Parsing

Geoffrey Matthews

December 25, 2022

Modeling Syntax

- `3 + 4`
- `3 4 +`
- `(+ 3 4)`

Modeling Syntax

- `3 + 4`
- `3 4 +`
- `(+ 3 4)`

```
1 (define-type AE
2   [num (n number?)]
3   [add (lhs AE?) (rhs AE?)]
4   [sub (lhs AE?) (rhs AE?)])
```

```
1 (add (num 3) (num 4))
```

Modeling Syntax

- `(3 - 4) + 7`
- `3 4 - 7 +`
- `(+ (- 3 4) 7)`

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])
```

Modeling Syntax

- `(3 - 4) + 7`
- `3 4 - 7 +`
- `(+ (- 3 4) 7)`

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])
```

```
(add (sub (num 3) (num 4))
     (num 7))
```

Read

```
> (read)
hello
'hello
> (read)
(+ 3 (- 4 5))
'(+ 3 (- 4 5))
> 'hello
'hello
> '(+ 3 (- 4 5))
'(+ 3 (- 4 5))
```

Read in Python

```
def parse(s):
    skip_blanks(s)
    if s[0] in string.digits:
        return parse_number(s)
    elif s[0] not in punctuation:
        return parse_symbol(s)
    elif s[0] == '(':
        s.pop(0)
        return parse_list(s)
```

Read in Python

```
1  def parse_number(s):
2      skip_blanks(s)
3      n = 0
4      while s and s[0] in string.digits:
5          n = n*10 + int(s.pop(0))
6      return n
7
8  def parse_symbol(s):
9      skip_blanks(s)
10     w = ''
11     while s and s[0] not in punctuation:
12         w += s.pop(0)
13     return w
```

Read in Python

```
1 def parse_list(s):
2     skip_blanks(s)
3     if s[0] == ')':
4         s.pop(0)
5         return []
6     car = parse(s)
7     cdr = parse_list(s)
8     return [car] + cdr
```

Homework: rewrite read in Scheme

- Translate into pure functional style:
  - No assignments statements.
  - No destructive operations (e.g. `pop`).
  - No loops except recursion.
- You will need to return two values:
  - the parsed object
  - the remainder of the string
- You need only handle the subset of Scheme handled by our Python implementation.

Parsing Scheme into datatypes

```
;; parse : sexp -> AE
;; to convert s-expressions into AEs

(define (parse sexp)
  (cond [(number? sexp) (num sexp)]
        [(list? sexp)
         (case (first sexp)
           [(+) (add (parse (second sexp))
                     (parse (third sexp)))]
           [(-) (sub (parse (second sexp))
                     (parse (third sexp)))])]))
```

```
> (parse '(+ 2 2))
(add (num 2) (num 2))
> (parse '(+ (- 2 3) 5))
(add (sub (num 2) (num 3)) (num 5))
> (parse '(+ (- 2 3) (- 12 3)))
(add
 (sub (num 2) (num 3))
 (sub (num 12) (num 3)))
```

Jobs for `read` and `parse`

- `read` will reject the first,
- `parse` will reject the second:

```
1   (+ 2 3 }
2   (+ 2 3 4)
```

Interpreting *vs.* Interpreted Language

- We will be using Scheme syntax for both languages.
- Racket allows you to use any brackets: `()` `[]` `{}`
- We will use only braces `{}` in the interpreted languages.
- Our interpreter will be written useing round and square brackets `()` `[]`
- Scheme program:

```
(+ (- 3 2) 5)
```

- Interpreted language program:

```
{+ {- 3 2} 5}
```