand and finally the 3'-most antiparallel strand. The default pairset for 4-stranded helices is the `rnamotif` variable `qu` and has one member, G:G:G:G.

```
descr
    q1( tag='qu1' )
        ss
    q2( tag='qu1' )
        ss
    q3( tag='qu1' )
        ss
    q4( tag='qu1' )
```

2.5. Parms.

The parm section is used to set the default values for the various structural elements. This section is optional, but if present must be the first section of an `rnamotif` descriptor preceding the required `descr` section. The parms section is introduced by the reserved word `parms` which is then followed by one or more parameter definitions. Each parameter definition takes the form of an assignment statement.

2.6. Sites.

One of the major shortcomings of pattern based descriptors is a general inability to incorporate context information. A good example would be a W/C helix which allows GU-pairing but requires that the third basepair is either an A:U or a G:C. Setting the `h5` and `h3` sequence constraints to `"rnn$"` and `"^nny"` is not sufficient because these two constraints will still admit a G:U at position 3. (The pair rule alone will reject A:C.) `rnamotif` provides two mechanisms to handle these

problems — a rather simple “sites” list discussed here and much more powerful score section dicussed in §6.

The sites section is an optional part of an `rnamotif` descriptor. If present it immediately follows the `descr` section and is introduced by the reserved word `sites`. This word is then followed by one more or site specifiers. A site specifier is a colon (`:`) separated list of structural elements followed by the reserved word `in` followed by a pairset containing the acceptable combinations of bases at the position specified by the site. A sites section with two sites is shown below

```
sites
    h5( tag='1', pos=2 ):h3( tag='1', pos=$-1 ) in { "a:u", "g:c" }
    h5( tag='2', pos=1 ):h3( tag='2', pos=$ ) in { "g:c" }
```

The first site specifies that the second base in the W/C helix named `1` is restricted to either an A:U or a G:C, and the second site specifies that the first base of the W/C helix named `2` be G:C. Here’s how the sites section works. Each site specifier in top to bottom order is tested against the current candidate to see if the bases at the specified positions are in the set of allowed pairs. If so, site testing continues until either the referenced bases are not in the site’s pairset (in which case the candidate is rejected) or all sites tests have passed (in which case the candidate is provisionally accepted subject to further testing in the score section.) If the score section is absent, the candidate is accepted.

Although both sites in the above example referred to paired positions in the same helix, this is not required. Thus sites provide a general way to specify long range covariation. For example, the site below

```
ss( tag='ss1', pos=3 ):ss( tag='ss2', pos=2 ) in { "a:u", "g:c" }
```

imposes the requirement that the bases at positions 3 and 2 of the single stranded regions named `ss1` and `ss2` be either an A:U or a G:C. The biggest limitation to using sites is they are “all or nothing”. The candidate sequence either passes all the site specifiers or it is rejected. This is often too severe, and the score section can be used to create more flexible acceptance criteria based on site-like objects.

3. Scoring.

This section describes the fourth and final part of an `rnamotif` descriptor, the score section. Introduced by the reserved word `score`, it follows both the descr and optional sites sections. As has been previously mentioned, pattern based descriptors have two limitations — they can't count and they have trouble using sequences that match an earlier part of a descriptor as context for what is to be accepted in a later part of the descriptor. The score section solves both of these problems.

The score section consists of the reserved word `score` followed by one or more "rules". Each rule is an "action" preceded by an optional expression. If the expression is present, it is evaluated and if true, the specified action is executed. If the expression is false the corresponding action is skipped and the next rule is checked. If the expression is absent, the specified action is always executed. The rules are evaluated from top to botton until 1) the current rule executes an `ACCEPT` statement, in which case the current match is accepted, 2) the current rule executes a `REJECT` statement in which case the current match is rejected or 3) all rules have been executed in which case the current match is accepted.

The actions and expressions are written in a simple AWK-like programming language that enables direct access to both the parameters of the descriptor and the substrings of the current match that correspond to each of the descriptor's structural elements. This is done by making the interpretation of the structural elements context sensitive. For example, in the `descr` section `h5( tag='p1' )` is a pattern that functionally defines the 5' side of a helix tagged `p1`. However in the `score` section, the same sequence of symbols refers to the substring of the current match that is the 5' side of helix `p1`. It is read only string variable whose contents can be examined by the various actions in the rules of the `score` section.

The scoring language contains the usual constructs for testing: `if` and `if`/`else`, looping: `for` and `while`, the standard arithmetic and string operations, and two specialized operations involving bases and base pairing. Variables can not be declared and take their type from the first expression they are assigned to. Unlike many interpreted languages `rnamotif` has rather strict typing. Only conversion between integer and real variables is permitted. In addition using an unassigned variable in a expression results in a run time error which terminates the search with an error message that variable *var* at line *N* is undefined.

3.1. rnamotif Variables.

`rnamotif` has two special predefined variables. The more important one is `SCORE`, a real variable whose value is printed as the number that immediately follows the sequence name in each line of output. In fact, setting this variable is the only way the user can report the result of applying the scoring rules to rate each match. Before applying the rules to the current match, `SCORE` is set to zero. The second special variable is `NSE`, a *read-only* integer that contains the number of structural elements in the descriptor. This variable is often used to compute some global measure of the current match, such as the total number of mispairs by looping over all the structural elements. In addition to these special variables, all variables defined in the `parms` section can be used in `score` expressions. These variables are also read-only. Only variables defined in the `score` section can be written. Once such a variable has been given a type from the type of the first expression assigned to it, it will be initialized to a type dependent default value before the rules are applied to the current match. Thus each match is evaluated without memory of previous matches.

Another important class of `rnamotif` variables are the substrings of the current match that were found by the pattern structural elements of the `descr` section. These substrings are read-only but may be otherwise examined, compared, concatenated and assigned to `score` section variables.

Of particular interest are the substrings of these strings, as it is by examining these individual bases, that users can write `score` programs that can count things like the percentage of certain bases or the number of bases between mispairs. These substrings are accessed with construct very similar to that representing the pattern, only here instead of parameters that define what is acceptable in this postition, the parameters are used to identify the structural element and the optional position of the first base (numbered from 1) and an optional length of the substring. If the length is missing, it means the substring from the starting position to the end of the string. As an example, consider the structural element `h5(tag='p1')` which has matched `ggagc` in the current candidate. Here are some substring selectors and their values.

Selector	Returns	Notes
`h5(tag='p1')`	`ggacg`	The entire string.
`h5(tag='p1',len=3)`	`gga`	The first three bases.
`h5(tag='p1',pos=2)`	`gacg`	The last four bases.
`h5(tag='p1',pos=2,len=3)`	`gac`	

Note that if the position is not specified as in the second of these substrings, the substring begins at position 1.

One disadvantage of this syntax is that it requires the accessed element to have a tag. Since many elements have no obvious names, `rnamotif` allows the user to reference a structural element via its "index". Structural elements are numbered from 5' to 3' beginning with one. If, in the above example, `h5(tag='p1')` was the fourth structural element, a second way of accessing the string it matched in the current candidate would have been to use the `index` parameter as shown below.

Selector	Returns	Notes
`h5(index=4)`	`ggacg`	The entire string.
`h5(index=4,len=3)`	`gga`	The first three bases.
`h5(index=4,pos=2)`	`gacg`	The last four bases.
`h5(index=4,pos=2,len=3)`	`gac`	

The big advantage of using `index` over `tag` is that is lends itself to looping over all the structural elements in a pattern as is shown in the following scoring rule:

```
{ for( i = 1; i <= NSE; i++ )
      SCORE += mispairs( se( index=i ) );
  if( SCORE != 1 )
      REJECT;
}
```

This rule sums the total number of mispairs into the special variable `SCORE`, which is written out immediately following the name of each matched sequence. The `for` loop causes the local variable `i` to range over the values $1, 2, \ldots,$ `NSE`. On each iteration it selects the $\texttt{i}^{\text{th}}$ structural element, which (since its secondary structure is unknown) is done with the generic structural element `se` and passes it to the builtin function `mispairs()`; this in turn returns the number of mispairs in the string that matched that element. For consistency, all elements of type `ss` are defined to have 0 mispairs. After the loop is completed, the `if` statement tests the value of `SCORE` and if it is not equal to 1, executes a `REJECT` statement, rejecting the current match. A rule like this would be used to search for structures that contain several helices, in which one, but only one has a single mispair. In fact, this is the only way to do this because the location of the mispair is unknown, so every helical element would have to

permit one mispair, thus accepting matches that contained a total of *H* mispairs where *H* is the number of helical elements.

A second disadvantage of this notation for selecting the strings that have matched the various structural elements is that it is long. Accordingly, `rnamotif` provides a second, shorter way to access these strings, one that does not use keywords. This shorter selector, as applied to `h5( tag='p1' )` is shown below.

Selector	Returns	Notes
`h5['p1']`	`ggacg`	The entire string.
`h5['p1',1,3]`	`gga`	The first three bases.
`h5['p1',2]`	`gacg`	The last four bases.
`h5['p1',2,3]`	`gac`	
`h5[4]`	`ggacg`	The entire string.
`h5[4,1,3]`	`gga`	The first three bases.
`h5[4,2]`	`gacg`	The last four bases.
`h5[4,2,3]`	`gac`	

While both the keyword and indexed forms provide the same access to the strings that have matched the structure elements, there are some important differences. Because the values are unambiguously assigned in the keyword version, their order is unimportant. However, in the indexed version, since all three values can be integers, their order is required to be id, pos, len. If only two values are present, the missing value is the length which is, as in the keyword version taken to be from the specified position to the end of the string.

We close this section on accessing the substrings that were found to match the various elements of the descriptor with a few final notes. In the examples, all the values were either literal strings or integers, but any string or integer valued expression can be used to select substrings. Tagged versions (`h5(tag='p1')`, `h5['p1']`) cannot be used with the generic structure element `se`, because, except in the case where it refers to an `ss` element, the reference is ambiguous as all elements of the same helix must either have no tag or the same tag. Finally, referencing a position beyond the end of a descriptor or indexing a descriptor < 1 or > `NSE` is an error, and will abort the search with error message describing the errant reference. However, a length that exceeds the length of the current string is silently reduced to that length.

3.2. rnamotif Expressions.

An `rnamotif` rule is a list of one or more statements enclosed in curly brackets. The rule may be preceded by an optional expression, whose syntax is described in this section; the syntax of `rnamotif` statements is covered in the next section.

3.3. rnamotif Statements.

Stmt-type	Syntax	Details
if	`if(` *expr* `)`*stmt*	X
if/else	`if(` *expr* `)`*stmt* `else` *stmt*	X
for	`for(` *e1*`;` *e2*`;` *e3* `)`*stmt*	X
while	`while(` *expr* `)`*stmt*	X
break	`break;`	X
continue	`continue;`	X
assignment	*var* `=` *expr*`;`	X
accept	`ACCEPT;`	X
reject	`REJECT;`	X
compound	`{` *stmts* `}`	X

3.4. score Examples.

This section give some examples from our work of scoring sections. Although they are not (yet) extensively annotated, we hope they are sufficiently self-explanatory to be helpful.

3.4.1. Base Percentages.

```
descr
      h5(minlen=7,maxlen=10)
            ss(len=4)
      h3

score
      { gcnt = 0;
        len = length( h5[1] );
        for( i = 1; i <= len; i++ ){
            b = h5[1,i,1];
            if( b == "g" || b == "c" )
                  gcnt++;
        }
      # require 80% GC in the stem!
        SCORE = 1.0 * gcnt / len;
        if( SCORE < .8 )
            REJECT;
      }
```

3.4.2. Preventing consecutive mispairs.

```
descr
      h5( minlen=10, maxlen=15, pairfrac=.8 )
            ss( minlen=5, maxlen=10, seq="^gcc" )
      h3

score
```

```
        {
          if( mispairs( h5( index=1 ) ) >= 2 ){
              len = length( h5( index=1 ) );
              mpos = 0;
              for( i = 1; i <= len; i++ ){
                    if( !paired( h5( index=1, pos=i, len=1 ) ) ){
                          if( mpos == 0 )
                                mpos = i;
                          else if( mpos == i - 1 )
                                REJECT;
                          else
                                mpos = i;
                    }
              }
          }
        }
```

3.4.3. Runs of bases.

```
descr
    h5( minlen=4, maxlen=6, seq="y2,4" ) ss( len=4 ) h3

score
    {
      nruns = 0; lyrun = 0; yrun = 0;
      for( i = 1; i <= length( h5[1] ); i++ ){
        if( h5[1,i,1] =~ "y" ){
            if( yrun == 0 )
               nruns++;
            yrun++;
            if( yrun > lyrun )
               lyrun = yrun;
        }else
            yrun = 0;
      }
      if( lyrun > 4 || nruns > 1 )
        REJECT;
    }
```

3.4.4. Correlation between parts of an internal loop.

```
parms
    wc += gu;

descr
    ss( len=3 )
```

```
    h5( len=3 )
        ss( minlen=1, maxlen=3, tag='5p', seq="c$" )
        h5( len=5 )
            ss( len=6, seq="^cagug" )
        h3
        ss( minlen=0, maxlen=1, tag='3p' )
    h3
    ss( len=3 )

score
    #
    # Very strict rules on the internal ss's:
    # 5p/3p = 3/1 OR 5p/3p = 1/0. No other solutions!
    #
    # 1/0 solutions which consist only of a 'c' bulge
    # recieve a perfect score of 1.000
    #
    # 3/1 solutions recieve a score that is the sum of the
    # contribution from both sides of the internal loop:
    #
    #   5p  val 3p  val
    #   tgc 0.5 c   0.5
    #   tac 0.4 t   0.2
    #   tcc 0.3 others 0.0
    #   ttc 0.2
    #   cnn 0.1
    #   others 0.0
    #
    #   thus tgc/c gets the perfect score of 1.0
    #
    {
      SCORE = 0.0;
      if( length(ss(tag='5p')) == 3 && length( ss( tag='3p')) == 1 ){
        if( ss(tag='5p') == "tgc" )
            SCORE = 0.5;
        else if( ss(tag='5p') == "tac" )
            SCORE = 0.4;
        else if( ss(tag='5p') == "tcc" )
            SCORE = 0.3;
        else if( ss(tag='5p') == "ttc" )
            SCORE = 0.2;
        else if( ss(tag='5p',pos=1,len=1) == "c" )
            SCORE = 0.1;
        if( ss(tag='3p') == "c" )
            SCORE += 0.5;
        else if( ss(tag='3p') == "t" )
            SCORE += 0.2;
        ACCEPT;
```

```
  }else if( length(ss(tag='5p' )) == 1 && length(ss(tag='3p')) == 0 ){
    SCORE = 1.0;
    ACCEPT;
  }else
    REJECT;
}
```

4. Algorithms

`rnamotif` is a program that searches nucleic acid sequence databases for entries that can adopt a particular secondary structure. This secondary structure is defined via a pattern language whose symbols represent helices and single stranded stretches. These elements can be parameterized as to length, sequence, pairing etc, provding a measure of control over the the sequences the pattern matches. Where `rnamotif` differs from its predecessors (`rnamot`, `palingol`, `patscan`) is that it combines these pattern elements with an `awk`-like scoring section that is used to add capabilities that patterns alone can not provide. In particular, this score section allows users to enforce arbitrary sequence context relations between the pattern elements and to perform counts on the bases of the actual sequences that matched the pattern elements.

The key to making this work was the realization that the symbols that specify the structure elements in the search pattern can have a second related meaning. In the pattern section they describe the motif, but in the score section they represent (with some qualification) the actual sequences that matched those pattern elements. For example, in the pattern section, the symbols `h5(tag='hlx-A')` represent the 5'-end of a Watson/Crick duplex labelled `hlx-A`, but the same symbols in the score section would represent the actual sequence that matched that part of the pattern. Allowing these symbols to act like read-only string variables with length and substring operations and combining them with standard programming constructs like assignments, expressions, testing and looping, makes it easy to implement the arbitrary context relations that `rnamotif`'s predecessors so sorely needed.

`rnamotif` uses a two-stage algorithm to perform the search. Stage one compiles the motif along with any scoring rules into a search list and a set of instructions for the scoring interpreter. Stage two executes the search, testing each entry in the sequence database first, for matches to the pattern, and then those that match the pattern are tested against any scoring rules. Matching sequences, along with the sequence name and location information are written to `stdout`.

4.1. Compilation.

The target secondary structure, along with its scoring rules and other information, is called a descriptor. Descriptors are stored as ASCII text files and are created via a text editor. The inputs to an `rnamotif` run are a descriptor and the database file to be searched.

A descriptor consists of four sections called parameters, descriptor, sites and score. The sections must be in that order, although only the descriptor section is required. The parameters section is used to define variables that are used in the rest of the descriptor. It is also used to set `rnamotif` defaults. The descriptor section defines the secondary structure or motif to search for. The sites section allows users to specify simple relations among the elements of the descriptor, and the score section is a set of condition/action pairs that provides direct access to the actual sequences that matched the elements of the descriptor. The score section is used for two things. It tests each sequence that matched the pattern to see that it also matches those elements of the motif that can not be represented in the pattern, and it allows users to evaluate each match and assign it a value (or values) that shows how closely it matched the motif.

The descriptor language is formally specified via `lex` and `yacc`. The lexical or text level is specified as a `lex` input file, which contains a set of regular expressions that define the basic symbols of the descriptor language. These are things like numbers, strings, reserved words, special characters, etc. The scanner generator `lex` converts this file into a function, called a scanner, that converts the stream of characters in the descriptor file into a stream of symbols in the `rnamotif` descriptor language. The syntax of this language is specified as a context free grammar for the parser generator `yacc` which uses it to produce a parser for the language. This parser takes the stream of symbols

produced by the scanner and assembles it into the actual search list and score program that will perform the search.

4.1.1. Preprocessor.

Compilation requires two passes over the descriptor. The first pass is a preprocessor that peforms both file inclusion and interpolation of command line definitions. Unlike the `rnamotif` language itself which is free format, file inclusion is line oriented. File inclusion is indicated via the preprocessor directive

```
# include file-name
```

whose action is to replace this line with the contents of the named file. File includes may be nested. Include files whose names are not absolute paths are assumed to be located in the current directory. Additional include directories can be specified on the command line. Source file name and line number information is saved in the expanded file in order to properly locate both compile and run time errors.

Interpolation of command line definitions is a way to add runtime flexibility to descriptors. Every descriptor may include a parameter section. This section, if present, contains one or more assignment statements of the form *var* `=` *expr*`;` Immediately after parsing the semicolon, the expression, *expr*, is evaluated and stored in the variable *var*. These variables, instead of their values, can then be used elsewhere in the descriptor, removing literals and increasing its readability. This same assignment mechanism is also used to change the default values of the `rnamotif` search algorithm. These defaults, which include the pairing rules and minimum and maximum lengths for each type of helix, are made available to the user as a set of predefined variables. When `rnamotif` is invoked, the user can place one or more variable definitions on the command line. The preprocessor interpolates these definitions into the expanded source immediately before the reserved word `descr` which starts the mandatory descriptor section. The result is that these command line definitions are the last parameter values to be assigned, which means that these new values will replace any previous values, including the default values of the builtin variables. For example, running `rnamotif` with the command line definition `-Dwc+=gu` makes GU pairing the default for all Watson/Crick helices in the current descriptor.

4.1.2. Parsing.

The syntax of the descriptor language is specified by a context free grammar. Such grammars provide a precise and convenient way of describing recursive structures like computer languages. Because nucleic acid secondary structures are also recursive — helices contain other helices which contain still other helices — we suspect that a context free grammar can be written that generates nucleic secondary structures and that it will be easy to integrate this new grammar with the one that describes the traditional computer language part of an `rnamotif` descriptor. Unfortunately, this is not the case, as context free grammars are not powerful enough to generate arbitrary pseudoknots, which are an important class of nucleic acid secondary structures. Nevertheless, the idea of decomposing complex objects into a well defined hierarchy of simple building blocks is a powerful one, and the insights gained from how a context free grammar can generate many important nucleic acid secondary structures are used throughout `rnamotif`.

Consider the following grammar. This grammar is extremely simple, yet it can generate a large number of nucleic acid secondary structures. It consists of one rule with three parts.

```
S       :  S h5 S h3 S
           |  ss
           |
```

This is a rewriting rule, and when used it replaces the generic secondary structure *S* on the left side of the rule with one of the more detailed secondary structures on the right side. The process continues until all of the unspecified secondary structure has been replaced by either Watson/Crick helices, single stranded stretches or nothing. The rule and the process it defines are a recursive definition of nucleic acid secondary structure: A secondary structure is either a Watson/Crick helix (shown as `h5` ... `h3`) both surrounded by and containing additional secondary structure, or a single stranded stretch (shown as `ss`), or nothing at all. In the example below, the rule is used four times to create a simple hairpin. In the first and third applications, the third part of the rule (*S* `->`) is used so that this structure begins and ends with a helix.

```
S       -> S h5 S h3 S
        -> h5 S h3 S
        -> h5 ss h3 S
        -> h3 ss h3
```

There are two ways to apply this rule to create more complex structures. If the second application is to the trailing (or leading) *S*, the result is two consecutive hairpins:

```
S       -> S h5 S h3 S
        -> S h5 S h3 S h5 S h3 S
```

The more interesting case is when the rule is applied to the interior *S*:

```
S       -> S h5 S           h3 S
        -> S h5 S h5 S h3 S h3 S
```

which creates an *interior* loop, (the outer `h5` ... `h3`) containing a hairpin. Repeated application of this rule easily creates more complicated structures. In the next example, the rule is used four times to create a tRNA cloverleaf.

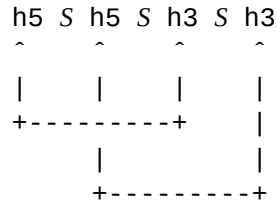
```
S       -> S h5 S                                 h3 S
        -> S h5 S h5 S h3 S                       h3 S
        -> S h5 S h5 S h3 S h5 S h3 S             h3 S
        -> S h5 S h5 S h3 S h5 S h3 S h5 S h3 S h3 S
             ^    ^    ^    ^    ^    ^    ^    ^
             |    |    |    |    |    |    |    |
             |    +----+    +----+    +----+    |
             +-------------------------------------+
```

Finally, this basic rule can be extended to create other types of helices. Here is an extension to create parallel duplexes, triplexes and quadruplexes, although in the case of the 3- and 4- strand helices, the rule does not define the strand order.

```
S       :  S h5 S h3 S
```

```
|  S p5 S p3 S
|  S t1 S t2 S t3 S
|  S q1 S q2 S q3 S q4 S
|  ss
|
```

As powerful as the basic rule is, there is an important class of structures, the pseudoknots, that it can not generate. The basic rule can only generate properly nested structures, ie, structures in which every helix is either completely nested inside of another helix or not nested inside any helix at all. Pseudoknots, however, are composed of improperly nested helices, helices in which no helix is completely contained in any other helix of the pseudoknot. The canonical pseudoknot consists of two helices, whose nesting is shown below. The two helices are clearly connected, but neither is contained inside the other.



That the rule can not generate such improperly nested structures follows immediately from the fact that its application replaces *one* piece of unspecified secondary structure (an *S*) with a *single, complete* helix, which is by definition properly nested. Since the original structure was properly nested, the single secondary structure the rule was applied to was either completely within some helix or not nested at all. And because the action of the rule is located solely in this single secondary structure, the new helix is either completely contained in the original containing helix or also not nested, which results in a new properly nested structure.

Pseudoknots, by their very nature are made of helices that have one end inside of another helix of the pseudoknot and the other end either outside of it entirely, or inside yet another helix of the pseudoknot. A rule generating these types of structures would require action at two *S* regions, one for the 5'-part and one for the 3'-part. Of course, we could extend the basic rule to generate canonical pseudoknots directly, and one such extension is shown below.

```
S       :  S h5 S h3 S
        |  S pk5 S pj5 S pk3 S pj3 S
        |  ss
        |
```

This rule generates two helix pseudoknots (`pk5`/`pk3`, `pj5`/`pj3`), but it cannot generate ones with three (or more) helices. We could add more rules for each class of pseudoknots that `rnamotif` is to support, but this would add complexity, lack elegance and worst of all, would limit the `rnamotif`'s descriptive and search abilities to pseudoknots of some fixed complexity, which is unacceptable.

Fundamentally, the problem with pseudoknots is not generating the correct sequence of `h5`/`h3` symbols, but in correctly linking them. This is because many sequences of these symbols are ambiguous — the various `h5`/`h3` elements can be paired up in several different ways. The symbols below illustrate this.

```
h5 S h5 S h3 S h3
```

If the outermost h5/h3 are paired, the result is an internal/hairpin loop, but if the first h5 is paired with first h3, the result is a pseudoknot. In the grammar under discussion, this ambiguity is resolved by pairing the h5/h3 elements so that an unpaired h3 is paired with the closest unpaired h5; all other pairings are not considered.

The advantage to using a context free grammar to generate nucleic acid secondary structures is that it would embody the structure of even very complicated motifs directly in the sequence of symbols that represent a set of basic motifs or building blocks. No additional linking information would be required. Unfortunately, this is not possible. However, as was mentioned at the beginning of this section, the idea of representing complex objects as a hierarchy of basic motifs is a powerful one and well worth keeping, even if their *hierarchy* can not be represented as a context free grammar.

The building blocks that rnamotif uses to describe arbitrary motifs are the Watson/Crick helix, the single stranded stretch, three other types of proper helix and arbitrary pseudoknots made from Watson/Crick duplexes. The other proper helices are the parallel duplex, a triplex based on Hoogsteen pairs and their analogs with strand order ↑↓↑ and a quadruplex where the strand order is ↑↓↑↓. All of these basic motifs, including the pseudoknots, have two important characteristics. First, they partition the sequence into three types of subsequences: the exterior, which precedes and follows the motif, the sequences involved in the helical strands themselves and one, two, three or more interior regions, depending if the motif is a duplex, triplex, quadruplex or pseudoknot. Their second characteristic is that they are atomic — no helical strand can be removed without destroying the motif. Pseudoknots differ from the other motifs only in that the helical strands on the two sides of an interior region are parts of different Watson/Crick duplexes.

The grammar that describes the rnamotif descriptor language is basically that of a simple awk-like language with only those extensions that are required to demarcate the sections of a descriptor and define the structure elements that are used to build the target motif. Both descriptor sections and structure elements are denoted by new reserved words. Changes to a structure element's default values are made by appending a parameter list to the reserved word for that particular element. Since many of these parameters have values of the same type, parameters are identified by keyword rather than position. For example, the motif fragment shown below contains three structure elements, the first two of which have parameters.

```
h5( minlen=4, maxlen=10, mispair=2 ) ss( seq="^gnra$" ) h3
```

These three elements describe a GNRA-loop with a stem length of 4-10, using the default pairing rule for Watson/Crick duplexes, but allowing up to two mispairs. The grouping of these structure elements into rnamotif's building blocks is done after the entire source has been parsed. This process is discussed in the next section.

4.1.3. Motif Discovery and Analysis.

If no errors are encountered during the parsing stage, the structure elements and associated data are translated into a search program. This process involves several steps. First, the structure elements are linked into helices. Then the parameters for each helix, which may be distributed among the parameter lists of the helix's individual strands, must be collected and checked for consistency. Next, improper helices are dectected and grouped into pseudoknots. At this point, all of the structure elements have been assembled into rnamotif's fundamental motifs and their hierarchical nesting is determined. This hierarchy is used to propagate length limits upward, allowing for a more efficient

search. The motif hierarchy is linearized to form a search list that is use to drive the actual search.

As was shown above, the context-free grammar that defines `rnamotif`'s input language is not powerful enough to directly generate linking information for arbitrarily complex improperly nested structures. Instead, the sequence of structure symbols is saved in the order they were encountered and then linked after the parsing is complete. Improperly nested helices (ie parts of pseudoknots) and all triplexes and quadruplexes must have each helical strand labelled. Structure elements with the same label are grouped and checked that they form a single helix with the correct number elements and that these elements are in the proper order. Then the unlabelled elements are grouped, using the rule that an unpaired `h3` element is paired with the closest (upstream) `h5` element. If any ungrouped helical elements remain after this process, the descriptor is incorrect and the search is aborted.

Next the parameters (if any) for the structure elements are collected and checked for consistency. In the case of helices these parameters may be distributed among parameter lists attached to any of the structure elements that define the strands of the helix. All structure elements may have length, sequence and mismatch constraints; helices may also include parameters that specify the pairing rule, mispair limits (both absolute number of mispairs and/or fraction that must be paired) and whether or not a helix can begin and/or end with a mispair. All explicit length constraints (`minlen`, `maxlen`, `len`) must agree. Implicit length constraints derived from sequence constraints must be consistent with any explicit length constraints. As sequence constraints are specified as simple regular expressions without alternation, this means that fully anchored regular expressions with a fixed length (eg `^gnra$`) must agree exactly with any explicit length constraints. Any regular expression with a variable length must match a sequence whose minimum length does not exceed the explicit maximum length (if set) of this helix (or single stranded region). If two or more helical strands have regular expressions with variable but differing lengths, the implied minimum length of this helix is set to the longest minimum length of these regular expressions. If any errors have been detected in this phase, the search is aborted.

After parameter processing, any improperly nested helices are grouped into pseudoknots. As was discussed above, `rnamotif` considers pseudoknots to be a basic motif, albeit one with a variable number of helical strands. `rnamotif` flattens the set of helices that compose the pseudoknot into an extended structure containing $2N - 1$ interior regions where N is the number of Watson/Crick duplexes in the pseudoknot. This new structure is anchored by the 5'-strand (`h5`) of its leftmost helix. All `rnamotif` pseudoknots must be composed of only Watson/Crick duplexes.

The last step in the motif discovery and analysis phase is to convert the set of fully linked and parameterized fundamental motifs that define the target secondary structure into a search list. The motifs in this set form a tree, and a preorder traversal of this tree is used to form the search list. A postorder traversal of this tree is used to propagate length limits defined for sub-motifs upward to the motif that contains them in order to improve the efficiency of the search. The tree is built recursively. Every fundamental motif is a tree. This tree has one subtree for the substructure(s) contained in each of its interior regions and one subtree for the substructure that follows it. Single stranded regions have at most one subtree, that of their successor. The root of the tree that is built from the target secondary structure is its leftmost motif. The preorder traversal of this tree is then: begin with the root and visit it, then visit each of its interior subtrees in left to right order and then visit the subtree that follows it. The postorder traversal also begins as the root, but first visits a node's interior subtrees, also in left to right order, then its successor subtree and finally the node itself. The following table shows several secondary structures, their associated trees and the search lists that are derived from those trees.

Motif	Tree	Search List
ss Single Stranded. No interior, at most a successor.	ss→	ss
h5 ss h3 Single Watson/Crick helix forming a hairpin.	h5→ ↓ ss→	h5, ss
$h5_1$ ss_1 $h5_2$ ss_2 $h3_2$ ss_3 $h3_1$ Two Watson/Crick helices arranged as an internal loop containing a hairpin.	$h5_1$→ ↓ ss_1→ $h5_2$→ ss_3→ ↓ ss_2→	$h5_1$, ss_1, $h5_2$, ss_2, ss_3
$h5_1$ ss_1 $h3_1$ ss_2 $h5_2$ ss_3 $h3_2$ Two consecutive Watson/Crick helices each forming a hairpin.	$h5_1$→ ss_2→ $h5_2$→ ↓ ↓ ss_1→ ss_3→	$h5_1$, ss_1, ss_2, $h5_2$, ss_3
$h5_1$ ss_1 $h5_2$ ss_2 $h3_1$ ss_3 $h3_2$ Two Watson/Crick helices arranged as a pseudoknot.	$h5_1$→ ↓ $h5_2$→ ss_1→ ss_2→ ss_3→	$h5_1$, $h5_2$, ss_1, ss_2, ss_3

4.2. Searching.

The `rnamotif` pattern finding algorithm takes two inputs: a search list describing the pattern and a string which is to be searched for all instances of that pattern. The algorithm uses a "divide and conquer" strategy: Begin with the first position of this string and look for the first motif on the search list. If this motif is found, it partitions the string into three types of substrings. These are the substrings that make up the motif, zero or more interior substrings between these motif strings (single stranded regions have zero interior regions) and a suffix, the part of the search string which immediately follows the last base of the last motif string. At this point, the search function recursively calls itself to search for the next motif on the search list, using as its search string either an interior substring or the suffix substring (depending on whether this next motif is nested inside the motif just found or follows it).

The algorithm is not complicated but does involve a lot of bookkeeping. It is implemented as a hierarchy of functions called by a driver. The driver takes a sequence from the database being searched and steps through the bases of this sequence in left to right order. At each step it stores the beginning and ending positions of the substring that is to be searched for instances of the motif into the first element of the search list. The beginning position of the search string is the driver's current position. The length of this substring is the smaller of these two values: the length of the longest possible instance of the motif and the remaining length of the sequence. The search terminates when the remaining length of the sequence becomes shorter than the shortest possible instance of the motif.

`rnamotif` does not require the user to specify the maximum length of the target motif, allowing it in principle to be unbounded. Thus it could span the entire sequence, irregardless of the sequence length. However, as helix finding is at least $O(N^2)$, `rnamotif` applies a default upper bound to motifs with no explicit maximum length. This default value is 6000 and is stored in the builtin variable `windowsize`. Like any builtin `rnamotif` variable, `windowsize` can changed

either explicitly in the parameter section or by a command line assignment.

The top level search routine is called `find_motif()`. It takes one argument, the search list. The search list is a list of the basic motifs that the search algorithm can find. Each element of this list contains the type of the motif to find and the beginning and ending sequence positions where that element must be found. Initially, when `find_motif()` is called by the driver, only the positions for the first element will have been set. However, as the search progresses and the various sub-motifs are found, the search positions for the other list elements will be filled in. The motif specifed by the first element of the search list must begin at that element's start position. Its end, however, will in general be somewhere between the first and last positions specified in the element, and `find_motif()` manages the loop that is used to range over this motif's possible ending points.

When `find_motif()` is called, the start and stop positions in the first list element are those of the substring that must hold not only the motif defined by that first element but also any motifs that follow it. Using the analysis performed in the post compilation phase, `find_motif()` adjusts the stop position of this first element so as to leave sufficient substring to hold at least the shortest instance of the motif that follows the first element. The loop to find the end of the first motif thus runs downward from this value to the value that is just long enough to hold the shortest instance of the first motif. At each iteration of the loop, this loop value replaces the original stop value in the first search list element, and the loop value plus one becomes the start value of the substring that must contain the motif that follows the first motif. The updated search list is passed to `find_1_motif()` which calls the individual search functions that actually find `rnamotif`'s basic motifs. These functions and the motifs they find are listed in the table below.

Function	Motif
`find_ss()`	Single stranded region.
`find_wchlx()`	Watson/Crick duplex, with strand directions ↑↓.
`find_phlx()`	Parallel duplex with strand directions ↑↑.
`find_triplex()`	Hoogsteen base triplex with strand directions ↑↓↑.
`find_4plex()`	Quadruplex with strand directions ↑↓↑↓.
`find_pknot()`	Arbitrary pseudoknot formed from Watson/Crick duplexes.

These six functions, while aimed at different motifs, have a similar structure. They all take a single argument, a search list. They all attempt to find the first motif on this list and by the time they are called, both the start and stop position of the substring that may contain the motif have been fixed and the motif must span this substring. While they obviously differ in how they find their target motif, they also differ in what happens when that motif is found. If the motif is helical, the strands of the helix demarcate one or more interior substrings. The sub-motifs nested between these helical strands must be found in these substrings and in order to do this, a helix finding routine sets the start and stop positions of the elements of the search list that correspond to the leftmost motifs of these interior substructures. The helical routine then recursively calls `find_motif()` with next element of the search list to find the remainder of the target motif.

The process is different when a single stranded motif is found. Single standed motifs have no interior regions so no start/stop positions in the search list are updated. If the single stranded motif has a successor, `find_ss()` recursively calls `find_motif()` to find the remainder of the motif. However, if the motif does not have a successor, this means that a candidate has been found that matches all the elements in the descriptor. In this case, `find_ss()` calls routines that check first that the candidate passes any site restrictions and then passes all scoring rules, at which time it accepted and is

printed along with identifying information and score values to `stdout`.

4.2.1. Optimization of the searching algorithm

Starting with version 2.0.0, *rnamotif* uses some heuristics to accelerate the search procedure. These are controlled by the *-O* flag. Basically, the new routine searches for fixed patterns inside the descriptor; if a long enough sequence string is found, it begins the search there, rather than by just starting at the beginning of the descriptor. For descriptors that have significant fixed sequences, this can dramatically improve run times; for other descriptors it may have little or no effect. Results of the program should be independent of the optimization level.

The new option is *-On*, where *n* is the minimumlength of best sequence to search for first. Set *-O0* ("minus captial O zero") to turn off all optimization. The default value for *n* is 2.5. Potential search strings are scored by the following parameters:

```
a,c,g,t         = 1.0
r, y, etc       = 0.5
[acg], etc      = 0.25
n               = 0
```

Thus "gnra" is scored at 2.5. The minimum length is: sequence length − number of mismatches, so that

```
ss( seq="^grna$", mismatch=1 )
```

will not be optimized. Speedup is about 2^n, where *n* is the mismatch-adjusted length.

The sequence to be searched first need not be pinned, but must occur in an element whose length is fixed. Hence

```
ss( seq="grna" )
```

is not optimized because it can start anywhere, but

```
seq( seq="gnra", len=6 )
```

is becuase it can start at most -2 from the gnra.

4.2.2. Imperfect Helices.

Many nucleic acid secondary structures contain helices that are not perfectly paired. These imperfections fall into two classes — mispairs and bulges. `rnamotif` helices may have mispairs, but not bulges. Bulges must be explicitly placed into the descriptor by splitting the helix at each bulge position and inserting a single stranded stretch into the gap between the helices. The minimum and maximum lengths of this single stranded stretch are the minimum and maximum lengths of the bulge; an optional bulge is created by a single stranded stretch with a minimum length of zero. The descriptor fragment shown below illustrates two helices interrupted by a optional one base bulge on their 5'-sides.

```
h5 ss(minlen=0,maxlen=1) h5 ... h3 h3
```

The rationale behind the exclusion of bulges from helices is simple: Helices with bulges, especially optional bulges are complicated; helices without them are not. This complexity appears both in the descriptor and score sections. In the descriptor section, it is needed to specify the positions and lengths of the bulges. In the score section, bulges, especially optional ones, complicate the addressing of the individual bases in the substrings that matched the structure elements of those bulged helices. In helices without bulges, all strands have the same length. If a helix has length L and its strands are antiparallel, the paired bases are numbered 1:L, 2:L−1, ..., L:1; if the strands are parallel, the paired bases are numbered 1:1, 2:2, ..., L:L. In bulged helices, however, the individual strands have different lengths and the numbering of the paired bases depends on the locations of the bulges.

In contrast to bulges, mispairs are simple. Every type of helix supported by `rnamotif` has a default pairing rule which may be overriden for any individual helix. A pairing rule is a set of pairs, triples or quads that, when found at corresponding positions in the strands of a helix, count as a pair (or triple, etc.) Any other combination of bases at these positions is a mispair. Mispair limits other than zero are specified either as a maximum number of mispairs allowed or as the fraction of the helix that must be paired. The latter is useful for those helices that can have a highly variable length. Helices are generally required to begin and end with a pair, but this too is under user control. The distribution of mispairs along a helix can not be specified at the descriptor level, but is easily handled in the score section. Mispairs are optional, although the score section can be used to reject any candidate with less than some number of mispairs.

4.2.3. Imperfect Sequences.

Every structure element may include a sequence constraint. Sequence constraints are specified as simple regular expressions without alternation. Mismatches are permitted as long as the regular expression contains no closure operators. The closure operator, indicated as a star (`*`), is used to specify zero of more instances of the previous regular expression. For example, in the regular expression `A*`, the `*` indicates that this regular expression matches any number (including zero) of `A`'s. The effect of this restriction is that only sequence constraints that have a fixed length can have mismatches. The mismatch limit is specified as the maximum number of mismatches allowed, with zero as the default. Like mispairs, mismatches are optional, but the score section can be used to accept only those candidates with the required number of mismatches.

4.3. The Sites List.

Many features of nucleic acid secondary structures can not be represented as a pattern based descriptor. `rnamotif` provides two mechanisms for incorporation of these features into its descriptors. The simpler and less powerful of these mechanisms is the sites list and is discussed here. Sites provide a way to specify a limited set of fixed sequence relationships between specific positions of the motif. Unlike structure elements, the elements of a site need not have functional (eg helical) relationship. Thus sites provide a way to incorporate longer range "tertiary" constraints into the search.

A site consists of an ordered list of two to four positions and a set of "pairs", "triples" or "quads" that represent the allowed bases as these positions. The quotes are used here to indicate that while these bases are specified using `rnamotif`'s syntax for true hydrogen bonded pairs, no such hydrogen bonding pairing relationship is implied among the positions specified in the site. For example, the site shown below involves a position from each of three different helices, and requires either a `g:a:a` or `g:g:a` at these positions.

```
h5(tag='A',pos=2):h3(tag='B',pos=1):h5(tag='C',pos=$-3) in {"g:a:a","g;g:a"}
```

Each time a candidate sequence matches the pattern based descriptor, the actual bases corresponding to the site's positions are extracted and checked to insure that they are among the "pairs", allowed at those positions. If they are not, the candidate is rejected. If the site is composed of acceptable bases, the candidate is provisionally accepted and the next site is checked. This process continues until a site rejects the candidate or all sites have been matched, in which case the candidate is then scored or (in the absense of a score section) is accepted. Because sites are all or nothing — a candidate either matches or fails — they lack nuance, and because of this, site style constraints along with threshhold values are generally converted into scoring rules.

4.4. Scoring.

In many cases it is impossible to completely describe the motif using patterns only. The motif may have long range sequence covariation or require specific base counts. Even if the pattern completely specifies the motif, the various matches may have different values and it is often necessary to apply some score function to the matches and discard those below some cutoff. The score section solves both these problems.

The score section was modelled after `awk`. It consists of one or more rules, where a rule is composed of an optional condition and an action. If the condition is present, it is evaluated; if true, the action is executed. If the condition is absent, the action is always executed. Like `awk` the rules are executed from top to bottom until 1) an `ACCEPT` statement is executed in which case the candidate is accepted; 2) a `REJECT` statement is executed in which case the candidate is rejected, or 3) all scoring rules have been executed in which case the candidate is accepted. The scoring program communicates its results to the output file via the built-in variable `SCORE`. The details of this process are discussed in a separate section below.

4.4.1. The Score Interpreter.

During the parsing phase, the scoring rules are translated into code for a simple stack machine. This machine has about 35 instructions, the majority of which are used to evaluate expressions. The remainder of the instructions are used for control flow — testing values, branching, looping and accepting or rejecting the candidate. The instructions themselves are untyped, with the actual operation being chosen at runtime based on the types of the operands. Expressions are evaluated on a stack whose entries contain both the type and value of the operands. Variables are stored in the symbol table. However, to speed up execution, the parser replaces references to variables in the score program with pointers to those variables' entries in the symbol table, allowing their values to be fetched or stored without a lookup.

All variables have one of these types: `undefined`, `integer`, `real`, `string` or `pairset`. Variables initially have type `undefined` until they are assigned a value, at which point they take the type of that value. The only type conversions supported are between `integer` and `real`. Variables defined in the parameter section may not be changed in the score section in order to insure that the descriptor is not changed during a search. This applies to the built-in variables as well, except for the special built-in variable `SCORE`, which is used to communicate the results of the score program to the output. Variables defined in the score section are read/write, but once such a variable has been assigned a value and a type, it will be reinitialized to the default value of that type immediately before the score section is executed ensuring that each invocation of the score program is identical.

All variables are scalars, with one exception. This is the special readonly variable `se` which represents the array of structure elements that define the target motif. This array is indexed from 1 to `NSE`, which is the built-in variable that holds the number of structure elements in the descriptor. This array makes it easy to perform tasks such as global counts. For example, the score fragment below

determines the total number of mispairs in the current candidate.

```
nmp = 0;
for( i = 1; i <= NSE; i++ )
      nmp += mispairs( se[ i ] );
```

The primary input to every score program is the set of sequences that matched the structure elements that define the target motif. The score interpreter treats these sequences as a set of readonly string variables which are accessed using the same symbols, with some qualification, that are used to define the structure elements of the motif in the descriptor section. This qualification is required because structure elements in the descriptor element need not be named, resulting in an ambiguity that must be resolved in order to get the correct strings. The ambiguity is resolved by appending to every such structure variable information that contains either its name, or its index. This appendix can contain substring information as well. The example below shows the four ways to refer to substring from the string the matched a structure element. This element is the 5'- strand of a Watson/Crick helix, whose name is `hlx-A`. It is also the fourth element in the descriptor. Strings begin at position 1, and the substring below is 2 bases long and begins at position 3.

```
h5(tag='hlx-a',pos=3,len=2)
h5(index=4',pos=3,len=2)
h5['hlx-a',3,2]
h5[4,3,2]
```

In addition to having a sequence, these structure element variables also have attributes. These are the total number of mispairs in the current match, the total number of mismatches to a sequence constraint, and for any element of type other than single stranded, whether or not two (or more) bases are paired (form a triple, etc). Attributes are accessed by applying a builtin function to the specified structure variable. The number of mispairs to a single strand stretch is defined to be zero as is the number of mismatches if no sequence constraint was specified. The built-in function `paired()` is a generic pairing relationship and works on all helical elements. The elements of `paired()` must all be from the same helical element, but need not be corresponding bases. In the example below, the call to `paired()` extracts the bases at positions `i`, `j` and `k` from the triplex named `Tr1` and tests if they are valid triples using the rule in force for this triplex.

```
mismatches( ss['A-Codon'] )
mispairs( h5['D-Loop'] )
paired( t1['Tr1',i,1], t2['Tr1',j,1], t3['Tr1',k,1] )
```

4.4.2. Nearest-Neighbor Energies.

One of the more useful capabilities of the score interpreter is its ability to evaluate the free energy of either the current match or any part of it. This is done by calling the builtin function `efn()`, which takes two arguments, the beginning and ending positions of the structure whose energy is to be evaluated. It uses the descriptor to convert the selected seqeunce into a base pair and connection table and then applies the Turner energy function to that table.

The fit between the nearest-neighbor (Turner) rules and `rnamotif` structures is not perfect, resulting in a number of clashes. The Turner rules are defined only for those structures composed of

perfectly nested Watson/Crick helices and therefore can not be applied to any structure containing (even a part of) a parallel duplex, triplex or quadruplex. If such a structure is passed to `efn()` the function returns an energy value of `INFINITY`, which following the convention of Zuker and Turner has the value of 160. The perfect nesting restriction also means the Turner rules can not handle pseudoknots, and they too are assigned the same energy value of `ININITY`.

A second type of clash involves mispairs. `rnamotif` considers corresponding bases of helical strands paired if they match the pairing rule in effect for that helical element. The Turner energy function however, divides mispairs into two classes. The two bases involved could form a non-canonical mispair, or they could form an internal 1×1 loop. Theses two cases have different 3-D structures and hence have different energies. `rnamotif` allows users to choose how mispairs are treated. The user can choose to create a base pairing table limited to the standard Watson/Crick pairs (possibly including GU pairs), in which case all other types of mispairs in an `rnamotif` structure element will be treated as loops. Alternatively, the user can tell `rnamotif` to create a base pair table that asserts that all corresponding bases in the two strands of helical element that meet that helix's pairing rule are paired, in which case the thermodynamic values for those bases as mispairs instead of loops will used.

We have implemented two versions of the nearest-neighbor energy rules, called *efn()* and *efn2()*. The *efn()* rules are based on those described by Serra *et al.* [10], implemented in a completely new recursive version based on the traversal of the tree formed by the helical nesting. To test our implementation, several hundred medium size rna's (100-300 nt's) were extracted from Genbank and folded by mfold-3.1. The resulting .ct files were then passed to *efn_drv* (part of the rnamotif distrib) and their energies calculated. When all agreed with the mfold-3.1 energies, I considered *efn()* to be solid and released it. Since its initial release in September, 2000, 3 bugs have been found and fixed: improper tetraloop indexing, improper indexing of 1x2 loops, and and = used instead of == in for stack tracking.

Starting with version 1.7.0, we have also implemented *efn2()*, based on revised rules described by Mathews *et al.* [11]. This was implemented by translating Dave Mathew's *efn2()* code (mfold-3.1/c++/*.cpp) into C. We cleaned up the implementation and removed the intermolcular sections that were not relevant to RNAMotif. We also rebased the arrays at 0 and reorderd all base codes so that RNAMotif's efn() and efn2() use the same ones. We found and fixed three bugs in the original. Two invovlved off by-1 (2) errors in reading some m-way branch loop data and the third treated a string position as a base.

Suitably chastised by early errors in the improper 4-loop indexing, we used a more rigorous approach to testing *efn2()*. We wrote a code that allowed one to take a descriptor and use RNAMotif to find instance of that descriptor, which were then passed to *efn2_drv* and mfold-3.1/bin/efn2. The computed energies were compared and when no differences were found in our test set, it was released. The test set included: simple 4-loops, hairpins with bulges and/or internal loops, and 2-, 3-, 4-, 5- and 6-way junctions with and without bulges and internal loops between the arms. More than 38,000 total structures were examined.

---

10. M.J. Serra, D.H. Turner, and S.M. Freier, "Predicting thermodynamic properties of RNA," Meth. Enzymol. **259,** 243-261 (1995).

11. D.H. Mathews, J. Sabina, M. Zuker, and D.H. Turner, "Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure," J. Mol. Biol. **288,** 911-940 (1999).

4.4.3. Writing the score.

Every program must have a way to communicate its results to the user and `rnamotif` scoring programs are no exception. Fortunately, `rnamotif`'s output requirements are simple. Each time `rnamotif` finds a sequence that matches the entire descriptor — pattern, sites and score, it must write out the details of the match (entry name, match location and the substrings that matched the pattern elements) along with any values computed by the scoring rules. `rnamotif` uses the builtin variable `SCORE` to hold those values. If the result is a single number, then simply assigning it to the `SCORE` will work, but if there are two or more results, some additional mechanism will be required to encode these results in a single variable. The solution chosen by `rnamotif` is based on the traditional `C`/`awk` character formatting function `sprintf()`.

The `rnamotif` builtin function `sprintf()` takes two or more arguments and converts them into a character string. A user wishing to report several score values simply evaluates them into several local score section variables then passes these variables, along with the appropriate format string to `sprintf()` which converts them into a single character string. If this character string is assigned to the special variable `SCORE` its value will be printed along with the match details each time a candidate is accepted.

The first argument to `sprintf()` is the format string which controls the conversion of the second and subsequent arguments into printable form. This format string is composed of a mixture of ordinary character and format specifiers and works like this. When `sprintf()` is called, the format string is scanned from left to right. Each time an ordinary character is encountered, it is simply copied to the output string. However, when a format specifier is encountered it is replaced by the character representation of the argument it references. Which argument is converted and the details of this conversion are determined by the format specifier.

Format descriptors can be quite complex involving not just the value to convert, but also information describing the width, precision and fill characters of the conversion. Complicating matters further, the width and/or precision information can be supplied at runtime from other arguments of the `sprintf()` call. Nonetheless, decomposing a format descriptor into its constituent parts, and supplying proper defaults for those parts that are missing is a straightforward if detailed process. Unfortunately, the conversion of numbers, especially those near the hardware's representation limits is not. For this reason `rnamotif` uses the following implementation of `sprintf()`.

When `sprintf()` is called, the stack is marked and the `sprintf()` parameters are evaluated in left to right order and pushed onto the stack. The format string is thus at the mark pointer and the number of arguments is the difference between the stack pointer and the mark pointer. The `sprintf()` code takes the format string and begins scanning it. Ordinary characters are copied to the end of an initially empty work buffer. When a format specifier is encountered, it is parsed to find its arguments and these arguments are passed to the appropriate C language (`rnamotif` is written in C) `sprintf()` routine which performs the actual conversion. The converted characters are concatenated to the end of the work buffer and the process repeated until the format string is exhausted, at which time, the arguments to `sprint()` are removed from the stack and replaced by the contents of the work buffer. Because, `rnamotif` has only three fundamental data types: `integer`, (C `int`), `real` (C `double`) and `string` (C \0-terminated strings), only those conversions that work on those data types are supported. Even so, `rnamotif`'s `sprintf()` can do a lot more than just print out numbers.

5. `rnamotif` descriptor summary.

This section gives a brief reference guide to the elements of an *rnamotif* descriptor.

`rnamotif` Types		
Type	Examples	Notes
integer	`1 2 300`	
real	`.3 1.3 5. 2e5 3E-3`	Period (.) or `e`/`E` required.
sequence	`"gnra"`	A *DNA* sequence: Upper case → lower case `u` → `t` Letters are IUPAC codes
string	`'h1'`	A literal character string: Characters are exactly as typed
pairset	`{ "g:u", "u:g" }`	The set of allowed "pairs" in a helix: Each sequence must contain 2-4 letters separated colons. Each sequence must contain the same number of letters. Upper case → lower case. `u` → `t`. Letters must be `a`, `c`, `g`, `t`.

`rnamotif` Structural Elements		
Symbol	Structure Type	Notes
`ss`	Single Strand	.
`h5` `h3`	W/C Duplex 5' W/C Duplex 3'	
`p5` `p3`	Parallel Duplex 5' Paralle Duplex 3'	

`t1`
`t2`
`t3`
	Triplex 5'
	Triplex -middle
	Triplex 3'	

`q1`
`q2`
`q3`
`q4`
	Quad-plex 5'
	Quad-plex 3'
	Quad-plex 5'
	Quad-plex 3'	

rnamotif Structural Element Parameters		
Parameter	Type	Notes
tag	string	All
minlen	integer	All
maxlen	integer	All
len	integer	All
seq	sequence	All
mismatch	integer	All
matchfrac	real	All
mispair	integer	All but ss
pairfrac	real	All but ss
ends	string	All but ss
pair	pairset	All but ss

rnamotif Default Pairing Rules		
Helix	Variable	Default Value
Duplex	wc	{ "a:u", "c:g", "g:c", "u:a" }
	gu	{ "g:u", "g:u" }
Triplex	tr	{ "a:u:u" }
Quadruplex	qu	{ "g:g:g:g" }

`rnamotif score` Operator Summary			
Operator	**Prec.**	**Associates**	**Meaning**
`( )`	8	No.	Index a descriptor, change evaluation order
`[ ]`	8	No.	Index a descriptor
`++ --`	7	No.	Increment, decrement
`-` (Unary)	6	?	Negate
`!`	6	?	Not
`:`	6	L to R	Form base pair
`* / %`	5	L to R	Multiplication, division, modulus
`+`	4	L to R	Addition
`-` (Binary)	4	L to R	Subtraction
`== !=`	3	No.	Comparison (Any type)
`< <= >= >`	3	No.	Comparison (Numeric, string only)
`=~ !~`	3	No.	Regular expression match
`in`	3	No.	Pair in pairset
`&&`	2	No.	And
`||`	1	No.	Or
`=`	0	No.	Assignment
`+= -= *= /= %=`	0	No.	Reflexive Assignement

IUPAC Codes			
Letter	Meaning	Letter	Meaning
`a`	`a`	`n`	`acgt`
`b`	`cgt`	`r`	`ag`
`c`	`c`	`s`	`cg`
`d`	`agt`	`t`	`t`
`g`	`g`	`u`	`t`
`h`	`act`	`v`	`acg`
`k`	`gt`	`w`	`at`
`m`	`ac`	`y`	`ct`