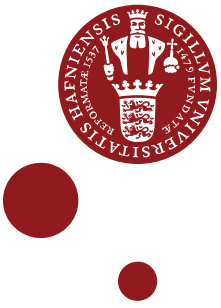
Advanced Programming

Assignment 3: Boa Parser

Gilles Souton

September 30, 2022



1 Warmup

Only the warmup with *readP* was implemented. I was out of time to apprehend and understand *Parsec*.

2 Boa concrete syntax

2.1 Correcting the given grammar

The initial grammar was ambigious regarding *Exp*. It was also necessary to rewrite certain rules and the order in which the rules are applied.

2.2 Expression

First we divided the rule *Exp* in the following manner.

$$
\begin{aligned}
Expr &\rightarrow \text{"}not\text{"}\ Exp \mid Exp' \\
Expr' &\rightarrow ExpNum\ ExpBool \mid ExpNum \\
\\
ExprBool &\rightarrow \text{" == "}\ ExpNum \mid \text{"!="}\ ExpNum \mid \text{" > "}\ ExpNum \mid \text{"}in\text{"}...\ ExpNum \\
\\
ExpNum &\rightarrow ExpTerm\ ExpNum' \\
ExpNum' &\rightarrow AddOp\ Expterm\ ExpNum' \mid \epsilon \\
AddOp &\rightarrow \text{" + "} \mid \text{" - "} \\
\\
ExpTerm &\rightarrow Const\ ExpTerm' \\
ExpTerm' &\rightarrow MulOp\ const\ ExpNum' \\
MulOp &\rightarrow \text{" * "} \mid \text{"//"} \mid \text{"\%"} \\
\\
Const &\rightarrow Num \mid String \mid Variable \mid True \mid False...
\end{aligned}
$$

The way the rules are written allows to respect different rules. Here a quick overview of the main important ideas that this grammar is trying to express.

1. Allowing nested *not* (see first rule).
2. Separating the numerical expression to the booleans, does allows to chain expression but not boolean expression. (i.e $x < y < z$) therefore removing associativity between boolean operators.
3. Rewriting the expression with operator so that we avoid left recursion.
4. Allowing precedence between operators. Whenever a 'ExpNum' will parsed we will check for 'ExpTerm' to ensure precedence between the operators $(+, -, *, /, \%)$

A complete and detailed grammar can be found in the file *Boaparser.hs*

The code that implement those rule is pretty straight forward since it is a direct application of the rules.

```
--  Exp              = "not" Exp
--                   |  Exp'
pExp :: Parser Exp
pExp =
  do string "not"; skip; exp <- pExp; return (Not exp)
    <|> do pExp'
```

```haskell
--  Exp'            = ExpNum ExpBool
--                  | ExpNum
pExp' :: Parser Exp
pExp' =
  do exp <- pExpNum; skip; pExpBool exp
    <|> pExpNum

--  ExpBool         = "==" ExpNum
--                  | "<"  ExpNum
--                  | ">"  ExpNum
--                  | "in"  ExpNum
pExpBool :: Exp -> Parser Exp
pExpBool left =
  do string "=="; skip; right <- pExpNum; return (Oper Eq left right)
    <|> do string "!="; skip; right <- pExpNum; return (Not (Oper Eq left right))
    <|> do string "<"; skip; right <- pExpNum; return (Oper Less left right)
    <|> do string "<="; skip; right <- pExpNum; return (Not (Oper Greater left right))
    <|> do string ">"; skip; right <- pExpNum; return (Oper Greater left right)
    <|> do string ">="; skip; right <- pExpNum; return (Not (Oper Less left right))
    <|> do string "in"; skip1; right <- pExpNum; return (Oper In left right)
    --really long line

--  ExpNum          = ExprTerm ExprNum'
pExpNum :: Parser Exp
pExpNum = do expTerm <- pExpTerm; pExpNum' expTerm

--  ExpNum'         = "+" ExpTerm ExprNum'
--                   | "-" ExpTerm ExprNum'
--                   | empty
pExpNum' :: Exp -> Parser Exp
pExpNum' exp =
  do char '+'; skip; expTerm <- pExpTerm; pExpNum' (Oper Plus exp expTerm)
    <|> do char '-'; skip; expTerm <- pExpTerm; pExpNum' (Oper Minus exp expTerm)
    <|> do return exp

--  ExpTerm         = Const ExpTerm'
pExpTerm :: Parser Exp
pExpTerm = do const <- pConst; pExpTerm' const

--  ExpTerm'        = "*" Const ExpNum'
--                  | "//" Const ExpNum'
--                  | empty
pExpTerm' :: Exp -> Parser Exp
pExpTerm' exp =
  do char '*'; skip; const <- pConst; pExpTerm' (Oper Times exp const)
    <|> do string "//"; skip; const <- pConst; pExpTerm' (Oper Div exp const)
    <|> do char '%'; skip; const <- pConst; pExpTerm' (Oper Mod exp const)
    <|> return exp
```

Some parts in the grammar of *Const* required some more work

```haskell
--  Const'          = num | string | var | True | False | None
--                  | "(" Expr ")"
--                  | ident "(" Exprz ")"
--                  | "[" Exprz "]"
--                  | "[" Exp ForClause Clausez"]"
pConst :: Parser Exp
pConst =
  do n <- pNum; return (Const (IntVal n))
    <|> do string <- pString; return (Const (StringVal string))
    <|> do ident <- pIdent; return (Var ident)
    <|> do string "True"; return (Const TrueVal)
    <|> do string "False"; return (Const FalseVal)
    <|> do string "None"; return (Const NoneVal)
    <|> do string "("; skip; exp <- pExp; skip; string ")"; skip; return exp
    <|> do ident <- pIdent; skip; string "("; skip; call <- pExpCall ident; skip;
↪  string ")"; skip; return call
    <|> do string "["; skip; exp <- pExpz; skip; string "]"; return (List exp)
    <|> do string "["; skip; exp <- pExp; skip; for_clause <- pFor; skip; res <-
↪  pClausez exp [for_clause]; string "]"; return res
```

Here *Expz* was divided into two different functions.

1. *pExpz* which return a list of expressions, used when parsing a list (i.e $[1, 2, 3]$)

```haskell
--  Exprz           = Exp Expz'
pExpz :: Parser [Exp]
pExpz =
  do exp <- pExp; skip; pExpz' [exp]
    <|> return []

--  Expz'           = "," Exp Expz'
--                  | empty
pExpz' :: [Exp] -> Parser [Exp]
pExpz' exp_list =
  do char ','; skip; exp <- pExp; pExpz' (exp_list ++ [exp])
    <|> return exp_list
```

2. *pExpCall* which returns this time an *Exp* which correspond to applying a function in **Boa** This function return directly an expression of type **Call**

```haskell
    --  ExpCall         = Exp ExpCall'
pExpCall :: String -> Parser Exp
pExpCall name =
  do exp <- pExp; skip; res <- pExpCall' name [exp]; return res
    <|> return (Call name [])

--  ExpCall'        = "," Exp Expz'
--                  | empty
pExpCall' :: String -> [Exp] -> Parser Exp
pExpCall' name exp_list =
  do char ','; skip; exp <- pExp; skip; pExpCall' name (exp_list ++ [exp])
    <|> return (Call name exp_list)
```

Note on Expz the list generation could have been factorized, however for simplicity and time reason it remains separated functions.

2.3 Clauses and list comprehension

The functions that describe the rules for the clauses *for, if* and list comprehension, also follows the rules nothing more complicated was added.

2.4 Whitespaces and comments

To skip whitespaces and comments, a function skip was created.

```
skip :: Parser ()
skip =
  do
    skipSpaces
    satisfy (== '#')
    munch (/= '\n')
    skip
    <|> skipSpaces
```

This function is doing the same than *skipSpaces* of *readP* however it skips comments by checking if a is present in the input stream. If present it consumes all the characters until find the end of the line. The function is called again recursivly to parse successive comments lines.

Allowing interspaces and no space with brackets To enforce to have at least one space after a keyword, a function *skip1* was added.

```
skip1 :: Parser ()
skip1 =
  do
    satisfy (== '#')
    munch (/= '\n')
    skip1
    <|> do s <- look; if not (null s) && (head s == '(' || head s == '[' )  then do
↪   return() else do munch1 isSpace; skip
```

It does the same thing than *skip* but leaves at least one whitespace So it enforces to have space after a keyword (i.e $for\ x\ in\ y$ is valid whereas $for\ x\ iny$ is considered invalid)

This function also checks if there is a parenthesis or a square bracket after a keyword and therefore allows the stream to continue to be parsed.

2.5 Tests

1. We do not understand what is a leading keyword...

```
leading keyword:              FAIL
  Exception: Grammar is ambigious
  CallStack (from HasCallStack):
    error, called at src/BoaParser.hs:65:8 in main:BoaParser
```

   It seems logic to me that the variables names $notx$, $forabc$, inc are valid variables names.

2. The parser is timing out when too many level of parenthesis are used

```
deep parens:                  TIMEOUT (1.03s)
  Timed out after 1s
deep brackets:                TIMEOUT (1.06s)
  Timed out after 1s
*empty parens:                OK
*deep parens ):               TIMEOUT (1.06s)
  Timed out after 1s
*( deep parens:               TIMEOUT (1.34s)
  Timed out after 1s
```

```
*deep brackets ]:            TIMEOUT (1.04s)
  Timed out after 1s
*[ deep brackets:            TIMEOUT (1.39s)
  Timed out after 1s
```

I think that the issue comes from the parsing of *'(' Exp ')'*

```haskell
--  Const'          = num | string | var | True | False | None
--                  | "(" Expr ")"
--                  | ident "(" Exprz ")"
--                  | "[" Exprz "]"
--                  | "[" Exp ForClause Clausez"]"
pConst :: Parser Exp
pConst =
  do n <- pNum; return (Const (IntVal n))
    <|> do ident <- pIdent; skip; string "("; skip; call <- pExpCall ident;
↪ skip; string ")"; skip; return call
    <|> do string "["; skip; exp <- pExpz; skip; string "]"; return (List
↪ exp)
    <|> do string "["; skip; exp <- pExp; skip; for_clause <- pFor; skip; res
↪ <- pClausez exp [for_clause]; string "]"; return res
```

It is surely necessary to do some left factorization...

Remarks The code is still flawed, and missing unit test, however most unit tests form onlinTA seems to pass except those two.