NAME

frag – DNA sequence simulator and shotgun data generator.

SYNOPSIS

frag [-s *seed]* [-unpFPN] *specfile*

DESCRIPTION

frag reads from the file *specfile* (a) a specification of a dna sequence in the form of either a stochastic grammar or the designation of a data file, (b) an optional set of 8 parameters describing a shotgun sampling of the generated dna sequence, and (c) an optional set of 4 parameters describing a double-barrelled pairing of the fragments sampled in (b). In formal grammatical terms:

```
<Input> <- <DNA_Sequence> (<Shotgun_Sample> <Double_Pairs> ? ) ?
```

It outputs either the generated dna sequence or the collection of shotgun fragments sampled from it to *stdout,* depending on whether a shotgun sampling is absent or requested, respectively.

The -s option permits one to specify a positive integer with which to seed random number generation so that one can consistently generate the same data set if desired. Otherwise the process id of the particular program invocation is used to seed the random number generator, resulting in a distinct data set with each invocation. The seed used in generating a data set is always reported in the output, so that it may be used to regenerate the same data set with the -s option.

The –u and –n specify the probability distribution used for the read lengths and insert lengths. The –u option specifies a uniform distribution, whereas –n (the default) specifies a normal distribution. In the case of a normal distribution, the min and max lengths specify the a =/- 3 standard deviation range.

The output of frag is exactly standard FASTA format, with the small addition that all lines beginning with '#' are comments. As such it is suitable for input to our UNIFAK fragment assembly suite, and other commonly-used fragment assemblers. Embedded in the comments of the output is a detailed description of the structure of the actual dna sequence generated and, if fragments are generated, a description of exactly how each was obtained from the dna strand. We have found this information invaluable in analyzing and/or debugging the performance of our assembly algorithms. Indeed, if the -p option is specified, then *frag* also reports the strings associated with each variable of the stochastic grammar within the comment lines of the output. For those cases, where no comments are desired, one should set the **-F** option (denoting "FASTA only"). For those cases where only fragment interval comments are desired, set the **–N** option (celsim uses this). The -P option (used by celsim when generating protoIO output) will delete any non actgACTG chars in any sequence read from a file (the assembler can't deal with them). The default behavior is to convert non actgACTG chars to 'n'.

1. Describing a DNA Sequence:

A DNA sequence may be described by (a) a file name or by (b) a special stochastic grammar:

```
<DNA_Sequence> <- <Input.File> | <Stochastic.Grammar>

<Input.File> <- "<" {A UNIX File/Path Name}
```

File Name. In this case, `frag` is assuming a sequence from some other source, like say the DNA databases, is being supplied as the target for simulated fragmentation. The "<" is meant to suggest the piping of the file to `frag` and the file name must be the name of an existing UNIX file. The file itself is assumed to be in FASTA format. If there is more than one sequence on the file, the first complete entry is assumed to be the sequence that should be fragmented.

Stochastic Grammar. In this case, *frag* admits a stochastic grammar describing the subject dna strand. The stochastic grammar takes the form of a sequence of up to 26 element definitions where each element is denoted by a distinct capital letter. A definition consists of the element being defined, followed by an equal-sign or tilde, followed by the definition proper, and terminated with a semi-colon. Characters between a '#' and the end of a line are treated as comments and discarded. The precise grammar is as follows:

```
<Stochastic.Grammar> <- ( <Inline> <FileImport> <Basis> | <Cncat> )+

<Inline> <- <Name> "<" <QuotedSequenceString> ";"
<FileImport> <- <Name> "<" <PathToFile> ";"
<Basis> <- <Name> "[=~]" <Nat.s>(-<Nat.l>) ? <Distribution>? <Instance>* ";"
<Cncat> <- <Name> "[=~]" <Instance>+ ";"

<Instance> <- <Name> <Orientation>? <Variance>? <Repetitions>? <Regenerate>?
              <Fracture>?

<Distribution> <- "p" "("!<Prob.A>", "?<Prob.C>", "?<Prob.G>")"?

<Orientation>  <- "o""("?<Prob.f>")"?  | "r"
<Variance>     <- "m""("?<Fract.min> ","?<Fract.max>)? ")"?
<Repetitions>  <- "n""("?<Nat.min> ","?<Nat.max>)? ")"?
                | "n""("?<Fract.min> ","?<Fract.max>)? ")"?
<Regenerate>   <- "!" ( "("?<Nat.min> ","?<Nat.max>)? ")"?
<Fracture> >   <- "f(" Fract.prefix "," Fract.suffix  "," Fract.random  ","
                       Fract.minLength  "," Fract.maxLength  )? )?)? )?  ")"
```

Definitions. There are three types of definitions: concatenation, basis, or constant. A basis definition specifies an element as a base sequence of some length and symbol distribution that may have embedded in it some number of smaller, previously-defined element instances. A concatenation definition specifies an element as the concatenation of some number of previously-defined elements. In both cases the definitions end with a list of "instances". For concatenations the instances are concatenated together, and for basis definitions the instances are randomly inserted into the basis sequence. For a basis definition the list of instances is optional. Constant definitions (<Inline> and <FileImport> above) define constant sequences. <Inline> definitions appear as quoted strings within the input, and <FileImport> definitions read a definition, in FASTA format, from a file. Constant definitions are of fixed length. When definitions are imported from a file, all characters that are not from [actgACTG] are stripped if the -P option is specified, and converted to 'n' otherwise.

Basis Definitions. A single integer or a pair of integers separated by a dash specifies the length range of a basis sequence, and this may optionally be followed by a distribution term which gives the probabilities with which nucleotides are selected during the generation of the basis sequence. If the distribution term is missing then each nucleotide is picked with equal probability. For example, `'A = 12;'`, defines `A` to be a sequence of 12 nucleotides and is identical to the definition, `'A = 12 p (.25, .25,. 25);'`, which explicitly asserts that each nucleotide is chosen with equal probability. The definition, `'B = 10-20 p (.4, .1, .1);'`, specifies a sequence whose length will be chosen uniformly between 10 and 20, and that on average will be 80% `at`-rich. Note that basis definitions without embedded elements are the only terminal definitions, i.e., ones that do not refer to other elements. It is important to note that even if a basis element does contain embedded elements, its length is within the specified range. That is, embedded elements are considered to contribute to the length of the basis, and only the nucleotides not in an embedded element are chosen according to the distribution qualifier. For example, if a basis element is to be of length 1000 and contain two embedded elements, each of length 200, then 600 nucleotides are generated according to whatever distribution is in effect and the two embedded elements are uniformly inserted into this shorter sequence to produce a 1000 nucleotide sequence.

Instances. Each instance or reference to an element on the right-hand side of a definition, may be qualified by a sequence of optional terms as indicated by the grammatical production for `<Instance>` above. An unqualified reference simply refers to the (unaltered) sequence of the element. For example, `'C = A B A;'`, defines `C` as the concatenation of a copy of `A` followed by a copy of `B` followed by another copy of `A`. The orientation, mutation, repetition, and regeneration qualifiers, stochastically modulate which orientation of an element's sequence is used, how much that element is to be randomly mutated, how many copies are inserted or concatenated (depending on whether the definition is a basis or

concatenation type), and how many times (if any) the element is regenerated from its definition. It should be clear after the following discussion that the unqualified reference, 'A',is the same as the qualified reference, `'A o(1.) m(0.) n(1)'`. For the moment, we will defer treating the regeneration qualifier and the difference between '=' and '~' definitions, until a final discussion on controlling the generation/regeneration of the defined stochastic elements. For now we proceed with a discussion of just the *o,m,n,x* qualifier sequence.

Orientation. The orientation term specifies the probability with which the element is utilized as opposed to its complementary strand. For example, qualified with `'o(.5)'` an element will be inserted as is, or complemented, half of the time. Unqualified an element will always be inserted as is, i.e., 'o(1.)'. For convenience, the qualifier 'r' is short-hand for 'o(0.)',i.e., always utilize the complementary strand. The parentheses are optional, i.e., one might write 'o.5' or 'o .5' if desired. In general, for all the qualifiers the punctuation is optional, but commas must be replaced with at least one white space in order to separate the numeric arguments of the qualifier.

Mutation. The presence of a mutation term indicates that each copy of the qualified element is to be mutated by the amount specified. We speak of copies in the plural because the repetition qualifier to be described next may require that several instances of the element be utilized. For example, with `'m(.02)'` each copy of the element is mutated by exactly 2%. It is important to appreciate that an exact number of point mutations will be inserted. The mutation rate is multiplied by the length of the element and the result is rounded up or down to the nearest integer with a probability that would give a sequence of integers whose average is the fractional number (e.g. 2.25 would be rounded to 2 with probability .75 and to 3 with probability .25). Given the integer number, exactly this many difference are introduced with uniform probability across the sequence with one-third of the mutations being substitutions, one-third being insertions, and one-third being deletions. If a range of mutation rates is given, as in `'m .02 .05'`, then for each copy, a rate is chosen uniformly from this range, and then applied to the copy.

Repetition. A repetition qualifier requests that some number of copies of the qualified element be utilized. For example, `'n (3)'`, concatenates three copies in the case of a concatenation definition, or inserts three copies in the case of a basis definition. If the element is also qualified by an orientation , mutation, or fracture term, then these qualifiers are reapplied to each copy. For example, `'A o(.2) m(.001, .005) n(100)'`, produces 100 copies of `A` about 80% of which will be complemented and where each is mutated between .1 and .5% with the median mutation rate being .3%. If one specifies a range of repetition, as in `'n(5, 10)'`, then the number of repetitions used is chosen uniformly between the two extremes. The minimum number of repetitions may be 0 but if the instance is part of a concatenation definition then at least one instance in the definition must have a non-zero minimum repetition limit.

In addition, the repetition qualify, if used in a basis definition, can instead specify how many copies of an element should be inserted as a *fraction* of the length of the basis sequence. For example, `'A n.2'` requested that 20% of the basis be composed of the element `A`. Just as for the mutation qualifier, if the length of `A` does not evenly divide 20% of the basis length, then the number of copies is rounded up or down to the nearest integer with a probability that over a series of trials converges to 20%.

Fracture. A fraction qualifier requests that only a substring of each copy of the qualified element be utilized. This serves to model recombinant effects in ALUs and other repeated elements. These “fractures” come in three varieties: prefix, suffix and random. Prefix fractures are taken from the start of the element, suffix from the end, and random from some random position in the element. The length of the fractures can also be specified as a fraction of the element’s length. The fracture qualifier can specify one or more of the following arguments:

Fract.prefix: the fraction of the elements that are prefix fractures
Fract.suffix: the fraction of the elements that are suffix fractures (default 0.0)
Fract.random: the fraction of the elements that are random internal fractures (default 0.0)
Fract.minLength: the minimum length of the fractured elements, as a fraction of element length (default 0.0)
Fract.maxLength: the maximum length of the fractured elements, as a fraction of element length (default 1.0)

For example, `'A o(.2) m(.001, .005) n(100) f(0.01, 0.01, 0.10, 0.1, 0.5)'`, produces 100 copies of `A` about 80% of which will be complemented and where each is mutated between .1 and .5% with the median

mutation rate being .3%. Each copy is prefix fractured 1% of the time, suffix fractured 1% of the time, and randomly fractured 10% of the time. When fractured, the length of the fractures is uniformly distributed between 10-50% of the length of A prior to mutation.

A Complete Example: A complex target dna sequence can be built up of a series of definitions subject to the restriction that any element referred to in the right-hand side of a definition must already be defined. Thus, recursive definitions are not possible, the grammar is strictly finite and hierarchical. The last element defined is assumed to be the subject strand from which shotgun fragments will be extracted. As a complete example, consider:

```
A = 150;
B = A Ar;
C = 4-6 p .35 .15 .15;
D = C o.5 m(.1,.2) n(30,100);
S = 50000 Bm(.08,.16)n(.1) f(0.01, 0.01, 0.05, 0.3) Dn(0,3);
```

Element `B` is defined to be a palindromic sequence of length 300. Element `D` is a tandem repeat of 30 to 100 copies of a 4-6 nucleotide, `at`-rich sequence `C`, where each copy is perturbed by 10-20% and may be oriented in either direction with equal probability. The final subject strand `S` is a 50,000 nucleotide sequence containing up to 3 copies of the tandem repeat `D` and 10% of which will be the palindromic element `B` each copy varying by 8-16%.

Regeneration: In the example above, note that once an element such as A has been constructed according to its stochastic definition, any reference to it in later instances, are assumed to be references to the given sequence. If such were not the case then B would not be a copy of a random 150 base pair sequence followed by its complement, but would be, effectively, a random 300 base pair sequence. But this is not always the effect one desires. For example, the definitions of

C and D together describe what a biologist might think of as an *at*-rich microsatellite repeat. In the example, one such repeat is generated and assigned to D. Thereafter, zero to three copies of this identical sequence are placed in the generated dna sample. What one might prefer, is that zero to three distinct microsatellite sequences, having the same structure, but different instantiations, be incorporated into the result.

To achieve this affect, one can use the '!' or regeneration qualifier. This qualifier has the effect of regenerating another instance of the qualified element according to its definition and applying the previous qualifiers to each of the requested regenerations. Thus in the example above, replacing the last definition with:

```
S = 50000 Bm(.08,.16)n(.1) D!(0,3);
```

results in a dna sample that has up to three distinct micro-satellites in it, each of which has the structure of 30-100 tandem repeats of a 4-6 *at*-rich segment.

Continuing with our example, suppose that we also wish to have two different instances of sequences with the palindromic structure given by B's definition. We achieve this as follows:

```
S = 50000 Bm( .08,.16)n(.05)!(2) D!(0,3);
```

In this case, 2 instances of B are generated, and S contains 5% of each type, where the copies of each are mutated by 8 to 16%. Note that not only is B regenerated, but recursively, anything comprising it is also regenerated in creating each new instance. For example, if A's definition were not reevaluated when a new instance of B is requested, then all the instances of B would be the same. Finally, note that writing '!' is the same as writing '!(1)', and that this is not the same as omitting the option, as the latter does not cause a regeneration whereas the former does. For example,

```
A = 4;
S = A! A A A A!;
```

Results in a string of length 20, where a given 4-tuple constitutes the 3 interior 4-tuples, and the outer ones are random with respect to each other and the three interior ones.

Static Definitions: Let's reconsider the microsatellite definition D in our example. Once we changed its reference in S's definition to 'D!(0,3)', we got up to three distinct microsatellites generated, each one with a different 4-6 tuple arranged in tandem and slightly mutated. This happens because each requested regeneration of D results in its definition being re-evaluated, which in turn causes C to be re-evaluated. Suppose that this is not the desired effect, but rather we want up to three micro-satellites involving different numbers of units, orientations, and mutations, but with all three involving the *same* satellite 4-6 tuple (this does happen in real sequences). To effect this one must stop C from being regenerated each time D is. To do so, replace the '='-sign of C's definition with a '~'. Such a definition is said to be *static* as it is never re-evaluated even if elements referring to it are. Naturally, constant definitions are static. So let's conclude with a final grammar example:

```
A < repeat.fasta;
B = A Ar;
C ~ 4-6 p .35 .15 .15;
D = C o.5 m(.1,.2) n(30,100);
S = 50000 Bm(.08,.16)n(.05)!(2) Dn(0,3);
```

which generates a dna sample that is 50,000 bases long, contains 5% each of two types of 300 base pair palindromic elements, and up to three instance of micro-satellite repeats involving 30-100 repeats of the same 4-6 nucleotide, *at*-rich subunit. Note that the definition of element A is read from the file repeat.fasta.

2. Shotgun Sampling Fragments:

If one terminates the input after the stochastic grammar or the simple four parameter variant, then *frag* outputs the sample

target sequence as its output string in FASTA format. If one otherwise continues with the following 8 parameters that specify a shotgun sampling of the target sample, then *frag* outputs the reads of the shotgun sampling in FASTA format.

The syntax of a shotgun sampling specification is as follows:

```
<Shotgun~Sample> <- <Nat.frag> <Nat.minlen> <Nat.maxlen> <Prob.forward>
                    <Prob.brate> <Prob.erate> <Prob.insert> <Prob.delete>
```

The first four numbers pertain to the sampling of fragments from the dna strand. The number of fragments to be sampled, `Nat.frag`, the minimum length possible for any fragment, `Nat.minlen`, and the maximum length possible for any fragment (+/- 3 standard deviations if gaussian distribution is employed), `Nat.maxlen`, are all specified as positive integers, in that order. A fragment's length is chosen uniformly over the minimum to maximum length range (see above comments on gaussian distributions). Each fragment is chosen uniformly over all possible fragments of that length that could be sampled from the subject dna strand. The final parameter, Prob. forward is the probability (a real value between 0 and 1) with which the fragment is chosen from the subject strand versus its complement strand (in which the direction is reversed as well).

The last four numbers are concerned with the introduction of errors into fragments and all four are expressed as probabilities (real values between 0 and 1). An error rate ramp over the length of a fragment is specified by giving the error rate at the start of the fragment, `Prob.brate` and the error rate at the end of the fragment, `Prob.erate`. The rate at which errors are introduced at intermediate points along the fragment is determined by interpolating the end point rates, i.e., the probability of error at position *i* in the fragment is *brate + (i/fragment_length)×(erate - brate).* Errors are generated in a Poisson fashion according to the current error rate. Thus the number of errors in a given strand has a binomial distribution about the expected mean. The last two numbers specify the probabilities that an error is an insertion, `Prob.insert`, or deletion, `Prob.insert`, of a character. The probability that an error is a substitution is 1 minus the other two probabilities. In the case of a substitution error one of the other three nucleotides is chosen with equal probability, and if a character is inserted, each nucleotide is also chosen with equal probability.

3. Double-Barrelled Shotgun Sampling:

If one has given a shotgun sampling specification, then one may further follow it with a four parameter specification of a dual-end sampling strategy. If so, then the output consist of a collection of reads, some of which are designated as being paired by virtue of having the same number followed by an f and r suffix, respectively (see later examples for further clarification). The specification is grammatically as follows

```
<Double_Pairs> <- <Prob.single> <Nat.minins> <Nat.maxins> <Prob.prate>
```

The first number, `Prob.single`, is the probability with which fragments are not paired. For example, sampling 100 fragments and asserting that .1 will be unpaired results in an average of 10 unpaired reads in the data set. All other fragments are sampled in pairs from the end of a hypothetical insert. The next two positive integers, `Nat.minins` and `Nat.maxins`, specify the length range of the insert from which the end reads are obtained. For each pair, the insert length is chosen uniformly from this range. Finally, the last number, `Prob.prate`, is the probability that a given pairing is uncorrelated, i.e., that the paired reads actually come from arbitrary places within the target strand. Note that in such a case, they may, by chance, be in the right orientation and distance apart. This is important as one may later wonder why there are fewer false pairs reported by a piece of software than were actually requested/generated by *frag*.

4. Output:

frag produces a single dna sequence or a collection of sampled fragments of a dna sequence in a FASTA-like format on *stdout.* Embedded within the comments fields of the output *frag* also writes a description of the definitions, parameters, and structure of the dna sequence it sampled from. The output file is made up of fragment sequences, fragment name lines, and comment lines. Each fragment line has an '='-sign as its first character and each comment line has a '>'-sign as its first character. All lines beginning with other characters are assumed to be sequence lines. Each name line is followed by a number of sequence lines giving the sequence for the fragment whose number is given by an integer following the '='-sign on the name line. Comment lines can be interspersed as desired save between lines containing the sequence of a given fragment. For example, in response to the simple input:

```
S = 100;
5 30 40 0.5 0.2 0.4 0.33 0.33
25 50 60 .25
```

frag produces the following output:

```
#
# Seed = 1336
#
# Element: S
#    Length = [100,100], ACGT odds = 0.25/0.25/0.25/0.25
# Fragments:
#    Number = 5, Length Range = [30,40], F/R odds = 0.50/0.50
# Edit Characteristics:
#    Error Ramp = 0.20->0.40, Ins/Del/Sub Odds = 0.33/0.33/0.34
# Dual-End Inserts:
#    Single Odds = 0.25
#    Insert Range = [50,60]
#    Pairing Error Rate = 0.25
#
#  S.0 (100)
#
# Generated Fragment a:
#
# Not Related to 1r!
#
# Interval: [50,82] REVERSED
#
#  at-cggtc-t-ttgt-attgca-ataagagaactgta
#    t  ---c g    t    a t    -        -
#
> 1f
acagtttcttatattcaataacaacagcgaat
#
# Not Related to 1f!
#
# Interval: [63,95] REVERSED
#
#  tt-gcaataagagaactgtagggaagaatagag-
#    t      g   g  a c  -           t
#
>     lr
actctattcttcctgctgtcctcctattgcaaa
#
# Interval: [48,86] REVERSED
#
#  ctatcggtctttgtat-tgcaataagagaactgtaggga
#   - -   c    -   g
#
> 3
tcctacagttctcttattgcacataaaaggccgtg
#
# Interval: [70,100] REVERSED
#  aagagaactgta-g--ggaagaatagagagcg-a
#      t    a- t-ac    - g       --t
```

```
#
>     4f
tactctctactttccgtattagttatctt
```

```
#
# Interval: [44,79]
#
#  aggtctat-cggtctttgtat-tgcaat-aagagaact
#       c    tg       --  c  t c g      c
#
> 4r
aggcctattgggtctttatctgtactgaagagacct
```

In the example above, note that the initial comments record the values of the input parameters that produced the output, including the number used to seed the random number generator. Further note that the comments immediately preceding each fragment sequence contain a record of where the fragment came from in the subject strand, whether or not it has been reversed, and a record of the errors that were introduced into it. For example, for the last fragment, `4r`, the fourth nucleotide was misread as a `c`, a `t` was inserted after the eighth nucleotide, the ninth nucleotide was misread as a `g`, and so on. Finally, note that the names of the paired fragment are suffixed with `f` and `r`, and that uncorrelated pairs are further noted in the comments.

In the case of an input involving a stochastic grammar, the initial comments record the element definitions and also the way in which elements were instantiated. For example, in response to the input:

```
A = 150;
B = A Ar;
C      ~ 4-6 p .35 .15 .15;
D = C o.5 m(.l,.2) n(3,l0);
S = 50000 Bm(.08,.16)n(.015)!(2) Dn(0,3);
```

a run of *frag* produced the following initial set of comments for the header:

```
#
# Seed = 544
#
# Element: A
#    Length = [150,150), ACGT odds 0.25/0.25/0.25/0.25
# Element: B
#    Concatenation of:
#     A: O'odds = 1.00, Mut's = 0.00-0.00, 1-1 Rep's, 0-0 Gen's
#     A: O'odds = 0.00, Mut's = 0.00-0.00, 1-1 Rep's, 0-0 Gen's
# Element: C (static)
#    Length = [4,6], ACGT odds = 0.35/0.15/0.15/0.35
# Element: D
#    Concatenation of:
#     C: O'odds = 0.50, Mut's = 0.10-0.20, 5-5 Rep's, 0-0 Gen's
# Element: S
#    Length = [50000,50000], ACGT odds = 0.25/0.25/0.25/0.25
#    Containing:
#     B: O'odds = 1.00, Mut's = 0.08-0.16, 0.015-0.015 % of Basis, 2-2 Gen's
#     D: O'odds = 1.00, Mut's = 0.00-0.00, 1-1 Rep's, 2-2 Gen's
#
#  S.0 (50000)
#  > D.4f at 4445-4465, mutated 0.00.
#    = C.Of at 4445-4449, mutated 0.18.
#    = c.0r at 4449-4453, mutated 0.14.
#    = c.0r at 4453-4457, mutated 0.17.
#    = c.0r at 4457-4461, mutated 0.16.
#    = c.Of at 4461-4465, mutated 0.13.
#  > B.2f at 6394-6694, mutated 0.09.
#    = A.2f at 6394-6544, mutated 0.00.
```

```
#    = A.2r at 6544-6694, mutated 0.00.
#  > B.lf at 18672-18973, mutated 0.12.
#    = A.lf at 18672-18822, mutated 0.00.
#    = A.lr at 18822-18972, mutated 0.00.
#  > B.lf at 25270-25571, mutated 0.12.
#    = A.lf at 25270-25420, mutated 0.00.
#    = A.lr at 25420-25570, mutated 0.00.
```

```
#  > B.2f at 26427-26726, mutated 0.09.
#=   A.2f at 26427-26577, mutated 0.00.
#=   A.2r at 26577-26727, mutated 0.00.
#    > 0.3f at 30584-30604, mutated 0.00.
#      =     C.Or at 30584-30588, mutated 0.12.
#        =   C.Of at 30588-30592, mutated 0.14.
#      =     C.Of at 30592-30596, mutated 0.15.
#        =   c.Or at 30596-30600, mutated 0.15.
#      =     C.Or at 30600-30604, mutated 0.13.
#
# DNA Sequence:
#
>0
tacttaagcttgctctcgtctttaacatgttgcgttgatacggacaaggt
agctgttcatgccgcacttcactcgagttsetegttacaettcctgccga
.....
```

Observe that the header describes each element as being the concatenation of some instances, or as containing some instances depending on whether it is a concatenation or basis definition, respectively. The nature of the basis and the qualification of each reference are also given. After having read the input, *frag* then generates the subject sequence and outputs additional comments describing the position of subelements within a given element so that its repeat structure is described. For example, two instances `B.1` and `B.2` of the palindromic element `B` are created and two or three copies of each are spread throughout `S`. For example, one of the copies of `B.2` was mutated 9% and then placed in the forward direction into positions 6394 to 6694 of `S`.

DIAGNOSTICS

Exit status is 0 is everything works normally, 1, if there is an error in the input, or not enough of it, and 2, if there is not enough memory.

AUTHORS

Gene Myers

`Created October 16, '98`

`Revised 1/5/99 SAK  Added ability to define constant elements using the < operator.`

`Revised 3/15/99 SAK  Imported data has non actgACTG chars stripped.`