A parser for the Core Language

Functional Languages

Francesco Bizzaro

Padova, February 2018

Project outline

Abstract

The project consists in writing a parser in Haskell for the *Core language*. A BNF syntax for the *Core language* is given beside. The project is split in two parts.

1st part

- Write Parser for Expressions except the first two productions (parseExpr)
- Write Parser for Definitions (parseDef)
- Write Parser for Alternatives (parseAlt)
- Write Parser for AExpr (parseAExpr)
- Write Paser for Variables (parseVar)

Programs	*program*	$\rightarrow$	sc_1 ; ... ; sc_n	$n \geq 1$
Supercombinators	*sc*	$\rightarrow$	$var\ var_1 \ldots var_n$ = *expr*	$n \geq 0$
Expressions	*expr*	$\rightarrow$	*expr aexpr*	Application
		$\mid$	$expr_1\ binop\ expr_2$	Infix binary application
		$\mid$	`let` *defns* `in` *expr*	Local definitions
		$\mid$	`letrec` *defns* `in` *expr*	Local recursive definitions
		$\mid$	`case` *expr* `of` *alts*	Case expression
		$\mid$	\ $var_1 \ldots var_n$. *expr*	Lambda abstraction ($n \geq 1$)
		$\mid$	*aexpr*	Atomic expression
	aexpr	$\rightarrow$	*var*	Variable
		$\mid$	*num*	Number
		$\mid$	`Pack{`*num*`,`*num*`}`	Constructor
		$\mid$	(*expr*)	Parenthesised expression
Definitions	*defns*	$\rightarrow$	$defn_1$; ... ; $defn_n$	$n \geq 1$
	defn	$\rightarrow$	*var* = *expr*	
Alternatives	*alts*	$\rightarrow$	alt_1 ; ... ; alt_n	$n \geq 1$
	alt	$\rightarrow$	<*num*> $var_1 \ldots var_n$ -> *expr*	$n \geq 0$
Binary operators	*binop*	$\rightarrow$	*arithop* $\mid$ *relop* $\mid$ *boolop*	
	arithop	$\rightarrow$	+ $\mid$ - $\mid$ * $\mid$ /	Arithmetic
	relop	$\rightarrow$	< $\mid$ <= $\mid$ == $\mid$ ~= $\mid$ >= $\mid$ >	Comparison
	boolop	$\rightarrow$	& $\mid$ |	Boolean
Variables	*var*	$\rightarrow$	$alpha\ varch_1 \ldots varch_n$	$n \geq 0$
	alpha	$\rightarrow$	*an alphabetic character*	
	varch	$\rightarrow$	*alpha* $\mid$ *digit* $\mid$ _	
Numbers	*num*	$\rightarrow$	$digit_1 \ldots digit_n$	$n \geq 1$

Figure 1.1: BNF syntax for the Core language

Project outline

$$
\begin{array}{lcll}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts & \\
 & | & \backslash\ var_1 \ldots var_n\ .\ expr & \\
 & | & expr1 & \\
 & & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 & \\
 & | & expr2 & \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 & \\
 & | & expr3 & \\
expr3 & \rightarrow & expr4\ relop\ expr4 & \\
 & | & expr4 & \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 & \\
 & | & expr5\ \texttt{-}\ expr5 & \\
 & | & expr5 & \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 & \\
 & | & expr6\ \texttt{/}\ expr6 & \\
 & | & expr6 & \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n & (n \geq 1)
\end{array}
$$

2nd part

Complete the parser, implementing the first two productions for Expressions (not considered in the previous part). To avoid infinite recursion and in order to obtain a deterministic syntactical tree (by ordering the operations), the original BNF for Expressions has been changed into the one at the left. This new BNF takes care of the priorities of all operators. Only the first two productions (the ones to implement in this part of the project) are affected by the change, the others stay the same.

Type definitions - 1

```haskell
type Name = String

data Expr a = EVar Name                   --Variables
 | ENum Int                               --Numbers
 | EConstr Int Int                        --Constructor tag arity
 | EAp (Expr a) (Expr a)                  --Applications
 | ELet IsRec [Def a] (Expr a)            --Let(rec) expressions
 | ECase (Expr a) [Alter a]               --Case expression
 | ELam [a] (Expr a)                      --Lambda abstractions
 deriving Show

type Program a = [ScDefn a]
type CoreProgram = Program Name
type ScDefn a = (Name,[a],Expr a)
type CoreScDefn = ScDefn Name
type Def a = (a,Expr a)                   --for let
type Alter a = (Int,[a],Expr a) --for case

data IsRec = NonRecursive | Recursive --to distinguish let/letrec
 deriving Show
```

Type definitions - 2

```
newtype Parser a = P(String->[(a,String)])

parse::Parser a -> String -> [(a,String)]
parse (P p) s = p s
```

```
class Applicative f => Alternative f where
 empty :: f a
 (<|>) :: f a -> f a -> f a
 many :: f a -> f [a]
 some :: f a -> f [a]

 many x = some x <|> pure []
 some x = pure (:) <*> x <*> many x
```

```
instance Alternative Parser where
 empty = P(\s -> [])
 --(<|>) :: Parser a -> Parser a -> Parser a
 pa <|> pb = P(\s -> case parse pa s of
  [] -> parse pb s
  [(v,out)] -> [(v,out)])
```

Making Parser a Monad

```
instance Functor Parser where
 -- fmap :: (a->b) -> Parser a -> Parser b
 fmap g p = P(\s -> case parse p s of
  [] -> []
  [(x,out)] -> [(g x,out)])

instance Applicative Parser where
 --pure :: a -> Parser a
 pure a = P(\s -> [(a,s)])
 --(<*>) :: Parser(a->b) -> Parser a -> Parser b
 pg <*> px = P(\s -> case parse pg s of
  [] -> []
  [(g,out)] -> parse (fmap g px) out)

instance Monad Parser where
 -- return :: a -> Parser a
 return = pure
 --(>>=):: Parser a -> (a -> Parser b) -> Parser b
 px >>= f = P(\s -> case parse px s of
  [] -> []
  [(x,out)] -> parse (f x) out)
```

Some simple parsers

```haskell
item::Parser Char
item = P(\s -> case s of
 [] -> []
 (x:xs) -> [(x,xs)])

sat::(Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else empty

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

alphanum::Parser Char
alphanum = sat isAlphaNum

nat::Parser Int
nat = do xs<-some digit
         return (read xs)

natural::Parser Int
natural = token nat
```

```haskell
string::String->Parser String
string [] = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)

space::Parser ()
space = do many (sat isSpace)
           return ()

token::Parser a -> Parser a
token p = do space
             v<-p
             space
             return v

symbol::String->Parser String
symbol xs = token (string xs)

character::Char->Parser Char
character xs = token (char xs)
```

Parse Variables

Variables	var	$\rightarrow$	$alpha\ varch_1 \dots varch_n$	$n \geq 0$
	$alpha$	$\rightarrow$	*an alphabetic character*	
	$varch$	$\rightarrow$	$alpha \mid digit \mid \_$	

```
parseVar::Parser(Name)
parseVar = do space
              v<-firstLet
              vs<-many (do alphanum
                           <|> char '_')
              case (v:vs) of
               "case" -> empty
               "let" -> empty
               "letrec" -> empty
               "Pack" -> empty
               "in" -> empty
               "of" -> empty
               _ -> return (v:vs)

firstLet::Parser Char
firstLet = do x <- item
              if ((isAlpha x)||(x == '_')) then return x else empty
```

Parse Definitions and Alternatives

Definitions	*defns*	$\to$	$defn_1$; ... ; $defn_n$	$n \geq 1$
	defn	$\to$	*var* = *expr*	
Alternatives	*alts*	$\to$	alt_1 ; ... ; alt_n	$n \geq 1$
	alt	$\to$	<*num*> $var_1 \dots var_n$ -> *expr*	$n \geq 0$

```
parseDef::Parser(Def Name)
parseDef = do v<-parseVar
              character '='
              e<-parseExpr
              return (v,e)

parseAlt::Parser(Alter Name)
parseAlt = do character '<'
              a<-natural
              character '>'
              vs<-many parseVar
              symbol "->"
              expr<-parseExpr
              return (a,vs,expr)
```

Parse AExpr

aexpr	$\rightarrow$	*var*	Variable
	|	*num*	Number
	|	`Pack{`*num*`,`*num*`}`	Constructor
	|	`(` *expr* `)`	Parenthesised expression

```
parseAExpr::Parser(Expr Name)
parseAExpr = do v<-parseVar --variable
                return (EVar v)
              <|> do n<-integer --number
                     return (ENum n)
              <|> do symbol "Pack" --constructor
                     character '{'
                     a1<-natural
                     character ','
                     a2<-natural
                     character '}'
                     return (EConstr a1 a2)
              <|> do character '(' --parenthesis
                     e<-parseExpr
                     character ')'
                     return e
```

Parse Expressions - 1

```
parseExpr::Parser(Expr Name)
parseExpr = do symbol "let" --let
               dnf<-parseDef
               dnfs<-many (do character ';'
                              parseDef)
               symbol "in"
               el<-parseExpr
               return (ELet NonRecursive (dnf:dnfs) el)
             <|> do symbol "letrec" --letrec
                    dnf<-parseDef
                    dnfs<-many (do character ';'
                                   parseDef)
                    symbol "in"
                    el<-parseExpr
                    return (ELet Recursive (dnf:dnfs) el)
             <|> do symbol "case" --case
                    ec<-parseExpr
                    symbol "of"
                    alt<-parseAlt
                    alts<-many (do character ';'
                                   parseAlt)
                    return (ECase ec (alt:alts))
             <|> do character '\\' --lambda
                    vs<-some parseVar
                    character '.'
                    exp<-parseExpr
                    return (ELam vs exp)
             <|> do e1<-parseExpr1 --expr1
                    return e1
```

$$
\begin{array}{lcl}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts \\
 & | & \backslash\ var_1 \ldots var_n\ \texttt{.}\ expr \\
 & | & expr1 \\
 & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 \\
 & | & expr2 \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 \\
 & | & expr3 \\
expr3 & \rightarrow & expr4\ relop\ expr4 \\
 & | & expr4 \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 \\
 & | & expr5\ \texttt{-}\ expr5 \\
 & | & expr5 \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 \\
 & | & expr6\ \texttt{/}\ expr6 \\
 & | & expr6 \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n
\end{array}
$$

Parse Expressions - 2

```
parseExpr1::Parser(Expr Name)
parseExpr1 = do e2<-parseExpr2
                character '|'
                e1<-parseExpr1
                return (EAp (EAp (EVar "|") e2) e1)
             <|> do e<-parseExpr2
                    return e

parseExpr2::Parser(Expr Name)
parseExpr2 = do e3<-parseExpr3
                character '&'
                e2<-parseExpr2
                return (EAp (EAp (EVar "&") e3) e2)
             <|> do e<-parseExpr3
                    return e

parseExpr3::Parser(Expr Name)
parseExpr3 = do e4_1<-parseExpr4
                op<-parseRelop
                e4_2<-parseExpr4
                return (EAp (EAp op e4_1) e4_2)
             <|> do e<-parseExpr4
                    return e
```

$$
\begin{array}{lcl}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts \\
 & | & \backslash\ var_1 \ldots var_n\ .\ expr \\
 & | & expr1 \\
 & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 \\
 & | & expr2 \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 \\
 & | & expr3 \\
expr3 & \rightarrow & expr4\ relop\ expr4 \\
 & | & expr4 \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 \\
 & | & expr5\ \texttt{-}\ expr5 \\
 & | & expr5 \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 \\
 & | & expr6\ \texttt{/}\ expr6 \\
 & | & expr6 \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n
\end{array}
$$

Parse Expressions - 3

```
parseExpr4::Parser(Expr Name)
parseExpr4 = do e5<-parseExpr5
                character '+'
                e4<-parseExpr4
                return (EAp (EAp (EVar "+") e5) e4)
              <|> do e5_2<-parseExpr5
                     character '-'
                     e5_3<-parseExpr5
                     return (EAp (EAp (EVar "-") e5_2) e5_3)
              <|> do e<-parseExpr5
                     return e

parseExpr5::Parser(Expr Name)
parseExpr5 = do e6<-parseExpr6
                character '*'
                e5<-parseExpr5
                return (EAp (EAp (EVar "*") e6) e5)
              <|> do e6_2<-parseExpr6
                     character '/'
                     e6_3<-parseExpr6
                     return (EAp (EAp (EVar "/") e6_2) e6_3)
              <|> do e<-parseExpr6
                     return e
```

$expr$	$\rightarrow$	`let` $defns$ `in` $expr$
	$\mid$	`letrec` $defns$ `in` $expr$
	$\mid$	`case` $expr$ `of` $alts$
	$\mid$	`\` $var_1 \ldots var_n$ `.` $expr$
	$\mid$	$expr1$
$expr1$	$\rightarrow$	$expr2$ `|` $expr1$
	$\mid$	$expr2$
$expr2$	$\rightarrow$	$expr3$ `&` $expr2$
	$\mid$	$expr3$
$expr3$	$\rightarrow$	$expr4$ $relop$ $expr4$
	$\mid$	$expr4$
$expr4$	$\rightarrow$	$expr5$ `+` $expr4$
	$\mid$	$expr5$ `-` $expr5$
	$\mid$	$expr5$
$expr5$	$\rightarrow$	$expr6$ `*` $expr5$
	$\mid$	$expr6$ `/` $expr6$
	$\mid$	$expr6$
$expr6$	$\rightarrow$	$aexpr_1 \ldots aexpr_n$

Parse Expressions - 4

```
parseExpr6::Parser(Expr Name)
parseExpr6 = do aexps<-some parseAExpr
                return (applicate aexps)

applicate::[(Expr Name)]->(Expr Name)
applicate [e] = e
applicate es =(EAp (applicate (init es))(last es))

parseRelop::Parser(Expr Name)
parseRelop = do symbol "=="
                return (EVar "==")
              <|> do symbol "~="
                     return (EVar "~=")
              <|> do symbol ">"
                     return (EVar ">")
              <|> do symbol ">="
                     return (EVar ">=")
              <|> do symbol "<"
                     return (EVar "<")
              <|> do symbol "<="
                     return (EVar "<=")
```

$$
\begin{array}{lcl}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts \\
 & | & \backslash\ var_1 \ldots var_n\ .\ expr \\
 & | & expr1 \\
 & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 \\
 & | & expr2 \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 \\
 & | & expr3 \\
expr3 & \rightarrow & expr4\ relop\ expr4 \\
 & | & expr4 \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 \\
 & | & expr5\ \texttt{-}\ expr5 \\
 & | & expr5 \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 \\
 & | & expr6\ \texttt{/}\ expr6 \\
 & | & expr6 \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n
\end{array}
$$

Parse Program

Programs	*program*	$\to$	sc_1 ; ... ; sc_n	$n \geq 1$
Supercombinators	*sc*	$\to$	$var\ var_1 \ldots var_n = expr$	$n \geq 0$

```
parseProg::Parser(Program Name)
parseProg = do p<-parseScDef
               do character ';'
                  ps<-parseProg
                  return (p:ps)
                <|> return [p]

parseScDef::Parser(ScDefn Name)
parseScDef = do v<-parseVar
                pf<-many parseVar
                character '='
                body<-parseExpr
                return (v,pf,body)
```

A simple example

```
main = double 21;
double = \ x . x*2;
double2 x = x + x
```

input.txt

```
>ghci CoreParser.hs
[("main",[],EAp (EVar "double") (ENum 21)),
("double",[],ELam ["x"] (EAp (EAp (EVar "*") (EVar "x")) (ENum 2))),
("double2",["x"],EAp (EAp (EVar "+") (EVar "x")) (EVar "x"))]
```

shell

A more complex example

```
Leaf = Pack{1,1};
Node = Pack{2,3};

depth t = case t of
           <1> _ -> 0;
           <2> t1 _ t2 -> let bigger = \ x y . max x y
                            in 1 + bigger (depth t1) (depth t2)
```

input.txt

```
>ghci CoreParser.hs
[("Leaf",[],EConstr 1 1),
("Node",[],EConstr 2 3),
("depth",["t"],ECase (EVar "t") [
  (1,["_"],ENum 0),
  (2,["t1","_","t2"],ELet NonRecursive
    [("bigger",ELam ["x","y"] (EAp (EAp (EVar "max") (EVar "x"))
      (EVar "y")))]
    (EAp (EAp (EVar "+") (ENum 1)) (EAp (EAp (EVar "bigger")
    (EAp (EVar "depth") (EVar "t1"))) (EAp (EVar "depth")
    (EVar "t2")))))
  ]
)]
```

shell