Lab 12

Parser combinators in Haskell - Part 2

Goals

In this lab you will learn to:

1. Understand LISP syntax
2. Write a parser for a subset of LISP

Resources

Table 12.1: Lab Resources

Resource	Link
An intuition for LISP syntax	https://stopa.io/post/265
COMMON LISP: A Gentle Introduction to Symbolic Computation	https://www.cs.cmu.edu/~dst/LISPBook/book.pdf
Structure and Interpretation of Computer Programs	https://opendocs.github.io/sicp/sicp.pdf
Functional parsing	https://youtu.be/dDtZLm7HIJs
Understanding parser combinators	https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/
Understanding parser combinators: a deep dive - Scott Wlaschin	https://www.youtube.com/watch?v=RDalzi7mhdY

12.1 Intro to LISP

12.1.1 Polish notation (prefix notation)

Concept 12.1.1: Polish notation (prefix notation)

Polish notation or Prefix notation is a way to write mathematical or logical expressions, where the operator is *before* the operands. For example instead of writing $1+2$, we write $+\ 1\ 2$ and instead of $(2-1)+(3*4)$ we write $+\ -\ 2\ 1\ *\ 3\ 4$. Notice that with this notation, *we can omit all parentheses* without introducing any ambiguity, though we can still use parentheses to improve clarity: $+\ (-\ 2\ 1)\ (*\ 3\ 4)$.

You've already used this notation with almost all programming languages, where function calls use prefix notation (since infix notation works only for binary functions). Thus in this style, we would write a function call as: `f a b c`, where `f` is the name of the function and `a, b, c` are its parameters.

If we want to call multiple functions, we can use the same notation, but with parentheses: `f (g a b) (h a) c`, which you've already seen while using Elm, Haskell and SML! Instead of placing the arguments in parentheses and after the function name, we place the function name and the arguments between the parentheses, with the function name being the first element.

12.1.2 LISP programs

LISP is a very simple, yet powerful language: a program is just a nested list of expressions. Expressions can be lists or *atoms*, which are numbers or *symbols* (identifiers). This allows LISP programs to be also considered as *data*, that can be manipulated by the program itself in a similar fashion to reflection in Java.

For example, the following are all valid LISP expressions: `(+ 1 2)`, `(+ 1 2 3)`. In each case, `+` is the first element in the list and it corresponds to a function call with an arbitrary number of arguments. We can also use multiple functions, using parentheses to denote function application: `(+ (- 2 1) (* 1 2 3))`.

The final aspect is that because the program is actually also data, we need a way to differentiate between code that we want to be executed (i.e. function to be called on its arguments) and data definition. To mark a list or a symbol as data, we use *quoting*, with the `'` character. For example, to create the equivalent of the following Haskell list `[1, 2, 3]` in LISP, we write `'(1 2 3)`, which means that this expression shouldn't be evaluated (i.e. don't treat `1` as function applied on `2` and `3`).

Now that we can define lists as data, we can also define functions to manipulate lists: `car`, which returns the first element of a list and `cdr`, which skips the first element and returns rest of the elements of the list. So `(car '(1 2 3))` is 1, `(cdr '(1 2 3))` is (2 3) and `(car (cdr '(1 2 3)))` is 2.

12.2 Assignment

Deadline: Monday, January 11, 23:55

12.3 Submission instructions

1. Unzip the `MiniLisp.zip` folder. You should find:
   - 2 files in the `src` folder:
     - `Parser.hs` - for the basic parsing library that you completed at the last lab
     - `MiniLisp.hs` - for the LISP parser that you partially completed at the last lab
2. Edit the first line of each of the source files as described in the comments.
3. Edit the source files in the `src` folder with your solutions.
4. When done, zip this `MiniLisp` folder and name the zip archive with the following format:

$$MiniLisp\_\langle FirstName\rangle\_\langle LastName\rangle\_\langle Group\rangle$$

   Examples of valid names:
   - `MiniLisp_John_Doe_30432.zip`
   - `MiniLisp_Ion_Popescu_30434.zip`
   - `MiniLisp_Gigel-Dorel_Petrescu_30431.zip`

   Examples of invalid names:
   - `Solutions.zip`
   - `MiniLisp.zip`
   - `Solutii_MiniLisp_Ion_Popescu.zip`

12.3.1 Preparation

Update your existing `MiniLisp.hs` file from the previous lab by adding the new definitions from the `MiniLisp.hs` file from this lab.

12.3.2 Exercises

Exercise 12.3.1 3p

Create a parser `sepBy sep p` with the signature `sepBy :: Parser a -> Parser b -> Parser [b]`

```
Haskell REPL

> runParser (sepBy (char ',') ident) "a,b,c"
Success ["a", "b", "c"] ""
> runParser (sepBy (char ',') ident) "abc"
Success ["abc"] ""
> runParser (sepBy (char ',') ident) "a,,b"
Success ["a"] ",,b"
> runParser (sepBy ws ident) "a  b c     d"
Success ["a", "b", "c", "d"] ""
```

Exercise 12.3.2 — 2p

Create a parser `lispAtom :: Parser lispAtom` which parses a LISP atom (either number or symbol).

```
Haskell REPL

> runParser lispAtom "abc"
Success (Symbol "abc") ""
> runParser lispAtom "123"
Success (Number 123) ""
```

Exercise 12.3.3 — 2p

Create a parser `lispList :: Parser [LispValue]` which parses a list of LISP values, using the `lisp` function (that you will write in Exercise 12.3.4).

```
Haskell REPL

> runParser lispList "(a b c)"
Success [Atom (Symbol "a"),Atom (Symbol "b"),Atom (Symbol "c")] ""
> runParser lispList "(a b c("
Error "Unexpected character '('"
```

Exercise 12.3.4 — 2p

Create a parser `lisp :: Parser LispValue` which parses a LISP value, using the `lispAtom` and `lispList` functions.

```
Haskell REPL

> runParser lisp "(a b c)"
Success (List [Atom (Symbol "a"),Atom (Symbol "b"),Atom (Symbol "c")]) ""
> runParser lisp "123"
Success (Atom (Number 123)) ""
> runParser lisp "abc1"
Success (Atom (Symbol "abc1")) ""
```