P1. Lexing & Parsing

Kelley van Evert
s4046854

Language = Haskell

- Previous choice was Ocaml
- “Haskell has too much sugar”, but in the end it convinced me anyway
  - e.g.
    - type classes
    - using data constructors as ordinary functions

```
data Parser s a = Parser (s -> [(a, s)])

instance Monad (Parser s) where
  return a  = Parser $ \cs -> [(a, cs)]
  p >>= f   = Parser $ \cs -> concat [parse (f a) cs' | (a, cs') <- parse p cs]
  fail _    = Parser $ \cs -> []

tuple = [ Tuple e1 e2 | “(“ ← next, e1 ← expr, “,” ← next, e2 ← expr, “)” ← next ]
```

Method = Parser Combinators

- From scratch (I wanted to learn)
  - ±150 lines each (including all the signatures & fluff):
    - Generic combinator library
    - Lexer
    - Expression parsing
    - Program parsing
- Pipelined:
  - Lexer : String → [Token]
  - Parser : [Token] → Program
  - (Separated)
    ExpressionParser : [Token] → Expr

Method = Parser Combinators

- Hutton, Graham, and Erik Meijer. "Monadic parser combinators." (1996).

- Ridge, Tom. "Simple, functional, sound and complete parsing for all context-free grammars." (2011).
  - “Don't transform the grammar, change the combinators!”
  - Though I didn't use this in the end

Expression Parsing

- Danielsson, Nils Anders, and Ulf Norell. "Parsing mixfix operators." (2011).
  - Mixfix operators with DAG as precedence relation
  - Works unambiguously given some basic assumptions

- => Transformed expression grammar, implemented with combinators
  - Not fully abstracted (didn't get to it, and not really necessary anyway)

```
Expr   ::= Cons | Or | And | Not | Eq | Comp | Add | Mul | Neg | Closed

Cons   ::= (Cons↑ “:”)+ Cons↑
Cons↑  ::=        Or | And | Not | Eq | Comp |                   Closed

Or     ::= (Or↑ “||”)+ Or↑
Or↑    ::=             And | Not | Eq | Comp |                   Closed

And    ::= (And↑ “&&”)+ And↑
And↑   ::=                   Not | Eq | Comp |                   Closed

Not    ::= “!”+ Not↑
Not↑   ::=                         Eq | Comp |                   Closed

Eq     ::= (Eq↑ (“==” | “!=”))+ Eq↑
Eq↑    ::=                              Comp | Add | Mul | Neg | Closed

Comp   ::= (Comp↑ (“<” | “<=” | “>” | “>=”))+ Comp↑
Comp↑  ::=                                     Add | Mul | Neg | Closed

Add    ::= Add↑ ((“+” | “-”) Add↑)+
Add↑   ::=                                           Mul | Neg | Closed

Mul    ::= Mul↑ ((“+” | “-”) Mul↑)+
Mul↑   ::=                                                 Neg | Closed

Neg    ::= “-”+ Neg↑
Neg↑   ::=                                                       Closed

Closed ::= Ident (“.” Field)* | Basic | FunCall | “(“ Expr ”)”
```

```haskell
parse_comp :: Parser [Token] Expr
parse_comp = chainr1 parse_comp_up (parse_binop ["<", ">", "<=", ">="])

parse_comp_up :: Parser [Token] Expr
parse_comp_up = first $ list_or [
                  parse_add,
                  parse_mul,
                  parse_neg,
                  parse_closed
                ]


parse_add :: Parser [Token] Expr
parse_add = chainl1 parse_add_up (parse_binop ["+", "-"])

parse_add_up :: Parser [Token] Expr
parse_add_up = first $ list_or [
                 parse_mul,
                 parse_neg,
                 parse_closed
                ]

parse_mul :: Parser [Token] Expr
parse_mul = chainl1 parse_mul_up (parse_binop ["*", "/", "%"])

parse_mul_up :: Parser [Token] Expr
parse_mul_up = first $ list_or [
                 parse_neg,
                 parse_closed
                ]

parse_neg :: Parser [Token] Expr
parse_neg = [ foldr E_UnOp e (map (const "-") ops) | ops <- many1 (element $ T_Op "-"),
                                                     e <- parse_neg_up ]

parse_neg_up :: Parser [Token] Expr
parse_neg_up = first $ list_or [
                 parse_closed
                ]
```

Demo

(Questions)