How to build language processors in a pure functional programming language

Recognizers – for terminals

```
term   wd (tok:toks) = [toks], if tok = wd
                     =   [], otherwise
term   x      []     =   []

e.g.
term “one” [“one”,“two”,“three”]
                           => [[“two”,”three”]]

term “two” [“one”,“two”,“three”] => []

Now  one  = term “one”
     two  = term “two”
     plus = term “+”    etc
```

Recognizers – alternatives

```
(p $orelse q) toks = p toks ++ q toks

e.g.

one $orelse two [“one”,”four”]   => [[“four”]]

one $orelse two [“two”,”six”]    => [[“six”]]

one $orelse two [“three”,”six”]    => []

num = one $orelse two $orelse three …
```

Recognizers – sequence

```
(p $then q) toks = concat (map p (q toks))

e.g.

(one $then two) [“one”,”two”,”three”] => [[“three”]]

(one $then two) [“one”,”six”,”four”]  => []

num $then num [“one”,”two”,”three”]   => [[“three”]]
```

Recognizers – complex

```
add_seq = num $orelse (num $then plus $then add_seq)

e.g.

add_seq [“one”,”+”,”three”,”+”,“six”]
                  => [[“+”,”three”,“+”,”six”],
                      [“+”,”six”],
                      []    ]

add_seq [“plus”,”six”,”four”]    => []
```

Interpreters – for terminals

```
term (wd,val) (tok:toks) = [(val,toks)], if tok = wd
                         =   [], otherwise
term   x       []        =   []

e.g.
term (“one”,1) [“one”,“two”,“three”]
                              => [(1,[“two”,”three”]])

Now  one  = term (“one”,1)
     two  = term (“two”,2)
     plus = term (“+”,(+))    etc
```

Interpreters – alternatives

```
(p $orelse q) toks = p toks ++ q toks

e.g.

one $orelse two [“one”,”four”]   => [(1,[“four”])]

one $orelse two [“two”,”six”]    => [(2,[“six”])]

one $orelse two [“three”,”six”]    => []

num = one $orelse two $orelse three …
```

Interpreters – sequence

```
(p1 $then p2) toks
      = [((v1,v2),t2) | (v1,t1) <- p1 toks;
                          (v2,t2) <- p2 t1]
e.g.
(one $then two) ["one","two","three"]
                          => [((1,2),["three"])]

(one $then plus $then two)
               ["one","plus","two"."three"]
                       => [(1,((+),2)),["three"])]
```

Interpreters – complex

```
add_seq
    = num
      $orelse
      ((num $then plus $then add_seq) $apply_rule app_op)

(p $applyrule f) inp    = [ (f v, r) | (v, r) <- p inp ]

app_op (v1, (op, v2)) = op v1 v

e.g
add_seq [“one”,”+”,”three”,”+”,“six”]
                  => [(1,[“+”,”three”,“+”,”six”]),
                      (4,[“+”,”six”]),
                      (10,[])   ]
```