PS1: Introduction and Parser Combinators

Lior Zur-Lotan, Avi Hayoun and Hodai Goldman

November 2, 2015

Contents

1 Quasi-quotation

Quasi-Quote is a list constructor in scheme. It’s more flexible than 
quote (’), list or cons as it allows for mixing symbols with expression 
evaluations in the context of one Quasi-Quote.

The Quasi-Quote has two helper mechanisms:

1. Unquote - denoted by a comma (,)

2. Unquote-splicing denoted by a comma followed by an 'at' sign 
(,@)

When starting a Quasi-Quote, a quote context is initiated. unquote 
allows us to momentarily escape the quote context and evaluate the expression right after the unquote. unquote-splicing escapes an entire list from the Quote context as well as "unwrapping" the list.

1.1 Example

Given the following definition:

1 (define b 5)
2 (define l (list 1 2))

Note the differences in the values of the following expressions:

<table>
<thead>
<tr>
<th></th>
<th>Quote</th>
<th>Quasi-Quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>’(a b c)</td>
<td>’(a b c)</td>
</tr>
<tr>
<td>Output</td>
<td>&gt; (a b c)</td>
<td>&gt; (a b c)</td>
</tr>
<tr>
<td>Input</td>
<td>’(a ,b c)</td>
<td>’(a ,b c)</td>
</tr>
<tr>
<td>Output</td>
<td>&gt; (a ,b c)</td>
<td>&gt; (a 5 c)</td>
</tr>
<tr>
<td>Input</td>
<td>’(a ,l c)</td>
<td>’(a ,l c)</td>
</tr>
<tr>
<td>Output</td>
<td>&gt; (a ,l c)</td>
<td>&gt; (a (1 2) c)</td>
</tr>
<tr>
<td>Input</td>
<td>’(a ,@l c)</td>
<td>’(a ,@l c)</td>
</tr>
<tr>
<td>Output</td>
<td>&gt; (a ,@l c)</td>
<td>&gt; (a 1 2 c)</td>
</tr>
</tbody>
</table>
1.2 Debugging with Quasi-Quote

Quasi-Quote can be used for several things, one of which is trace-debugging (debugging functions calls). Consider the factorial function:

```lisp
(define fact (lambda (n)
  (if (<= n 0)
      0
      (* n (fact (- n 1))))))
```

This implementation has a bug. If we want to get the list of recursive calls to fact we can use Quasi-Quote like so:

```lisp
(define fact (lambda (n)
  (if (<= n 0)
      0
      '(* ,n ,(fact (- n 1))))))

> (fact 5)
(* 5 (* 4 (* 3 (* 2 (* 1 0))))))
```

2 Parser Combinators

- **Top-Down parsers** are parsers that try to build the parsing tree from a single starting non-terminal and expend it. Then they expand the new non-terminals created from the derivation and so on.

- **LL parsers** are a variation of Top-Down parsers. They consume the input from left to right and they apply the derivation rules on the leftmost non-terminal in the current stage of the parse tree. We’ll discuss LL parsers and their significance in PS the next week or two. Just know for now that we are using a parser generator package that creates LL parsers (actually LL Recursive Descent parsers).

- **Recursive Descent Parsers** are based on (perhaps mutually) recursive procedures where each procedure is usually responsible for a single derivation rule (sometimes it takes a few procedures to implement some rule and sometimes a few rules are implemented by one procedure). This creates a parser which is similar in structure to the CFG which defines the parser, and makes for very a easy-to-modify parser.

The parser you will write in this course (as part of the compiler) will be written using *parser combinators*. 


2.1 Parser Combinators according to Wikipedia

In functional programming, a parser combinator is a higher-order function which accepts several parsers as input and returns a new parser as its output. In this context, a parser is a function accepting strings as input and returning some structure as output. Parser combinators enable a recursive descent parsing strategy which facilitates modular piece-wise construction and testing. This parsing technique is called combinatory parsing. Parsers built using combinators are straightforward to construct, readable, modular, well-structured and easily maintainable.

Remark. What this means in essence is that when creating a parser using parser combinators, we will construct only very basic, simple, production rules explicitly, and define more complex production rules by iteratively composing simpler parsers. An example follows.

2.2 Example: Simple arithmetic expressions

In this example, we will construct a parser for a simple grammar using parser combinators. The grammar will be of chained summation and subtraction of numbers, e.g. $100 - 5 + 72 - 32 - 1 + 0 + 7$.

Remark. As we will construct the parser using parser combinators, we will want to define only the most basic production rules of the grammar explicitly.

2.2.1 Plan

1. Define a parser for operations ($+$ or $-$).

2. Define a parser for digits.

3. Define a parser for numbers (we will stick to natural numbers for simplicity’s sake). Do this using composition rules.

4. Define a parser for an expression.

5. Expand the expression parser into recursive parser for a chained expressions.

6. Output valid scheme code which will evaluate to the correct result.

Remark. For simplicity’s sake, the parser will assume right associativity of the arithmetical expression, so $1 + 2 - 3$ will evaluate to $1 + (2 - 3)$. 

3
2.2.2 The operator parser

\(<\text{op}> \rightarrow \{+\} | \{.\}\)
To demonstrate the fact that the .ml implementation is comparable (and, in fact, almost identical) to the scheme implementation, compare the <nat> parser defined in Scheme above to the _nat parser defined in Ocaml below:

```plaintext
let _nat =
  disj
  (pack
    (char '0')
    fun _ -> 0)
  (pack
    (caten
      _digit_1_9
      (star
        _digit_0_9))
    fun (x, xs) -> int_of_string (list_to_string x::xs))
```

### 2.2.5 The arithmetic expression parser

<expr> → <expr> <op> <nat> | <nat>

```plaintext
(define <expr>
  (disj
    (pack
      (caten
        (delayed (lambda () <expr>)) ; allow recursion
        (caten
          <op>
          <nat>)))
      (lambda (s)
        (let ((<expr> (car s))
          (op (cadr s))
          (n (cddr s)))
          (', (op, expr, n))))
      <nat>)))
```

**Left recursion:** Immediate left recursion occurs in rules of the form \( A \to A\alpha|\beta \) where \( \alpha \) and \( \beta \) are sequences of nonterminals and terminals, and \( \beta \) doesn’t start with \( A \).

The production \(<\text{expr}> \to <\text{expr}> <\text{op}> <\text{nat}>\) is a left recursive production rule. In \( LL \) parsers (such as the one we are defining in the example above), such production rules cause an infinite loop. We will discuss left recursion in more depth in a future class. For now, it is enough to realize that in the case of our grammar, the result is the same if we make the rule right recursive instead: \(<\text{expr}> \to <\text{nat}> <\text{op}> <\text{expr}>\).
2.2.6 The arithmetic expression parser - fixed

\[ <\text{expr}> \rightarrow <\text{nat}> <\text{op}> <\text{expr}> | <\text{nat}> \]

1 (define <expr>
2   (disj
3     (pack
4       (caten
5         <\text{nat}>
6       (caten
7         <\text{op}>
8         (delayed (lambda () <\text{expr}>))) ; allow recursion
9       (lambda (\text{expr} rest)
10         (let ((\text{op} (car rest))
11             (n (cdr rest)))
12             (,\text{op} ,\text{expr} ,n)))
13       <\text{nat}>))

Remark. If we would move the <\text{nat}> at line 13 to before line 3 (make it the first argument to disj), this parser would consume a <\text{nat}> token and return the rest of the string as a remainder even if that remainder fits the rest of the rule (i.e. "<\text{op} > <\text{expr}>."). This is because parsers made using parsing combinators have a defined order of precedence when parsing disjunctions (it's not commutative).

2.3 Packing

The \text{pack} function is also a parser combinator, meaning it takes a parser as input and composes it with some extra functionality. Unlike the other combinators (such as \text{caten} and \text{disj}), the packing combinator doesn't compose multiple parsers together, but rather allows us to transform the output of a parser before returning it.

As an example, consider the code (\text{pack} (\lambda () 0)) from the <\text{nat} > parser. In this case, we are taking whatever is returned from the parser (char #\0) and returning the number 0 instead.