Course Work Part 1

Runfa Lv 6510951

November 5, 2015

1 Task1.1

1.1 Breif Comments

Add repeat and until to keyword and add it to the command produciton. The code is written in a similar way as other commands such as while.

1.2 Answers

Here only shows parts of production which are modified

1.2.1 Lexical Syntax

Keyword → **begin** | **const** | **do** | **else** | **end** | **if** | **in**
| **let** | **then** | **var** | **while** | **repeat** | **until**

1.2.2 Context-Free Syntax

Command | **repeat** *Command* **until** *Expression*

1.2.3 Abstract Syntax

Command | **repeat** *Command* **until** *Expression* *CmdRepeat*

1.3 Code

```
Token.hs
    | Repeat      -- ^ \"repeat\"
    | Until       -- ^ \"until\"
Scanner.hs
            mkIdOrKwd "repeat"= Repeat
            mkIdOrKwd "until" = Until

Parser.y
    REPEAT        { (Repeat, $$) }
    UNTIL         { (Until, $$) }

          | REPEAT command UNTIL expression
          { CmdRepeat {crCond = $4, crBody = $2, cmdSrcPos = $1} }
AST.hs
    | CmdRepeat {
          crCond      :: Expression,        -- ^ Stop-loop-condition
          crBody      :: Command,           -- ^ Loop-body
          cmdSrcPos   :: SrcPos
      }
PPAST.hs
ppCommand n (CmdRepeat {crCond = e, crBody = c, cmdSrcPos = sp}) =
    indent n . showString "CmdRepeat" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c
```

2 Task1.2

2.1 Breif Comments

Add new token '?'. Add this new kind of expression both in CFS and AST.

The code is written in a similar way as other expressions.

2.2 Answers

Here only shows parts of production which are modified

2.2.1 Lexical Syntax

Token → *Keyword* | *Identifier* | *IntegerLiteral* | *Operator* | , | ; | : | := | = | (|) | eot | ?

2.2.2 Context-Free Syntax

Expression | *Expression* ? *Expression* : *Expression*

Operator	Precedence	Associativity
^	1	right
* /	2	left
+ -	3	left
< <= == ! = >= >	4	non
&&	5	left
||	6	left
? :	7	right

2.2.3 Abstract Syntax

Expression | *Expression* ? *Expression* : *Expression* *ExpB*

2.3 Code

```
Token.hs
     | Qmark        -- ^ \"?\"
Scanner.hs
          mkOpOrSpecial "?"   = Qmark

Parser.y
      '?'      { (Qmark, $$) }

    %right '?' ':'

      | expression '?' expression ':' expression
          { ExpB {ebCond      = $1,
                        ebExp1       = $3,
                        ebExp2       = $5,
                        expSrcPos = srcPos $1} }
AST.hs
    | ExpB {
          ebCond      :: Expression,      -- ^ Conditional expression
          ebExp1      :: Expression,      -- ^ expression 1
          ebExp2      :: Expression,      -- ^ expression 2
          expSrcPos   :: SrcPos
      }
PPAST.hs
ppExpression n (ExpB {ebCond = ec, ebExp1 = e1, ebExp2 = e2, expSrcPos =
    indent n . showString "ExpB" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) ec
    . ppExpression (n+1) e1
    . ppExpression (n+1) e2
```

3 Task1.3

3.1 Breif Comments

Add elseif to keyword. Add new non-terminal vriables CmdElsifs and CmdElsif Add productions for new non-terminal virables

3.2 Answers

Here only shows parts of production which are modified

3.2.1 Lexical Syntax

Keyword → **begin** | **const** | **do** | **else** | **end** | **if** | **in**
| **let** | **then** | **var** | **while** | **repeat** | **until** | **elseif**

3.2.2 Context-Free Syntax

Command | **if** *Expression* **then** *Command CmdElsifs*
| **if** *Expression* **then** *Command CmdElsifs* **else** *Command*
CmdElsifs → *CmdElsif*
| *CmdElsif CmdElsifs*
CmdElsif → **elsif** *Expression* **then** *Command*

3.2.3 Abstract Syntax

Command | **if** *Expression* **then** *Command CmdElsifs* *CmdThen*
| **if** *Expression* **then** *Command CmdElsifs* **else** *Command* *CmdElse*
CmdElsifs → *CmdElsif* *CmdElsifSeq*
| *CmdElsif CmdElsifs*
CmdElsif → **elsif** *Expression* **then** *Command* *CmdElsif*

3.3 Code

```
Token.hs
    | Elsif        -- ^ \"elsif\"
Scanner.hs
            mkIdOrKwd "elsif" = Elsif

Parser.y
    ELSIF          { (Elsif, $$) }

    | IF expression THEN command
        { CmdThen {ctCond = $2, ctThen = $4,
        ctMyElsif = Nothing, cmdSrcPos = $1} }
    | IF expression THEN command cmdelsif
        { CmdThen {ctCond = $2, ctThen = $4,
        ctMyElsif = Just $5, cmdSrcPos = $1} }
    | IF expression THEN command ELSE command
        { CmdElse {ceCond = $2, ceThen = $4,
        ceMyElsif = Nothing, ceElse = $6, cmdSrcPos = $1} }
    | IF expression THEN command cmdelsif ELSE command
        { CmdElse {ceCond = $2, ceThen = $4,
        ceMyElsif = Just $5, ceElse = $7, cmdSrcPos = $1} }
        cmdelsifs :: { [CmdElsif]}
cmdelsifs
    : cmdelsif                  { [$1] }
    | cmdelsif cmdelsifs        { $1 : $2 }

cmdelsif :: { CmdElsif }
cmdelsif
    : ELSIF expression THEN command
        { CmdElsif {ceiCond = $2, ceiThen = $4, cmdElsifSrcPos = $1} }
    | cmdelsifs
        { if length $1 == 1 then
              head $1
          else
              CmdElsifSeq {cesCmds = $1}
        }

AST.hs
    -- | If no else
    | CmdThen {
          ctCond      :: Expression,        -- ^ Condition
          ctThen      :: Command,           -- ^ Then-branch
          ctMyElsif   :: Maybe CmdElsif,-- ^ Elsif-branch
          cmdSrcPos   :: SrcPos
      }
    -- | If then else
    | CmdElse {
          ceCond      :: Expression,        -- ^ Condition
          ceThen      :: Command,           -- ^ Then-branch
          ceMyElsif   :: Maybe CmdElsif,-- ^ Elsif-branch
          ceElse      :: Command,           -- ^ Else-branch
          cmdSrcPos   :: SrcPos
      }

instance HasSrcPos Command where
    srcPos = cmdSrcPos

-- | Abstract syntax for the syntactic category CmdElsif
```

```
data CmdElsif
    = CmdElsif {
          ceiCond    :: Expression,          -- ^ Condition
          ceiThen    :: Command,             -- ^ Then-branch
          cmdElsifSrcPos :: SrcPos
      }
    | CmdElsifSeq {
          cesCmds    :: [CmdElsif],          -- ^ Elsifs
          cmdELsifSrcPos :: SrcPos
    }

instance HasSrcPos CmdElsif where
    srcPos = cmdElsifSrcPos

PPAST.hs
ppCommand n (CmdThen {ctCond = e, ctThen = c,
ctMyElsif = ctme, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c
    . ppOpt (n+1) ppCmdElsif ctme
ppCommand n (CmdElse {ceCond = e, ceThen = c1,
ceElse = c2, ceMyElsif = ceme, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c1
    . ppOpt (n+1) ppCmdElsif ceme
    . ppCommand (n+1) c2
--------------------------------------------------------------------------------
-- Pretty printing of elsif
--------------------------------------------------------------------------------
ppCmdElsif :: Int -> CmdElsif -> ShowS
ppCmdElsif n (CmdElsif {ceiCond = e, ceiThen = c}) =
    indent n . ppExpression (n+1) e
    . ppCommand (n+1) c
ppCmdElsif n (CmdElsifSeq {cesCmds = ces, cmdELsifSrcPos = csp}) =
    indent n .ppSeq (n+1) ppCmdElsif ces
```

4 Task1.4

4.1 Breif Comments

According to the grammar, the Character Literal is a character between two quotation marks.
Firstly, scan and find quotation mark.
Then, match with five escape characters. If it does not match any one of them, it is a non-control character. Finally change it into token, and print after parsing.

4.2 Answers

Here only shows parts of production which are modified

4.2.1 Lexical Syntax

Add the productions which are given in the question

4.2.2 Context-Free Syntax

$$\mathit{PrimaryExpression} \qquad\qquad | \ \underline{\mathit{CharLiteral}}$$

4.2.3 Abstract Syntax

$$\mathit{PrimaryExpression} \qquad | \ \underline{\mathit{CharLiteral}} \qquad \mathit{ExpLitChar}$$

4.3 Code

```
Token.hs
    | LitChar{lchVal :: Char}              -- ^ Char literals

Scanner.hs
            scan l c ('\'' : s) = scanLitChar l c s

              -- scanLitChar :: Int -> Int -> String -> D a
            scanLitChar l c ('\\' : x : '\'' :xs) =
                        retTkn (mkChar x) l c (c+4) xs
```

```
scanLitChar l c (x : '\'' :xs)          =
        retTkn (mkChar x) l c (c+3) xs

scanLitChar l c (x :xs)                 = do
                emitErrD (SrcPos l c)
            ("Lexical error: Illegal \
               \character "
                ++ show x
                ++ " (discarded)")
           scan l (c + 1) xs

mkChar :: Char -> Token
mkChar 'n' = LitChar {lchVal = '\n'}
mkChar 'r' = LitChar {lchVal = '\r'}
mkChar 't'= LitChar {lchVal = '\t'}
mkChar '\\' = LitChar {lchVal = '\\'}
mkChar '\'' = LitChar {lchVal = '\''}
mkChar char = LitChar {lchVal = char}
```

```
Parser.y
    | LITCHAR
        { ExpLitChar {elchVal = tspLchVal $1, expSrcPos = tspSrcPos $1}

tspLchVal :: (Token,SrcPos) -> Char
tspLchVal (LitChar {lchVal = n}, _) = n
tspLchVal _ = parserErr "tspLIVal" "Not a LitChar"
AST.hs
    | ExpLitChar {
          elchVal    :: Char,              -- ^ Integer value
          expSrcPos  :: SrcPos
      }
PPAST.hs
ppExpression n (ExpLitChar {elchVal = v}) =
    indent n . showString "ExpLitChar". spc . shows v . nl
```