

A Monadic Parser and Interpreter for Minijava Language

李寰¹ 章瀚元²

¹16210240012@fudan.edu.cn

²16210240024@fudan.edu.cn

Project Overview

- MiniJava language
 - A subset of Java
 - Can be compiled by a regular Java compiler
 - Basic features
 - Variables assignment
 - arithmetic operation
 - If, while, recursion, functions invoking each other
- 分工
 - Parser^[1]: 李寰
 - Interpreter: 章瀚元

^[1]Graham Hutton, Erik Meijer. Monadic parsing in Haskell. Journal of functional programming, 1998, 8(04): 437-444.

The Type Parser

```
newtype Parser a = Parser { parse :: String -> [(a,String)] }
```



```
instance Monad Parser where
    return a = Parser ( \cs -> [(a,cs)] )
    p >=> f = Parser ( \cs -> do { (a,cs') <- parse p cs; parse (f a) cs' } )
```



```
(+++ ) :: Parser a -> Parser a -> Parser a
p +++ q = Parser ( \cs -> case parse p cs ++ parse q cs of
    [] -> []
    x:xs -> [x] )
```



```
zero :: Parser a
zero = Parser ( \cs -> [] )
```

Combinators - Char and String

```
item :: Parser Char
item = Parser f
    where f [] = []
          f (c:cs) = [(c,cs)]

sat :: (Char -> Bool) -> Parser Char
sat p = do { c <- item; if p c then return c else zero }

sat' :: (Char -> Bool) -> Parser String
sat' p = do { x <- sat p; return [x] }

char :: Char -> Parser Char
char c = sat (c==)

string :: String -> Parser String
string "" = return ""
string (c:cs) = char c >> string cs >> return (c:cs)

next :: String -> Parser String
next cs = string cs +++ (item >> next cs)
```

Combinators - Applying a Parser

```
asterisk :: Parser a -> Parser [a]
asterisk p = plusSign p +++ return []

plusSign :: Parser a -> Parser [a]
plusSign p = do { x <- p; xs <- asterisk p; return (x:xs) }

space :: Parser String
space = sat' Char.isSpace

comments :: Parser String
comments = (string "/*" >> next "*/") +++ (string "//" >> next "\n")

white :: Parser String
white = asterisk (space +++ comments) >=> return . concat

token :: Parser a -> Parser a
token p = do { a <- p; white; return a }

apply :: Parser a -> String -> [(a,String)]
apply p = parse (white >> p)
```

Combinators - Parsing Sequence

```
-- ( (a op a) op a ) op a
lass :: Parser a -> Parser op -> (a -> b) -> (b -> op -> a -> b) -> Parser b
lass a op single cons = a >>= rest . single
  where rest x = ( do
                    y <- op
                    z <- a
                    rest $ cons x y z ) +++ return x

-- a op ( a op (a op a) )
rass :: Parser a -> Parser op -> (a -> b) -> (a -> op -> b -> b) -> Parser b
rass a op single cons = a >>= rest
  where rest x = ( do
                    y <- op
                    z <- a
                    r <- rest z
                    return $ (cons x y r) ) +++ return (single x)
```

Parsing Arithmetic Expressions - Operators and Ints

```
data AddOp    = Plus | Minus
```

```
plus         = token $ string "+"  >> return Plus
```

```
minus        = token $ string "-"  >> return Minus
```

```
addOp        = plus +++ minus
```

```
data MulOp    = Times | Slash | Modulo
```

```
times        = token $ string "*"  >> return Times
```

```
slash        = token $ string "/"  >> return Slash
```

```
modulo       = token $ string "%"  >> return Modulo
```

```
mulOp        = times +++ slash +++ modulo
```

```
data BasicExpr = Bool Bool | Int Int | Nul | ...
```

```
int = token $ plusSign (sat Char.isDigit) >=> return . Int . read
```

Parsing Arithmetic Expressions - Nodes in AST

```
data AddExpr = AMul MulExpr  
             | AddExpr AddExpr AddOp MulExpr
```

```
addExpr = lass mulExpr addOp AMul AddExpr
```

```
data MulExpr = AUnary UnaryExpr  
             | MulExpr MulExpr MulOp UnaryExpr
```

```
mulExpr = lass unaryExpr mulOp AUnary MulExpr
```


Parsing Arithmetic Expressions - An Example

Terminal

```
ghci> fst . head $ apply addExpr "10 / 3 - 4 * 5 % 7"
(AddExpr
  (MulExpr
    (int 10)
    (op "/")
    (int 3)
  )
  (op "-")
  (MulExpr
    (MulExpr
      (int 4)
      (op "*")
      (int 5)
    )
    (op "%")
    (int 7)
  )
)
```

The Type Interp

```
newtype Env = Env (Map.Map Ident Int, Map.Map Ident Func)

newtype Interp a = Interp {
  interp :: [Env] -> String -> Either (a,[Env],String) (Int,[Env],String) }

instance Monad Interp where
  return a = Interp ( \es0 cs0 -> Left (a,es0,cs0) )
  i >>= h = Interp ( \es0 cs0 -> case interp i es0 cs0 of
    Left (a,es,cs) -> interp (h a) es cs
    Right (a,es,cs) -> Right (a,es,cs) )

return_ :: Int -> Interp a
return_ a = Interp ( \es cs -> Right (a,es,cs) )
```

Interpreting Arithmetic Expressions

```
addExpr :: AddExpr -> Interp Int
addExpr (AMul a) = mulExpr a
addExpr (AddExpr a b c) = do
    i <- addExpr a
    j <- mulExpr c
    case b of
        Plus  -> return $ i + j
        Minus -> return $ i - j

mulExpr :: MulExpr -> Interp Int
mulExpr (AUnary a) = unaryExpr a
mulExpr (MulExpr a b c) = do
    i <- mulExpr a
    j <- unaryExpr c
    case b of
        Times  -> return $ i * j
        Slash  -> return $ i `div` j
        Modulo -> return $ i `mod` j
```

Interpreting Functions

```
data Func = Funci { funci :: Interp Int }
           | Funcf { funcf :: Int -> Interp Func }

env0 = [] :: [Env]
env1 = Env (Map.empty, Map.empty) :: Env

closure :: Interp Int -> Interp Int
closure i = Interp (\es0 cs0 -> case interp i (env1:es0) cs0 of
                                Right (a,e:es,cs) -> Left (a,es,cs)
                                Left (a,e:es,cs) -> Left (a,es,cs) )

valf :: Ident -> Interp Func

apply :: [Int] -> Interp Func -> Interp Int

invoke :: [Int] -> Ident -> Interp Int
invoke xs id = closure . apply xs . valf $ id
```

Interpreting Functions

```
func :: Maybe Paras -> Block -> Func
func aa b =
  case aa of
    Nothing -> Funci $ block b
    Just (APara (Para _ id)) -> Funcf $ \x -> (exti id x) >> h
      where h = return $ func Nothing b
    Just (Paras (Para _ id) _ p) -> Funcf $ \x -> (exti id x) >> h
      where h = return $ func (Just p) b

eval :: Program -> Interp Int
eval (Program [ClassDecl a b c d es f] EOF) = closure h
  where h = methods es >> invoke [] (Ident "main")
        where methods ...
```