Language.CoreErlang.Parser

```haskell
-----------------------------------------------------------------------------
-- |
-- Module      :  Language.CoreErlang.Parser
-- Copyright   :  (c) Henrique Ferreiro García 2008
--                (c) David Castro Pérez 2008
--                (c) Eric Bailey 2016
-- License     :  BSD-style (see the file LICENSE)
--
-- Maintainer  :  Alex Kropivny <alex.kropivny@gmail.com>
-- Stability   :  experimental
-- Portability :  portable
--
-- CoreErlang parser.
-- <http://www.it.uu.se/research/group/hipe/cerl/>

-----------------------------------------------------------------------------
module Language.CoreErlang.Parser (
  -- * Lexical definitions
  sign, digit,
  uppercase, lowercase, inputchar, control, space, namechar,
  escape,
  octaldigit, octal, ctrlchar, escapechar,
  -- * Terminals
  integer, float, atom, echar, estring, variableName,
  -- * Non-terminals
  annotatedModule, module_, moduleHeader, exports,  attributes, moduleAttribute,
  funDef, funName, fun, constant, literal, nil, variables,
  ecase, clause, guard, letrec, app, modCall, primOpCall, etry,
  receive, timeout, eseq, ecatch,
  -- * Static semantics
  -- ** Annotations
  annotation, annotated,
  -- ** Module definitions
  parseModule,
  -- * Parse Error (from -- "Text.Parsec")
  ParseError
  ) where

import           Language.CoreErlang.Syntax

import           Prelude                               hiding (exp)

import           Control.Monad                         (liftM)
import           Data.Char                             (chr, isControl)
import           Numeric                               (readOct)

import           Text.Parsec.Char                      (char, lower, noneOf,
                                                        oneOf, satisfy, upper)
import qualified Text.Parsec.Char                      as PChar
import           Text.ParserCombinators.Parsec         (ParseError, Parser,
                                                        choice, count, eof,
                                                        many, many1, option,
                                                        parse, try, (<|>))
import           Text.ParserCombinators.Parsec.Language
import           Text.ParserCombinators.Parsec.Token   (TokenParser,
                                                        makeTokenParser)
import qualified Text.ParserCombinators.Parsec.Token   as Token
```

Grammar

Lexical definitions

$$
\begin{aligned}
\textit{sign} &::= +|-\\
\textit{digit} &::= 0|1|\ldots|9\\
\textit{uppercase} &::= \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{\textbackslash u00c0} \mid \ldots \mid \texttt{\textbackslash u00d6} \mid \texttt{\textbackslash u00d8} \mid \ldots \mid \texttt{\textbackslash u00de}\\
\textit{lowercase} &::= \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{\textbackslash u00df} \mid \ldots \mid \texttt{\textbackslash u00f6} \mid \texttt{\textbackslash u00f8} \mid \ldots \mid \texttt{\textbackslash u00ff}\\
\textit{inputchar} &::= \text{any character except } \texttt{CR} \text{ and } \texttt{LF}\\
\textit{control} &::= \texttt{\textbackslash u0000} \mid \ldots \mid \texttt{\textbackslash u001f}\\
\textit{space} &::= \texttt{\textbackslash u0020}\\
\textit{namechar} &::= \textit{uppercase} \mid \textit{lowercase} \mid \textit{digit} \mid \texttt{@} \mid \texttt{\_}\\
\textit{escape} &::= \texttt{\textbackslash}\ (\textit{octal} \mid (\texttt{\^{}}\ \textit{ctrlchar}) \mid \textit{escapechar})\\
\textit{octaldigit} &::= \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{7}\\
\textit{octal} &::= \textit{octaldigit}(\textit{octaldigitoctaldigit}?)?\\
\textit{ctrlchar} &::= \texttt{\textbackslash u0040} \mid \ldots \mid \texttt{\textbackslash u005f}\\
\textit{escapechar} &::= \texttt{b} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{v} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\textbackslash}
\end{aligned}
$$

```
sign, digit :: Parser Char
```

```haskell
sign  = oneOf "+-"
digit = PChar.digit

uppercase, lowercase, inputchar, control, space, namechar :: Parser Char
uppercase = upper
lowercase = lower
inputchar = noneOf "\n\r"
control   = satisfy isControl
space     = char ' '
namechar  = uppercase <|> lowercase <|> digit <|> oneOf "@_"

escape :: Parser Char
escape    = char '\\' >> (octal <|> ctrl <|> escapechar)
  where
    ctrl :: Parser Char
    ctrl = char '^' >> ctrlchar

octaldigit, octal, ctrlchar, escapechar :: Parser Char
octaldigit = oneOf "01234567"
octal      = do chars <- tryOctal
                let [(o, _)] = readOct chars
                return (chr o)
  where
    tryOctal :: Parser [Char]
    tryOctal = choice [ try (count 3 octaldigit)
                      , try (count 2 octaldigit)
                      , try (count 1 octaldigit)
                      ]
ctrlchar   = satisfy (`elem` ['\x0040'..'\x005f'])
escapechar = oneOf "bdefnrstv\"\'\\"
```

Terminals

```haskell
-- | > Integer (i):
-- >    sign? digit+
integer :: Parser Integer
integer = do i <- positive <|> negative <|> decimal
             whiteSpace -- TODO: buff
             return $ i
  where
    positive :: Parser Integer
    positive = char '+' >> decimal
    negative :: Parser Integer
    negative = char '-' >> decimal >>= return . negate

-- | > Float:
```

```
-- >    sign? digit+ . digit+ ((E | e) sign? digit+)?
float :: Parser Double
float = Token.float lexer

atom :: Parser Atom
atom = do _ <- char '\''
--          ((inputchar except control and \ and ')|escape)*
--          inputchar = noneOf "\n\r"
          a <- many (noneOf "\n\r\\\'")
          _ <- char '\''
          whiteSpace -- TODO: buff
          return $ Atom a

echar :: Parser Literal
-- char = $((inputchar except control and space and \)|escape)
echar = do _ <- char '$'
           c <- noneOf "\n\r\\ " <|> escape
           whiteSpace -- TODO: buff
           return $ LChar c

estring :: Parser Literal
-- string = "((inputchar except control and \\ and \"")|escape)*"
estring = do _ <- char '"'
             s <- many $ noneOf "\n\r\\\""
             _ <- char '"'
             return $ LString s

variableName :: Parser VarName
-- variable = (uppercase | (_ namechar)) namechar*
variableName = identifier
```

Non-terminals

```
annotatedModule :: Parser (Ann Module)
annotatedModule = annotated module_

module_ :: Parser Module
module_ = do reserved "module"
             name    <- atom
             header  <- moduleHeader
             fundefs <- many funDef
             reserved "end"
             return $ Module name header fundefs

moduleHeader :: Parser ModHeader
moduleHeader = do funs  <- exports
```

```
                 attrs <- attributes
                 return $ ModHeader funs attrs

exports :: Parser [FunName]
exports = brackets $ commaSep funName

attributes :: Parser Attributes
attributes = reserved "attributes" *> brackets (commaSep moduleAttribute)

moduleAttribute :: Parser ModAttribute
moduleAttribute = do a <- atom
                     _ <- symbol "="
                     c <- constant
                     return (a,c)

funDef :: Parser FunDef
funDef = do name <- annotated funName
            _    <- symbol "="
            body <- annotated fun
            return $ FunDef name body

funName :: Parser FunName
funName = do a <- atom
             _ <- char '/'
             i <- decimal
             whiteSpace -- TODO: buff
             return (a,i)

fun :: Parser Exp
fun = do reserved "fun"
         vars <- parens $ commaSep (annotated variableName)
         _    <- symbol "->"
         expr <- expression
         return $ Fun vars expr

constant :: Parser Const
constant = liftM CLit   (try literal)    <|>
           liftM CTuple (tuple constant) <|>
           liftM CList  (elist constant)

literal :: Parser Literal
literal = try (liftM LFloat float) <|> liftM LInt integer <|>
          liftM LAtom atom <|> nil <|> echar <|> estring

nil :: Parser Literal
nil = brackets (return LNil)
```

```haskell
pattern :: Parser Pat
pattern = liftM PAlias (try   alias)   <|> liftM PVar    variableName      <|>
          liftM PLit   (try   literal) <|> liftM PTuple  (tuple   pattern) <|>
          liftM PList  (elist pattern) <|> liftM PBinary (ebinary pattern)

alias :: Parser Alias
alias = do v <- variableName
           _ <- symbol "="
           p <- pattern
           return $ Alias v p

patterns :: Parser Pats
patterns = liftM Pat pattern <|> liftM Pats (angles $ commaSep pattern)

expression :: Parser Exps
expression =  try (liftM Exps (annotated $ angles $ commaSep (annotated sexpression))) <|>
              liftM Exp (annotated sexpression)

sexpression :: Parser Exp
sexpression = app <|> ecatch <|> ecase <|> elet <|>
              liftM FunName (try funName) {- because of atom -} <|>
              fun <|> letrec <|> liftM Binary (ebinary expression) <|>
              liftM List (try $ elist expression) {- because of nil -} <|>
              liftM Lit literal <|> modCall <|> primOpCall <|> receive <|>
              eseq <|> etry <|> liftM Tuple (tuple expression) <|>
              liftM Var variableName

tuple :: Parser a -> Parser [a]
tuple = braces . commaSep

elist :: Parser a -> Parser (List a)
elist a = brackets $ list a

list :: Parser a -> Parser (List a)
list x = do xs <- commaSep1 x
            option (L xs) (do _ <- symbol "|"
                              t <- x
                              return $ LL xs t)

ebinary :: Parser a -> Parser [BitString a]
ebinary p = do _  <- symbol "#"
               bs <- braces (commaSep (bitstring p))
               _  <- symbol "#"
               return bs
```

```haskell
bitstring :: Parser a -> Parser (BitString a)
bitstring p = do _  <- symbol "#"
                 e0 <- angles p
                 es <- parens (commaSep expression)
                 return $ BitString e0 es

elet :: Parser Exp
elet = do reserved "let"
          vars <- variables
          _    <- symbol "="
          e1   <- expression
          _    <- symbol "in"
          e2   <- expression
          return $ Let (vars,e1) e2

variables :: Parser [VarName]
variables =
  do { v <- variableName; return [v] } <|> angles (commaSep variableName)

ecase :: Parser Exp
ecase = do reserved "case"
           exp <- expression
           reserved "of"
           alts <- many1 (annotated clause)
           reserved "end"
           return $ Case exp alts

clause :: Parser Clause
clause = do pat <- patterns
            g   <- guard
            _   <- symbol "->"
            exp <- expression
            return $ Clause pat g exp

guard :: Parser Guard
guard = do reserved "when"
           e <- expression
           return $ Guard e

letrec :: Parser Exp
letrec = do reserved "letrec"
            defs <- many funDef
            reserved "in"
            e <- expression
            return $ Letrec defs e
```

```haskell
app :: Parser Exp
app = do reserved "apply"
         e1 <- expression
         eN <- parens $ commaSep expression
         return $ App e1 eN

modCall :: Parser Exp
modCall = do reserved "call"
             e1 <- expression
             _  <- symbol ":"
             e2 <- expression
             eN <- parens $ commaSep expression
             return $ ModCall (e1, e2) eN

primOpCall :: Parser Exp
primOpCall = do reserved "primop"
                a <- atom
                e <- parens $ commaSep expression
                return $ PrimOp a e

etry :: Parser Exp
etry = do reserved "try"
          e1 <- expression
          reserved "of"
          v1 <- variables
          _  <- symbol "->"
          e2 <- expression
          reserved "catch"
          v2 <- variables
          _  <- symbol "->"
          _  <- expression
          return $ Try e1 (v1,e1) (v2,e2)

receive :: Parser Exp
receive = do reserved "receive"
             alts <- many $ annotated clause
             to <- timeout
             return $ Rec alts to

timeout :: Parser Timeout
timeout = do reserved "after"
             e1 <- expression
             _  <- symbol "->"
             e2 <- expression
             return $ Timeout  e1 e2
```

```haskell
eseq :: Parser Exp
eseq =  do reserved "do"
           e1 <- expression
           e2 <- expression
           return $ Seq e1 e2

ecatch :: Parser Exp
ecatch = do reserved "catch"
            e <- expression
            return $ Catch e

annotation :: Parser [Const]
annotation = do _  <- symbol "-|"
                cs <- brackets $ many constant
                return cs

annotated :: Parser a -> Parser (Ann a)
annotated p = parens (Ann <$> p <*> annotation) <|> Constr <$> p

lexer :: TokenParser ()
lexer = makeTokenParser
            (emptyDef {
             --    commentStart = "",
             --    commentEnd = "",
                   commentLine = "%",
             --    nestedComments = True,
                   identStart = upper <|> char '_',
                   identLetter = namechar
             --    opStart,
             --    opLetter,
             --    reservedNames,
             --    reservedOpNames,
             --    caseSensitive = True,
               })


angles, braces, brackets :: Parser a -> Parser a
angles     = Token.angles   lexer
braces     = Token.braces   lexer
brackets   = Token.brackets lexer

commaSep, commaSep1 :: Parser a -> Parser [a]
commaSep   = Token.commaSep  lexer
commaSep1  = Token.commaSep1 lexer

decimal :: Parser Integer
decimal = Token.decimal lexer

identifier :: Parser String
identifier = Token.identifier lexer

parens :: Parser a -> Parser a
parens = Token.parens lexer

reserved :: String -> Parser ()
reserved = Token.reserved lexer

symbol :: String -> Parser String
symbol = Token.symbol lexer

whiteSpace :: Parser ()
whiteSpace = Token.whiteSpace lexer

-- | Parse of a string, which should contain a complete CoreErlang module
parseModule :: String -> Either ParseError (Ann Module)
parseModule input = parse (do whiteSpace
                              x <- annotatedModule
                              eof
                              return x) "" input
```