PS3: Using pc.ml

Lior Zur-Lotan and Tal Barami

October 30, 2018

Contents

1 Expressions language
   1.1 Plan ........................................... 1
   1.2 Infrastructure .................................. 2
   1.3 The tokens .................................... 2
   1.4 Numbers Parser ................................ 4
   1.5 Expressions parsers ............................ 4
      1.5.1 Fold-left/Fold-right ....................... 6

1 Expressions language

In our last session we defined a CFG for our expressions language. We removed the ambiguity and the left recursion from the intuitive CFG we proposed and broke our language into tokens. This was the result:

Token parsers

1. <lparen> → <space>* '(' <space>*
2. <digit> → '0'-'9'
3. <rparen> → <space>* ')' <space>*
4. <expop> → '^' <space>*
5. <mulop> → <space>* '*' <space>*
6. <addop> → <space>* '+' <space>*

Production rules

1. <num> → <digit>+
2. <paren> → <num> | <lparen> <expr> <rparen>
This week, we’ll implement the parsers for this language using `pc.ml`, as well as an evaluation function for the resulting parse tree.

### 1.1 Plan

1. Define our internal representation - the internal representation of the input
2. Define the "token" parsers (digits, '+' '-' )
3. Define a parser for numbers.
4. Define a parser for expressions.

### 1.2 Infrastructure

```ml
type expr = Exp of expr * expr | Mul of expr list | Add of expr list | Num of int;;
```

Our AST will be composed of `expr` nodes. This is our internal representation.

### 1.3 The tokens

```ml
let _spaces_ = star (char ' ');;
let _digit_ = range '0' '9';;

let _lparen_ =
  let _lp_ = char '(' in
  let _spaced_ = caten (caten _spaces_ _lp_) _spaces_ in
  pack _spaced_ (fun ((l, p), r) -> p);;

let _rparen_ =
  let _rp_ = char ')' in
  let _spaced_ = caten (caten _spaces_ _rp_) _spaces_ in
```
At line 1, we use the `char` parser constructor to produce a parser that accepts only a single char, and then we apply the `star` operator to create a parser that can accepts zero or more of that single char.

At line 3 we use the `range` parser constructor that takes two chars and returns a parser that accepts a single char in the given character range.

At line 7 we use the `caten` operator to catenate together two parsers into a new parser that accepts the catenation of their languages.

At line 8 we use the `pack` combinator to transform the accepted tokens (characters) into an internal representation used in our parse tree. The `caten` operator produces tuples, the type of the tuple elements depends on the packing of the underlying parsers. Since we nest two `caten` operators, our packing function takes two nested tuples. We only want the data stored in the `_lp_` parser, so the function’s body only returns that. When parsers aren’t packed, their input tokens are outputted as is (in the case of our pc.ml, that means chars, char lists, char tuples etc.).

On lines 7, 12, 17 and 22 we define the delimiters for `<lparen>`, `<rparen>`, `<mulop>` and `<addop>` respectively - they are delimited on both sides with 0 or more spaces. At line 27 we define the delimiter for `<expop>` which only allows a space after the operator sign.

Because we defined all the tokens and their delimiters here, we don’t have to worry about spaces or other delimiters going forward - we can build the rest of our parsers using these “token parsers”.

**Remark.** Our code for `<lparen>`, `<rparen>`, `<mulop>` and `<addop>` is essentially almost the same. A better way to write this is to factor out their common portions.
See [expr.ml](expr.ml) for an example how to do this.

### 1.4 Numbers Parser

```ml
let _num_ =
  let _digits_ = plus _digit_ in
  pack _digits_ (fun (ds) -> Num (int_of_string (list_to_string ds)));
```

At line 2 we apply the `plus` operator to produce a parser that takes one or more digits. Note that we accept numbers with leading zeros, like `0123`. While, technically, natural numbers mustn't start with 0, we do this here for simplicity's sake. Because `plus` returns a list and `_digit_` returns a digit (int), the `_digits_` parser will return a list of digits (ints).

At line 3 We first turn our list of digits into a string using `list_to_string` (defined in `pc.ml`) and then we use the built-in function `int_of_string` to convert it to an int. Finally, we apply the `Num` type constructor to create an `expr` since that's what our parse tree holds.

### 1.5 Expressions parsers

```ml
let rec _paren_ s =
  let _nested_ = pack (caten (caten _lparen_ _expr_ _) _rparen_)
    (fun ((l, e), r) -> e) in
  (disj _num_ _nested_) s

and _exp_ s =
  let _head_ = star (pack (caten _paren_ _expop_) (fun (p, o) -> p)) in
  let _tail_ = _paren_ in
  let _chain_ = caten _head_ _tail_ in
  let _packed_ = pack _chain_
    (fun (hd, tl) -> match hd with
      | [] -> tl
      | hd -> List.fold_right
        (fun e aggr -> Exp (e, aggr))
        hd
        tl) in

  _packed_ s

and _mul_ s =
  let _head_ = _exp_ in
```
First thing to note is that this set of parsers is mutually recursive - `<paren>` depends on `<expr>` (see line 2) which indirectly depends on `<paren>`. For this reason we define all these parsers with the `let rec` keyword at line 1, and we chain functions with the `and` keyword instead of subsequent `lets` (for example at line 7).

Another thing to note is the way we define our parsers to take a parameter (for instance look at line 40). An alert programmer would notice that `_expr_ s = _add_ s;;` should be equivalent to `_expr_ = _add_;;` (this equivalency is called \(\eta\)-reduction). The reason we need to define these parsers with that superfluous parameter \(s\) is because of a limitation in in Ocaml's `let rec/and` mechanism. You can only define functions with the `let rec/and` mechanism, since defining variables recursively can result is self contradictions in the type system. To work around this limitation we define the parsers as a function taking a parameter \(s\) and simply passing it along to the internal parser. This doesn't affect our function's signature thanks to Ocaml's auto-currying.

If you would define `_expr_` as a variable equal to the function `_add_` (i.e. `and _expr_ = _add_;;`), you would get the following error:

```
and _expr_ = _add_;;
```

**Error:** This kind of expression is not allowed as right-hand side of `let rec`
1.5.1 Fold-left/Fold-right

Last thing to note is our use of `fold_right` at line 15. The functions `fold_left` and `fold_right` and `map` in the module `List` are the most versatile group of functions for any list manipulation you will want to perform.

`map` is the origin of SQL’s `select` and is used to project a list (i.e. create a new list by performing an action on each of the list’s elements). It’s the functional counterpart of `foreach`.

`fold-left` and `fold-right` are aggregator functions. This means they take a list and iterate it, applying an aggregator function to each element in order to create some result. This result can be an element, a list (not necessarily the same length as the input list) or anything else. Any time you want to manipulate a list and your output is not a list (or a list of different length), you should consider using fold - it will often yield shorter and easier to understand code. A mathematical formulation of `fold-left` and `fold-right` are:

\[
\text{fold-left}(f, \text{base}, [x_1; x_2; \ldots; x_n]) = f(f(f(\text{base}, x_1), x_2), \ldots, x_n)
\]

\[
\text{fold-right}(f, [x_1; x_2; \ldots; x_n], \text{base}) = f(x_1, f(x_2, \ldots f(x_n, \text{base})\ldots))
\]

For instance, let’s evaluate the value of our parsed expressions:

\[
\text{let } \text{exp x y} = (\text{float_of_int x}) ** (\text{float_of_int y}) |> \text{int_of_float}
\]

\[
\text{let rec eval = function}
\]

\[
| \text{Num}(n) -> n
\]

\[
| \text{Exp}(b,e) -> \text{exp} (\text{eval b}) (\text{eval e})
\]

\[
| \text{Mul}(l) -> \text{List.fold_left (fun prod operand -> prod * (eval operand)) 1 l}
\]

\[
| \text{Add}(l) -> \text{List.fold_left (fun sum operand -> sum + (eval operand)) 0 l};
\]

To see how this works, we’ll take a look at the `Mul(1)` pattern. The above definitions we can see that we get (using `mul` as a the multiplication function):

\[
\text{fold-left}(\text{mul}, 1, [e_1, e_2, \ldots, e_n]) = \text{mul}(\ldots\text{mul}(\text{mul}(1, \text{eval}(e_1)), \text{eval}(e_2))\ldots, \text{eval}(e_n))
\]

And converting this expression to infix multiplication with the `*` notation:

\[
\text{mul}(\text{mul}(\ldots\text{mul}(\text{eval}(e_1), 1), \ldots, \text{eval}(e_{n-1}), \text{eval}(e_n)) = 1*\text{eval}(e_1)*\text{eval}(e_2)*\ldots*\text{eval}(e_n))
\]

And it’s easy to see (and prove) this is the correct implementation.