Compiler course work 1 report

Task I.1:

Aim is to extend MiniTriangle with a repeat-loop.

Should add extra token, extend AST have new print method and should let scanner and parser can identify new token.

First extend grammar:

Lexical Syntax:

Keyword repeat | until

Context-Free Syntax:

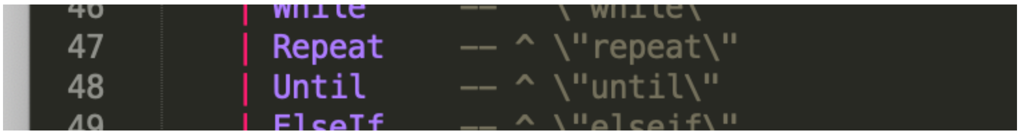
Command |repeat *command* until *expression*

Abstract Syntax:

Command |repeat command until expression CmdRepeat

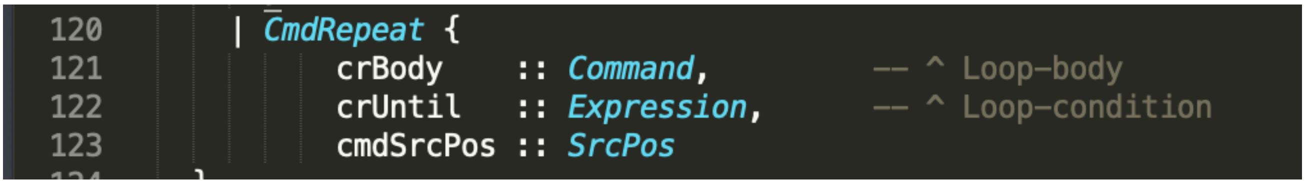
Coding part:

Extend 'repeat', 'until' token to token.hs



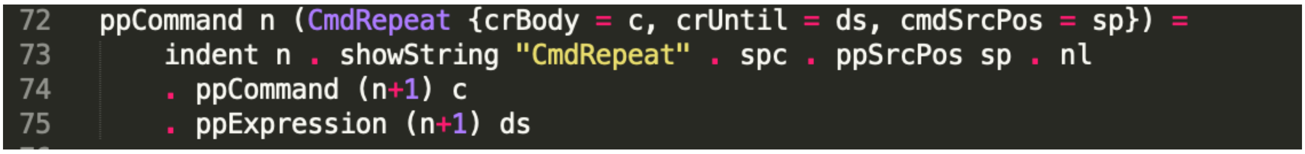
```
                | While        -- ^ \"while\"
                | Repeat       -- ^ \"repeat\"
                | Until        -- ^ \"until\"
                | ElseIf       -- ^ \"elseif\"
```

Extend command in AST.hs



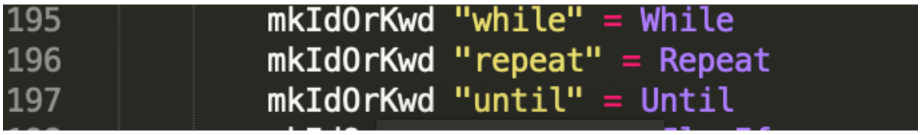
```
    | CmdRepeat {
          crBody    :: Command,             -- ^ Loop-body
          crUntil   :: Expression,          -- ^ Loop-condition
          cmdSrcPos :: SrcPos
```

Extend ppcommand in PPAST.hs to make parser print properly



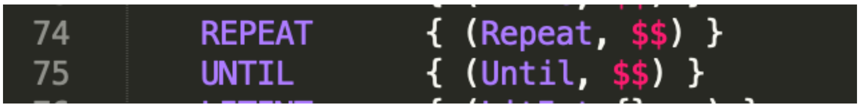
```
ppCommand n (CmdRepeat {crBody = c, crUntil = ds, cmdSrcPos = sp}) =
    indent n . showString "CmdRepeat" . spc . ppSrcPos sp . nl
    . ppCommand (n+1) c
    . ppExpression (n+1) ds
```

Extend mkIdOrKwd in scanner.hs to let compiler can properly identify the new token.

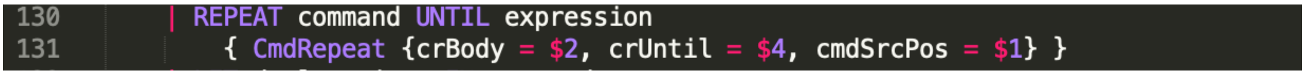


```
        mkIdOrKwd "while" = While
        mkIdOrKwd "repeat" = Repeat
        mkIdOrKwd "until" = Until
```

Extend token and command in parser.y



```
    REPEAT      { (Repeat, $$) }
    UNTIL       { (Until, $$) }
```



```
    | REPEAT command UNTIL expression
        { CmdRepeat {crBody = $2, crUntil = $4, cmdSrcPos = $1} }
```

Task I.2

Should add extra token, extend AST have new print method and should let scanner and parser can identify new token.

First extend grammar:

Context-Free Syntax:

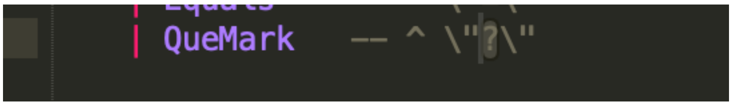
Expression | expression ? expression : expression

Abstract Syntax:

Expression | expression ? expression : expression   ExpCond

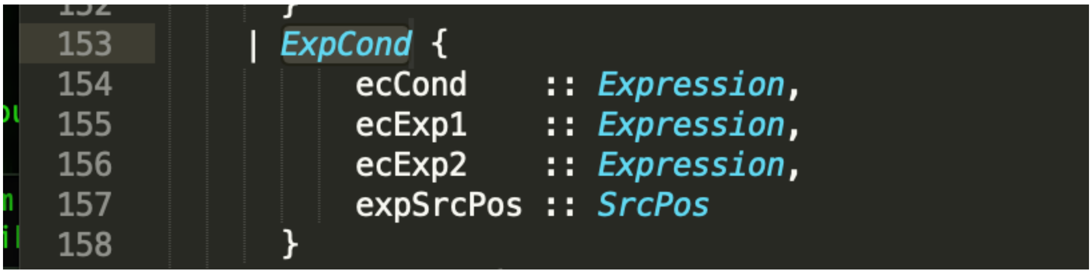
Coding part:

Extend '?' token to token.hs since '?' is already there



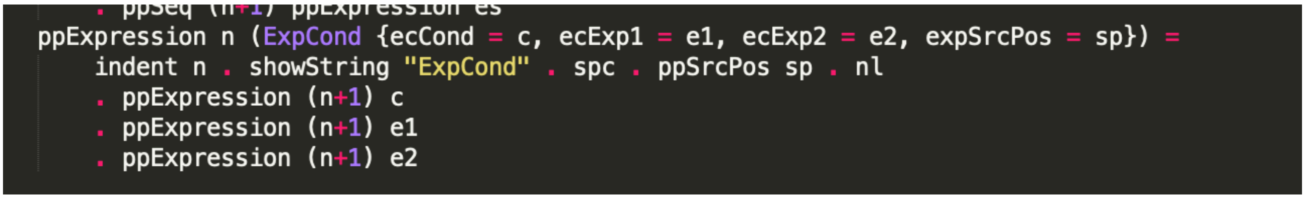
```
    | QueMark    -- ^ \"?\"
```

Exrend AST.hs:



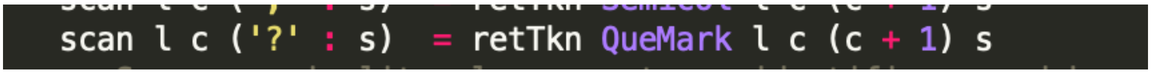
```
        | ExpCond {
              ecCond    :: Expression,
              ecExp1    :: Expression,
              ecExp2    :: Expression,
              expSrcPos :: SrcPos
          }
```

Extend ppexpression to PPAST:



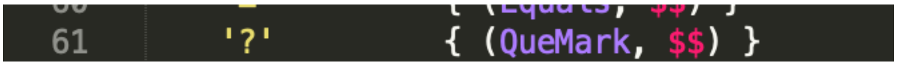
```
      . ppSeq (n+1) ppExpression es
ppExpression n (ExpCond {ecCond = c, ecExp1 = e1, ecExp2 = e2, expSrcPos = sp}) =
    indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) c
    . ppExpression (n+1) e1
    . ppExpression (n+1) e2
```

Scanner.hs:

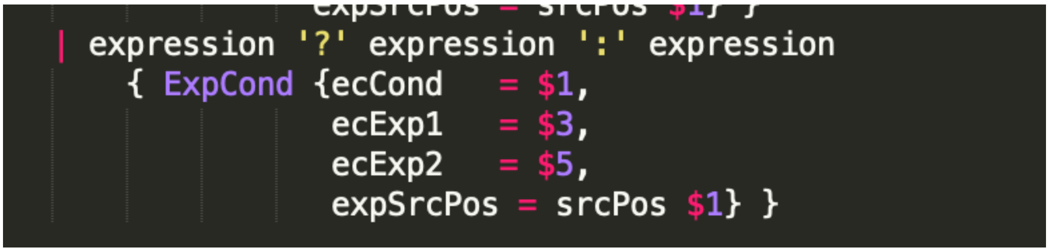


```
scan l c ('?' : s)  = retTkn QueMark l c (c + 1) s
```

Parser.y



```
        '?'           { (QueMark, $$) }
```



```
| expression '?' expression ':' expression
    { ExpCond {ecCond    = $1,
               ecExp1    = $3,
               ecExp2    = $5,
               expSrcPos = srcPos $1} }
```

Task I.3:

This task should use maybe keyword to allow the condition command exist, and use the ppSeq to allow multiple Elseif command.

First extend grammar:

Lexical Syntax:

Keyword elseif

Context-Free Syntax:

Command | if expression then command

| if expression then command elseifcommands else command

| if expression then command elseifcommands

Elseifcommands -> Elseifcommand | elseifcommand elseifcommands

Elseifcommand -> elseif expression then command

Abstract Syntax:

Command | if expression then command CmdIf

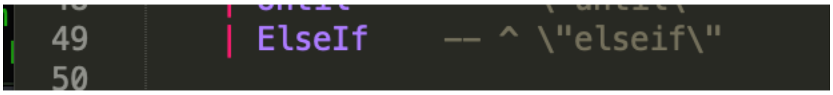
| if expression then command elseifcommand* else command CmdIf

| if expression then command elseifcommand* CmdIf

Elseifcommand -> elseif expression then command CmdElIf

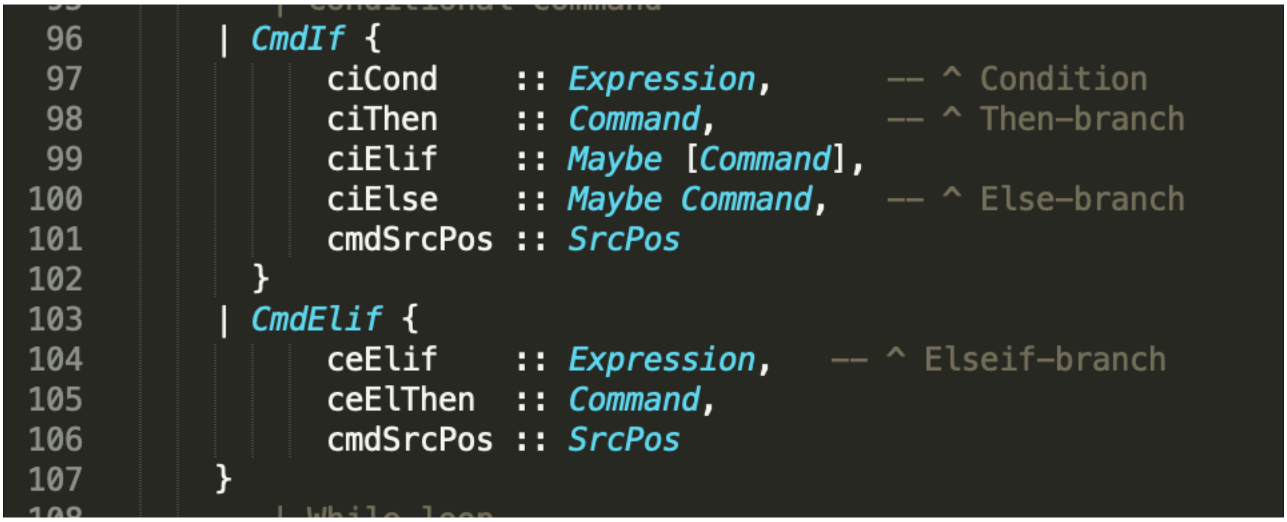
Coding part:

Token.hs:



```
        | ElseIf    -- ^ \"elseif\"
```

AST.hs:

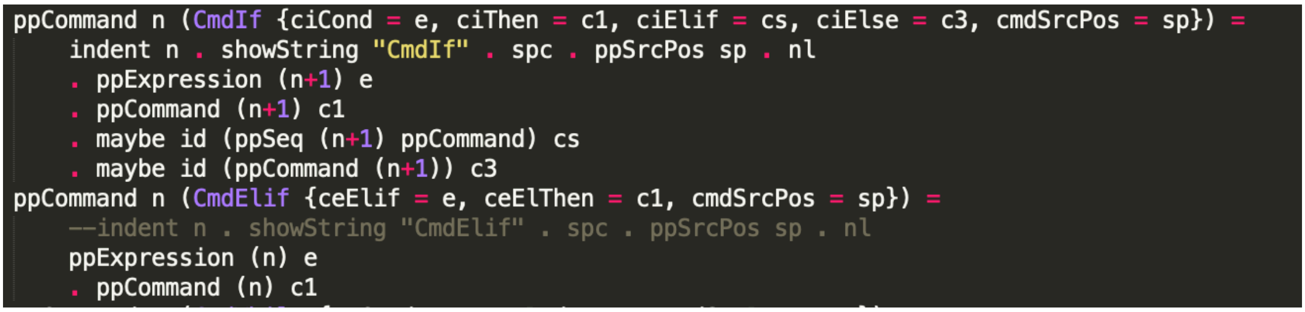


```
    | Conditional command
    | CmdIf {
        ciCond    :: Expression,       -- ^ Condition
        ciThen    :: Command,          -- ^ Then-branch
        ciElif    :: Maybe [Command],
        ciElse    :: Maybe Command,    -- ^ Else-branch
        cmdSrcPos :: SrcPos
      }
    | CmdElif {
        ceElif    :: Expression,    -- ^ Elseif-branch
        ceElThen  :: Command,
        cmdSrcPos :: SrcPos
    }
      | While loop
```

Add maybe [command] means this is sequence of command and it’s conditional.

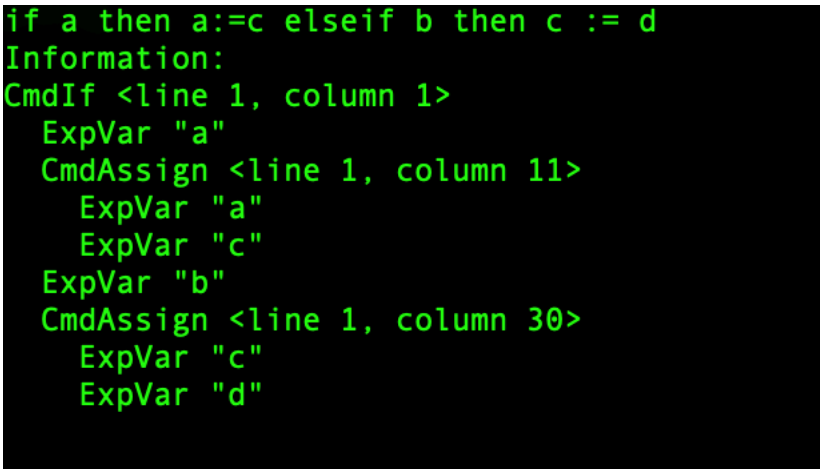
And add CmdElif is that I consider else if is a kind of command but it’s following by CmdIf in parser.y do not have a command start with elseif it prevent error grammar like “elseif a then b” without if command.

PPAST.hs:



```
ppCommand n (CmdIf {ciCond = e, ciThen = c1, ciElif = cs, ciElse = c3, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c1
    . maybe id (ppSeq (n+1) ppCommand) cs
    . maybe id (ppCommand (n+1)) c3
ppCommand n (CmdElif {ceElif = e, ceElThen = c1, cmdSrcPos = sp}) =
    --indent n . showString "CmdElif" . spc . ppSrcPos sp . nl
    ppExpression (n) e
    . ppCommand (n) c1
```

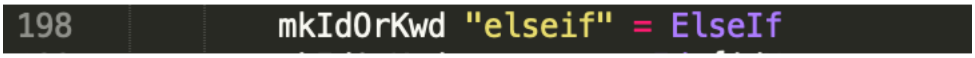
The ppcommand of cmdElif is a bit different to other command since I should make the command indent properly.



```
if a then a:=c elseif b then c := d
Information:
CmdIf <line 1, column 1>
  ExpVar "a"
  CmdAssign <line 1, column 11>
    ExpVar "a"
    ExpVar "c"
  ExpVar "b"
  CmdAssign <line 1, column 30>
    ExpVar "c"
    ExpVar "d"
```

Two CmdAssign have same indent.

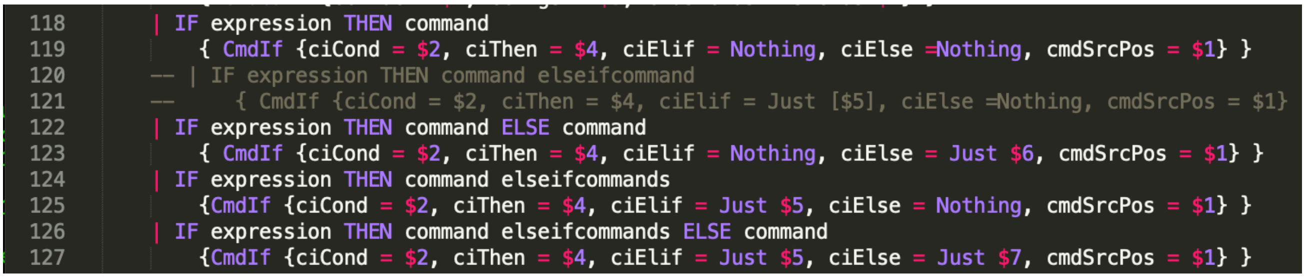
Scanner.hs:



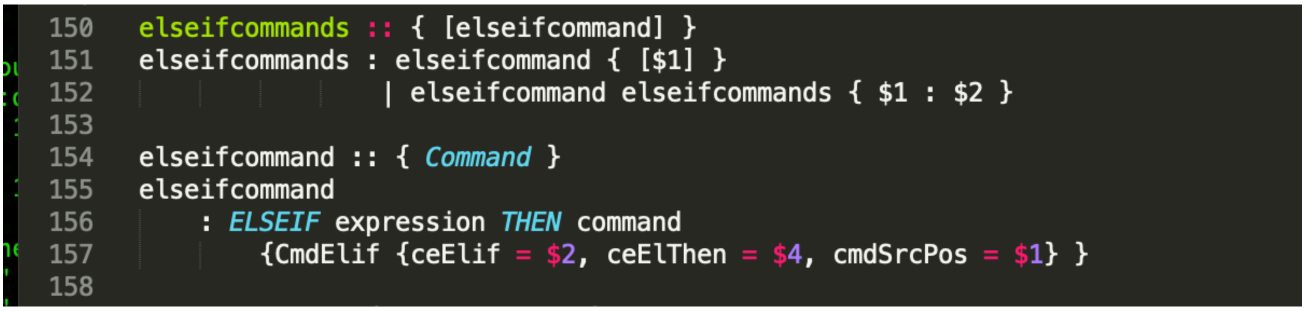
```
        mkIdOrKwd "elseif" = ElseIf
```

Simply add one token to it like task I.1

Parser.y:



```
    | IF expression THEN command
      { CmdIf {ciCond = $2, ciThen = $4, ciElif = Nothing, ciElse =Nothing, cmdSrcPos = $1} }
    -- | IF expression THEN command elseifcommand
    --     { CmdIf {ciCond = $2, ciThen = $4, ciElif = Just [$5], ciElse =Nothing, cmdSrcPos = $1}
    | IF expression THEN command ELSE command
      { CmdIf {ciCond = $2, ciThen = $4, ciElif = Nothing, ciElse = Just $6, cmdSrcPos = $1} }
    | IF expression THEN command elseifcommands
      {CmdIf {ciCond = $2, ciThen = $4, ciElif = Just $5, ciElse = Nothing, cmdSrcPos = $1} }
    | IF expression THEN command elseifcommands ELSE command
      {CmdIf {ciCond = $2, ciThen = $4, ciElif = Just $5, ciElse = Just $7, cmdSrcPos = $1} }
```



```
elseifcommands :: { [elseifcommand] }
elseifcommands : elseifcommand { [$1] }
               | elseifcommand elseifcommands { $1 : $2 }

elseifcommand :: { Command }
elseifcommand
    : ELSEIF expression THEN command
        {CmdElif {ceElif = $2, ceElThen = $4, cmdSrcPos = $1} }

```

should add extra command to it just follow the extended grammar.

Task I.4

This part should focus on extend the scanner, make scanner can identify the new grammar.

First extend grammar:

Lexical Syntax:

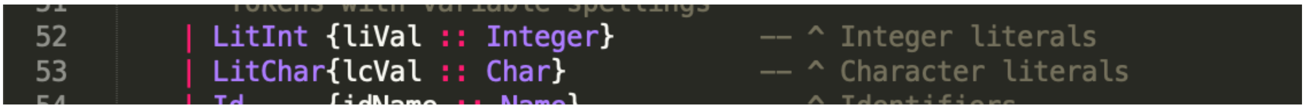
CharacterLiteral -> Graphic | EscapeCharacter

Graphic -> non-contril character except ‘ and \

EscapeCharecter -> \ (n | r | t | \ | ’)

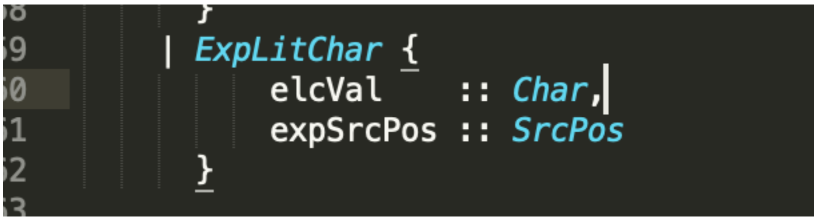
Coding part:

Token.hs:



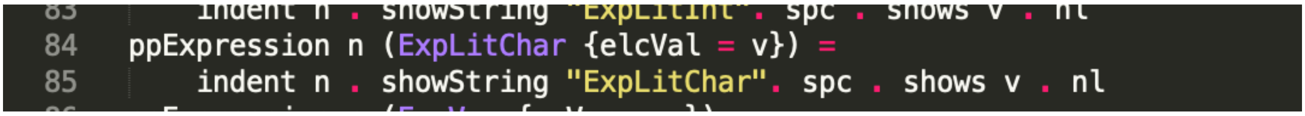
```
    -- Tokens with variable spellings
    | LitInt {liVal :: Integer}          -- ^ Integer literals
    | LitChar{lcVal :: Char}             -- ^ Character literals
    | Id     {idName :: Name}            -- ^ Identifiers
```

AST.hs:



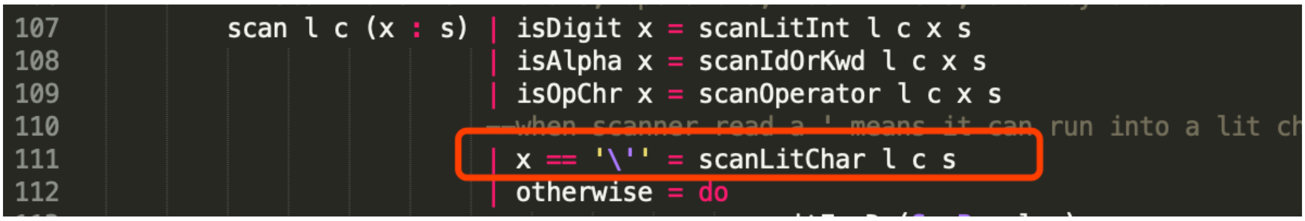
```
        }
    | ExpLitChar {
          elcVal    :: Char,
          expSrcPos :: SrcPos
      }
```

PPAST.hs:



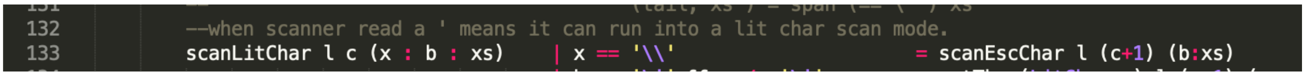
```
    indent n . showString "ExpLitInt". spc . shows v . nl
ppExpression n (ExpLitChar {elcVal = v}) =
    indent n . showString "ExpLitChar". spc . shows v . nl
```

Scanner.hs:



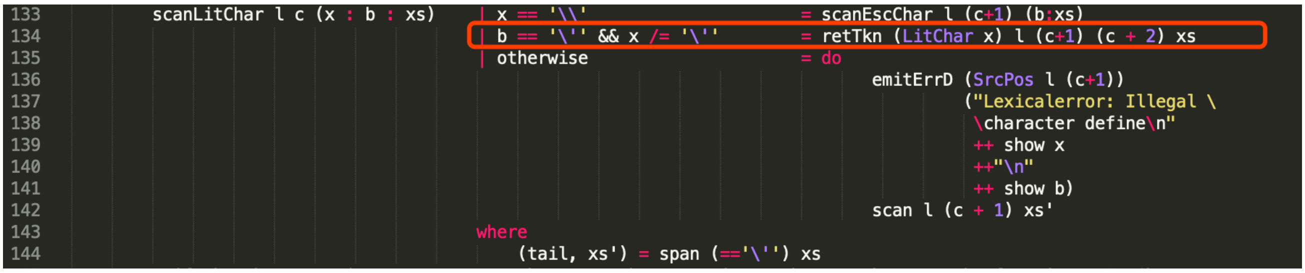
```
        scan l c (x : s) | isDigit x = scanLitInt l c x s
                         | isAlpha x = scanIdOrKwd l c x s
                         | isOpChr x = scanOperator l c x s
                         --when scanner read a ' means it can run into a lit ch
                         | x == '\'' = scanLitChar l c s
                         | otherwise = do
```

When reach a ‛ start scanlitchar



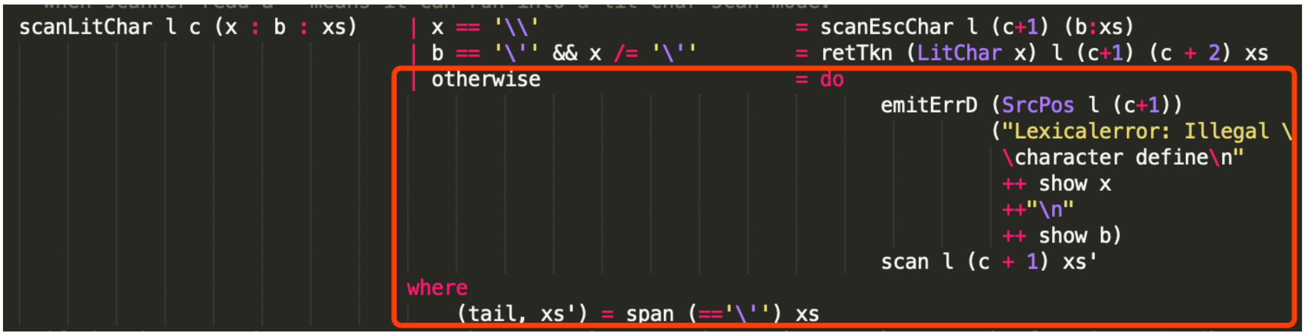
```
        --when scanner read a ' means it can run into a lit char scan mode.
        scanLitChar l c (x : b : xs)    | x == '\\'                     = scanEscChar l (c+1) (b:xs)
```

When reach a \ start scanescchar



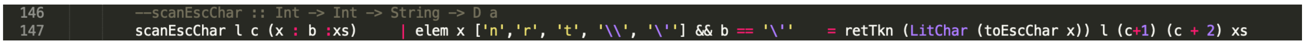
```
        scanLitChar l c (x : b : xs)    | x == '\\'                     = scanEscChar l (c+1) (b:xs)
                                        | b == '\'' && x /= '\''        = retTkn (LitChar x) l (c+1) (c + 2) xs
                                        | otherwise                     = do
                                                                             emitErrD (SrcPos l (c+1))
                                                                                      ("Lexicalerror: Illegal \
                                                                                       \character define\n"
                                                                                       ++ show x
                                                                                       ++"\n"
                                                                                       ++ show b)
                                                                             scan l (c + 1) xs'
                                        where
                                           (tail, xs') = span (=='\'') xs
```

Return literal character with source position.



```
scanLitChar l c (x : b : xs)    | x == '\\'                     = scanEscChar l (c+1) (b:xs)
                                | b == '\'' && x /= '\''        = retTkn (LitChar x) l (c+1) (c + 2) xs
                                | otherwise                     = do
                                                                     emitErrD (SrcPos l (c+1))
                                                                              ("Lexicalerror: Illegal \
                                                                               \character define\n"
                                                                               ++ show x
                                                                               ++"\n"
                                                                               ++ show b)
                                                                     scan l (c + 1) xs'
                                where
                                   (tail, xs') = span (=='\'') xs
```

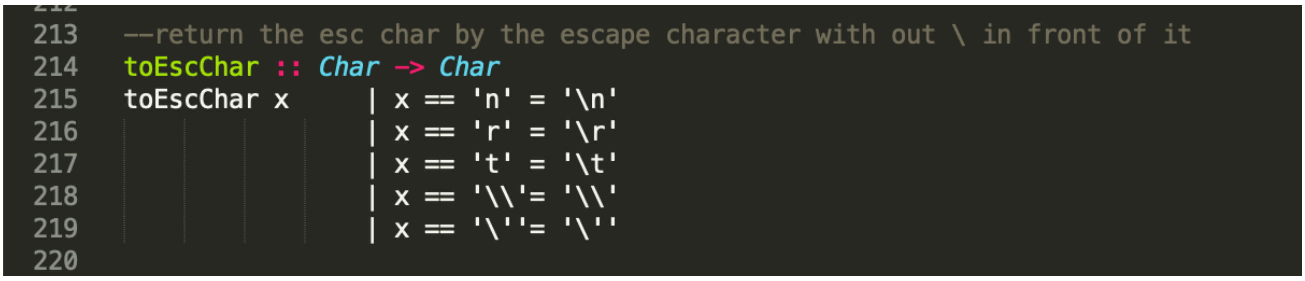
print error message properly.



```
--scanEscChar :: Int -> Int -> String -> D a
scanEscChar l c (x : b :xs)     | elem x ['n','r', 't', '\\', '\''] && b == '\''    = retTkn (LitChar (toEscChar x)) l (c+1) (c + 2) xs
```

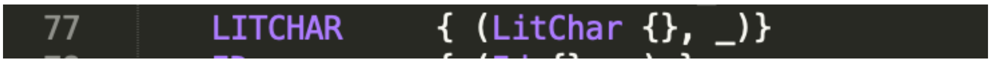
When reach escape character return litchar, using a helper function toEscChar

for change the single escape character to full escape charcter etc. n -> \n

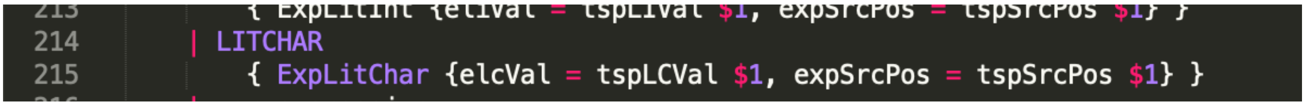


```
--return the esc char by the escape character with out \ in front of it
toEscChar :: Char -> Char
toEscChar x     | x == 'n' = '\n'
                | x == 'r' = '\r'
                | x == 't' = '\t'
                | x == '\\'= '\\'
                | x == '\''= '\''
```

Parser.y:



```
    LITCHAR     { (LitChar {}, _)}
```



```
        { ExpLitInt {eliVal = tspLIVal $1, expSrcPos = tspSrcPos $1} }
      | LITCHAR
        { ExpLitChar {elcVal = tspLCVal $1, expSrcPos = tspSrcPos $1} }
```