# A Portable Packrat Parser Library for Scheme

Tony Garnock-Jones <tonyg@lshift.net>

27th August 2005

Packrat parsing is a memoizing, backtracking recursive-descent parsing technique that runs in time and space linear in the size of the input text. The technique was originally discovered by Alexander Birman in 1970 [1], and Bryan Ford took up the idea for his master's thesis in 2002 [4, 3, 2]. For detailed information on the technique, please see Bryan Ford's web page at

`http://pdos.csail.mit.edu/~baford/packrat/`

This document describes an R5RS Scheme library of parsing combinators implemented using the packrat parsing algorithm. The main interfaces are the `packrat-parse` macro (section 3) and the combinators into which it expands (section 2), the `base-generator->results` function (section 1.2), and the accessors for `parse-result` records (section 1.1).

## 1 Data Structures

This section describes the data structures that make up the core of the packrat parsing algorithm, and some of the low-level procedures that operate on them.

### 1.1 parse-result

A parse-result record describes the results of an attempt at a parse at a particular position in the input stream. It can either record a successful parse, in which case it contains an associated semantic-value, or a failed parse, in which case it contains a parse-error structure.

**(parse-result? <object>) → <boolean>**

This is a predicate which answers #t if and only if its argument is a parse-result record.

**(parse-result-successful? <parse-result>) → <boolean>**

This predicate returns #t if its argument represents a successful parse, or #f if it represents a failed parse.

**(parse-result-semantic-value <parse-result>) → <object> or #f**

If the argument represents a successful parse, this function returns the associated semantic-value; otherwise, it will return #f.

**(parse-result-next <parse-result>) → <parse-results> or #f**

If the argument represents a successful parse, this function returns a parse-results record representing the parsed input stream starting immediately after the parse this parse-results represents. For instance, given an input stream [a, b, c, d, e], if the parse-result given to parse-result-next had completed successfully, consuming the [a, b, c] prefix of the input stream and producing some semantic value, then the parse-results returned from parse-result-next would represent all possible parses starting from the [d, e] suffix of the input stream.

**(parse-result-error <parse-result>) → <parse-error> or #f**

If the argument represents a failed parse, this function returns a parse-error structure; otherwise, it may return a parse-error structure for internal implementation reasons (to do with propagating errors upwards for improved error-reporting - see section 3.2.4 of [3]), or it may return #f.

**(make-result <semantic-value> <next-parse-results>) → <parse-result>**

This function constructs an instance of parse-result representing a successful parse. The first argument is used as the semantic value to include with the new parse-result, and the second argument should be a parse-results structure representing the location in the input stream from which to continue parsing.

**(make-expected-result <parse-position> <object>) → <parse-result>**

This function constructs an instance of parse-result representing a failed parse. The parse-position in the first argument and the value in the second argument are used to construct a variant of a parse-error record for inclusion in the parse-result that reports that a particular kind of value was expected at the given parse-position.

**(make-message-result <parse-position> <string>) → <parse-result>**

This function constructs an instance of parse-result representing a failed parse. The parse-position in the first argument and the string in the second argument are used to construct a variant of a parse-error record for inclusion in the parse-result that reports a general error message at the given parse position.

**(merge-result-errors <parse-result> <parse-error>) → <parse-result>**

This function propagates error information through a particular parse result. The parse-error contained in the first argument is combined with the parse-error from the second argument, and the resulting parse-error structure is returned embedded in the error field of a copy of the first argument.

## 1.2 parse-results

A parse-results record notionally describes all possible parses that can be attempted from a particular point in an input stream, and the results of those parses. It contains a parse-position record, which corresponds to the position in the input stream that this parse-results represents, and a map associating "key objects" with instances of parse-result.

Atomic input objects (known as "base values"; usually either characters or token/semantic-value pairs) are represented specially in the parse-results data structure, as an optimisation: the two fields `base` and `next` represent the implicit successful parse of a base value at the current position. The `base` field contains a pair of a token-class-identifier and a semantic value unless the parse-results data structure as a whole is representing the end of the input stream, in which case it will contain #f.

**(parse-results? <object>) → <boolean>**

This is a predicate which answers #t if and only if its argument is a parse-results record.

**(parse-results-position <parse-results>) → <parse-position> or #f**

Returns the parse-position corresponding to the argument. An unknown position is represented by #f.

**(parse-results-base <parse-results>) → (cons <kind-object> <value-object>) or #f**

If the argument corresponds to the end of the input stream, this function returns #f; otherwise, it returns a pair, where the car is to be interpreted as a base lexical token class identifier (for instance, "symbol", "string", "number") and the cdr is to be interpreted as the semantic value of the token.

**(parse-results-token-kind <parse-results>) → <kind-object> or #f**

This function returns the car (the token class identifier) of the result of parse-results-base, if that result is a pair; otherwise it returns #f.

**(parse-results-token-value <parse-results>) → <value-object> or #f**

This function returns the cdr (the semantic value) of the result of parse-results-base, if that result is a pair; otherwise it returns #f.

**(parse-results-next <parse-results>) → <parse-results> or #f**

This function returns the parse-results record representing the position in the input stream immediately after the argument's base token. For instance, if the base tokens used represented characters, then this function would return the parse-results representing the next character position; or, if the base tokens represented lexemes, then this function would return a representation of the results obtainable starting from the next lexeme position. The value #f is returned if there is no next position (that is, if the argument represents the final possible position before the end-of-stream).

**(base-generator->results <generator-function>) → <parse-results>**

This function is used to set up an initial input stream of base tokens. The argument is to be a nullary function returning multiple-values, the first of which is to be a parse-position record or #f, and the second of which is to be a base token, that is a pair of a token class identifier and a semantic value. The argument is called every time the parser needs to read a fresh base token from the input stream.

**(prepend-base <parse-position> <base-value> <parse-results>) → <parse-results>**

This function effectively prepends a base token to a particular parse-results. This can be useful when implementing extensible parsers: using this function in a suitable loop, it is possible to splice together two streams of input.

For instance, if `r` is a parse-results representing parses over the input token stream `'((b . 2) (c . 3))`, then the result of the call

```
(prepend-base #f '(a . 1) r)
```

is a new parse-results representing parses over the input stream `'((a . 1) (b . 2) (c . 3))`.

The first argument to prepend-base, the parse-position, should be either a parse-position representing the location of the base token being prepended, or #f if the input position of the base token is unknown.

**(prepend-semantic-value <parse-position> <key-object> <semantic-value> <parse-results>) → <parse-results>**

This function is similar to prepend-base, but prepends an already-computed semantic value to a parse-results, again primarily for use in implementing extensible parsers. The resulting parse-results is assigned the given parse-position, and has an entry in its result map associating the given key-object with the given semantic-value and input parse-results.

**(results->result <parse-results> <key-object> <result-thunk>) → <parse-result>**

This function is the central function that drives the parsing process. It examines the result map in the parse-results given to it, searching for an entry matching the given key-object. If such an entry is found, the parse-result structure associated with the key is returned; otherwise, the nullary result-thunk is called, and the resulting parse-result is both stored into the result map and returned to the caller of results->result.

## 1.3 parse-error

Parse-error structures represent collected error information from attempted parses. They contain two kinds of error report, following [3]: a collection of "expected token" messages, and a collection of free-format message strings.

**(parse-error? <object>) → <boolean>**

This is a predicate which answers #t if and only if its argument is a parse-error record.

**(parse-error-position <parse-error>) → <parse-position> or #f**

Retrieves the parse-position in the input stream that this parse-error is describing. A #f result indicates an unknown position.

**(parse-error-expected <parse-error>) → (list-of <object>)**

Retrieves the set (represented as a list) of token class identifiers that could have allowed the parse to continue from this point.

**(parse-error-messages <parse-error>) → (list-of <string>)**

Retrieves the list of error messages associated with this parse-error.

**(make-error-expected <parse-position> <object>) → <parse-error>**

Constructs an “expected token” parse-error record from its arguments. Called by make-expected-result (section 1.1).

**(make-error-message <parse-position> <string>) → <parse-error>**

Constructs an “general error message” parse-error record from its arguments. Called by make-message-result (section 1.1).

**(parse-error-empty? <parse-error>) → <boolean>**

Returns #t if its argument contains no expected tokens, and no general error messages; otherwise returns #f. Used internally by merge-parse-errors (section 1.3).

**(merge-parse-errors <parse-error> <parse-error>) → <parse-error>**

Merges two parse-error records, following [3]. If one record represents a position earlier in the input stream than the other, then that record is returned; if they both represent the same position, the “expected token” sets are unioned and the general message lists are appended to form a new parse-error record at the same position. The standard parsing combinators call this function as appropriate to propagate error information through the parse.

## 1.4 parse-position

A parse-position record represents a character location in an input stream.

**(make-parse-position <filename> <linenumber> <columnnumber>) → <parse-position>**

Constructs a parse-position record from its arguments. The given filename may be #f if the filename is unknown or not appropriate for the input stream the parse-position is indexing into.

**(parse-position? <object>) → <boolean>**

This is a predicate which answers #t if and only if its argument is a parse-position record.

**(parse-position-file <parse-position>) → <string> or #f**

Retrieves the filename associated with a parse-position record. Returns #f if the filename is absent or not appropriate for this input stream.

**(parse-position-line <parse-position>) → <number>**

Retrieves the line number this parse-position represents. Line numbers begin at 1; that is, all characters on the very first line in a file will have line number 1.

**(parse-position-column <parse-position>) → <number>**

Retrieves the column number within a line that this parse-position represents. Column numbers begin at 0; that is, the very first character of the very first line in a file will have line number 1 and column number 0.

**(top-parse-position <string>) → <parse-position>**

Constructs a parse-position representing the very beginning of an input stream. The argument is passed into make-parse-position as the "filename" parameter, and so may be either a string or #f.

**(update-parse-position <parse-position> <character>) → <parse-position>**

Given a position, and the character occurring at that position, returns the position of the next character in the input stream. Most characters simply increment the column number. Exceptions to this rule are: `#\return`, which resets the column number to zero; `#\newline`, which both resets the column number to zero and increments the line number; and `#\tab`, which increments the column number to the nearest multiple of eight, just as a terminal with an eight-column tab stop setting might do.

**(parse-position->string <parse-position>) → <string>**

Converts a parse-position record into an `emacs`-compatible display format. If the filename in the parse-position is unknown, the string "`<??>`" is used in its place. The result is of the form

```
filename:linenumber:columnnumber
```

for example,

```
main.c:33:7
```

**(parse-position>? <parse-position> <parse-position>) → <boolean>**

Returns #t if the first parse-position is more advanced in the input stream than the second parse-position. Either or both positions may be #f, representing unknown positions; an unknown position is considered to be less advanced in the input stream than any known position. Note that the filename associated with each parse-position is completely ignored — it is the caller's responsibility to ensure the two positions are associated with the same input stream.

## 2 Parsing Combinators

Parsing combinators are functions taking a parse-results structure and returning a parse-result structure. Each combinator attempts to parse the input stream in some manner, and the result of the combinator is either a successful parse with an associated semantic value, or a failed parse with an assocated error record.

This section describes the procedures that produce the mid-level parsing combinators provided as part of the library.

The type of a parser combinator, written in ML-like notation, would be

```
parse-results → parse-result
```

**(packrat-check-base <kind-object> <semantic-value-acceptor>) → <combinator>**

Returns a combinator which, if the next base token has token class identifier equal to the first argument ("kind-object"), calls the second argument ("semantic-value-acceptor") with the semantic value of the next base token. The result of this call should be another parser combinator, which is applied to the parse-results representing the remainder of the input stream.

The type of the semantic value acceptor, written in ML-like notation, would be

```
semanticValue → parserCombinator
```

or, more fully expanded,

```
semanticValue → parse-results → parse-result
```

These types recall the types of functions that work with monads[1].

**(packrat-check <combinator> <semantic-value-acceptor>) → <combinator>**

Returns a combinator which attempts to parse using the first argument, and if the parse is successful, hands the resulting semantic value to the semantic-value-acceptor (which has the same type as the semantic-value-acceptors passed to packrat-check-base) and continues parsing using the resulting combinator.

**(packrat-or <combinator> <combinator>) → <combinator>**

Returns a combinator which attempts to parse using the first argument, only trying the second argument if the first argument fails to parse the input. This is the basic combinator used to implement a choice among several alternative means of parsing an input stream.

**(packrat-unless <string> <combinator> <combinator>) → <combinator>**

The combinator returned from this function first tries the first combinator given. If it fails, the second is tried; otherwise, an error message containing the given string is returned as the result. This can be used to assert that a particular sequence of tokens does not occur at the current position before continuing on. (This is the "not-followed-by" matcher, from section 4.1.6 of [3].)

## 3 The packrat-parser macro

The packrat-parser macro provides syntactic sugar for building complex parser combinators from simpler combinators. The general form of the macro, in an EBNF-like language, is:

```
(packrat-parser <result-expr> <nonterminal-definition>*)
```

where

[1] and, of course, in languages like Haskell, it is the norm to implement parser combinators and related code in a monadic style.

```
<nonterminal-definition> :==
  (<nonterminal-id> (<sequence> <body-expr>+)*)
<sequence> :== (<part>*)
<part> :== (! <part>*)
       |   (/ <sequence>*)
       |   <var> <- '<kind-object>
       |   <var> <- @
       |   <var> <- <nonterminal-id>
       |   '<kind-object>
       |   <nonterminal-id>
```

Each nonterminal-definition expands into a parser-combinator. The collection of defined nonterminal parser-combinators expands to a `(begin)` containing an internal definition for each nonterminal.

The result of the whole packrat-parser form is the `<result-expr>` immediately following the `packrat-parser` keyword. Since `(begin)` forms within `(begin)` forms are flattened out in Scheme, the `<result-expr>` can be used to introduce hand-written parser combinators which can call, and can be called by, the nonterminal definitions built in the rest of the parser definition.

Each nonterminal definition expands into:

```
(define (<nonterminal-id> results)
  (results->result results 'nonterminal-id
    (lambda ()
      (<...> results))))
```

where `<...>` is the expanded alternation-of-sequences combinator formed from the body of the nonterminal definition.

An alternation (either implicit in the main body of a nonterminal definition, or introduced via a `<part>` of the form `(/ <sequence> ...)`) expands to

```
(packrat-or <expansion-of-first-alternative>
            (packrat-or <expansion-of-second-alternative>
                        ...))
```

This causes each alternative to be tried in turn, in left-to-right order of occurrence.

Whereever a `<part>` of the form "`<var> <- ...`" occurs, a variable binding for `<var>` is made available in the `<body-expr>`s that make up each arm of a nonterminal definition. The variable will be bound to the semantic value resulting from parsing according to the parser definition to the right of the arrow (the "`...`" above).

The `(! <part> ...)` syntax expands into an invocation of packrat-unless.

The "`@`" syntax in "`<var> <- @`" causes `<var>` to be bound to the parse-position at that point in the input stream. This can be used for annotating abstract syntax trees with location information.

`<part>`s of the form `'<kind-object>` expand into invocations of packrat-check-base; those of the form `<nonterminal-id>` expand into invocations of packrat-check, with the procedure associated with the named nonterminal passed in as the combinator argument.

## 4 Porting the library to other Scheme implementations

The library depends on R5RS Scheme's multiple-values support in only one place, the interface to the procedure `base-generator->results`. The macro `packrat-parser` is implemented using R5RS `syntax-rules`. The library also depends on these SRFIs:

- SRFI-1 (lists)
- SRFI-9 (records)
- SRFI-6 (basic string ports) (only used in one place, for error reporting, in the `packrat-parser` macro)

# 5 Examples

## 5.1 A base-generator for an input stream of characters

This generator reads characters from a Scheme port, maintaining an input position and producing base tokens with each character in both token class identifier and semantic value position. When using a parse-results record over an input stream built from this generator, the functions `parse-results-token-kind` and `parse-results-token-value` will both return the same character. To use the generator, pass the result of the `generator` function to `base-generator->results`.

```
(define (generator filename port)
  (let ((ateof #f)
        (pos (top-parse-position filename)))
    (lambda ()
      (if ateof
          (values pos #f)
          (let ((x (read-char port)))
            (if (eof-object? x)
                (begin
                  (set! ateof #t)
                  (values pos #f))
                (let ((old-pos pos))
                  (set! pos (update-parse-position pos x))
                  (values old-pos (cons x x)))))))))
```

## 5.2 A base-generator for a higher-level input stream of lexemes/tokens

This generator sketches the construction of more complicated lexeme or token-based substrates for the packrat parser combinators. It reads tokens from a precomputed list of token-class/semantic-value pairs; in a more realistic situation, the list of lexemes would be computed on demand. Since we're reading from a list, with no real position information available, we return #f as the first of the two values expected from the generator, to indicate "unknown location" at every step of the way.

```
(define (generator tokens)
  (let ((stream tokens))
    (lambda ()
      (if (null? stream)
          (values #f #f)
          (let ((base-token (car stream)))
            (set! stream (cdr stream))
            (values #f base-token))))))
```

### 5.3 A simple calculator

This example builds on the generator from section 5.2. It implements a simple calculator, supporting addition, multiplication, and grouping operators. The parser expects an input stream of base-tokens with token class identifiers drawn from the set `(num oparen cparen + *)`.

```
(define calc (packrat-parser expr
    (expr ((a <- mulexp '+ b <- mulexp)
           (+ a b))
          ((a <- mulexp) a))
    (mulexp ((a <- simple '* b <- simple)
             (* a b))
            ((a <- simple) a))
    (simple ((a <- 'num) a)
            (('oparen a <- expr 'cparen) a))))
```

### 5.4 An example calculator session

This session uses the definitions of `generator` and `calc` from sections 5.2 and 5.3.

```
Welcome to MzScheme version 209, Copyright (c) 2004 PLT Scheme, Inc.
> (require "packrat.ss")
> (define (generator tokens) [...])
> (define calc [...])
> (define g (generator
    '((num . 1) (+) (num . 2) (*) (num . 3))))
> (define r (calc (base-generator->results g)))
> (parse-result-successful? r)
#t
> (parse-result-semantic-value r)
7
> (define g (generator
    '((oparen) (num . 1) (+) (num . 2) (cparen) (*) (num . 3))))
> (define r (calc (base-generator->results g)))
> (parse-result-successful? r)
#t
> (parse-result-semantic-value r)
9
>
```

## References

[1] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, August 1973.

[2] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation.

[3] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Sep 2002.

[4] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.