Chio:
A String Processing Library
for Common Lisp

Jonathan E. Spingarn

September 20, 2003

spingarn@toiling-in-obscurity.net

http://www.toiling-in-obscurity.net/chio/

Chio: A String Processing Library for Common Lisp

Please send comments and suggestions to the author at

spingarn@toiling-in-obscurity.net,

For up-to-date information about Chio, the latest iteration of this document, and information about the software, see the Chio web site.

Table of Contents

Introduction

Chio is a string processing package for Common Lisp. Its three main macros -- `with-test-binds`, `with-test-format`, and `with-test-split` — perform matching, substitution, and splitting operations on strings. As an extension of Lisp, Chio grants the user full access to the Lisp programming environment at all times, thus offering flexibility and power that make it ideal for complex or highly algorithmic searches that might be cumbersome in more rigid languages. At the same time, it is simple and intuitive for routine tasks. (The "ch" in "Chio" is pronounced softly, as in `char`.)

The atomic unit of a Chio search is a *simple-test*. Postponing its definition, we mention now only that a simple-test may be compiled from a regular expression, but can also result from evaluating any suitable Lisp expression. The possibilities for what a simple-test might do are therefore not limited by the particular regular-expression syntax that Chio happens to provide. In addition to many ready-made simple-tests, and tools for creating new ones, Chio provides the `simple-test-reader`, a reader macro that is able to manufacture simple-tests from regular expressions. For example, the simple-test-reader reads `#~"a+b+"` as code that returns a simple-test that searches, as one would expect, for a run of `a`'s followed by a run of `b`'s.

Chio does not have any regexp "engine" to process regular expressions in the manner of a black box. Indeed, the moment the `simple-test-reader` reads it, a regular expression ceases to exist. The use of this reader macro gives Chio some attractive features. For instance, by examining the Lisp reader's code expansion of a regexp, a user receives immediate feedback as to what the regexp does. This feature is useful not only for debugging, but also for its instructional role — if you want to write a simple-test, it is sometimes helpful to see the code the simple-test-reader writes to handle a similar task. The `simple-test-reader` can be regarded as an independent module of the Chio package.

When we say that simple-tests are "atomic" we mean that you cannot

look inside one to see its parts. For example, Chio does not grant access to the portion of a string that matches the `"b+"` portion of an `#~"a+b+"` match. Surrounding that part with parentheses, `#~"a+(b+)"`, is legal but would not (as it would, for instance, in Perl) grant any kind of access to that portion of the match.

To gain such access, you need to move up to the next level of complexity, the *binding-tree*. The leaves of a binding-tree are simple-tests, and its internal nodes are keyword symbols, called *operation-keywords,* that represent operations combining simple-tests into compound searches. For instance, the binding-tree `(:&0 #~"a+" (:&1 #~"b+"))` has two internal nodes `(:&0` and `:&1)` and represents concatenation of `#~"a+"` with `#~"b+"`. Chio can grant memory access to the match results corresponding to an internal node of a binding-tree. In this example, `:&0` grants access to the entire `"a+b+"` match, and `:&1` grants access to the `"b+"` portion of the match. When the symbol-name of the operation-keyword has length greater than one (such as `:&0`), access is granted for future access. However, when the symbol-name has length one (such as `:&`), the results are not retained. This is the distinction between *basic-operation-keywords* (like `:&` and `:o`) and *remembering-operation-keywords* (like `:&1` and `:ox`).

Binding-trees are compiled into *compiled-binding-trees*. Since the user has control over the compilation, it can be kept outside of iterated code, where it would waste resources. A `NIL` leaf on the binding-tree can serve as a place holder for a simple-test that is unavailable at the time of compilation, and there is an efficient mechanism for filling `NIL` slots in such an *incomplete binding-tree* at a later time.

Chio supports a rich regular expression syntax. In addition to the familiar quantifiers `*, +,` and `?,` Chio assigns special roles to `!, <,` and `>` when they occur in certain contexts. The `!` causes a simple-test to return only the first match it finds. This can speed execution, especially when a test fails, and also clarify a programmer's intentions. It is a destructive operation in the sense that it causes certain valid matches to be overlooked. The `<` and `>` symbols cause search results to be returned in a

sorted ascending or descending order. These six basic quantifiers correspond to the six Chio functions `T*, T+, T?, T!, T<,` and `T>` which take a simple-test as a single argument and return an appropriately modified new simple-test. Quantifiers can be composed by writing them sequentially. Many useful such combinations are handled by specially tailored routines to enhance efficiency. For example, the `simple-test-reader` handles `+!` via a call to the `T+!` routine, rather than by composing `T!` with `T+.` Efficiency is further enhanced by handling character classes and their quantifiers by specially optimized routines. For example, `#~"[a\d]+!",` which searches for a longest-only run of `a`'s and digits, expands into a call to `chtes+!` and does not use `T+!`

The three major macros include among their arguments a binding-tree (either uncompiled or compiled), a target string (to be bound to the global variable `*STRING*`) to be searched, and several keyword arguments that permit the user to choose among options and return values in numerous ways.

Each major macro operates in one of two possible *modes*: `loop` mode and `once-only` mode. In `once-only` mode, the body of the macro is executed just one time, while in `loop` mode the body is enclosed in a `(LOOP …)` to iteratively process additional matches.

The first argument to each major macro is `PREFIX,` a symbol. `PREFIX` provides the search with a name that permits access to search results. In addition, the symbol-name of `PREFIX` serves as a prefix for the symbol-names of several variables to be bound to match results. For example, in the body of a call that begins `(WITH-TEST-BINDS AA ...,` the symbol `AA-COUNT` is bound to an iteration count, and the symbols `AA-MRS` and `AA-MRE` point to the start and end of the match results in Chio's workspace `*STACK*.` `PREFIX` also serves to name an enclosing block in the case of the `with-test-binds` and `with-test-split` macros, thereby providing an alternative means to exit from the body of the macro by using `(RETURN-FROM PREFIX &OPTIONAL VALUE).`

The user accesses match results via `MREF.` For example, the substring

that matches the `"b+"` portion of the binding-tree `(:&0 #~"a+" (:&1 #~"b+"))` would be accessed (assuming `PREFIX=AA`) as `(MREF AA 1 :S)`. Here, the number `"1"` specifies the second remembering internal node of the binding-tree, and the argument `:S` instructs `MREF` that that substring be read *as a string* (as opposed to `:I` which would read it as an integer, or `:R` which would read it as the Lisp reader would read it, among several other choices). An additional feature provides unlimited flexibility to `MREF:` a function name or lambda expression in the place of the keyword can specify customized instructions to read a match in any manner one might desire. For example,

```
(MREF AA 1 (lambda (start end) (- end start)))
```

returns the length of the match and

```
(MREF AA 1 (lambda (start end)
             (nreverse (subseq *string* start end))))
```

returns the match backwards as a string.

When major macros are nested, the naming of each one eliminates any reference ambiguity. For instance, you could have

```
(with-test-binds AA ...
  (with-test-binds BB ...
     ...(MREF BB 0 :S)...
     ...(MREF AA 7 :R)...))
```

and it would be clear that the first `MREF` refers to the inner call and the second to the outer one.

In addition to `MREF,` two other access macros are provided. Neither does anything that `MREF` can't already do, but they are convenient nonetheless. `EMPTY-MATCH-P` returns the position of a match of length zero or `NIL` if the match has positive length, and `MWRITE` writes a match to a stream without first copying it.

Simple-Tests

Fundamental concepts: simple-test and tret

Chio uses the notion of *simple-test* to encapsulate a search task such as might be described by a regular expression. A simple-test is a function of three variables, `*STRING*, START,` and `*END*.` However, only `START` is passed as an argument to the simple-test, the other two variables being global. A form `(FUNCALL SIMPLE-TEST START)` returns a *thunk* (a function of zero arguments) which we will call a *tret* (pronounced "tray" - a function that a Test RETurns). `*STRING*` is bound to the string currently being searched. Each substring of `*STRING*` that satisfies the test (each *match*) is required to have its starting index exactly equal to `START` and must end at or before `*END*.` Each time the tret is called, it returns one possible index for the end of the match, until the tret finally returns `NIL` to indicate that it is exhausted. After the tret returns `NIL` once, *it must never be called again*. (Failure to observe this convention may cause stack space belonging to other tests to be overwritten, as well as other problems). Some simple-tests return trets that return *all* possible endings for a match, whereas others return only some subset of the possible endings; the documentation for a simple-test (or for any function that creates simple-tests) should state clearly how its trets behave.

For example, if the goal of a simple-test `V` is to search for a non-empty run of consecutive vowels, `*STRING*="XYAEIZE"` and `*END*=6,` then `(funcall V 2)` is a tret that might return `5,4,3,` and finally `NIL.` Or, perhaps it might return the endings in the opposite order: `3,4,5, NIL.` Or, perhaps it might just return `5;` it all depends on the intentions of the programmer who designs `V.` The definition of "simple-test" is very permissive; it does not require that the tret return a sequence of *distinct* integers, nor does it even require that this sequence have finite length (though that is certainly to be desired).

The simple-test is only the most elementary buildling block for a Chio search. Simple-tests can be assembled into *binding-trees* to construct more complicated searches that grant access to particular substrings of the match. Algebraic operations can be performed that act on simple-tests to produce new simple-tests. These ideas

will be discussed further in the sections on *binding-trees* and *algebraic operations on simple-tests*.

The simple-test-reader

The *simple-test-reader,* defined in the file `simple-test-reader.lisp,` is a reader macro that processes regular expressions. It reads an expression of the form `#~"..."` as a Lisp expression that evaluates to a simple-test. The test can be made case-insensitive by inserting the letter `"i"` after the `"~".` For example, `#~i"\b[xy]{2,}\d*"` is a case-insensitive search for two or more occurences of `"x"` or `"y",` starting at a word-boundary, and followed by zero or more digits. The `simple-test-reader` reads this regular expression as

```
(SCAT WORD-BOUNDARY
  (CHTES-N "xy" NIL NIL NIL 2 :INFINITY NIL T)
  (CHTES+ NIL NIL (DIGIT-CHAR-P) NIL :STAR T))
```

Here, `SCAT` builds a simple-test by concatenating three simple-tests. The first one is the zero-width test `WORD-BOUNDARY.` The second tests for two or more occurrences of the letters `"x"` or `"y"` in a case-insensitive manner, and the third tests for zero or more digits. (See the documentation strings for `SCAT, CHTES-N,` and `CHTES+` for more information.)

Chio's regular expression syntax will be discussed in more detail in a section devoted to that topic.

Keyboard shortcuts

With version 1.0 of Chio, the following keyboard shortcuts are available when running Macintosh Common Lisp. If you place the cursor in front of a regular expression such as `#~"[aeiou]+"` and type `c-x c-r,` you can view the code produced by the `simple-test-reader.` In many cases, the code that you see will be a macro call, so if you type `c-m,` you will observe the macroexpansion which is the actual code for the simple test. To easily type a regular expression like `#~"[aeiou]+",` you can either

(1) Type `meta-~` and then type `[aeiou]+,` or

(2) Select `[aeiou]+` and then type `meta-~`

Testing a simple-test

Chio provides the function

(CALLL *SIMPLE-TEST* &OPTIONAL **STRING** *START* **END**)

for the purpose of debugging simple-tests. It applies SIMPLE-TEST to the other arguments to produce a tret. That tret is then exhausted; that is, called over and over until NIL is returned, printing each result. For example:

```
(calll #~"[aeiou]+" "eyzaiouaisqv" 3 12)
                                        prints=>  9 8 7 6 5 4 NIL
```

A silent version, CALLL-SILENTLY, useful for timing simple-tests, does the same thing without printing the results. Or, to test a tret directly, use TEST-TRET to call it repeatedly, printing each result until NIL is returned. For example,

```
(with-string ("eyzaiouaisqv")          ; WITH-STRING binds *STRING* and *END*
  (let ((tret (funcall #~"[aeiou]+" 3)))
    (test-tret tret)))
                                        prints=>  9 8 7 6 5 4 NIL
```

Some tools for building simple-tests

+NIL-THUNK+ (Constant)

is the tret that a simple-test should return to indicate that no match can be found. When called, it returns NIL:

```
(defconstant +nil-thunk+ #'(lambda ()))
```

Since this outcome is so common, it is better for a simple-test to return this constant object than to create a new one each time.

RETURN-NUM-RANGE *first last* (Function)

returns a tret that returns an increasing or decreasing sequence of consecutive integers. For example, DOT+ (= #~".+") which returns *end*, *end*-1,…,start+1, could be defined like this:

```
(defun DOT+ (start)
    (if (> *end* start)
       (return-num-range *end* (+ start 1))
       +nil-thunk+))
```

`RETURN-ONCE` *val* (Function)

returns a tret that returns `val` and then `NIL`. For an example, the simple-test `DOT (= #~".")` which matches an arbitrary character, could be defined like this:

```
(defun DOT (start)
    (if (> *end* start)
       (return-once (+ start 1))
       +nil-thunk+))
```

`TEST-ONCE` *simple-test &optional start* (Function)

calls `simple-test` to produce a tret, and then returns the value that results by calling the tret just once. It is used in the definition of `T!` and other destructive simple-tests. For example,

```
(with-string ("xxabbac") (test-once #~"[ab]+" 2)) => 6
(with-string ("xxabc") (test-once #~".+"))  =>  5
```

The use of `TEST-ONCE`, or of the closely related function `T!`, or the `!` regular-expression operator can be considered to result in a destructive operation since it may cause possible matches to be discarded. For example, `(calll #~"[ab]+" "xxabbac" 2)` prints all possible values `6,5,4,3,` whereas `(calll #~"[ab]+!" "xxabbac" 2)` prints only `6`. Any pending items that a `SIMPLE-TEST` places on the `*STACK*` are cleared by `TEST-ONCE`.

`TRET-RETURNS-VALUE-P` *simple-test start value* (Function)

returns `t` if the tret `(funcall simple-test start)` is capable of returning `value`. Conceptually, this is the same as calling `(calll simple-test start)` and checking whether `value` is one of the printed results.

Tests for arbitrary characters

These simple-tests match arbitrary characters:

name	*regexp*	*matches…*
`DOT`	`#~"."`	any single character
`DOT+`	`#~".+"`	any nonempty character sequence, longer preferred
`DOT*`	`#~".*"`	any character sequence, possibly empty, longer preferred
`DOT+<`	`#~".+<"`	any nonempty character sequence, shorter preferred

DOT*<	#~".*<"	any character sequence, possibly empty, shorter preferred
DOT+!	#~".+!"	longest-only nonempty character sequence
DOT*!	#~".*!"	longest-only, possibly empty, character sequence

Matching a literal string

A string of length two or more is handled by the simple-test `(string-test string &optional start end)`. There is also a case-insensitive version `ci-string-test`. For example, `#~"abc"` expands as `(string-test "abc" 0 3)` and `#~i"abc"` expands as `(ci-string-test "abc" 0 3)`.

Zero-width simple-tests

A *zero-width simple-test* has the property that its trets either return `START` and then `NIL`, or simply return `NIL`. In other words, they either accept or reject the current position. `WORD-BOUNDARY`, `NON-WORD-BOUNDARY`, `START-OF-LINE`, and `END-OF-LINE` are zero-width simple-tests provided by Chio.

Additional zero-width simple-tests can be conveniently defined using the macro `(LOOKAHEAD-SIMPLE-TEST (START) &BODY BODY)`. The body of the macro should return `T` if `START` is accepted, and `NIL` otherwise. For example, a zero-width simple-test that accepts a boundary between a digit and an alphabetical character could be defined as follows:

```
(let ((alpha-num-boundary
       (lookahead-simple-test (start)
         (when (< 0 start *end*)
           (let ((ch- (char *string* (- start 1)))
                 (ch+ (char *string* start)))
             (or (and (alpha-char-p ch-)
                      (digit-char-p ch+))
                 (and (alpha-char-p ch+)
                      (digit-char-p ch-)))))))))
  (calll alpha-num-boundary "aaa444" 3))
       prints=>  3 NIL
```

For further insight, macroexpand the call to `lookahead-simple-test`. The idea of

a lookahead-simple-test is adapted from Perl.

Predicate-simple-tests

Somewhat dual to the idea of the lookahead-simple-test, Chio introduces what may be a new concept, the *predicate-simple-test*. Whereas the lookahead-simple-test always returns the `start` of a string (or `NIL`), the predicate-simple-test always returns the `*end*` of the target string (or `NIL`). In other words, a predicate simple-test either accepts or rejects an *entire* string. Predicate simple-tests are especially useful when used in conjunction with `T_AND` to filter matches that satisfy some desired property. For example, `#~"[+-]?\d+!"` matches an integer. However, if you want to match only integers divisible by seven, you could use the simple-test `(t_and #~"[+-]?\d+!" div7), div7` being a predicate-simple-test that matches integers divisible by seven.

Predicate simple-tests can be conveniently defined using the macro `(PREDICATE-SIMPLE-TEST (START) &BODY BODY).` The body of the macro should return `T` if the entire string (the substring of `*STRING*` between `START` and `*END*)` is accepted, and otherwise `NIL.` For example, a predicate-simple-test that accepts integers divisible by seven could be defined as follows:

```
(let ((div7 (predicate-simple-test (start)
                (= 0 (mod (parse-integer *string*
                                         :start start :end *end*) 7)))))
    (calll (t_and #~"[+-]?\d+" div7) "1435"))
                                        ; use \d+! if you only want the longer match
                                                        prints=> 4 2 NIL
```

Here, both values `1435` and `14` are accepted. But why is `14` accepted? To make complete sense of this, you will need to understand how `t_and` works, which will be explained in the section on the algebraic operations on simple-tests. For now, this explanation will do — the digit test finds four possibilities: `4,3,2,1.` `t_and` offers each of these (`"1435", "143", "14",` and `"1"`) to `div7` to either accept or reject, and it accepts the two that are divisible by `7.`

Design considerations

A simple-test should be designed so that it can be used over and over again, and

work the same predictable way. It should not be possible for a simple-test to modify state in such a way that it will behave differently the next time it is called. (Of course, a programmer can always choose to violate this rule in a local context if he knows what he is doing. This is just a suggestion for keeping out of trouble.)

For example, consider the function `LENGTH-RANGE`. `(LENGTH-RANGE *LOW* &OPTIONAL *(HIGH LOW)*)` returns a simple-test matching any string whose length lies in the interval `[LOW,HIGH]`, preferring a longer match. `HIGH` may have the value `:INFINITY` to indicate that any string whose length is greater than or equal to `LOW` is acceptable. Consider the following definition for `LENGTH-RANGE`, which seems to work just fine, but contains a subtle bug:

```
(defun length-rangex (low &optional (high low))
 #'(lambda (start)
   (when (eql high :infinity) (setf high (- *end* start)))
    (if (< *end* (+ start low))
     +nil-thunk+
     (return-num-range (min *end* (+ start high)) (+ start low)))))
```

A few examples give the impression that `length-rangex` works as intended:

```
(calll (length-rangex 2 4) "aaaaaaa" 2) => 6 5 4 NIL
(calll (length-rangex 2) "aaaaaaa" 2) => 4 NIL
(calll (length-rangex 1 :infinity) "aaaaaaa" 2) => 7 ... 3 NIL
```

However, after executing these examples, we find that the following now prints `"5"` and then `"3"`, and that is clearly not what is desired (it should print `"5"` both times):

```
(with-string ("abcde")          ; binds *string* to "abcde" and *end* to 5
 (let ((simple-test (length-rangex 3 :infinity)))
   (print (test-once simple-test 2))   ; prints "5"
   (print (test-once simple-test 0))   ; prints "3" but should print "5"
   nil))
```

The problem, of course, is the `SETF` in the definition of `LENGTH-RANGEX`. The simple-test is a lexical closure. Once the binding for `HIGH` in the lexical closure has been altered, the simple-test behaves in a manner that was not intended.

We also remark that simple-tests are designed under the assumption that they will be only called with `START ≤ *END*`, and that their trets will never be called again once they have returned `NIL`.

Regular Expressions

Regular expression syntax

To learn about regular expressions (or "regexps"), consult the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly, 1997). This section presents information you will need to use Chio's flavor of regular expressions.

A Chio regexp is translated at read-time by the `simple-test-reader` into a Lisp expression whose value is a simple-test. The regular expression is placed between delimiters (usually quotation marks) following the dispatch sequence `#~` (or `#~i` for case-insensitive searches). For example, `#~"ab[cd]"` searches for either `"abc"` or `"abd"`, while `#~i"ab[cd]"` accepts anything that is `string-equal` to one of these. Any character can be used as a delimiter, except `#\i and #\\` (or wierd stuff like #\delete).

Within regular expressions, the backslash `#\\` is used as an escape character. The escape character itself can be inserted by entering it twice. Escaped three-digit decimal codes `\ddd` and two-digit hex codes `\xdd` are replaced with the `code-char` of that value. For example, `#~"[\009\032]+"` or `#~"[\x09\x20]+"` searches for a run of tabs and spaces. In fact you can even replace the brackets and plus sign: `#~"\x58\x09\x20\x5D\x28".` A delimiting character can itself be placed in the regular expression if it is escaped. Thus all of these have the same meaning: `#~"abc", #~cab\cc, #~/abc/, #~xabcx, #~a\abca`.

The usual conventions governing character classes are followed. For example, `#~"[ac-f\s]"` matches the letter `"a",` any letter `"c"` through `"f",` or any whitespace character, and `#~"[^ac-f\s]"` matches the complement of that set. Chio treats the expression `#~"[^]"` in the same way as `#~"."` which matches an arbitrary character, the justification being that the complement of the empty class contains all characters. The empty class `#~"[]"` is not defined.

The following characters, when escaped, serve as abbreviations for *special characters or character predicates* within a regular expression. These can be used inside or outside of character classes.

`t` or `T`	#\tab
`n` or `N`	#\newline
`r` or `R`	#\return
`d`	any digit (as determined by `digit-char-p`)
`s`	any whitespace (as determined by `whitespacep`)
`a`	any alphabetical (as determined by `alpha-char-p`)
`w`	any word character (as determined by `word-char-p`)
`D`	any non-digit
`S`	any non-whitespace character
`A`	any non-alphabetical character
`W`	any non-word character

To modify these definitions or to add additional escape abbreviations, it is only necessary to modify the function `char-class-esc` or the association list `char-test-data` and recompile Chio.

The following represent *quantifiers:*

`+`	one or more repetition, longer preferred
`*`	zero or more repetitions, longer preferred
`?`	one or zero occurrence, one preferred
`!`	first value only
`<`	return values in increasing order
`>`	return values in decreasing order
`{n}`	exactly n repetitions
`{n,}`	n or more repetitions, longer preferred
`{n,m}`	between n and m repetitions inclusive, longer preferred

When escaped, these are treated as ordinary characters. However, the (unescaped) `!, <,` and `>` are only considered to be quantifiers when they follow another quantifier or an unescaped right parenthesis. In all other contexts, they are considered to be ordinary characters. The justification for this rule is that `!, <,` and `>` have no effect on character classes anyhow. A character class tret returns only one value, so if you sort that value in increasing or decreasing order or take only the first value, you are really doing nothing at all. Thus `!, <,` and `>` are only *needed*

when they are applied to something more complicated than a character class. This is fortunate, because it would be a nuisance to have to escape these useful characters everytime they were needed to match ordinary text. Thus, for example, `#~"a!"` searches for an `a` followed by a `!`, `#~"a!+"` searches for an `a` followed by one or more `!`. Also, in `#~"a!!"` the first `!` is an ordinary character because it follows an ordinary character, and therefor the second is an ordinary character for the same reason, so the search is for an `a` followed by two `!`.

Quantifiers can be composed by writing them sequentially. For example `#~"a+<"` is the same, conceptually, as `#~"((a)+)<)"`. It tests for a run of consecutive `a`'s and returns the matches in increasing order:

```
(calll #~"a+<" "aaaabb") => 1 2 3 4 NIL
```

The fact that `#~"a+<" and #~"((a)+)<)"` are conceptually the same does not, however, imply that the parentheses are without effect. Although the results are the same, the task is handled differently depending on how parentheses are placed. This is because a quantifier that follows a closing parenthesis handles the expression inside the parentheses in a manner that is independent of what that expression might be — it does not "see inside" the parentheses.

Consider the following examples, which are listed in order of decreasing efficiency:

example 1: `#~"a+<"` is handled efficently by the `CHTES+<` routine which is optimized to handle the `+<` composition for character classes. The single letter `a` here is handled the same as would be the character class `[a]`, so `#~"a+<"` is the same as `#~"[a]+<"`.

example 2: `#~"(a)+<"`. Chio passes the character class `[a]` to the routine `T+<`, which handles the `+<` composition efficiently. Thus `#~"(a)+<"` is equivalent to `(T+< #~"a")`. This is less efficient than example 1 because `T+<` cannot take advantage of the fact that its argument is a character class.

example 3: `#~"((a)+)<"`. The `simple-test-reader` reads this as `(T< (T+ #~"a"))`. This is very inefficient because, in order to return the results in increasing order, `T<` must exhaust the tret returned by `(T+ #~"a")` before returning even the first result.

The following characters, when escaped, and not inside a character class, represent *zero-width simple-tests:*

`b`	word-boundary (as determined by `word-char-p`)
`B`	non-word-boundary
`^`	start of line
`$`	end of line

`#~"\b"` matches any position between a word and a non-word character, or position `0` when the first character in `*STRING*` is a word character, or position `*END*` when the last character in `*STRING*` is a word character. `#~"\^"` matches position `0` and any position that follows a `#\NEWLINE,` while `#~"\$"` matches position `*END*` and any position that preceeds a `#\NEWLINE.` Quantifiers are treated as ordinary characters when they follow these zero-width tests. For example, `#~"\^*"` matches an asterisk at the beginning of a line.

The vertical line character `"|"` represents the "or" operation. The `simple-test-reader` translates it into the `T_OR` function. `(T_OR simple-test-1 ... simple-test-n)` is a simple-test that matches substrings satisfying at least one of `simple-test-1` ... `simple-test-n.`

The maximum length of a regular expression (excluding escapes) is specified by the compile-time constant `+MAX-REGEXP-LENGTH+.`

Examples:

Parentheses can be used to group expressions, but they have no memory function (as they would in Perl):

```
(calll #~"([ab]+c)+" "aaacbbbcababc")    => 13 8 4 NIL
```

The `<` operator causes results to be returned in ascending order:

```
(calll #~"([ab]+c)*<" "aaacbbbcababc")   => 0 4 8 13 NIL
```

When `<` or `>` or `!` follows a character class it is treated as an ordinary character rather than a quantifier:

```
(calll #~"[ab]<+c?" "a<<<<c")            => 6 5 4 3 2
```

In the following, the effect of the `!` operator is that whenever three `a` are found, no backtracking will be performed to allow just two `a` to be used.

```
(calll #~"(a{2,3}!.)+" "aaaaaaaaaaxaaaxaaa")
                                          => 15 11 8 4 NIL
```

Without the `!` operator, more values would be returned:

```
(calll #~"(a{2,3}.)+" "aaaaaaaaaaxaaaxaaa")
                         => 18 15 14 11 8 10 7 4 9 6 3 NIL
```

and adding the `>` operator would cause these to be returned in descending order:

```
(calll #~"(a{2,3}.)+>" "aaaaaaaaaaxaaaxaaa")
                        => 18 15 14 11 10 9 8 7 6 4 3 NIL
```

It is important to beware the danger of using the `!` operator — it causes perfectly good matches to be discarded. However, when you are sure that you really do want to throw out those matches, it increases efficiency and clarity of code.

In the following case-insensitive regexp, `\A` matches any character that is not alphabetical and `\s+!|\d+!` matches either a longest-only run of whitespace or digits:

```
(calll #~i"\A([ab]+(\s+!|\d+!))*"  "_AaaB  aaab77aaab9")
                        => 18 13 7 1 NIL
```

The following picks up groups of three `a,` but accepts a group of two at the end of the string

```
(calll #~"a{2,3}!*" "aaaaaaaaaaa") => 11 9 6 3 0 NIL
```

and `!*!` does the same but accepts only the first value returned:

```
(calll #~"a{2,3}!*!" "aaaaaaaaaaa") => 11 NIL
```

The `!*<` composition moves forward-only through a string, picking up first matches only. When using `<` (except on a character class), all the work must be done in advance. Thus for this example, the entire string is scanned before any values are returned:

```
(calll #~"(a+b+)!*<" "aaabbbbaaaaaaabbbbbaaaaabbbbb")
                                => 0 7 19 29 NIL
```

The `!+<` composition would do the same thing without returning `0.` Note that `!*<` returns zero first. If you wanted zero returned last, you would use `((a+b+)!+<)?`

```
(calll #~"((a+b+)!+<)?" "aaabbbbaaaaaaabbbbbaaaaabbbbb")
                                => 7 19 29 0 NIL
```

Composite quantifiers

Although quantifiers can be composed by writing them sequentially, Chio handles certain combinations with customized routines. Those combinations are represented in Figure 1:

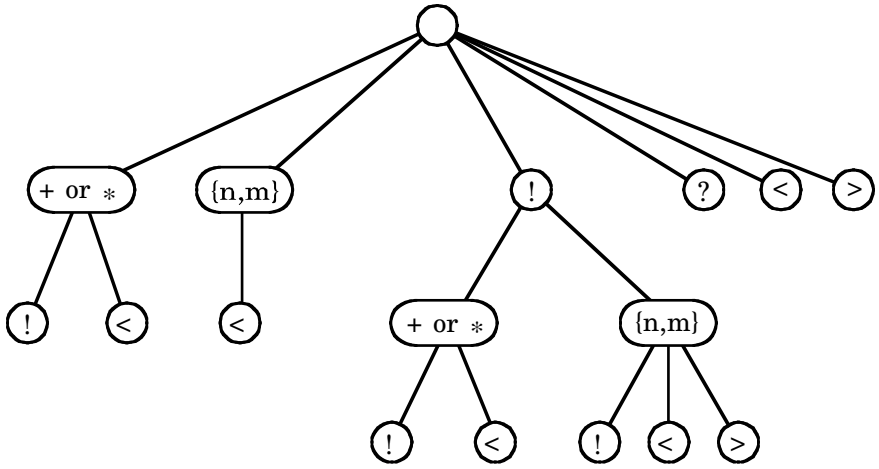

Figure 1

Any sequence of quantifiers that forms a path from the root of the tree in Figure 1 corresponds to a specially tailored Chio routine. In other words, the following sequences correspond to Chio routines:

sequences of length 1: `+` `*` `{n,m}` `!` `?` `<` `>`
sequences of length 2: `+!` `+<` `*!` `*<` `{n,m}<` `!+` `!*` `!{n,m}`
sequences of length 3: `!+!` `!*!` `!+<` `!*<` `!{n,m}!` `!{n,m}<` `!{n,m}>`

(where, in this listing, any numerical quantifier `{n,m}` can be replaced with a related quantifier `{n}` or `{n,}`.) The names of these routines are formed by prefixing the letter `"T"` to the symbol combination. For example, `T!*<` is the routine that handles the `!*<` composition. For those quantifier sequences that contain a numerical quantifier, the numerical quantifier is replaced by the letter `"N"`. For example, `T!N<` is the routine that handles the `!{n,m}<` composition.

Those routines are used by the `simple-test-reader` to expand a regular expression in which the corresponding symbols follow an expression in parentheses. Thus, the `simple-test-reader` expands `#~"(a+|b+)!*<"` first as `(T!*< #~"a+|b+")` (before it goes on to expand `#~"a+|b+"`).

For character classes, the routines represented in Figure 1 are not used. Rather, routines designed specifically for character classes handle the task. All of the

quantifier sequences shown in Figure 1 can be used following a character class (or single character), and are handled efficiently, except for those that begin with `!` or `<` or `>` (recall that `!, <,` and `>` are not considered to be quantifiers when they follow a single character or a character class.)

Although the symbol combinations shown in Figure 1 are the ones a user is most likely to use, the quantifier symbols can occur in any sequence of arbitrary length. The `simple-test-reader` groups combinations of symbols (proceeding from left to right) according to the tree of Figure 1. For example, `#~"(abc)+!<?{2,3}<+"` would be read as

```
(T+ (TN< (T? (T< (T+! (STRING-TEST "abc" 0 3)))) 2 3)).
```

More information about composite quantifiers will be presented in the section on algebraic operations on of simple-tests.

Algebraic operations on Simple-Tests

Chio provides many routines that take simple-tests as arguments and return a new simple-test. Some of these are used by the `simple-test-reader` to translate regular expressions into simple-tests.

Concatenation

```
(scat &rest simple-tests)
```

Simple-test concatenation is performed by the `scat` routine. `scat` is also used by the `simple-test-reader` to paste together the various parts of a regular expression sequentially. For example, the regular expression `#~"abcd+[^a-z]*!"` is read as

```
(SCAT (STRING-TEST "abc" 0 3)
      (CHTES+ "d" NIL NIL NIL NIL NIL)
      (CHTES+! NIL ((#\a . #\z)) NIL :- :STAR NIL))
```

which is a concatenation of three simple-tests: the first handles the literal string `"abc"`, the second handles repetitions of the letter `"d"`, and the third handles a longest only substring not containing lowercase `"a"` through `"z"`.

The possible matches that result from a concatenation of simple-tests form a tree. When `scat` is used to find these matches, the branches of the tree are searched in a depth-first order. To illustrate how this works, consider the following `scat` call:

```
(calll #~"[cx]+<[^c]+[^x]?" "ccxcxcxxxccx")
           => 4 3 6 5 10 9 8 7 10 9 8 10 9 12
```

which applies `#~"[cx]+<[^c]+[^x]?"`, a concatenation of three simple-tests to the string `"ccxcxcxxxccx"`. The first simple-test, `#~"[cx]+<"`, matches every position 1 through 12 in ascending order. The second test, `#~"[^c]+"`, matches consecutive characters not equal to `#\c` in descending order, or fails if there are no such characters. The third test, `#~"[^x]?"`, which always succeeds, matches 1 and 0 additional positions in descending order if the next character is not `#\x`, and otherwise matches 0 additional positions. The tree that results is displayed in Figure 2. The results returned by the tret appear in the bottom row in the order in which

they are returned.

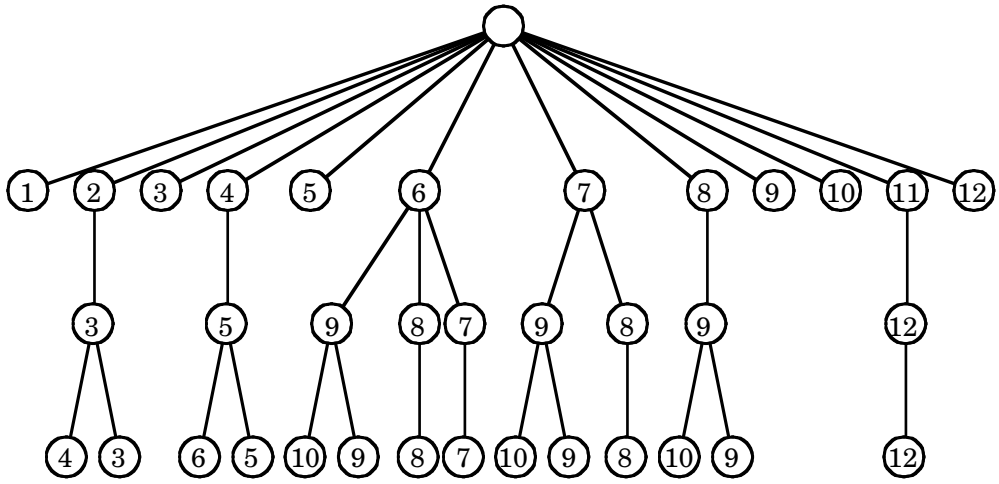

Figure 2

The OR *operation*

`(T_OR &rest simple-tests)` (function)
handles choice among alternative simple-tests. It returns a new simple-test that is satisfied if one of `simple-tests` is satisfied. `T_OR` works by exhausting the first simple-test, then the next, and so forth. For example

```
(calll (t_or #~"a{1,3}" #~"a{4,6}<") "aaaaaaaa")
                                    => 3 2 1 4 5 6 NIL
```

because the first simple-test returns `3,2,1` and then the second simple-test kicks in with `4,5,6.` If the order of the simple-tests were reversed, the values returned would be `4 5 6 3 2 1 NIL.`

The `simple-test-reader` uses `T_OR` to handle alternatives separated by vertical lines `"|".` For instance, the `simple-test-reader` would, as a first step, expand the regular expression `#~"abc+|(abab)*"` as `(t_or #~"abc+" #~"(abab)*").`

The AND *operation*

`(T_AND first-test &rest other-tests)` (function)
finds all matches to `first-test` that also match `other-tests.` The operation is perhaps misnamed, since the outcome depends on which of the arguments is `first-`

`test`. An alternative name could be `FILTER` since it filters out the matches to `first-test` that fail to match one of `other-tests`.

Let us refer to the tret returned by

```
(*)   (funcall (T_AND first-test &rest other-tests) start)
```

as the "outer" tret. `T_AND` works by using the "inner tret"

```
(**)  (funcall first-test start)
```

The outer tret (*) returns the same values, and in the same order, as the inner tret (**), except that each such value is tested against each of `other-tests` using the routine `tret-returns-value-p` to see if those `other-tests` are also *capable* of returning that value. Only those values that pass this requirement for all `other-tests` are returned by the outer tret. More precisely, the algorithm is:

step 0) Let $other\text{-}tests = (test_1 \ldots test_n)$
When the form (*) is evaluated, the inner tret (**) is computed.

Each time the outer tret is called, go to step 1:

step 1) Let `e = (funcall inner-tret).`
If `e=NIL,` outer-tret returns `NIL.`
`Otherwise, let i=1.`

step 2) With `*end*` bound to `e,`
let $tret_i$ `= (funcall` $test_i$ `start).`
Call $tret_i$ repeatedly until it returns `e` or `NIL.`
If it returns `NIL,` go to step 1.

step 3) Let `i=i+1.`
If `i>n,` outer-tret returns `e.`
Otherwise, go to step 2.

Notice that when a value `e` is returned by the inner-tret, the `other-tests` are consulted with `*end*` bound to `e` to see if they are also capable of returning `e.` For example, if the target string is `"abcdef"` and the inner tret returns the value 3, the other tests are consulted to see if they are capable of returning 3 for the target string `"abc"` (rather than the target string `"abcdef"`).

In a typical application for `T_AND,` we choose `first-test` to be one which specifies the *structure* of the desired match, and choose `other-tests` to be

predicates, perhaps defined using the `predicate-simple-test` macro, that describe *properties* the match should satisfy. For example, `first-test` might select a number in some specific format, and the `other-tests` might verify that the number possesses some desired properties.

The following example creates a simple-test that accepts any alphabetic string having length six and that contains at most two letters. The test `#~".{6}"` selects strings having the correct length. Those strings are then filtered by the `only-two-letters` predicate:

```
(let ((only-two-letters
       (predicate-simple-test (start)
         (let (letters)
           (dotimes (i (- *end* start) t)
             (pushnew (char *string* (+ start i)) letters)
             (if (third letters) (return))))))
      (target "xyxyxyuuuuvv"))
  (calll (t_and #~".{6}" only-two-letters) target))

                  => 6 NIL
```

When `T_AND` is called, `*end*` is bound to the length of the string, or 12. However, before `only-two-letters` is called to verify the value 6, `*end*` is re-bound to 6, so only the first six letters of the string are checked for the only-two-letters property. This rebinding is handled by the `T_AND` routine -- the design of `only-two-letters` can ignore the issue.

`T_AND` is not used by the `simple-test-reader`.

Memoized simple-tests

`(MEMOIZE-SIMPLE-TEST` *simple-test*`)` (function)
returns a new simple-test whose trets return the same values as those of `simple-test`, except that the same value is never returned twice. For example,

```
(calll #~"[cx]+<[^c]+[^x]?" "ccxcxcxxxccx")

             => 4 3 6 5 10 9 8 7 10 9 8 10 9 12
```

but,

```
(calll (memoize-simple-test #~"[cx]+<[^c]+[^x]?") "ccxcxcxxxccx")
                => 4 3 6 5 10 9 8 7 12.
```

`memoize-simple-test` works by marking the positions that have been returned in a bit vector of size `*end*-start+1` which it stores on the `*stack*`.

`MEMOIZE-SIMPLE-TEST` is not currently used by the `simple-test-reader`.

Repetition: `T+` *and* `T*`

`(T+` *simple-test*`)` (function)
`(T*` *simple-test*`)` (function)

`(T+ simple-test)` is a simple-test that performs `simple-test` one or more times, each time starting at the conclusion of the previous match. `(T* simple-test)` does the same, but performing the test *zero* or more times. Back-tracking is done both by the inner `simple-test` and the outer `(T+ simple-test)`, ensuring that no possible matches can be overlooked. In effect, `T+` performs a depth-first search of the match tree. Unnecessary work is avoided by tracking the positions already visited, so the search will never start a second time from the same position and no value is returned more than once. Without such tracking, costs could grow exponentially. For example, `(calll #~"(a|a)+" "aaaaa")` returns the values `5,4,3,2,1,NIL,` but it would return `62` values before returning NIL if tracking were not used.

`T+` and `T*` do not advance on matches having length zero, since doing so would create an infinite loop. For example,

```
(calll #~"(\b)+" "abc") => NIL
(calll #~"(a?)+" "b") => NIL
```

both fail immediately, even though both tests are successfull in the sense that the start of the first string is a word boundary and the `a?` test is satisfied at position 0 of the second string. Contrast this with

```
(calll #~"\b" "abc") => 0 NIL
(calll #~"a?" "b") => 0 NIL.
```

Of course, we still have

```
(calll #~"(\b)*" "abc") => 0 NIL
(calll #~"(a?)*" "b") => 0 NIL.
```

The `simple-test-reader` uses `T+` and `T*` to expand regular expressions using `+` or `*` quantifiers unless the quantifier follows a single character or character class. For example, `#~"a+"` and `#~"[ab]+"` would not be handled by `T+,` but `#~"(a)+"` does expand using `T+` because of the parentheses, and `#~"a++"` expands as `(T+ #~"a+")` because quantifiers are always recognized as such when they follow other quantifiers.

`T+` uses `3L+2` words on the `*stack*,` where `L= *end*-start` is the length of the string. Of these, `L+1` are used as a bit vector to mark the positions that have been visited and `2L+1` are used as a stack to store string positions and trets.

Examples:

```
(calll #~"(a{5}|a{3})+" "aaaaaaaaa")    => 8 5 9 6 3 NIL
(calll #~"(@+\a+)+" "@@aa@@bb@@cc@@77") => 12 11 8 7 4 3 NIL
```

Repetition without backtracking: `T+!` *and* `T*!`

`(T+!` *simple-test*`)` `=` `(T!+!` *simple-test*`)` (function)
`(T*!` *simple-test*`)` `=` `(T!*!` *simple-test*`)` (function)

`(T+! simple-test)` is a simple-test that performs `simple-test` one or more times, each time starting where the previous match ends. However, unlike `T+,` no back-tracking is done, and only the end of the final match is returned. Like all destructive operations using `!,` it should be used with care. Since it does so little bookkeeping, it can save a lot of time, especially when it fails. `T+!` is equivalent, conceptually, to the composition of `T!` with `T+.` Since in this composition, `T+` is not called on to do any backtracking, nothing would change if `T+` were replaced with `T!+.` For this reason, it follows that `T!+!` and `T+!` are the same operations, that is `T!+! = T+!.` Chio therefore assigns these two names to the same routine. (To say that `T!+! = T+!` means that `(calll T!+! simple-test str)` and `(calll T+! simple-test str)` always produce identical sequences of numbers). Likewise, `T*!` and `T!*!` are two names for a routine that performs a test zero or more times without backtracking.

These routines do not use the `*stack*.`

Examples:

```
(calll #~"(a{5}|a{3})+!" "aaaaaaaaa")      =>  8 NIL
```

```
(calll #~"(@+\a+)+!" "@@aa@@bb@@cc@@77")    =>  12 NIL
(calll #~"[ab]+!b" "aaaaaab")               =>  NIL
```

Replacing `+!` with `+` would cause a match to be found in the third example.

Dumbed repetition with backtracking: `T!+` *and* `T!*`

`(T!+ simple-test)` (function)
`(T!* simple-test)` (function)

If `test` is a simple-test, then `(T!+ test)` is a simple-test that performs `(T! test)` one or more times, each time starting where the previous match ends. In other words, one match to `test` is found starting at `START` (and no match will ever again be found starting at `START` — that is a key point). Another (just one!) match is then found starting where the first match ends, and then another starting where the second ends, this process continuing as long as we keep moving forward. The endings of all those matches are then returned in reverse order by the tret. Although `(T!+ test)` backtracks, `test` itself never does. Thus `T!+` works a little harder than `T!+!`, but possibly much less than `T+.`

Examples:

```
(calll #~"(\s\d+)!+" " 111 222 333 ")        => 12 8 4 NIL
(calll #~"(a{5}|a{3})!+" "aaaaaaaaa")        =>  8 5 NIL
```

The `simple-test-reader` uses `T!+` and `T!*` to expand regular expressions in parentheses followed by `!+` or `!*.` The `!+` and `!*` quantifiers cannot directly follow a single character or character class since `!` is not regarded as a quantifier in that context. Thus `#~"[ab]!+"` is a search for an `a` or `b` followed by one or more `!.`

`T!+` uses `*end*-start` words on the `*stack*`. These are used as a bit vector to mark the positions where some match ends.

Short repetition: `T+<` *and* `T*<`

`(T+< simple-test)` (function)
`(T*< simple-test)` (function)

If `test` is a simple-test, then `(T+< test)` is a simple-test that performs `test` one or more times, each time starting where the previous match ends. `T+<` examines

the same tree of all possible matches as `T+` but the results are returned in ascending order and the order of the search is different.

The `T+<` algorithm works as follows. Let `Y` denote the tret `(funcall (T+< test) start)`. The first time `Y` is called, the tret `(funcall test start)` is created, exhausted, and all the values that it returns are marked to record that those positions are reachable. The smallest of those marked values (greater than `start`), say val_1, is returned by `Y`. The next time `Y` is called, the tret `(funcall test` val_1`)` is created, exhausted, and all the values that it returns are marked (if not already marked). The smallest marked value (greater than val_1), call it val_2, is then returned by `Y`. This process continues until either the `*end*` of the `*string*` is returned or there are no more marked values.

Although conceptually equivalent to the composition of `T<` with `T+`, `T+<` is more efficient because it only does the computations needed to advance by one match each time the tret is called. By contrast, using `T<` would require exhausting the `T+` tret before any values could be returned.

Examples:

```
(calll #~"(a{5}|a{3})+<" "aaaaaaaaaaaaaaa")
            => 3 5 6 8 9 10 11 12 13 14 15 NIL
(calll #~"(\a+<abc)+<" "abacabcabbcaaccabcaaaabc")
            => 7 18 24 NIL
```

The `simple-test-reader` uses `T+<` and `T*<` to expand regular expressions within parentheses that are followed by `+<` or `*<`. Single characters and character classes are handled by the specialized routine `chtes+<`.

`T+< and T*<` use `L+1` words on the `*stack*`. These are used as a bit vector to mark the reachable positions.

Dumbed short repetition: `T!+<` *and* `T!*<`

`(T!+<` *simple-test*`)` (function)
`(T!*<` *simple-test*`)` (function)

If `test` is a simple-test, `(T!+< test)` is a simple-test that performs `(T! test)` one or more times, each time starting where the previous match ends. The first time the tret `Y = (funcall (T!+< test) start)` is called, one match to `test` is

sought starting at `start`. If found, the position of its end, say val_1, is returned provided $val_1 > start$, and otherwise `NIL` is returned. If `Y` is called again, another match is then sought starting at val_1. If found, the position of its end, say val_2, is returned provided $val_2 > val_1$, and so forth. `T!+<` returns the same values as `T!+`, but in the reverse order.

`T!+<` is more efficient than composing `T<` with `T!+` because in the latter case all values to be returned by the tret would be calculated in advance and saved so that those values can be returned one by one each time the tret is called. By contrast, `T!+<` only causes a match to be computed when it is needed, moving forward by just one match each time the tret is called.

`T!+<` and `T!*<` do not use the `*stack*`.

The `simple-test-reader` uses `T!+<` and `T!*<` to expand regular expressions within parentheses that are followed by `!+<` or `!*<`.

Examples:

```
(calll #~"(\s\d+)!+<" " 111 222 333 ")       =>  4 8 12 NIL
(calll #~"(a{5}|a{3})!+<" "aaaaaaaaa")       =>  5 8 NIL
```

The optional operator: `T?`

`(T? simple-test)` (function)

The `simple-test-reader` uses `T?` to handle single characters, character classes, or regular expressions within parentheses when followed by an unescaped question mark. `T?` is defined as follows:

```
(defun t? (simple-test)
  #'(lambda (start)
     (let ((itret (funcall simple-test start)))   ; inner tret
       #'(lambda ()                               ; outer tret
          (when itret
             (cond ((funcall itret))
                   (t (setf itret nil)
                      start)))))))
```

The outer tret returned by `(T? simple-test)` returns the same values returned by the inner tret returned by `simple-test`, with the exception that the outer tret returns `start` once when the inner tret is exhausted.

Examples:

```
(calll #~"a?" "aaa"))              =>  1 0 NIL
(calll #~"a+?" "aaa"))             =>  3 2 1 0 NIL
(calll #~"a+<?" "aaa"))            =>  1 2 3 0 NIL
```

The once-only operator: T!

`(T! simple-test)` (function)

The `simple-test-reader` uses `T!` to handle regular expressions within parentheses when followed by an unescaped `!`. `(T! simple-test)` creates a tret that returns only the first value that would be returned by `simple-test`, and then `NIL`. Since it causes all other possible matches to be discarded, it should be regarded as a destructive operation, only to be used when one is certain that the discarded matches either do not exist or are of no interest. `T!` is defined as follows:

```
(defun t! (simple-test)
  #'(lambda (start)                          ; returned simple-test
      #'(lambda ()                           ; and the tret it creates
          (when start
            (prog1
              (test-once simple-test start)
              (setf start nil))))))
```

The first time the tret is called, `test-once` is called to find just one match beginning at `start`. If the tret is called again, it returns `NIL`. If the form

`(test-once simple-test start)`

were replaced in this definition by the form

`(funcall (funcall simple-test start)),`

which has the same value, the definition of `t!` would still work, in the sense that its trets would return the same values. However, there would be a subtle bug. The problem is that in the second form, the tret returned by `(funcall simple-test start)` is evaluated only once and not exhausted. Due to this, if `simple-test` allocates any space on the `*stack*`, that space would never be de-allocated. The first form using `test-once` evaluates and returns the second form and also de-allocates any `*stack*` space that has been allocated by `simple-test`. To see how this is done, see Chio's code for `test-once`.

Examples:

Without the `!` this first example would return `6 5 4:`

```
(calll #~"(a+b+)!" "aaabbb")        =>  6 NIL
```

The test in the next example fails with or without the `!.` The advantage of using the `!` operator is that it causes the test to fail *quickly*. Without the `!` the test would back-track, looking for a `c` after each `b:`

```
(calll #~"((ab)+c)!" "abababababab")     =>  NIL
```

If you know that your data permits use of the `!` operator, it can save much effort in situations where a test fails. Another advantage of using the `!` operator is that it makes your regular expressions easier to read by declaring that only a subset of the possible matches are under consideration.

The sorting operators: `T<` *and* `T>`

`(T<` *simple-test*`)` (function)
`(T>` *simple-test*`)` (function)

`(T< simple-test)` creates a tret (the "outer" tret) that returns the same values as would be returned by the "inner tret" created by `simple-test,` but in ascending order, and without duplicates. `T<` works by creating and exhausting the "inner" tret produced by `simple-test,` and marking the values in a bit vector on the `*stack*` so they can be returned one by one as the outer tret is called repeatedly. `T>` works in identical manner with decreasing order.

Using the sorting operators directly is not always a good idea. Since they create and exhaust a tret for `simple-test` before even a single value can be returned, they are inherently inefficient. They do provide the advantage of omitting all duplicate values, but that task would be handled more efficiently by `memoize-simple-test.`

Although the sorting operators may only be needed in limited circumstances, the main reason they have been included with the Chio language is to provide a consistent and sensible naming strategy for the numerous compound quantifiers. For example, the operator `T!+<` does not actually use the `T<` operator at all. However it behaves, in terms of the results it produces, *as if* it were the composition of `T<, T+,` and `T!.` As long as the sorting operators play this useful role for naming the compound quantifiers, it seems sensible to actually include them in the language.

Examples:

```
(calll #~"[cx]+<[^c]+[^x]?" "ccxcxcxxxccx")
                          => 4 3 6 5 10 9 8 7 10 9 8 10 9 12 NIL
(calll #~"([cx]+<[^c]+[^x]?)<" "ccxcxcxxxccx")
                          => 3 4 5 6 7 8 9 10 12 NIL
(calll #~"([cx]+<[^c]+[^x]?)>" "ccxcxcxxxccx")
                          => 12 10 9 8 7 6 5 4 3 NIL
```

The `simple-test-reader` uses **`T<`** and **`T>`** to handle regular expressions, other than single characters or character classes, that are followed by an unescaped `<` or `>.` In other contexts, < and > are treated as ordinary characters. For example, `#~"<tag>"` searches for the literal string `"<tag>".`

Numerical quantifiers: `TN`

`(TN` *`simple-test L &optional (U L))`* (function)

`TN` repeats `simple-test n` times, where `L ≤ n ≤ U.` If `U = :infinity,` then `simple-test` is repeated `L` or more times. `L` should be a fixnum. `U` should be a fixnum or `:infinity,` and defaults to `L.` Full back-tracking is performed, so no possibilities can be missed. `TN` is used by the simple-test-reader to handle numerical repetition counts. For example,

`#~"(ab){n,m}"`	expands as `(TN #~"ab" n m)`
`#~"(ab){n}"`	expands as `(TN #~"ab" n n)`
`#~"(ab){n,}"`	expands as `(TN #~"ab" n :infinity).`

`TN` does not handle repetition counts applied to single characters or character classes (unless enclosed in parentheses) – these are handled by a specialized and very efficient routine `chtes-n.`

In Chio 1.0, `TN` delegates its work to two different routines. One routine, `tn-low,` handles the `:infinity` case, and a less efficient routine, `tn-low-high,` handles the ordinary case. Chio can sometimes deduce, based on the length of the target string, that a regexp can be handled by `tn-low` even though it contains a finite upper bound. For example,

```
(calll #~"(a){5,10}" "aaaaaaa") => 7 6 5 NIL
```

uses `tn-low` because Chio knows that 10 matches cannot possibly fit into a string of

length 7. But Chio is not smart enough to know that

```
(calll #~"(ab){5,10}" "abababababababab") => 14 12 10 NIL
```

could have been handled by `tn-low` because it assumes a worst-case scenario where each match has length one.

Whenever `tn-low` finds a match, it records its depth, that is the number of matches since `start`. If, later, another match ends at that same position with a lower depth, the recorded value is changed to the smaller value. On the other hand, if another match later ends at that same position with the same or higher depth then there is no point in proceeding further along that branch of the search tree. This can sometimes save effort.

`tn-low-high`, on the other hand, searches the entire tree except, of course, that it cuts off a branch whenever the upper limit has been exceeded. This can lead to exponential explosions of complexity. So be careful with numerical quantifiers, especially when the upper bound is a large number. For instance, try timing this form

```
(calll-silently #~"(a|a){15}" "aaaaaaaaaaaaaaaaaaaaa")
```

The time roughly doubles when 15 is changed to 16. For contrast, try replacing {15} with {15,}.

Dumbed numerical quantifiers: `T!N` *and* `T!N!`

`(T!N` *simple-test L* `&optional (`*U L*`))` (function)
`(T!N!` *simple-test L* `&optional (`*U L*`))` (function)

`T!N!` keeps finding matches for `simple-test`, each time starting where the previous match ends. It stops when `U` matches have been found or when there are no more matches. `U` can have the value `:infinity`. If `U` is not reached, the end of the last match found is returned if at least `L` matches have been found. No back-tracking is performed.

`T!N` does the same work as `TN!`, but whereas `TN!` returns at most one value (and then `NIL`), `T!N` back-tracks to return the endings of all the valid matches in descending order. The reason why `T!N` and `TN!` are so efficient is that their search tree is a simple path.

Examples:

```
(calll #~"(a+b+)!{2,4}"  "aaabbbaaabbbaaabbb   ") => 18 12 NIL
(calll #~"(a+b+)!{2,4}!" "aaabbbaaabbbaaabbb   ") => 18 NIL
```

The `simple-test-reader` does not apply `T!N` to characters or character classes because `!` is not considered to be a quantifier in that context.

Note that there is no specially defined `TN!` Chio routine. Although it is permissible (and sometimes useful) to put `!` after a numerical quantifier, it is not handled in any special way – `T!` is just composed with the quantifier. However, the `simple-test-reader` does use a specialized routine to apply the `N!` composition to characters or character classes.

Short numerical quantifier: `TN<`

`(TN< simple-test L &optional (U L))` (function)

`TN<` returns the same values as `TN` but in ascending order. When `U` is `:infinity`, `TN<` passes its labor on to the efficient routine `TN<-lbound`. When `U` is a fixnum, Chio 1.0 simply composes `T<` with `TN,` so that no improvements in efficiency are achieved. `TN<` is used by the `simple-test-reader` to handle quantifiers of the form `{n,m}<` or `{n,}<` (other than character classes, which are handled by the specialized routine `chtes-n`).

`TN<-lbound` works very much like `T+<.` However, in addition to keeping track of reachable positions, it keeps track of their depth as well. When a position is revisited with lower depth, the search may be truncated. When revisited at a higher depth, the depth is updated to the higher value.

Examples:

```
(calll #~"(a+b+){2,4}<" "aaabbbaaabbbaaabbb")
                              => 10 11 12 16 17 18 NIL
(calll #~"(.{5}|a{3}){3,}<"  "aaaaaaaabbbbbbb")
                              => 11 13 15 NIL
```

Dumbed short numerical quantifier: `T!N<`

`(T!N< simple-test L &optional (U L))` (function)

returns the same values as `T!N` but in ascending order. When the tret is called, `L`

matches are attempted, each starting at the end of the previous one. If this succeeds the end of match `L,` x_L , is returned. If the tret is called again and `U>L,` another match is attempted starting at x_L. If it succeeds, the end x_{L+1} of the match is returned. If the tret is called again and `U>L+1,` another match is attempted starting at x_{L+1}, and so forth.

Examples:

```
(calll #~"(a+b+)!{2,4}<" "aaabbbaaabbbaaabbb")
                                => 12 18 NIL
(calll #~"(c+|.{5})!{2,}<"  "cccxccccxxxxxcc")
                                => 8 13 15 NIL
```

Binding-Trees

Definition of binding-tree

Binding-trees provide a mechanism for glueing simple-tests together to describe a search. To a limited extent, this role is already performed by the `simple-test-reader`. For example, the simple-test `#~"[ab]+(c{3}d{5}|d{3}c{5})e?"` can be represented by the tree shown in Figure 3

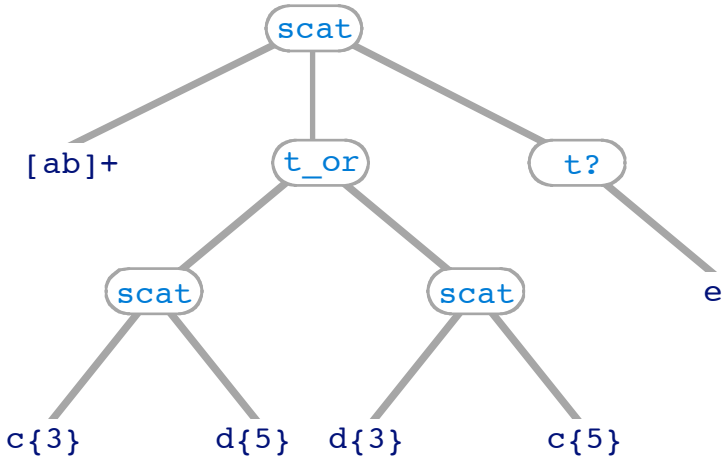

Figure 3

where the internal nodes represent operations on simple-tests (`scat`=concatenation, `t_or`=alternation, `t?`=option) and the leaves of the tree are simple-tests. The `simple-test-reader` glues these simple-tests together into one simple-test. However, the `simple-test-reader` cannot glue together arbitrary simple-tests – it can only handle those representable with Chio's regular-expression syntax. But a simple-test is just a Lisp procedure following certain rules, *and we ought to be able to include any simple-test as a leaf of a search tree, not merely those that can be understood by the* `simple-test-reader.`

Another shortcoming of the `simple-test-reader` is that it provides no way to capture sub-matches. For example, placing parentheses around `[ab]+` does not give access, as it would in some regular expression dialects, to the part of a match that matches `[ab]+.` One way to fix this would be to program more capability into the `simple-test-reader.` This approach, however, would not suffice – we need to be able to capture arbitrary sub-matches, not just the ones that correspond to Chio regular-expressions, or portions thereof.

To satisfy these needs, Chio provides the notion of *binding-tree*.

The internal nodes of a binding-tree are *operation-keywords,* of which there are three *basic-operation-keywords:*

- `:&` represents concatenation (like `scat`)
- `:o` represents alternation (like `t_or`) (the letter "o")
- `:?` represents option (like `t?`)

In addition to the basic-operation-keywords, any keyword symbol whose symbol-name has length greater than one and whose first character is `&`, `o`, or `?` is called a *remembering-operation-keyword.* For example, all of the following are remembering-operation-keywords:

```
:&0      :&foo     :&3      :ox      :o11      :?5
```

Remembering-operation-keywords perform grouping and binding functions, whereas basic-operation-keywords merely perform a grouping function. The first character, `&`, `o`, or `?`, specifies the operation.

The definition of *binding-tree* is recursive:

i. Any Lisp expression whose value is a simple-test is a binding-tree with a single node.
ii. If `op` is an operation-keyword and `L` is a non-empty list of binding-trees, then `(cons op L)` is a binding-tree.

Note that a binding-tree contains Lisp expressions that evaluate to simple-tests, rather than actual compiled simple-tests.

Example: These binding trees represent the same search:

i.
```
#~"[ab]+(c{3}d{5}|d{3}c{5})e?"
```
(binding-tree with single node)

ii.
```
(:& #~"[ab]+" #~"c{3}d{5}|d{3}c{5)" #~"e?")
```
(three leaves and one internal node)

iii.
```
(:& #~"[ab]+" (:o #~"c{3}d{5}" #~"d{3}c{5)") #~"e?")
```

iv.
```
(:& #~"[ab]+"
    (:o (:& #~"c{3}" #~"d{5}") (:& #~"d{3}" #~"c{5)"))
    (:? #~"e"))
```
(the binding tree of this example looks exactly like the tree of Figure 3, except that the internal nodes are operation-keywords.)

Naming conventions

All remembering-operation-keywords whose names start with the same character (`&`, `o`, or `?`) are indistinguishable; for example, it never matters whether you use `:&3` or `:&foo`. The names play no role in the bindings that will be formed. The only things that matter to Chio about the name of an operation-keyword are the first character, which identifies the operation, and whether or not there is another character after it. However, for the sake of code clarity, it is suggested to use ascending numerical values such as `:&0, :&1, :o2, :?3, :&4,…` to identify the remembering-operation-keywords in a binding-tree in the order in which they occur. The reason this scheme is so helpful is that *captured match results are referenced by the number (counting from left to right and starting with zero) of the remembering-operation-keyword.* For example, here is a binding-tree following this naming convention. It contains two basic- and three remembering-operation-keywords:

```
(:& #~"."                              ; single character
    (:&0 #~"[^+-]+!"                   ; longest-only run of not + or -
         (:o1 #~"[+]+"                 ; run of pluses
              #~"[-]+"                 ; run of minuses
              (:? #~i"[a-eiou]+"))     ; run of certain chars, ignoring case
         #~"\d+")                      ; run of digits
    (:&2 #~".{3,4}<"))                 ; 3 or 4 characters, preferring 3
```

and this would be read by the `simple-test-reader` as

```
(:& dot
    (:&0 (chtes+! "+-" nil nil :- nil nil)
         (:o1 (chtes+ "+" nil nil nil nil nil)
              (chtes+ "-" nil nil nil nil nil)
              (:? (chtes+ "iou" ((#\a . #\e)) nil nil nil t)))
         (chtes+ nil nil (digit-char-p) nil nil nil))
    (:&2 (length-range-short 3 4)))
```

which is a tree in which every leaf is a Lisp expression whose value is a simple-test, as shown in Figure 4. In such a list, internal tree structure is recognized as lists whose cars are keywords. Leaves of the tree are recognized as either symbols that are not keywords (like `dot`), or lists whose cars are not keywords (like `(length-range-short 3 4)`).

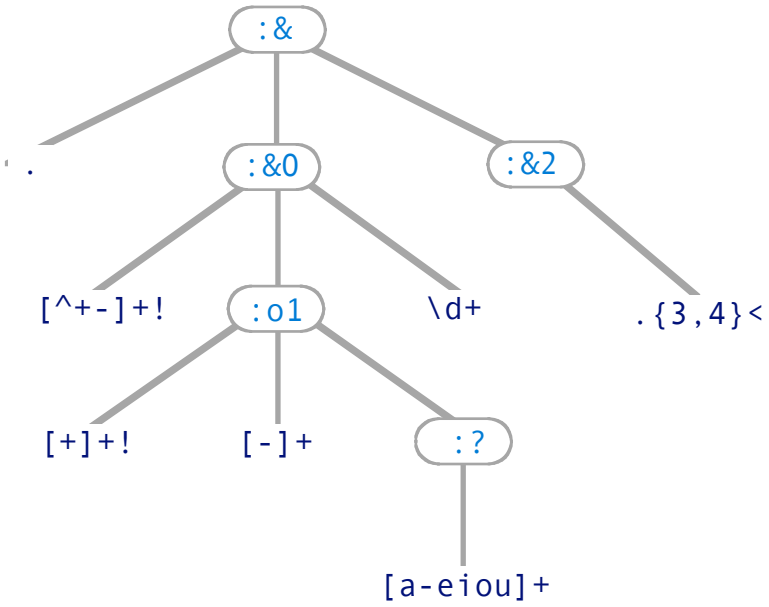

Figure 4

Compiled-binding-trees

Before it can be used, a binding-tree must be compiled. The resulting *compiled-binding-tree* structure contains a tree identical to the binding-tree, except that the code for its simple-tests has been compiled, and it is uses vectors rather than lists to enable faster access. The structure also contains information about the number of remembering-operation-keywords and the amount of space on the `*stack*` that need to be reserved to execute the search. A binding-tree is compiled using the macro `compile-test`. For example,

```
(compile-test (:& #~"."
                  (:&0 #~"[^+-]+!"
                       (:o1 #~"[+]+" #~"[-]+" (:? #~i"[a-eiou]+"))
                       #~"\d+") (:&2 #~".{3,4}<")))
```

returns something like this:

```
#S(compiled-test :tree
   (-3 .
       #(#<compiled-function dot-simple-test (non-global)>
         (0 .
          #(#<anonymous function>
            (7 .
             #(#<compiled-lexical-closure chtes+-simple-test>
               #<compiled-lexical-closure chtes+-simple-test>
               (-1 . #(#<compiled-lexical-closure chtes+-simple-test>))))
            #<compiled-lexical-closure chtes+-simple-test>))
         (12 .
          #(#<compiled-lexical-closure>))))
   :vars 6
   :ssize 9)
```

In the compiled structure, the internal nodes are fixnums that encode the operation type (`&`, `o`, or `?`) and, for each remembering node, its position. The `:vars` component is two times the number of remembering nodes, and the `:ssize` component indicates the required `*stack*` space.

The three main macros `with-test-binds`, `with-test-format`, and `with-test-split` accept either a binding-tree or a compiled-binding-tree as an argument. When the user provides a binding-tree, the macro will automatically compile it. Thus explicit compilation by the user is optional. However, when used inside an iterative control like a `loop`, `do`, or `while`, a binding-tree should always be compiled explicitly outside of the loop so that the compilation is not needlessly repeated.

Recall that a simple-test is also a binding-tree. When `compile-test` is applied to a simple-test, it is treated as a binding-tree with the single *remembering*-operation-keyword `&0`. For example, `#~"abc"` is compliled in the same way as `(:&0 #~"abc")`.

Incomplete binding-trees

The value `NIL` can be used for a leaf of a binding-tree as a place-holder for a simple-test to be inserted *after* compilation. A binding tree that contains one or more `NIL` leaves is an *incomplete binding-tree*. When `compile-test` is applied to an incomplete binding-tree having `n` such `NIL` leaves, a function of `n` variables is returned. Let's call that function `g`. The function `g` contains the compiled-binding-tree (still with `NIL` leaves) in its lexical-closure. When `g` is called, it inserts its arguments into their reserved spots and returns the modified compiled-binding-tree.

The ordering of the arguments of `g` corresponds to the order of the `NIL` leaves in the binding-tree.

Incomplete binding-trees are useful in situations where just a portion of the search instructions change each time a compiled-binding-tree is to be used, typically within iterated code. Avoiding the necessity to allocate a new data structure, the existing compiled-binding-tree is destructively modified so just the simple-tests that need to be changed are replaced with their new values. This replacement is fast because the function `g` knows exactly where to put the new simple-tests – it doesn't have to search through the compiled-binding-tree to find the right spots.

As an example,

```
(let ((incomplete
       (compile-test (:& #~"."
                         (:&0 #~"[^+-]+!"
                              (:o1 #~"[+]+" #~"[-]+" (:? NIL))
                              #~"\d+")
                         (:&2 NIL))))
      (first-insertion #~i"[a-eiou]+")
      (second-insertion #~".{3,4}<"))
  (funcall incomplete first-insertion second-insertion))
```

returns the same compiled-binding-tree as the previous example.

Access to Match Results

Chio provides the macros `mref, mwrite,` and `empty-match-p` to access results captured by binding-trees. When a binding-tree is used successfully, the coordinates (start and end) of each remembered match are stored on the `*stack*.` The access macros provide a convenient interface to these saved values.

These access macros are designed to be used only within the body of the major macros `with-test-binds, with-test-format,` and `with-test-split.` The macros are therefore called only in the context of a *successful* match. The first argument to each major macro is `prefix,` a symbol which serves to identify the macro call and provide a key to access results. The access macros may also be used within the afterform of `with-test-format` in once-only mode. (Afterforms do not exist in once-only mode with the other two macros).

MREF *prefix i fmt* &optional *empty-val* [Macro}
accesses the substring of `*string*` matching the `i`th remembering-operation-keyword of the binding-tree (starting with `i=0`). If the match has length zero, or if the match does not exist (because it refers to an unused or failed alternative of an `:o` operation-keyword) then `empty-val` (which defaults to `NIL`) is returned. If the match exists and has positive length, then the match can be read in various ways, depending on the value of `fmt`

`(MREF prefix i :A) =`	index in `*string*` of start of match
`(MREF prefix i :E) =`	index in `*string*` of end of match
`(MREF prefix i :P) =`	coordinates of the match as a pair (start . end)
`(MREF prefix i :S) =`	match, read as a string (a copy is made)
`(MREF prefix i :I) =`	match, read as an integer (as by `parse-integer`)
`(MREF prefix i :R) =`	match, read as the Lisp reader would read it
`(MREF prefix i :C) =`	the first character of the match
`(MREF prefix i :N) =`	the match, read as a number (as by Lisp reader)

The `:R` option causes one object to be read from the match. For example, if the match is `"(+ 3 4)zzz",` then `(+ 3 4)` would be read, and `zzz` ignored. Likewise, the `:I` option causes one integer to be parsed, and may not use up the entire match.

The `:N` option uses the Lisp reader to read one object, checks that the object is a number and that it uses up the entire match, signaling an error otherwise.

It is easy for a user to customize these options. It is merely necessary to modify the function `format-translation` in the file `access.lisp` and define (or redefine) the corresponding reader function. For example, if you wanted `:R` to signal an error when the whole match is not consumed, you would look at the code for `format-translation` and note that the reader corresponding to `:R` is called `obj-reader`. You would then change the code for `obj-reader` to make it behave as you want.

In addition to the above keyword values for `fmt`, a user can supply the name of a function (or lambda expression) of two variables (start end) to be applied to the match. For example,

```
(MREF prefix i (lambda (start end) (- end start)))
```

would return the length of the match, and

```
(MREF prefix i (lambda (start end)
                 (position #\d *string* :start start :end end)))
```

would return the position of the first `d` within the match. Using the `:I` option is equivalent to

```
(MREF prefix i (lambda (start end)
                 (parse-integer *string*
                                :start start :end end))).
```

Here is an example that uses `MREF with` several of its keyword tags. The string `"abc  345  3/4"` contains three items separated by whitespace. `"abc"` is match 0, `"345"` is match 1, and `"3/4"` is match 2:

```
(let* ((ws #~"\s+")              ; whitespace
       (nws #~"\S+")             ; not whitespace
       (test (compile-test
              (:& (:&0 nws) ws (:&1 nws) ws (:&2 nws)))))
  (with-test-binds X (test "abc  345  3/4")
    (list (mref X 0 :s)          ; match 0 as string
          (mref X 0 :r)          ; match 0 read as symbol
          (mref X 1 :i)          ; match 1 as integer
          (mref X 1 :c)          ; first char of match 1
```

```
(mref X 2 :n)       ; match 2 as number
(mref X 1 :a)       ; start of match 1
(mref X 1 :e)       ; end of match 1
(mref X 1 :p))))    ; match 1 as pair
```

returns => `("abc" ABC 345 #\3 3/4 6 9 (6 . 9))`

MWRITE *prefix i stream* [Macro]

outputs the match to `STREAM` without first making a copy of it. It is equivalent to

```
(mref prefix i (lambda (start end)
                 (write-string *string* stream
                               :start start :end end)))
```

EMPTY-MATCH-P *prefix i* [Macro]

returns the position `start=end` if the remembered match I has length zero. If the match has positive length, it returns `NIL`. And there is a third possibility: `end` will have a value *less* than `start` if the match does not exist because it is an unchosen or failed alternative of an `:o` remembering-operation-keyword; in this situation, `empty-match-p` returns `T`.

For example, the following reads either a number or a word and uses `empty-match-p` to distinguish which outcome it is:

```
(with-test-binds AA ((:o0 (:&1 #~"\d+") (:&2 #~"\a+")) " 123 ")
  (format t "the ~a is ~a" (if (empty-match-p AA 1)
                               "word" "number")
                           (mref AA 0 :s))
  (subseq *stack* AA-mrs AA-mre))          ; return the match vector
```

prints => `the number is 123`
returns => `#(1 4 1 4 0 -1)`

In this example, match 2 has negative "length" since it represents an unchosen alternative. `(empty-match-p AA 2)` would return `T`. Note that whereas `empty-match-p` returns a fixnum for a successful match of length zero, it returns `T` for an unused or failed alternative of an `:o` remembering-operation-keyword. It is probably rare that the need exists to distinguish between these two outcomes, since almost always it will be the case that successful matches necessarily have positive length, but `empty-match-p` makes it possible to make the distinction in those rare cases when it is needed.

Matching: with-test-binds

This section documents the macro with-test-binds, which executes code subject to bindings established by matching a binding-tree to a string.

WITH-TEST-BINDS *prefix*
(test string &key start end until afterform tags)
&body body [Macro]

Summary

WITH-TEST-BINDS finds match(es) to TEST in STRING. When the match is successful, BODY is executed subject to the bindings described by the remembering-operation-keywords of the binding-tree for TEST. The behavior of this macro, which is highly flexible, is governed by TAGS, which is a list of keywords. TEST must be either a binding-tree or a compiled-binding-tree. If TEST is a binding-tree it is compiled into a compiled-binding-tree.

Modes

There are two modes: once-only and loop.

Loop mode is signalled when one of the tags :G, :MAP, or :MAP-IF is present. Otherwise, the mode is once-only, meaning that only a single match is sought.

UNTIL and AFTERFORM are available only in loop mode.

Description

In once-only mode,...

if a match is found, BODY is executed subject to the bindings described by the remembering-operation-keywords of the binding-tree for TEST, and the last form in BODY is returned. If no match is found, BODY is never executed, and NIL is returned.

Alternatively, (return-from PREFIX val) can exit the containing block named PREFIX and return val.

In `loop` mode with `:G` tag,…

the `BODY` is placed inside of a `(loop…)` where it is executed once for each match found, subject to the bindings described by the remembering-operation-keywords of the binding-tree for `TEST.` Any `(return)` inside of `BODY` or any non-`NIL` `UNTIL` form exits the loop. When the entire `STRING` has been searched or the loop has been exited by a `(return)` or `UNTIL` form, the `AFTERFORM` (which defaults to `NIL`) is evaluated and returned.

Alternatively, any `(return-from PREFIX val)` inside `BODY` can exit with `val` as return value (and, in this case, the `AFTERFORM` is never executed because it is inside the block named `PREFIX`).

In `loop` mode with `:MAP` or `:MAP-IF` tags,…

the `BODY` is placed inside of a `(loop…)` and is executed once for each match found, subject to the bindings described by the remembering-operation-keywords of the binding-tree for `TEST.` Any `(return)` inside of `BODY` or any non-`NIL` `UNTIL` form exits the loop. Each time `BODY` is executed, the value of the last form in `BODY` is pushed onto a list, with the exception that with `:MAP-IF, NIL` values are not pushed. When the entire `STRING` has been searched or the `loop` has been exited by a `(return)` or `UNTIL` form, the `AFTERFORM` (which defaults to `NIL`) is evaluated for side-effects only. Then the collected list is nreversed and returned.

Alternatively, any `(return-from PREFIX val)` inside `BODY` exits the block with `val` as return value. This value takes precedence over the list that would have been returned by `:MAP` or `:MAP-IF` and, since `AFTERFORM` is inside the block named `PREFIX,` it is never executed.

Keyword arguments

`:TAGS`

Tags are included in a list following the keyword `:TAGS`. Permissible tags are:

`:G` Find all matches `(loop mode)`. A leftmost match is sought, starting where the previous match ends. So `#~"aa"` would only find one match in `"aaa"` and only two in `"aaabaaa"`.

`:MAP` Find all matches `(loop mode)` and return a list of the results.

`:MAP-IF` Find all matches `(loop mode)` and return a list of the results, but omitting `NIL` values.

`:anchorL` In `once-only` mode, anchors match to `START` of `STRING` (note: this has nothing to do with newlines, which are given no special treatment).

In `loop` mode, anchors first match to `START` of `STRING` and each successive match to end of the previous match.

`:anchorR` In `ONCE-ONLY` mode, anchors match to `END` of string.

`:START`

Starting index in `STRING` of target substring.

`:END`

Ending index in `STRING` of target substring.

`:UNTIL`

In `loop` mode, this form is evaluated at the conclusion of each iteration to test for an early exit from the loop. The `UNTIL` form is evaluated *after* `PREFIX-COUNT` has been incremented.

`:AFTERFORM`

In `loop` mode, this form is evaluated after the loop is exited normally (because there are no additional matches or because an `UNTIL` form returns a non-nil value).

With the `:G` tag, the `AFTERFORM` is returned. With the `:MAP` or `:MAP-IF` tags, the `AFTERFORM` is evaluated for side-effects only.

Since the `AFTERFORM` is placed inside the block named `PREFIX`, any `(return-from PREFIX val)` inside of `BODY` will cause the `AFTERFORM` to be skipped.

Local variables and block

The macro encloses its code in a block named `PREFIX`, allowing the use of `(return-from PREFIX &optional value)` to exit returning `value`. The `symbol-name` of `PREFIX` is also used as a prefix for several other symbols that are interned in the current package:

`PREFIX-END`

The end of the most recent match (or of the single match in `once-only` mode)

`PREFIX-COUNT`

In `loop` mode, when `PREFIX-COUNT` occurs inside `BODY`, it is bound to the index of the current iteration (starting with 0 during the first iteration). `PREFIX-COUNT` is incremented after each complete execution of `BODY` and before the `UNTIL` form. Within an `AFTERFORM`, `PREFIX-COUNT` equals the number of times `BODY` has been executed, or zero if `BODY` was never executed. However, if the loop is exited by a `(return)` inside of `BODY`, then that final execution of `BODY` is considered to be incomplete, so `PREFIX-COUNT` is not incremented for that last iteration. Consequently, within the `AFTERFORM`, `PREFIX-COUNT` would still equal the index of the iteration during which the exit occurred.

`PREFIX-COUNT` is not used in `once-only` mode.

`PREFIX-MRS` and `PREFIX-MRE`

The match results (that is, the coordinates of the substrings of `STRING` that match the remembering-operation-keywords of the binding-tree for `TEST`) are stored in the subvector of `*stack*` that starts at `PREFIX-MRS` and ends at `PREFIX-MRE`. (`"MRS"` and `"MRE"` stand for "match-results-start" and "match-results-end"). A user will normally access match those results indirectly via the macros `mref`, `mwrite`, and `empty-match-p` which expand into expressions that use `PREFIX-MRS` and `PREFIX-MRE`. Therefore, a user does not normally need to use these two variables directly.

The start and end of the substring that matches the i^{th} remembering-operation-keyword (starting with i=0) are stored at

```
(svref *stack* (+ prefix-mrs (* 2 i)))
```

and

```
(svref *stack* (+ prefix-mrs (* 2 i) 1)),
```

respectively (as can be observed by examining the macro-expansions of `(mref prefix i :a)` and `(mref prefix i :e))`.

If no match is found for the i^{th} remembering-operation-keyword because it is a failed or unused alternative of an `:o` operation-keyword, then the second of these values is smaller than the first and `(empty-match-p prefix i)` returns `T`.

In `loop` mode, match results are unpredictable within the scope of an `AFTERFORM` because results from the most recent successful match may be overridden by results from a more recent unsuccessful match attempt. For this reason, the behavior of the macros `mref, mwrite,` and `empty-match-p` is undefined when they are used in an `AFTERFORM`.

The variable `PREFIX-MRE` is used only when the global variable `*CHECK-MATCH-BOUNDS*` is not `NIL`.

Macro-expansion outlines

Three pseudo-code macro-expansions are included here to show approximately how the code looks that the macro produces. Many questions about the behavior of the macro can be answered by studying these code outlines. For example, if you have questions about exactly when prefix-count is incremented, or how the use of a `(return)` affects the prefix-count, glancing at the outline can provide the answer.

In `once-only` mode, the expression

```
(with-test-binds prefix (test str &key start end tags) &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(let* ((*string* STR)      ; the target string
       (*end* (or end (length *string*)))
       (prefix-end ...)    ; end of match
       (prefix-mrs ...)    ; the first position in *stack* where
                           ; match results are stored
       (prefix-mre ...)    ; the first position in *stack* following the match results
                           ; (this variable is not needed if *check-match-bounds* is nil)
       (#:gen0 ...)        ; the compiled-test is bound to a gensym
       ... other bindings ...)
  (block PREFIX            ; block can be executed by (return-from prefix &optional value)
    ... look for only one match ...
    (when match has been found
          ...BODY...)))
```

In `loop` mode with the `:G` tag, the expression

```
(with-test-binds prefix (test str &key start end
                                  until afterform tags) &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(let* ((*string* str)      ; the string being matched
       (*end* (or end (length *string*)))
       (prefix-count 0)    ; iteration counter
       (prefix-end ...)    ; end of match
       (prefix-mrs ...)    ; the first position in *stack* where match results are stored
       (prefix-mre ...)    ; the first position in *stack* following the match results
                           ; (not needed if *check-match-bounds* is nil)
       (#:gen0 ...)        ; the compiled-test is bound to a gensym
       ... other bindings ...)
  (block prefix            ; block can be executed by (return-from prefix &optional value)
    (loop
      (cond                ; loop can be exited by placing (return) inside of body
        (..not finished..  ; finished when string has been entirely searched
            (when ..match is found..
              (setf prefix-end ..index of end of match..)
              ,@body
              (incf prefix-count)
              (if untilform (return))))   ; untilform defaults to nil
        (t (return))))
    afterform))            ; afterform (defaults to nil) supplies the return value
                           ; afterform is skipped when block is exited by (return-from...)
```

In `loop` mode with the `:MAP` or `:MAP-IF` tags, the expression

```
(with-test-binds prefix (test str &key start end
                          until afterform tags) &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(let* ((*string* str)      ; the string being matched
       (*end* (or end (length *string*)))
       (prefix-end ...)    ; end of match
       (prefix-count 0)    ; iteration counter
       (prefix-mrs ...)    ; the first position in *stack* where match results are stored
       (prefix-mre ...)    ; the first position in *stack* following the match results
                           ; (not needed if *check-match-bounds* is nil)
       (#:gen0 ...)        ; the compiled-test is bound to a gensym
       (#:gen1 nil)        ; to accumulate list of results
       ... other bindings ...)
  (block prefix            ; block can be executed by (return-from prefix &optional value)
    (loop
      (cond                ; note that loop can be exited by placing (return) inside of body
        (..not finished..
          (when ..match is found..
            (setf prefix-end ..index of end of match..)
            (let ((#:gen2 (progn ..,@body...)))
              (push #:gen2 #:gen1)) ; with :map-if:  (if #:gen2 (push #:gen2 #:gen1))
            (incf prefix-count)
            (if untilform (return))))
        (t (return))))
    afterform              ; for side-effects only -- does not supply the return value
                           ; afterform is skipped when block is exited by (return-from...)
    (nreverse #:gen1)))    ; return list of results
```

Substitution: with-test-format

This section documentations the macro with-test-format, which performs substitutions in a string subject to bindings established by matching a binding-tree to the string.

WITH-TEST-FORMAT *prefix destination*
(test string &key start end afterform tags)
&body body [Macro]

where either destination = (variable stream)
or destination = variable, the latter being equivalent to
destination = (variable NIL)

Modes

There are two modes: once-only and loop. Loop mode is signalled when the :G tag is present. Otherwise, the mode is once-only, meaning that only a single match is sought.

Description

With VARIABLE bound to STREAM, WITH-TEST-FORMAT reads STRING and copies it to STREAM. The match (or *matches* in loop mode) to TEST, if any, is/are however not copied to STREAM. Rather, for each such match, the code in BODY is executed instead. Typically, BODY contains instructions sending output to STREAM intended as a substitution for the match(es). Or, the absence of such instructions in BODY has the effect of deleting the match(es). BODY is executed subject to the various bindings described by the remembering-operation-keywords of the binding-tree for TEST. TEST must be either a binding-tree or a compiled-binding-tree. If TEST is a binding-tree it is compiled to produce a compiled-binding-tree.

If STREAM=NIL, output is directed to a string-output-stream.

The behavior of this macro, which is highly flexible, is governed by TAGS, which is a list of keywords.

In either mode, AFTERFORM (which defaults to NIL) is returned. AFTERFORM is executed within the scope of the bindings for the local variables prefix-out (the output collected by the string-output-stream when STREAM=NIL), prefix-end (the

end of the last match), and prefix-count (the number of matches, or zero if there are none).

Keyword arguments

:TAGS

Tags are included in a list following the keyword :TAGS. Permissible tags are:

:G Find all matches (loop mode) and execute BODY subordinate to the bindings established by each match.

:anchorL In once-only mode, anchors match to START of STRING (note: this has nothing to do with newlines, which are given no special treatment).

In loop mode, anchors first match to START of STRING and each successive match to end of the previous match.

:anchorR In ONCE-ONLY mode, anchors match to END of string.

:SET When STREAM=NIL (so that output goes to a newly allocated string), causes STRING, which must be a settable generalized variable, to be SET to the output string.

:HALT In either mode, this causes formatting to halt after last match, so that the tail portion of STRING following the last match is not copied to STREAM. Has no effect if there is no match.

:SKIP-IF-NONE
Governs the behavior of the macro in the event that no match is found. If :SKIP-IF-NONE tag is present and no match is found, no output is sent to STREAM. If :SKIP-IF-NONE tag is not present and no match is found, the entire STRING is sent to STREAM.

Consequently if STREAM=NIL and no match is found, a fresh copy of STRING is bound to the variable prefix-out when the tag is not present, and an empty string is bound to prefix-out when the tag is present.

:START

Starting index in STRING of target substring.

:END

Ending index in STRING of target substring.

:AFTERFORM

AFTERFORM is evaluated and returned after all other processing has been completed. This also allows for the insertion of code, possibly for side-effect only, to be executed after formatting is completed, but still within the scope of the bindings of PREFIX-OUT, PREFIX-END, and PREFIX-COUNT. The AFTERFORM provides the only mechanism whereby the macro WITH-TEST-FORMAT returns a value.

remark: although there is no :UNTIL form, the same effect can be achieved by placing a (return) inside of BODY to exit the loop.

Local variables

The symbol-name of PREFIX is used as a prefix for several other symbols that are interned in the current package. In contrast to the macros with-test-binds and with-test-split, with-test-format does *not* enclose its code in a block named PREFIX. Hence return-from cannot be used to exit and return a value (values may only be returned by AFTERFORM.)

PREFIX-OUT

If STREAM=NIL, then PREFIX-OUT is bound to the string collected by the output-stream-string. For example, :AFTERFORM PREFIX-OUT causes the output string to be returned.

PREFIX-END

The end of most recent match (or of the single match in once-only mode). This variable comes in handy if you want to halt the matching process and then resume it later from the same spot.

PREFIX-COUNT

In once-only mode, PREFIX-COUNT is zero during the execution of BODY. Within an AFTERFORM, PREFIX-COUNT is one if BODY has been executed (because a match has been found) and zero otherwise.

In loop mode, when PREFIX-COUNT occurs inside BODY, it is bound to the index of the current iteration (starting with zero during the first iteration). PREFIX-COUNT is incremented after each complete execution of BODY. During execution of an AFTERFORM, PREFIX-COUNT equals the number of times BODY has been executed, which would be zero if BODY was never executed. However, if the loop is exited by a (return) inside of BODY, then that final partial execution of BODY is considered to be incomplete, so PREFIX-COUNT is not incremented for that last iteration. Consequently, within the AFTERFORM, PREFIX-COUNT is still equal to the index of the iteration during which the exit occurred. (The code skeleton below makes it clear why this happens).

PREFIX-MRS and PREFIX-MRE

See the description of these variables in the documentation for with-test-binds.

Macro-expansion outlines

Two pseudo-code macro-expansions are included here to show approximately how the code looks that the macro produces. Many questions about the behavior of the macro can be answered by studying these code outlines. In `once-only` mode, the expression

```
(with-test-format prefix (variable stream)
                  (test string &key start end afterform tags)
                  &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
 (let* ((*STRING* STR)      ; the target string
        (*END* (or END (length *STRING*)))
        (PREFIX-COUNT 0)    ; match counter
        (PREFIX-MRS ...)    ; the first position in *stack* where match results are stored
        ; include the following line only if  *CHECK-MATCH-BOUNDS* is T
        (PREFIX-MRE ...)    ; the first position in *stack* following the match results
        ; include the following line only if STREAM=NIL
        (PREFIX-OUT ...)    ; the result string
        (PREFIX-END ...)    ; end of match
        (#:GEN0 ...)        ; the compiled-test is bound to a gensym
        (VARIABLE (or STREAM (make-string-output-stream)))
        ... other bindings ...)
   (... look for only one match ...
     (cond
       (..match found..
         ..copy portion of string preceeding match to stream..
         (setf PREFIX-END  ..index of end of match..)
         ,@BODY             ; code formatting match to stream
         (incf PREFIX-COUNT)  ; increment counter
         ; omit the following line when :HALT tag is present
         ..copy portion of string following match to stream..)
       (..match not found..
         ; omit the following line when :SKIP-IF-NONE tag is present
         ..copy entire string to stream..)))
    ;include the following line only if STREAM=NIL
   (setf PREFIX-OUT         ; with :SET tag, STRING is also SETF to this value.
         (get-output-stream VARIABLE))
   AFTERFORM)
```

In `loop` mode, the expression

```
(with-test-format prefix (variable stream)
                  (test string &key start end afterform tags)
                  &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
 (let* ((*STRING* STR)      ; the target string
        (*END* (or END (length *STRING*)))
        (PREFIX-COUNT 0)    ; match counter
        (PREFIX-MRS ...)    ; the first position in *stack* where match results are stored
        ; include the following line only if  *CHECK-MATCH-BOUNDS* is T
        (PREFIX-MRE ...)    ; the first position in *stack* following the match results
        ; include the following line only if STREAM=NIL
        (PREFIX-OUT ...)    ; the result string
        (PREFIX-END ...)    ; end of match
        (#:GEN0 ...)        ; the compiled-test is bound to a gensym
        (VARIABLE (or STREAM (make-string-output-stream)))
        ... other bindings ...)
   (loop                    ; note that loop can be exited by (RETURN) inside of BODY
     (cond
       (..match found..
         ..copy portion of string preceeding match to stream..
         (setf PREFIX-END  ..index of end of match..)
         ,@BODY             ; code formatting match to stream
         (incf PREFIX-COUNT)) ; increment counter
       (t (return))))       ; break out of loop when no match is found
   (unless (or (:SKIP-IF-NONE tag is present and no match found)
               (:HALT tag is present and at ≥1 match found))
      ..copy portion of string after last match to stream,
        or copy entire string if no match has been found)
   ;include the following line only if STREAM=NIL
   (setf PREFIX-OUT         ; with :SET tag, STRING is also SETF to this value.
         (get-output-stream VARIABLE))
   AFTERFORM)
```

Splitting: with-test-split

This section contains the documentation for Chio's splitting macro, with-test-split.

WITH-TEST-SPLIT *prefix*
(*sep test string*
&key *start end until afterform tags*)
&body *body* [Macro]

Summary

WITH-TEST-SPLIT splits STRING into fields separated by separators matching the simple-test SEP. There are two modes: once-only and loop. Loop mode is signalled when one of the tags :G, :MAP, or :MAP-IF is present. otherwise, the mode is once-only.

In once-only mode, BODY is executed *just one time* subject to bindings under which match i refers to the substring of field i that matches TEST's unique remembering-operation-keyword. The value of the last form in BODY is returned.

In loop mode, BODY is executed *once for each field* subject to the bindings established by applying TEST to the leftmost successful match in the field. Match i refers to the substring of the field matching the i^{th} remembering-operation-keyword of TEST. With the tags :MAP and :MAP-IF results can be collected and returned as a list. With the :G tag, an AFTERFORM can be evaluated for a return value.

Description

In either mode,…

STRING is split into fields numbered 0,1,… by separator substrings that match SEP, which must be a simple-test. The number of fields is one more than the number of substrings that match SEP. The macro expands into code that is contained within a block named PREFIX. Thus (RETURN-FROM PREFIX VAL) inside BODY exits and returns VAL, overriding the normal return values described below, and skipping the AFTERFORM.

In once-only mode,…

TEST can be either

1. a binding-tree with exactly one remembering-operation-keyword
2. a compiled-binding-tree compiled from a binding-tree with exactly one remembering-operation-keyword
3. NIL (to match entire field)

When TEST is a binding-tree, it is compiled to produce a compiled-binding-tree. When TEST is a simple-test, it is handled as would be the binding-tree (:&0 TEST). TEST=NIL is handled as would be the binding-tree (:&0 #~".*!") which matches and remembers the entire field.

STRING is first split into fields by separator substrings that match SEP. In each field, the leftmost substring matching TEST is then found, storing the coordinates of the match to TEST's single remembering-operation-keyword onto the *stack* for later access. The BODY is then executed just one time subject to bindings under which match I refers to the substring of field i that matches TEST's single remembering-operation-keyword. The value of the last form in BODY is returned.

For example, within BODY, (mref PREFIX i :s) would return a copy of the substring of field i matching TEST's single remembering-operation-keyword.

Even when you know that you want to match each entire field, use of an appropriate TEST (rather than simply making TEST=NIL) permits verification of the data. For example, to verify that each field contains only digits, you could use TEST=#~"\d+" with :anchorL and :anchorR tags.

Unless the :PERSIST tag is present, an error is signaled if TEST fails to find a match in some field.

In loop mode,…

BODY is executed once for each field, subject to the bindings established by applying TEST once to its leftmost successful match in the field. For example, during the i^{th} execution of BODY, (mref PREFIX k :s) returns a copy of the substring of field i matching the k^{th} remembering-operation-keyword of the binding-tree for TEST (starting with i=0 for field 0 during iteration 0).

With the :G tag, AFTERFORM (default=NIL) is evaluated for the return value.

With the :MAP or :MAP-IF tags, a list is returned containing the value, for each field, of the last form in BODY. :MAP-IF is the same as :MAP except that NIL values are not included in the list. With :MAP or :MAP-IF, the AFTERFORM is evaluated, but for side-effects only.

TEST can be either

1. a binding-tree
2. a compiled-binding-tree
3. NIL (to match entire field)

When TEST is a binding-tree, it is compiled to produce a compiled-binding-tree. When TEST is a simple-test, it is handled as would be the binding-tree (:&0 TEST). TEST=NIL is handled as would be the binding-tree (:&0 #~".*!") which matches and remembers the entire field.

Unless the :PERSIST tag is present, an error is signaled if TEST fails to find a match in some field.

Any (return) inside of BODY or any non-NIL UNTIL form exits the loop. One difference between these two ways to achieve an early exit is that with a (return) the PREFIX-COUNT is not incremented for the iteration during which the exit occurs.

`with-test-split`

Keyword arguments

`:TAGS`

Tags are included in a list following the keyword `:TAGS.` Permissible tags are:

`:G` (`loop` mode) Match `TEST` once against the leftmost successful match in each field.

`:MAP` (`loop` mode) Match `TEST` once against the leftmost successful match in each field, collecting in a list the value of the last form in `BODY.`

`:MAP-IF` (`loop` mode) Same as `:MAP,` but omit `NIL` results from the list.

`:anchorL` Anchors `TEST` to the start of each field.

`:anchorR` Anchors `TEST` to the end of each field.

`:persist` Indicates that execution should persist (and no error be signalled) in the event that `TEST` fail to find a match in a field. In the presence of the `:persist` tag…

- `PREFIX-COUNT` is incremented even for fields that fail to match `TEST` because `PREFIX-COUNT` is a counter for fields, not successful matches.
- In `once-only` mode, when `TEST` fails to match in field `i,` a match of negative length is recorded, and `(empty-match-p PREFIX i)` returns `T.`
- In `loop` mode, when `TEST` fails to match in a field, `BODY` is not executed for that field (and nothing is pushed onto the return list if the `:MAP` or `:MAP-IF` tag is present.)

The `:PERSIST` tag has no effect if `TEST` is `NIL` (or `DOT*! = #~".*!"`) since that test never fails to match.

`:START`

Starting index in `STRING` of target substring.

`:END`

Ending index in `STRING` of target substring.

`:UNTIL`

In `loop` mode, this form is evaluated at the conclusion of each iteration to test for an early exit from the loop. An `UNTIL` form is evaluated after `PREFIX-COUNT` has been incremented.

:AFTERFORM

In loop mode, this form is evaluated after the loop is exited normally (because there are no additional matches or because an UNTIL form or a (return) causes an early exit from the loop).

With the :G tag, the AFTERFORM is returned. With the :MAP or :MAP-IF tags, the AFTERFORM is evaluated for side-effects only.

Since the AFTERFORM is placed inside the block named PREFIX, any (return-from PREFIX val) inside of BODY will cause the AFTERFORM to be skipped.

:FIELDS

In once-only mode, this integer is an upper bound on the number of fields. Its default value is supplied by the constant +DEFAULT-SPLIT-FIELDS+. If the actual number of fields exceeds this, there is no problem unless you try to access the contents of the extra fields using access macros such as mref, in which case you will either get nonsensical results or, if the global variable *check-match-bounds* is not NIL, signal an error. If you are not able to provide an upper bound for the number of fields, you can avoid the issue by working in loop mode.

Local variables and block

The macro with-test-split encloses its code in a block named PREFIX, allowing the use of (return-from PREFIX &optional value) to exit returning value. The symbol-name of PREFIX is also used as a prefix for several other symbols that are interned in the current package:

PREFIX-COUNT

In loop mode, when PREFIX-COUNT occurs inside BODY, it is bound to the index of the current field (starting with 0 during the first field). PREFIX-COUNT is incremented after each complete execution of BODY and before the UNTIL form. During execution of an AFTERFORM, PREFIX-COUNT is the number of fields that have been processed.

However, if the loop is exited by a (RETURN) inside of BODY, then that final execution of BODY is considered to be incomplete, so PREFIX-COUNT is not incremented for that last partial iteration. Consequently, within the AFTERFORM, PREFIX-COUNT is still equal to the index of the field during which the exit occurred.

In once-only mode, PREFIX-COUNT equals the total number of fields. It can never be zero since there is always at least one field.

PREFIX-MRS and PREFIX-MRE

The match results are stored in the subvector of *stack* that starts at PREFIX-MRS and ends at PREFIX-MRE. A user does not normally access these results directly, but uses instead the access macros mref, mwrite, and empty-match-p.

In once-only mode, the match vector contains, for each field, the start and end of the substring matching TEST's single remembering-operation-keyword. More precisely, for field i the start and end of that substring are equal to

```
(aref *stack* (+ PREFIX-MRS (* 2 i)))
```

and

```
(aref *stack* (+ PREFIX-MRS (* 2 i) 1)).
```

(Compare these with the macroexpansions of `(mref PREFIX i :a)` and `(mref PREFIX i :e))`. If `TEST` fails to find a match in field `i` and the `:PERSIST` tag is used to avoid signalling an error, then the second of these entries is made smaller than the first so that a match of negative length is recorded and `(empty-match-p PREFIX i)` returns `T`.

In `loop` mode, during iteration `i` (while field `i` is being examined), the match vector contain the start and end of the substrings of field `i` that match the remembering-operation-keywords of `TEST`. More precisely, during iteration `i` the start and end of the substring of field `i` that matches the k^{th} remembering-operation-keyword of `TEST` are equal to

```
(aref *stack* (+ PREFIX-MRS (* 2 k)))
```

and

```
(aref *stack* (+ PREFIX-MRS (* 2 k) 1)).
```

If `TEST` fails to find a match in field `i` (and the `:PERSIST` tag is used), then `BODY` is not executed during the i^{th} iteration so these entries have no possible relevance. If `TEST` succeeds in finding a match in field `i`, but no match is found for the k^{th} remembering-operation-keyword of `TEST` (because it is a failed or unused option of an `:o` operation-keyword) then a match of negative length is recorded so that `(empty-match-p PREFIX k)` returns `T`.

The contents of match results should not be accessed within the scope of an `AFTERFORM`, neither directly nor indirectly by calling the macros `mref`, `mwrite`, or `empty-match-p`.

Macro-expansion outlines

Three pseudo-code macro-expansions are included here to show approximately how the code looks that the macro produces. Many questions about the behavior of the macro can be answered by studying these code outlines.

In `once-only` mode, the expression

```
(with-test-split prefix (sep test string
                 &key start end fields tags)
                 &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(let* ((*string* STRING) ; the target string
       (*end* (or END (length *string*)))
       (PREFIX-MRS…)       ; the first position in *stack* where match results are stored
       (PREFIX-COUNT 0)    ; field counter
       …other bindings…)
  (block PREFIX     ; block can be exited by (return-from prefix &optional val)
     Match each field to TEST, storing start
     and end for single remembering-operation-
     keyword in *stack*, and setting PREFIX-
     COUNT to the number of fields.
      ; the variable PREFIX-MRE is used only if *CHECK-MATCH-BOUNDS* is T
     (let ((PREFIX-MRE…))      ; the first position in *STACK* following the match results
        BODY)))
```

In `loop` mode with `:G` tag, the expression

```
(with-test-split prefix (sep test string
                 &key start end until afterform tags)
                 &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(LET* ((*string* STRING) ; the target string
       (*end* (or END (length *string*)))
       (PREFIX-MRS…)       ; the first position in *stack* where match results are stored
       (PREFIX-COUNT 0)    ; field (and iteration) counter
       …other bindings…)
  (block PREFIX     ; block can be exited by (return-from prefix &optional val)
    ; the variable PREFIX-MRE is used only if  *CHECK-MATCH-BOUNDS* is T
    (let ((PREFIX-MRE…)) ; the first position in *stack* following the match results
      (loop                ; loop can be exited by a (RETURN) inside of BODY
        …try to find leftmost match for TEST in current field…
        …if not found and :PERSIST tag is not present
           signal error…
        (when ..match is found..
          ,@BODY)
        (incf PREFIX-COUNT)  ; note that count is incremented before UNTILFORM
        (if UNTILFORM (return)) ; test for early exit from loop
        (cond (..no more fields.. (return))
              (t  ..continue with next field..))))
    AFTERFORM)) ; return value, defaults to NIL.
```

In `loop` mode with the `:MAP` or `:MAP-IF` tag, the expression

```
(with-test-split prefix (sep test string
                 &key start end until afterform tags)
                 &body body)
```

macroexpands into code that resembles this pseudo-code skeleton:

```
(LET* ((*string* STRING) ; the target string
       (*end* (or END (length *string*)))
       (PREFIX-MRS…)       ; the first position in *stack* where match results are stored
       (PREFIX-COUNT 0)    ; field (and iteration) counter
       (#:gen1 nil)        ; to accumulate list of results
       …other bindings…)
  (block PREFIX     ; block can be exited by (return-from prefix &optional val)
    ; the variable PREFIX-MRE is used only if  *CHECK-MATCH-BOUNDS* is T
    (let ((PREFIX-MRE…)) ; the first position in *stack* following the match results
      (loop              ; loop can be exited by a (RETURN) inside of BODY
        …try to find leftmost match for TEST in current field…
        …if not found and :PERSIST tag is not present
           signal error…
        (when ..match is found..
          (push (progn ..,@BODY...) #:gen1))
          ; with :MAP-IF, a NIL value would not be pushed
        (incf PREFIX-COUNT)  ; note that count is incremented before UNTILFORM
        (if UNTILFORM (return)) ; test for early exit from loop
        (cond (..no more fields.. (return))
              (t  ..continue with next field..))))
    AFTERFORM)) ; return value, defaults to NIL.
```

Managing the Stack

Chio uses the simple-vector `*stack*` as a workspace for holding intermediate results. A casual Chio user does not need to be aware of it, but a programmer writing algorithms such as those in the file `algebra.lisp` for creating and manipulating simple-tests, needs to know more. In this section, we discuss the stack, and how it is used.

The global variable `*stack*` is defined in the file `stack.lisp.` It is a simple vector of size `+stack-size+.` The global variable `*fp*` acts as a pseudo fill-pointer for `*stack*,` always pointing to the next position to be filled. It is not a real fill-pointer because `*stack*` , being a simple-vector, does not have a fill-pointer. This is done to save time – a simple-vector is faster than a vector with fill-pointer. Calling `(s-display)` will display the value of `*fp*` and also the maximum value, `*max-fp*,` that `*fp*` has attained since last being reset to zero. This will provide an idea how much stack space you are using (and therefore whether you need to increase `+stack-size+).`

To illustrate ideas, consider this example. The simple-test `#~"a+"` does not use the `*stack*.` Evaluating

```
(s-reset)                ; reset the stack
(s-display)              ; display current fill-pointer
```

prints: current fill pointer `*FP* = 0`
maximum value of fill pointer since last calling (s-reset) `*MAX-FP* = 0`.

If we then evaluate

```
(calll #~"a+" "aaaaa") ;=> 5 4 3 2 1 NIL
(s-display)
```

prints: current fill pointer `*FP* = 0`
maximum value of fill pointer since last calling (s-reset) `*MAX-FP* = 0`

so we see that the `*stack*` was not used, since otherwise `*MAX-FP*` would have a nonzero value. However, if we put parentheses around the `a,` then `#~"(a)+"` is read by the `simple-test-reader` as a call to the routine `T+` which does use the `*stack*.` Now, evaluating

```
(calll #~"(a)+" "aaaaa"))) ;=> 5 4 3 2 1 NIL
(s-display)
```

prints: current fill pointer `*FP* = 0`

maximum value of fill pointer since last calling (s-reset) `*MAX-FP* = 17`

we see that the `*stack*` was used. The fact that the fill pointer is now zero might lead us to believe that all the extra memory we have caused to be allocated is now free to be disposed by the garbage-collector. Unfortunately, this is false. Evaluating

```
(subseq *stack* 0 17)      ;take a look at the *stack* space we used
```

prints:

```
#(7 7 #<Anonymous Function #xBA41D6>
  #<COMPILED-LEXICAL-CLOSURE #x127841E> 2
  #<COMPILED-LEXICAL-CLOSURE #x127843E> 3
  #<COMPILED-LEXICAL-CLOSURE #x127845E> 4
  #<COMPILED-LEXICAL-CLOSURE #x127847E> 0 0 1 1 1 1 1)
```

showing that several trets that were created by `T+` are still in the `*stack*` and cannot be garbage collected. In order to release them, you would have to execute `(s-reset),` which would replace the first 17 elements in `*stack*` with zeros. As a practical matter, you probably would not want to bother to do this because your next use of the `*stack*` would overwrite those elements anyhow, releasing the trets that the array entries point to. That is why the main macros do not truly clear the stack, but only reset `*fp*` to zero; it would mostly be a waste of time to zero out the memory, but you are free to do so at any time by executing `(s-reset).`

When evaluated at top-level, `*fp*` should always be zero. If not, then someone has either made a mistake, or simply created and called some trets without exhausting them. The main macros, `with-test-binds, with-test-format,` and `with-test-split,` all use `unwind-protect` to reset `*fp*` to zero even if an error occurs. If you find that `*fp*` is nonzero at top-level, you can reset the stack by calling `(s-reset)` which not only sets `*fp*` to zero, but also replaces the contents of `*stack*` with zeros.

To get an idea how `*fp*` can obtain a nonzero value, consider the following:

```
(s-reset)                ; reset the stack just in case
(s-display)              ; display current fill-pointer
```

prints: current fill pointer `*FP*=0`
maximum value of fill pointer since last calling (s-reset) `*MAX-FP*=0`

```
(with-string ("aaaaa")     ; binds *string* to "aaaaa" and *end* to 5
    (funcall (funcall #~"(a)+" 0)))        ; create a tret and call it just
```

```
once
(s-display)
```

prints: current fill pointer `*FP*=17`
maximum value of fill pointer since last calling (s-reset) `*MAX-FP*=17`

What has happened here? The parentheses around the `a` have caused the routine `T+` to be called, and `T+` uses the `*stack*`. The form `(funcall #~"(a)+" 0)` returns a tret and that tret has been called just once. But it needs to be called six times before it is exhausted and `T+` can release its stack space — and this never happens. The problem become serious if you make a mistake like this inside an iterative construct. For instance,

```
(loop
  (with-string ("aaaaa")
    (funcall (funcall #~"(a)+" 0))))
```

prints > Error: `*STACK*` overflow
> While executing: `ALLOCATE-STACK`

One way to avoid problems like this is to use the macro `(with-stack-restore &body body)`, which executes `BODY` and restores `*fp*` to the value it started with:

```
(s-reset)
(with-string ("aaaaa")
  (with-stack-restore
    (funcall (funcall #~"(a)+" 0))))
(s-display)
```

prints current fill pointer `*FP* = 0`
Maximum value of fill pointer since last calling `(s-reset)*MAX-FP*=17`

Dealing with the `*stack*` is much less of a problem than on might suppose. Normally, a user does not create trets directly. If you use Chio's main tools, like the three main macros, all the book-keeping is handled for you, and you don't have to think about the `*stack*`.

Examples

The author has provided a collection of Chio examples at the url

http://www.toiling-in-obscurity.net/chio/examples/

References

Friedl, Jeffrey E. F., Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools, O'Reilly 1998.

Excellent source for information about regular expressions and various languages that use them.

Graham, Paul, On Lisp, Advanced Techniques for Common Lisp, Prentice Hall 1994.

A wonderful book for Lisp programmers who already know the basics. This book does an excellent job of explaining macros, what they are for, when they should be used, and how to write them. Currently available at the author's web site, http://www.paulgraham.com/onlisp.html.

Steele, Guy L., Jr., Common Lisp, The Language, Second Edition, Digital Press (Butterworth-Heinemann) 1990.

Definition of the Common Lisp programming language.