MaxPC API Documentation

Max Rottenkolber

Friday, 16 April 2021

Table of Contents

1 maxpc (Package)

Max's Parser Combinators[1] is a simple and pragmatic library for writing parsers and lexers based on combinatory parsing. MaxPC is capable of parsing deterministic, context-free languages, provides powerful tools for parse tree transformation and error handling, and can operate on *sequences* and *streams*. It supports unlimited backtracking, but does not implement *Packrat Parsing* (http://pdos.csail.mit.edu/~baford/packrat/thesis/). Instead, MaxPC achieves good performance through its optimized primitives, and explicit separation of matching and capturing input. In practice, MaxPC parsers perform better on typical computer languages—when compared to Packrat parsers—at the expense of not producing linear-time parsers.[2]

- 1. MaxPC is a complete rewrite of *MPC* (https://github.com/eugeneia/mpc) with was in turn a fork of Drew Crampsie's *Smug* (http://smug.drewc.ca/).
- 2. See *MaxPC: Why? How? / Packrat Parsing* (http://mr.gy/blog/maxpc.html#section-3-1) on why the book keeping costs of Packrat parsing diminish the gain in execution time for typical grammars and workloads.

1.1 Basic Concepts

MaxPC *parsers* are *functions* that accept an *input* (#section-4) as their only argument and return three values: the remaining *input* or `nil`, a *result value* or `nil`, and a *generalized boolean* that indicates if a *result value* is present. The *type* of a parser is:

`(function (`*input*`) (or` *input* `null) * boolean)`

A parser can either succeed or fail when applied to an input at a given position. A succeeding parser is said to *match*. When a parser matches it can optionally produce a *result value*, which may be processed by its parent parsers. New parsers can be created by composing existing parsers using built-in or user defined *parser combinators*. A parser combinator is a higher-order *function* that includes one or more parsers as its arguments and returns a parser.

By convention, all parsers are defined as higher-order *functions* that return the respective parser, even if they are nullary. For instance, the `?end` parser is obtained using the form "`(?end)`".

A lexical convention is used to make three different types of parsers easily distinguishable:

- Parsers whose names start with a question mark (`?`) never produce a result value.
- Parsers whose names start with an equals sign (`=`) always produce a result value.
- Parsers whose names start with a percent symbol (`%`) may produce a result value depending on their arguments.

1.2 Example

We will define a parser for a simplistic grammar for Email addresses of the form:

email-address := *user* `@` *host*

First we define a parser for the valid characters in the *user* and *host* parts of an address. Just for this example, we choose these to be alphanumeric characters and then some. `?address-character` uses the `%or` combinator to form the union of two instances of `?satisfies` that match different sets of characters.

```
(defun ?address-character ()
  (%or (?satisfies 'alphanumericp)
       (?satisfies (lambda (c)
                     (member c '(#\- #\_ #\. #\+))))))
```

Then we use `?address-character` to implement our address parser which matches two sequences of "address characters" separated by an `@` character, and produces a list of user and host components as its result value. We match `?address-character` one or more times using `%some`, and produce the matched subsequence as the result value using `=subseq`. Both parts of the address separated by an @ character are matched in sequence using `=list`, whose result value is finally transformed by `=destructure`.

```
(defun =email-address ()
  (=destructure (user _ host)
      (=list (=subseq (%some (?address-character)))
             (?eq #\@)
             (=subseq (%some (?address-character))))
    (list user host)))
```

We can now apply `=email-address` using `parse`:

```
(parse "foo_bar@example.com" (=email-address))
 ⇒ ("foo_bar" "example.com"), T, T
(parse "!!!@@@.com" (=email-address))
 ⇒ NIL, NIL, NIL
(parse "foo_bar@example.com@baz" (=email-address))
 ⇒ ("foo_bar" "example.com"), T, NIL
```

1.3 Overview

`parse` is the entry point of `MaxPC`. It applies a parser to an input and returns the parser's result value, and two *boolean* values indicating if the parser matched and if there is unmatched input remaining, respectively.

1.3.1 Basic Parsers

- `?end` matches only when there is no further input.
- `=element` unconditionally matches the next element of the input *sequence*.
- `=subseq` produces the subsequence matched by its argument as its result value.
- `?satisfies`, `?test`, and `?eq` match input conditionally.
- `%maybe` matches, even if its argument fails to match.

1.3.2 Logical Combinators

- `%or` forms the union of its arguments.
- `%and` forms the intersection of its arguments.
- `%diff` forms the difference of its arguments.
- `?not` negates its argument.

1.3.3 Sequence Combinators

- `?seq` matches its arguments in sequence.
- `=list` matches its arguments in sequence and produces a list of their results as its result value.
- `%any` matches its argument repeatedly any number of times.
- `%some` matches its argument repeatedly one or more times.

1.3.4 Transformation

- `=transform` produces the result of applying a *function* to its argument's result value as its result value.
- `=destructure` is a convenient destructuring *macro* on top of `=transform`.

1.3.5 Error Handling

- `?fail` never matches and evaluates its body when it is called.
- `%handler-case` and `%restart-case` allow you to set up *handlers* and *restarts* across parsers.
- `get-input-position` can be called in error situations to retrieve the current position in the input.

1.4 Caveat: Recursive Parsers

A recursive parser can not refer to itself by its constructor, but must instead call itself by *symbol*—calling its constructor would otherwise result in unbounded recursion. In order to do so the parser *function* needs to be *bound* in the *function namespace* using `setf`. The example below implements a parser for balanced parentheses, and illustrates how to avoid this common caveat.

```
(defun ?parens ()
  (?seq (?eq #\() (%maybe '?parens/parser) (?eq #\))))

(setf (fdefinition '?parens/parser) (?parens))

(parse "((()))" (?parens)) ⇒ NIL, T, T
```

1.5 %and (Function)

Syntax:

— Function: **%and** `&rest` *parsers*

Arguments and Values:

parsers—*parsers*.

Description:

`%and` applies *parsers*, and matches the last *parser* only if all previous *parsers* succeed. If the last *parser* produces a result value then `%and` produces that value as its result value. It can be said that `%and` forms the set intersection of *parsers*.

Examples:

```
(parse '(:foo) (%and (?satisfies 'symbolp)
                     (?satisfies 'keywordp)))
→ NIL, T, T
(parse '(foo) (%and (?satisfies 'symbolp)
                    (?satisfies 'keywordp)))
→ NIL, NIL, NIL
(parse '(foo) (%and))
→ NIL, NIL, NIL
(parse '(foo) (%and (?satisfies 'symbolp)
                    (=element)))
→ FOO, T, T
(parse '() (%and (%maybe (?fail (princ 'foo)))
                 (%maybe (?fail (princ 'bar)))
                 (%maybe (?fail (princ 'baz)))))
▷ FOOBARBAZ
→ NIL, T, T
```

1.6 %any (Function)

Syntax:

— Function: **%any** *parser*

Arguments and Values:

parser—a *parser*.

Description:

`%any` matches *parser* in sequence any number of times. If *parser* produces a result value and matches at least once then `%any` produces a *list* of the values as its result value.

Examples:

```
(parse '(a b c) (%any (=element))) → (A B C), T, T
(parse '() (%any (=element))) → NIL, T, T
```

1.7 %diff (Function)

Syntax:

— Function: **%diff** *parser* `&rest` *not-parsers*

Arguments and Values:

parser—a *parser*.

not-parsers—*parsers*.

Description:

`%diff` matches *parser* only if applying *not-parsers* fails. If *parser* produces a result value then `%diff` produces that value as its result value. It can be said that `%diff` forms the set difference of *parser* and the union of *not-parsers*.

Examples:

```
(parse '(foo) (%diff (?satisfies 'symbolp)
                     (?satisfies 'keywordp)))
→ NIL, T, T
(parse '(:foo) (%diff (?satisfies 'symbolp)
                      (?satisfies 'keywordp)))
→ NIL, NIL, NIL
(parse '(foo) (%diff (?satisfies 'symbolp)))
→ NIL, T, T
(parse '(:foo) (%diff (?satisfies 'symbolp)))
→ NIL, T, T
```

1.8 %handler-case (Macro)

Syntax:

— Macro: **%handler-case** *parser* `&body` *clauses*

clauses::= {↓*error-clause*}*

error-clause::= (*typespec* ([*var*]) {*declaration*}* {*form*}* *parser-form*)

Arguments and Values:

parser—a *parser*.

typespec—a *type specifier*.

var—a *variable name*.

declaration—a `declare` *expression*; not evaluated.

form—a *form*.

parser-form—a *form* that evaluates to a *parser*.

Description:

`%handler-case` executes *parser* in a *dynamic environment* where handlers are active as if by `handler-case`. If a *condition* is handled by `%handler-case`, *parser-form* is evaluated and the resulting *parser* is applied.

Examples:

```
(defun assert-digit (c)
  (or (digit-char-p c)
      (error "Not a digit: ~c" c)))

(parse "01x2"
       (%any (%handler-case (%and (?satisfies 'assert-digit)
                                  (=element))
               (error (e)
                 (format t "Error at position ~a: ~a~%"
                         (get-input-position) e)
                 (?seq (=element))))))
▷ Error at position 2: Not a digit: x
→ (#\0 #\1 #\2), T, T
```

See Also:

`handler-case`.

1.9 %maybe (Function)

Syntax:

— Function: **%maybe** *parser*

Arguments and Values:

parser—a *parser*.

Description:

`%maybe` matches *parser* or nothing all, but always succeeds. If *parser* matches and produces a result value then `%maybe` produces that value as its result value.

Examples:

```
(parse '(a) (%maybe (=element))) → A, T, T
(parse '() (%maybe (=element))) → NIL, T, T
(parse '(42) (%maybe (?satisfies 'evenp))) → NIL, T, T
```

1.10 %or (Function)

Syntax:

— Function: **%or** `&rest` *parsers*

Arguments and Values:

parsers—*parsers*.

Description:

`%or` attempts to successfully apply *parsers*, and matches the first succeeding *parser*, if any. If that *parser* produces a result value then `%or` produces that value as its result value. It can be said that `%or` forms the set union of *parsers*.

Examples:

```
(parse '(a) (%or (?eq 'a) (?eq 'b))) → NIL, T, T
(parse '(b) (%or (?eq 'a) (?eq 'b))) → NIL, T, T
(parse '(a) (%or)) → NIL, NIL, NIL
(parse '(a) (%or (=element)
                 (?fail (format t "No element.~%"))))
→ A, T, T
(parse '() (%or (?fail (princ 'foo))
                (?fail (princ 'bar))
                (?fail (princ 'baz))))
▷ FOOBARBAZ
→ NIL, NIL, T
```

1.11 %restart-case (Macro)

Syntax:

— Macro: **%restart-case** *parser* `&rest` *clauses*

clauses::= {↓*restart-clause*}*

restart-clause::= (*case-name* ([*lambda-list*]) ⟦`:interactive` *interactive-expression* | `:report` *report-expression* | `:test` *test-expression*⟧ {*declaration*}* {*form*}* *parser-form*)

Arguments and Values:

parser—a *parser*.

case-name—a *symbol* or `nil`.

lambda-list—an *ordinary lambda list*.

interactive-expression—a *symbol* or a *lambda expression*.

report-expression—a *string*, a *symbol*, or a *lambda expression*.

test-expression—a *symbol* or a *lambda expression*.

declaration—a `declare` *expression*; not evaluated.

form—a *form*.

parser-form—a *form* that evaluates to a *parser*.

Description:

`%restart-case` executes *parser* in a *dynamic environment* where restarts are active as if by `restart-case`. If control is transferred to a *restart-clause*, *parser-form* is evaluated and the resulting *parser* is applied.

Examples:

```
(parse "012x3"
       (%any (%restart-case
                 (=transform
                  (=element)
                  (lambda (c)
                    (if (digit-char-p c)
                        c
                        (error "Not a digit: ~c" c))))
               (skip-element ()
                 :report "Skip character."
                 (?seq (=element))))))
▷ Error: Not a digit: x
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Skip non-digit character.
▷  2: Return to Lisp Toplevel.
▷ Debug> :continue 1
▷ Invoking restart: Skip character.
→ (#\0 #\1 #\2), T, T
```

See Also:

`restart-case`.

1.12 %some (Function)

Syntax:

— Function: **%some** *parser*

Arguments and Values:

parser—a *parser*.

Description:

`%some` matches *parser* in sequence one or more times. If *parser* produces a result value then `%some` produces a *list* of the values as its result value.

Examples:

```
(parse '(a b c) (%some (=element))) → (A B C), T, T
(parse '() (%some (=element))) → NIL, NIL, T
```

1.13 =destructure (Macro)

Syntax:

— Macro: **=destructure** (`&rest` *lambda-list*) *parser* `&body` *forms*

Arguments and Values:

lambda-list—a *destructuring lambda list*.

parser—a *parser*.

forms—an *implicit progn*.

Description:

`=destructure` matches *parser* and destructures its result value as if by `destructuring-bind`. The _ (underscore) symbol can occur in *lambda-list* any number of times, and is substituted with a *fresh*, *uninterned symbol* and declared `ignorable`. If *parser* matches `=destructure` evaluates *forms* and produces the value of the last *form* as its result value. If no *forms* are supplied the value of the last, *interned* variable defined in *lambda-list* is produced as the result value instead.

Examples:

```
(parse '(10 % 3) (=destructure (x _ y)
                   (=list (=element) (?eq '%) (=element))
                  (mod x y)))
→ 1, T, T

(parse "a," (=destructure (x _)
               (=list (=element) (?eq #\,))))
→ #\a, T, T

(parse '(a b c) (=destructure (x &rest xs)
                    (%some (=element))))
→ '(B C), T, T
```

Exceptional Situations:

If the result value of *parser* does not match the destructuring pattern, an *error* of *type* `program-error` is signaled.

See Also:

`destructuring-bind`

1.14 =element (Function)

Syntax:

— Function: **=element** <*no arguments*>

Description:

`=element` matches the next element and produces that element it as its result value.

Examples:

```
(parse '(a) (=element)) → A, T, T
(parse '() (=element)) → NIL, NIL, T
```

1.15 =list (Function)

Syntax:

— Function: **=list** `&rest` *parsers*

Arguments and Values:

parsers—*parsers*.

Description:

`=list` matches *parsers* in sequence, and produces a *list* of the result values of *parsers* as its result value.

Examples:

```
(parse '(a) (=list (=element) (?end)))
→ (A NIL), T, T
(parse '(a b) (=list (=element) (?end)))
→ NIL, NIL, NIL
(parse '(a) (=list))
→ NIL, T, NIL
```

1.16 =subseq (Function)

Syntax:

— Function: **=subseq** *parser*

Arguments and Values:

parser—a *parser*.

Description:

=subseq matches *parser,* and produces the subsequence matched by *parser* as its result value.

Examples:

```
(parse '(1 2 3) (=subseq (%any (?satisfies 'numberp))))
→ (1 2 3), T, T
(parse "123" (=subseq (%any (?satisfies 'digit-char-p))))
→ "123" T, T
```

1.17 =transform (Function)

Syntax:

— Function: **=transform** *parser function*

Arguments and Values:

parser—a *parser*.

function—a *designator* for a *function* of one argument.

Description:

=transform matches *parser* and produces the result of applying *function* to the result value of *parser* as its result value.

Examples:

```
(parse '(41) (=transform (=element) '1+)) → 42, T, T
(parse '() (=transform (=element) '1+)) → NIL, NIL, T
```

1.18 ?end (Function)

Syntax:

— Function: **?end** <*no arguments*>

Description:

`?end` matches the end of input.

Examples:

```
(parse '() (?end)) → NIL, T, T
(parse '(a) (?end)) → NIL, NIL, NIL
```

1.19 ?eq (Function)

Syntax:

— Function: **?eq** *x* `&optional` *parser*

Arguments and Values:

x—an *object*.

parser—a *parser*. The default is `(=element)`.

Description:

`?eq` matches *parser* if its result value is `eq` to *x*.

Examples:

```
(parse '(a) (?eq 'a)) ⇒ NIL, T, T
(parse '(b) (?eq 'a)) ⇒ NIL, NIL, NIL
```

See also:

`?satisfies`

1.20 ?fail (Macro)

Syntax:

— Macro: **?fail** `&body` *forms*

Arguments and Values:

forms—*forms*.

Description:

`?fail` always fails to match, and evaluates *forms* when it is applied.

Examples:

```
(parse '(a b c) (?fail (format t "Position: ~a~%"
                               (get-input-position))))
▷ Position: 0
→ NIL, NIL, NIL
```

1.21 ?not (Function)

Syntax:

— Function: **?not** *parser*

Arguments and Values:

parser—a *parser*.

Description:

`?not` matches the next element unless *parser* matches.

Examples:

```
(parse '(:foo :baz) (?not (?seq (?eq :foo) (?eq :bar))))
→ NIL, T, NIL
(parse '() (?not (?eq :baz))
→ NIL, NIL, NIL
```

1.22 ?satisfies (Function)

Syntax:

— Function: **?satisfies** *test* `&optional` *parser*

Arguments and Values:

test—a *designator* for a *function* of one argument that returns a *generalized boolean*.

parser—a *parser*. The default is `(=element)`.

Description:

`?satisfies` matches *parser* if its result value *satisfies the test*.

Examples:

```
(parse '(1) (?satisfies 'numberp)) → NIL, T, T
(parse '(a) (?satisfies 'numberp)) → NIL, NIL, NIL
(parse '(a b c)
       (?satisfies (lambda (s)
                     (intersection s '(b c d)))
                   (%any (=element))))
⇒ NIL, T, T
```

1.23 ?seq (Function)

Syntax:

— Function: **?seq** `&rest` *parsers*

Arguments and Values:

parsers—*parsers*.

Description:

`?seq` matches *parsers* in sequence.

Examples:

```
(parse '(a) (?seq (=element) (?end)))
→ NIL, T, T
(parse '(a b) (?seq (=element) (?end)))
→ NIL, NIL, NIL
(parse '(a) (?seq))
→ NIL, T, NIL
```

1.24 ?test (Macro)

Syntax:

— Macro: **?test** (*test* `&rest` *arguments*) `&optional` *parser*

Arguments and Values:

test—a *designator* for a *function* that returns a *generalized boolean*.

arguments—*objects*.

parser—a *parser*. The default is `(=element)`.

Description:

`?test` matches *parser* if its result value *satisfies the test* with *arguments* as additional arguments to *test*.

Examples:

```
(parse '(a) (?test ('member '(a b)))) ⇒ NIL, T, T
(flet ((power-of-p (n e) (= (mod n e) 0)))
  (parse '(42) (?test (#'power-of-p 2)))) ⇒ NIL, T, T
```

Notes:

```
(?test ({fun} {args}*))
≡ (?satisfies (lambda (x)
                (funcall {fun} x {args}*)))
```

Exceptional Situations:

If *test* accepts less arguments than the number of *arguments* plus one an *error* of *type* `program-error` is signaled.

See also:

`?satisfies`

1.25 get-input-position (Function)

Syntax:

— Function: **get-input-position** <*no arguments*>

→ *position*

→ *position*, *line*, *column*

Arguments and Values:

position, *column*—non-negative *integers*.

line—a positive *integer*.

Description:

`get-input-position` returns the number of elements read from the input so far. Additionally, *line* and *column* positions are returned if the input's *element type* is `character`. Lines are counted starting at one while columns are counted starting from zero.

`get-input-position` may only be called from within the body of `?fail`, the handlers of `%handler-case` or the restarts of `%restart-case`.

Exceptional Situations:

If `get-input-position` is not evaluated within the dynamic context of `?fail`, `%handler-case` or `%restart-case` an *error* of *type* `simple-error` is signaled.

1.26 parse (Function)

Syntax:

— Function: **parse** *input-source parser*

→ *result*, *match-p*, *end-p*

Arguments and Values:

input-source—an *input source*.

parser—a parser.

result—an *object*.

match-p, end-p—generalized booleans.

Description:

`parse` applies *parser* to the *input* and returns the parser's *result value* or `nil`. *Match-p* is *true* if *parser matched* the *input-source*. *End-p* is *true* if *parser matched* the complete *input-source*. *Input* is derived from *input-source* by using `maxpc.input:make-input`.

`parse` accepts *input sources* of *type* `sequence` and `stream` out of the box.

See Also:

input (#section-4), *maxpc.input.stream* (#section-5)

2 maxpc.char (Package)

Utility parsers for character inputs.

2.1 *whitespace* (Variable)

Initial Value:

```
(#\Tab #\Newline #\PageUp #\Page #\Return #\ )
```

Value Type:

a *list* of *characters*.

Description:

The *value* of `*whitespace*` is a *list* of *characters* considered to be *whitespace characters*.

2.2 =line (Function)

Syntax:

— Function: **=line** `&optional` *keep-newline-p*

Arguments and Values:

keep-newline-p—a *generalized boolean*. The default is *false*.

Description:

`=line` matches zero or more *characters* in sequence followed by a `#\Newline` *character* or the end of input, and produces the *string* of *characters* as its result value. The terminating `#\Newline` *character* is not included in *string* unless *keep-newline-p* is *true*.

Examples:

```
(parse (format nil "foo~%bar~%baz") (%any (=line)))
→ ("foo" "bar" "baz"), T, T
```

Exceptional Situations:

If an element attempted to be matched is not a *character* an *error* of *type* `type-error` is signaled.

2.3 ?char (Function)

Syntax:

— Function: **?char** *char* `&optional` *case-sensitive-p*

Arguments and Values:

char—a *character*.

case-sensitive-p—a *generalized boolean*. The default is *true*.

Description:

`?char` matches *char*. `?char` is case sensitive unless *case-sensitive-p* is *false*.

Exceptional Situations:

If the next element is not a *character* an *error* of *type* `type-error` is signaled.

2.4 ?newline (Function)

Syntax:

— Function: **?newline** <*no arguments*>

Description:

`?newline` matches the `#\Newline` *character*.

2.5 ?string (Function)

Syntax:

— Function: **?string** *string* `&optional` *case-sensitive-p*

Arguments and Values:

string—a *string*.

case-sensitive-p—a *generalized boolean*. The default is *true*.

Description:

`?string` matches the *characters* in *string* in sequence. `?string` is case sensitive unless *case-sensitive-p* is *false*.

Exceptional Situations:

If an element attempted to be matched is not a *character* an *error* of *type* `type-error` is signaled.

2.6 ?whitespace (Function)

Syntax:

— Function: **?whitespace** <*no arguments*>

Description:

`?whitespace` matches an element that is a member of `*whitespace*`.

Exceptional Situations:

If the next element is not a *character* an *error* of *type* `type-error` is signaled.

3 maxpc.digit (Package)

Parsers for digit numerals in character inputs.

3.1 =integer-number (Function)

Syntax:

— Function: **=integer-number** `&optional` *radix*

Arguments and Values:

radix—a *number* of *type* `(integer 2 36)`. The default is `10`.

Description:

`=integer-number` matches one or more digit *characters* in the specified *radix*, optionally lead by a sign character, in sequence, and produces the *integer* represented by that sequence as its result value. The leading sign can be `#\+` and `#\-` for positive and negative values respectively. The default is a positive value.

Examples:

```
(parse "101010" (=integer-number 2)) → 42, T, T
(parse "+101010" (=integer-number 2)) → 42, T, T
(parse "-101010" (=integer-number 2)) → -42, T, T
(parse "x101010" (=integer-number 2)) → NIL, NIL, NIL
```

Exceptional Situations:

If an element attempted to be matched is not a *character* an *error* of *type* `type-error` is signaled.

3.2 =natural-number (Function)

Syntax:

— Function: **=natural-number** `&optional` *radix*

Arguments and Values:

radix—a *number* of *type* `(integer 2 36)`. The default is `10`.

Description:

`=natural-number` matches one or more digit *characters* in the specified *radix* in sequence, and produces the natural *number* represented by that digit sequence as its result value.

Examples:

```
(parse "234" (=natural-number 2)) → NIL, NIL, NIL
(parse "101010" (=natural-number 2)) → 42, T, T
```

Exceptional Situations:

If an element attempted to be matched is not a *character* an *error* of *type* `type-error` is signaled.

3.3 ?digit (Function)

Syntax:

— Function: **?digit** `&optional` *radix*

Arguments and Values:

radix—a *number* of *type* `(integer 2 36)`. The default is `10`.

Description:

`?digit` matches a single digit *character* in the specified *radix*.

Exceptional Situations:

If the next element is not a *character* an *error* of *type* `type-error` is signaled.

4 maxpc.input (Package)

The generic *input* interface allows extensions to parse other *types* of *input sources*. To add a new *input source type*, `make-input` has to be specialized on that *type*. The following methods have to be defined for *inputs*:

- `input-empty-p`
- `input-first`
- `input-rest`

The following methods can optionally be defined for *inputs*:

- `input-position`
- `input-element-type`
- `input-sequence`

4.1 input-element-type (Generic Function)

Syntax:

— Generic Function: **input-element-type** *input*

→ *typespec*

Arguments and Values:

input—an *input*.

typespec—a *type specifier*.

Description:

`input-element-type` returns a *type specifier* that designates the *type* of the elements in *input*.

4.2 input-empty-p (Generic Function)

Syntax:

— Generic Function: **input-empty-p** *input*

→ *empty-p*

Arguments and Values:

input—an *input*.

empty-p—a *generalized boolean*.

Description:

`input-empty-p` returns *true* if *input* is empty.

4.3 input-first (Generic Function)

Syntax:

— Generic Function: **input-first** *input*

→ *element*

Arguments and Values:

input—a non-empty *input*.

element—an *object* of the *type* designated by the *type specifier* returned by `input-element-type` when called on *input*.

Description:

`input-first` returns the first element in *input*.

Exceptional Situations:

If *input* is empty the behavior of `input-first` is unspecified.

4.4 input-position (Generic Function)

Syntax:

— Generic Function: **input-position** *input*

→ *position*

Arguments and Values:

input—an *input*.

position—an *integer* between 0 and `array-dimension-limit` inclusively.

Description:

`input-position` returns the *position* of *input*.

4.5 input-rest (Generic Function)

Syntax:

— Generic Function: **input-rest** *input*

→ *rest*

Arguments and Values:

input—a non-empty *input*.

rest—the remaining *input*.

Description:

`input-rest` returns the remaining *input* without the first element.

Exceptional Situations:

If *input* is empty the behavior of `input-rest` is unspecified.

4.6 input-sequence (Generic Function)

Syntax:

— Generic Function: **input-sequence** *input length*

→ *sequence*

Arguments and Values:

input—an *input*.

length—an *integer* between 0 and `array-dimension-limit` inclusively.

sequence—a *sequence*.

Description:

`input-sequence` returns a *sequence* of the next *length* elements in *input*.

Exceptional Situations:

If the number of elements in *input* are less than *length* the behavior of `input-sequence` is unspecified.

4.7 make-input (Generic Function)

Syntax:

— Generic Function: **make-input** *input-source*

Arguments and Values:

input-source—an *object*.

Description:

`make-input` returns an *input* for *input-source*.

5 maxpc.input.stream (Package)

Implements support for *input sources* of *type* `stream`. Input from *streams* is copied into a temporary buffer lazily as required by the parser. *Streams* of *type* `file-stream` are read in as chunks of customizable size.

5.1 *bound* (Variable)

Initial Value:

`NIL`

Description:

`*bound*` can be set to limit the number of elements read from *stream inputs* in a single call to to `parse`.

5.2 *chunk-size* (Variable)

Initial Value:

`1000000`

Description:

`*chunk-size*` controls the size by which the buffer used for *stream inputs* grows, and the number of elements read at a time when parsing from *streams* of *type* `file-stream`.

5.3 *element-type* (Variable)

Initial Value:

`NIL`

Description:

`*element-type*` can be set to enforce a specific stream element type when reading from *stream inputs*. This can be useful when dealing with bivalent streams.