Max's Parser Combinators: Why? How?

Max Rottenkolber <max@mr.gy>

Thursday, 15 June 2017

Table of Contents

Update, 5 September 2016: MaxPC is now in Quicklisp.

Introduction

Years ago, after failing to write a recursive descent parser for the *Mk2* (http://inters.co/geneva/mk2.html) markup language, I searched for sane ways to write parsers, and found *Smug* (http://smug.drewc.ca/), a Common Lisp library that implements *combinatory parsing* (https://en.wikipedia.org/wiki/Parser_combinator). I fell in love with this technique: it allowed me to define parsers for context-free grammars semi-declaratively using function composition. I used Smug for a while, and forked *MPC* (https://github.com/eugeneia/mpc) off it when my patches were rejected.

MPC is basically a snapshot of Smug from 2013 with some bug fixes and practical features such as input position tracking and error handling helpers. It served me well for quite some time, but at one point it struck me: its a horrible fit for my use case, which is parsing reasonably simple computer languages.

- MPC is unnecessarily powerful. It is uncertain if I will ever need to parse an ambiguous language in my lifetime. I never felt the need for `=plus`, the "non-deterministic choice operator."
- The monadic *return* and *bind* operators are too fundamental to be part of a parser generator API.
- MPC's dictionary is overwhelming. Many functions try to mimic Common Lisp (`=funcall`, `=if`, `=when`, `=unless`, `=let*`, `=prog1`, `=prog2`, `=progn`, ...), but are not particularly useful to eloquently express a grammar. Other functions do have their uses, but are too exotic to be commonly useful (`=one-to`, `=one-of`, `=none-of`, `=zero-to`, `=one-to`, ...)
- MPC's performance is mediocre, and its monadic core makes its performance difficult to predict. MPC parsers always produce result values (capture input), regardless of whether that value is at all interesting or not, leading to lots of work being performed on what is garbage, essentially.

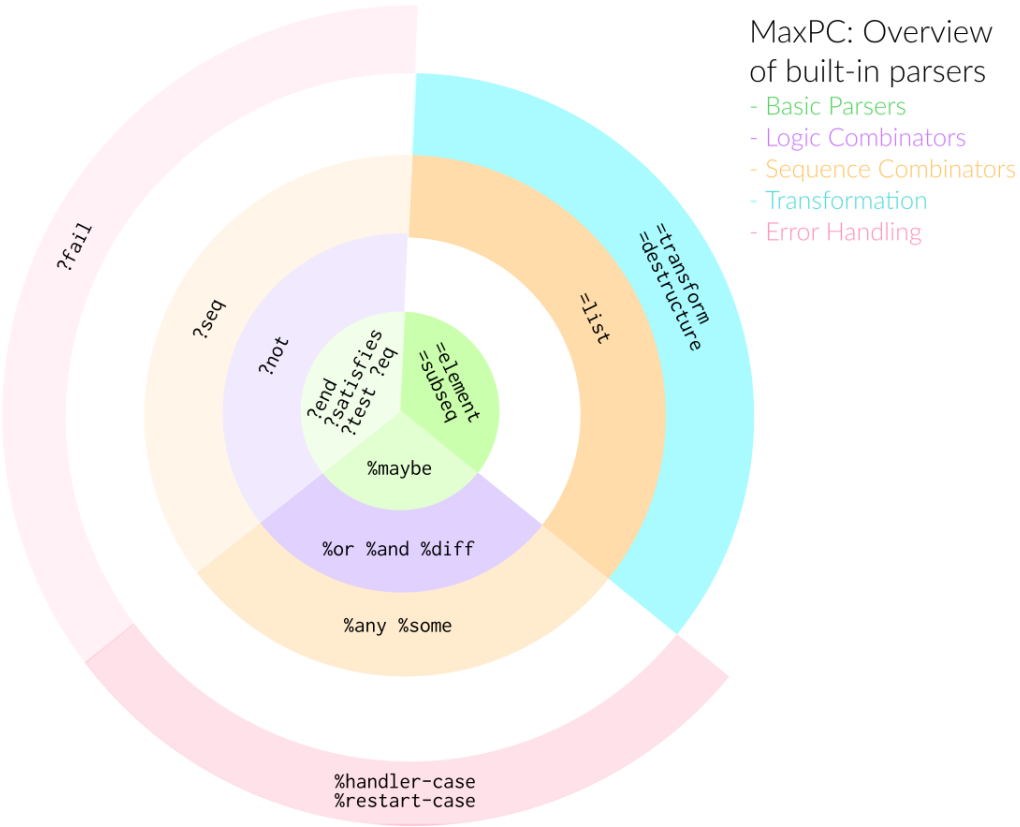
I decided to rewrite MPC from scratch with a limited scope: no support for ambiguous grammars, no claims for theoretical beauty. The new library should be simple and obvious to use, exclusively focused on effortlessly parsing the stuff that computers throw at us. Maybe it could even be reasonably efficient given its reduced feature set. Enter *MaxPC* (https://github.com/eugeneia/maxpc). I dropped the "M" from MPC, which stands for "Monadic", and inserted "Max", which stands for, well, my personal influence. MaxPC is my pragmatic takeaway from combinatory parsing, inspired by my experience in writing parsers for things such as markup and programming languages, HTTP requests, Email addresses, and URLs.

A Sane API

In MaxPC, there are five informal *categories* (/software/maxpc/api.html#section-1-3) of primitive parsers: basic parsers, logical combinators, sequence combinators, transformation, and error handling. Each category consists of only a handful of parsers, and it should be obvious which built-in to use in most situations. MaxPC models the input as a *finite* sequence of *elements*, and is aware of the *subsequence* comprised by a match (`?end`, `=element`, `=subseq`). It supports conditional matches (`?satisfies`,

`?test`, `?eq`), as well as optional matches (`%maybe`). The logical combinators (`%or`, `%and`, `%diff`, `%not`) are based on well known operations from set theory: union, intersection, difference, and negation. The sequence combinators allow for fixed sequences (`?seq`, `=list`) as well as unbounded sequences (`%any`, `%some`).

These fifteen primitives provide the core functionality expected from a parsing library, but any serious contestant must also cover transformation of matches and, of course, error handling. MaxPC features a Lispy destructuring combinator modeled after Common Lisp's `destructuring-bind` (`=destructure`), and a generic functional transformation device (`=transform`), for cases where the former is not flexible enough. Finally, MaxPC provides a mechanism to fail early (`?fail`), and two abstractions for hooking into the Common Lisp condition system (`%handler-case`, `%restart-case`). Of course, you can retrieve the input position in failure situations—line position even, for character inputs (`get-input-position`).



Furthermore, parsers are divided into three *types* (/software/maxpc/api.html#section-1-1): a parser either never produces a result, always produces a result, or maybe produces a result depending on its arguments.

This classification might seem confusing and complicated (read "not sane") at first, but hear me out.

```
(=element)        ; Always produces the next element
                  ; (if any) as its result value.

(?eq 'oddp)       ; Matches the next element if it is
                  ; ODDP, but does not produce a
                  ; result value.

(%or (?eq 'oddp) ; Matches the next element, but
     (=element)) ; only produces it as its result
                  ; if its not ODDP.
```

First of all, there is a naming convention (`?`/`=`/`%` prefixes, respectively) to make it easily distinguishable which type a given parser belongs to. Given this convention it is lexically transparent when a `%`-parser will produce a result value, and when it will not. But most of all, there is a good reason for the explicit distinction. Parsers perform two fundamental jobs, one of them is *matching* input, and the other one is *capturing* input. Originally, MPC parsers always had a result value, the default being `nil`—they were capturing by default. While uniform, an issue with that approach was that a parser equivalent to the regular expression `a*` would cons up a list of every matched element, even if there was no intent to capture these elements at all. This lead to a lot of unnecessary consing and space complexity. Furthermore, in cases where capturing was actually intended, often a list was not the desired result, but rather a string of the captured input, for instance. MaxPC solves this issue with the described protocol, and efficiently supports the mentioned cases.

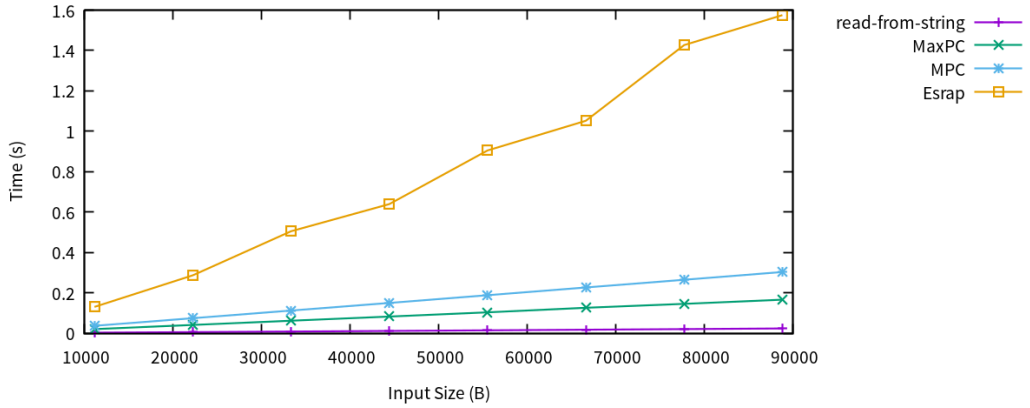
```
(%any (?eq #\a)) ; Matches the character 'a' any number of
                 ; times without additional consing.

(=subseq (%any #\a)) ; Matches the character 'a' any number
                     ; of times, and transparently produces
                     ; the matched subsequence as its
                     ; result.

(%any (=destructure (n _)              ; Matches comma
          (=list (=integer)            ; separated integers,
                 (%maybe (?eq #\,))))) ; and produces a list
                                       ; of integers as its
                                       ; result.
```

To summarize, MaxPC distinguishes between matching and capturing input, the former being the default, and capture being explicit. Because capture is explicit, and thanks to the built-in `=subseq`, both intuitive matching idioms as well as transparent capturing of subsequences can be efficient.

"Transparent"—in this case—means that the type of a captured subsequence reflects its input source. For instance, if the input source is a string, vector, or stream, then any subsequence will be a *displaced vector* (with the same element type as the input source) into that string, vector, or in case of a stream, its underlying buffer. If the input source is a list on the other hand, then any captured subsequence will be a list as well.

Many built-in combinators, such as `%any`, exhibit a dualism based on whether their arguments produce a result value, or not. This allows them to be used in both matching and capturing situations, while keeping the total number of built-ins to a minimum.

Performance Considerations



Performance of (simplistic) S-Expression parsers implemented in MaxPC, MPC (Quicklisp/mpc-20160208-git), and Esrap (Quicklisp/esrap-20160531-git) compared to `read-from-string` (CCL 1.11 on an Intel Xeon E31245 at 3.30 GHz).

MaxPC reaches its performance goal, being almost twice as fast as its predecessor MPC. There is still a sizable overhead over an optimized hand-written parser like `read-from-string`. Esrap is looking quite bad here. I only included it to make a point on Packrat parsing later on, but consider it as an example of what a reasonably popular parser generator can get away with. Of course, this comparison only tests a very specific workload, that can be considered a relatively simple language. But, according with the project objectives, I feel that S-expressions are a good approximation of your typical computer language, that does not go out of its way to punish parser generators.

Packrat Parsing

Packrat (http://pdos.csail.mit.edu/~baford/packrat/thesis/) parsing promises reduced time complexity of backtracking parsers by caching the results of each individual rule. The problem with Packrat is that its difficult to implement efficiently. Take Esrap as an example, which naively uses a hash table as the cache, and consider what *Sam* (https://github.com/eugeneia/sam) reports its doing:

```
CL-USER> (sam:profile ()
           (null (esrap:parse 'sexp-grammar::sexp
                              test-sexp)))
15%  CCL::%%EQUALHASH  <no source>
11%  GETHASH  <no source>
10%  CCL::LOCK-FREE-PUTHASH  <no source>
10%  CCL::%GROWHASH-PROBE  <no source>
 8%  CCL::%HASH-PROBE  <no source>
 8%  CCL::%LOCK-FREE-REHASH  <no source>
 8%  (:INTERNAL ESRAP::RULE/TRANSFORM ESRAP::COMPILE-RULE)...
 3%  CCL::ATOMIC-SWAP-GVECTOR  <no source>
```

All Esrap ever does is query a hash table, and that is why it performs so bad. In fact, Esrap's example S-expression parser is its own worst benchmark, because it actually never backtracks (the parse tree is decidable based on the next input character). I would guess that Esrap would actually perform competitively with MaxPC if the Packrat caching was omitted, depending on the quality of its code generator. Still, while being the worst case for a Packrat parser, the benchmark shows the inherent cost of doing Packrat parsing, which is looking up and storing computed values. A Packrat parser has to perform this bookkeeping on every rule application, so it has to be extremely cheap.

Let's put the laws of physics aside for a second, and assume that I am able to implement this bookkeeping so efficiently that it comes at zero cost. What kind of gains could I expect from my magical Packrat implementation? I *experimented* (https://github.com/eugeneia/maxpc/compare/packrat2) briefly with the workloads I am interested in, and as it turns out I happen to have whopping cache hit rate of roughly 1%. Note that this percentage does not directly translate to possible gains, because a cached result can contain the answer to computations of varying cost. Consider this parser

```
(let ((=alpha (=subseq (%any (?satisfies 'alphanumericp)))))
  (%or (=list =alpha (?eq #\.))
       (=list =alpha (?eq #\,))))
```

...which will perform twice as good using Packrat, when run on a very long input string of the form

```
"fooo...ooooo,"
```

...because the caching saves the work of applying `=alpha` twice. Of course the parser could be equivalently rewritten like so

```
(=list (=subseq (%any (?satisfies 'alphanumericp)))
       (%or (?eq #\.) (?eq #\,)))
```

. . . and would no longer require any backtracking at all. The gains from Packrat are affected by. . .

- the grammar of the language
- (de)duplication of rules
- parser rule style

In the end, Packrat parsing is an optimization strategy with a linear overhead for the general case that yields wildly fluctuating gains in a small set of special cases. Note that I have only discussed the time complexity of Packrat parsing, ignoring its huge overhead in space complexity and the inherent paging activity.

I have decided against implementing Packrat for MaxPC, because I lack workloads that would derive substantial benefits from this optimization. The hardly predictable nature of Packrat parsing makes it difficult to optimize and measure in general. I would love to be proven wrong, but I doubt that many parsers could benefit from this technique, given the cost to be amortized.

To Cons, or Not to Cons?

I initially attributed a big portion of the improvement over MPC to reduced consing. My theory was that less consing means better performance. After all, the reduced consing from making capture explicit in MaxPC gives it a clear edge over MPC. Following this logic I came up with a smart "optimization" for a common idiom using `=list` together with the macro `=destructure`. See, this pattern is used very often in MaxPC parsers

```
(=destructure (a b)
    (=list {a} {b})
  {form}...)
```

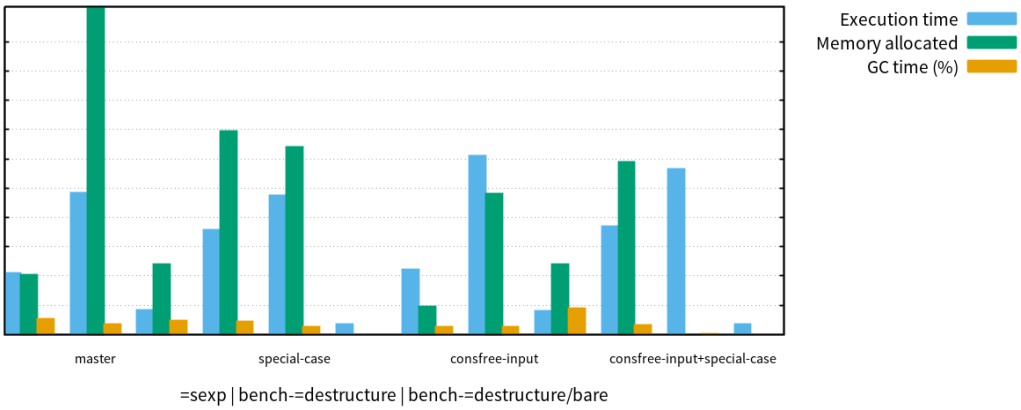
. . . and it expands to an expression like this one

```
(=transform (=list {a} {b})
            (lambda (#:G232)
              (destructuring-bind (a b)
                  #:G232
                {form}...)))
```

Now note that this expression destructures a list, even though its length is known at compile time. So consing up the list, as well as destructuring it, is really a wasted effort. I figured why not *special-case* `=destructure` on `=list` to generate garbage-free code on compile time, and save cycles. When the `=destructure` of the `special-case` branch sees that its second argument is of the form `(=list ...)`, it will expand to an expression like the following:

```
(let ((#:|df240| (lambda (a b) {form}...)))
  (lambda (#:|input239|)
    (block maxpc::=destructure-=list
      (let ((#:|value239|
             (funcall
              #:|df240|
              ;; Get value for A...
              (multiple-value-bind (#:|rest239| #:|value239|)
                  (funcall a #:|input239|)
                (unless #:|rest23998|
                  (return-from maxpc::=destructure-=list))
                (setf #:|input239| #:|rest239|)
                #:|value239|)
              (multiple-value-bind (#:|rest239| #:|value239|)
              ;; Repeat for B...
              ))))
      (values #:|input239| #:|value239| t)))))
```

Pretty huh? The expression is semantically equivalent to the previous expansion, but it does not cons or destructure a list. Proud about my smarts, I went on to measure the difference, and was thoroughly disappointed. Cons less it did, but to my surprise the impact on realistic situations was marginal, at best, even though the optimization was visible in artificially isolated experiments (`maxpc.bench:bench-=destructure/bare`). Intrigued by my findings, I hacked up the `consfree-input` branch, which replaces MaxPC's consing input abstraction with a garbage-free alternative, to get a better idea of the overall impact of consing.



Execution time vs garbage collection time of `maxpc.example-sexp` and `=destructure` (see `maxpc.bench`) on *master* (https://github.com/eugeneia/maxpc/tree/consing-2016-08-04), *special-case* (https://github.com/eugeneia/maxpc/compare/consing-2016-08-04...050a409), *consfree-input* (https://github.com/eugeneia/maxpc/compare/consing-2016-08-04...fe2c56c), and *consfree-input+special-case* (https://github.com/eugeneia/maxpc/compare/consing-2016-08-04...f0d551d) (CCL 1.11 on an Intel Core i3-4010U at 1.70 GHz).

The diagram is interesting in many ways, and I can only explain some of the peculiars. First of all, we can see that `master` is the winner in the benchmark that counts, which is reasonably real-world S-expression parsing. While the `special-case` branch indeed speeds up `=destructure` in isolated micro-benchmarks, the gains can hardly justify the increased code complexity. Its primary merit is that it proves the optimization indeed works as expected, and avoids heap allocation.

My theory that less consing means better performance is falsified, we can see that neither the total memory allocated or garbage collection time correlate with execution time. The real story is more delicate than that. On CCL, garbage collection is done in two phases. First, the garbage collector searches the stack for live objects on the heap and marks them, then it frees the objects that are not marked live. I have learned on CCL's IRC channel, that marking is expensive when there are lots of pointers in the live space (because it has to follow each pointer to mark the objects it points to), while the actual freeing is cheap. So while its cheap to generate a lot of garbage, having a lot of live pointers into the heap is what kills you. This would explain why explicit capturing gives MaxPC an edge (it creates fewer pointers), but `consfree-input` fails to move the needle despite consing so much less (thanks to a fast generational garbage collector).

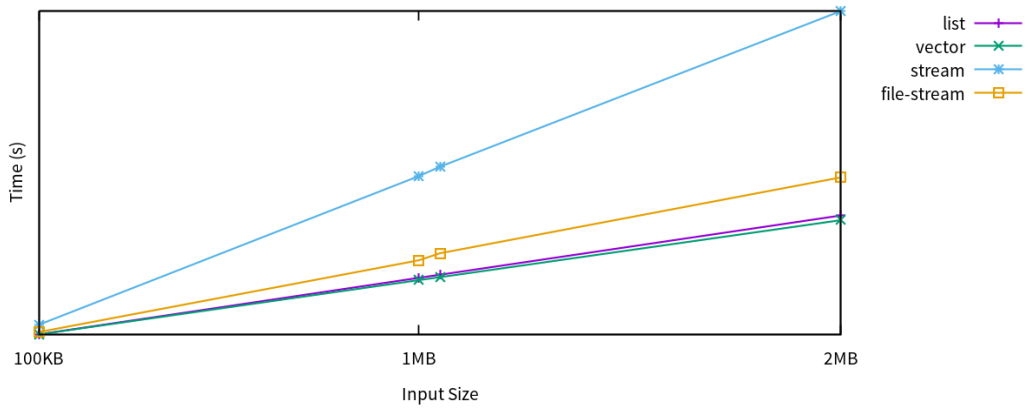
Looking at the isolated benchmarks raises even more questions. Why does `consfree-input` perform so badly in `bench-=destructure`,

but okay on `bench-=destructure/bare`? I assume `master` is triggering garbage collection more frequently, because it allocates more frequently. The garbage collector may compact, or otherwise reorganize the live data that survives garbage collection, and doing so might reduce the number of cache lines, pages, and other finite resources MaxPC needs to process that data. On the `consfree-input` branch, activating the garbage collector less frequently might actually have a negative impact on locality.

Another detail is completely inexplicable to me: the `special-case` branch allocates way more memory on the S-expression benchmark than `master`, even though isolated benchmarks show that it in fact allocates less in the code paths that differ. Riddle me this… I will update this section once I gain more insight. Until then, let it be known that outsmarting the compiler is easier said than done.

Inputs

Besides coming with support for sequences and streams out of the box, MaxPC features a public *input interface* (/software/maxpc/api.html#section-4) for you to implement. The input interface allows you to add support for new input sources and specialized implementations. The input interface is unsafe by definition. When the specification of a method says that the behavior is "unspecified" in a given situation, the implementer may choose to signal an error or open a gate to hell, at his merit. MaxPC will never use the input interface in a way that is unspecified.



Relative performance of input sources (CCL 1.11 on an Intel Xeon E31245 at 3.30 GHz). Note that MaxPC's default `maxpc.input.stream:*chunk-size*` is 1e6. E.g., 1 MB of ASCII characters.

The built-in implementations for sequences and streams naturally have different performance characteristics and trade-offs. The implementations for sequences perform better than the implementations for streams, and the different types of implementations for sequences perform practically equal. The stream implementations show more variation. Both copy the input read from a stream into a temporary buffer for fast access and backtracking, and they both grow their buffer at the same rate (which is configurable).

The `file-stream` implementation has a substantial advantage over generic streams because it reads the input in chunks at the same rate as it extends the buffer. It relies on the fact that latency for file I/O is usually low, and reading continuous sequences using `read-sequence` will seldom block for too long. If you look closely, you can see a dent in performance right after the 1 MB mark, this is where `read-sequence` is called the second time to read the input that did not fit it into the first chunk. The `file-stream` implementation is the best choice to parse from files. Minus the time spent reading, it is on par with the sequence implementations. I.e., if you add the time required to read the file into a vector to the run time of the `vector` implementation, you will get the run time of the `file-stream` implementation. An additional advantage of the stream implementations is that they will only read as much input as required for the parser to either succeed or fail.

The generic `stream` implementation reads the input one element at a time because it has no way of telling if more than a single element can be read without blocking, and therefore is substantially slower than the other implementations. Still, it is the implementation best suited for parsing from socket streams, for instance, as it will not block beyond the availability of the next input element. A bound can be configured to set an upper limit to the input read from a stream, to protect against denial of service attacks. Finally, a specific element type can be enforced, which might be necessary to deal with multivalent streams for which `stream-element-type` reports a different element type than desired.

Conclusion

Overall, I feel like MaxPC turned out well. It hits a satisfying middle ground between performance and expressiveness, and its input implementations should make it a good fit for many different situations—additionally, the public input interface grants extensibility. If you decide to use MaxPC, which I encourage, please report any issues you may encounter on GitHub or to me directly. This includes any mistakes

in this write-up, as well as insights you may have regarding my optimization riddles. Thanks for reading!