THE UNIVERSITY OF NOTTINGHAM NINGBO CHINA

AE3CMP COMPLIERS

COURSERWORK 01

21st March 2017

Student Name: Yu QU

Student ID: 6513261

Module Converner: Haifei HUANG

School of Computer Science

Task 1.1

In this task, the *Minitriangle Lexical Syntax* has been modified as follows:

```
...
Keyword -> begin | const | do | else | end | if | in | let | then | var | while |
repeat | until
...
```

The *MiniTriangle Context-Free Syntax* has been modified as follows:

```
...
Command -> VarExpression := Expression
    | VarExpression ( Expressions )
    | if Expression then Command
        else Command
    | while Expression do Command
    | let Declarations in Command
    | begin Commands end
    | repeat Commands until Expression
...
```

The *MiniTriangle Abstract Syntax* has been modified as follows:

```
...
Command -> Expression := Expression CmdAssign
    | Expression ( Expression* ) CmdCall
    | begin Command* end CmdSeq
    | if Expression then Command
        else Command CmdIf
    | while Expression do Command CmdWhile
    | let Declaration* in Command CmdLet
    | repeat Commands until Expression CmdRepeat
...
```

According to these changes we can modified the codes like the contents illustrated below:

1. Add new tokens *Repeat* and *Until* in file *Token.hs*:

```
data Token
    ...
    --Keywords
    | Repeat      -- ^ \"repeat\"
    | Until   -- ^ \"until\"
    ...
```

2. Add new keywords *repeat* and *until* in file *Scanner.hs*:

```
mkIdOrKwd :: String -> Token
...
mkIdOrKwd "repeat" = Repeat
```

```
        mkIdOrKwd "until" = Until
        ...
```

3. Add new command *CmdRepeat* in file *AST.hs*:

```
    data Command
        ...
        -- | Repeat-loop
        | CmdRepeat {
              crCond      :: Expression,    -- ^ Loop-condition
              crBody      :: Command,          -- ^ Loop-body
              cmdSrcPos :: SrcPos
        }
        ...
```

4. Add tokens *Repeat* and *Until* and command *REPEAT* in file *Parser.y*:

```
    %token
    ...
    REPEAT    { (Repeat, $$) }
    UNTIL      { (Until, $$) }
    ...
    command :: { Command }
    ...
  | REPEAT command UNTIL expression
  { CmdRepeat {crCond = $4, crBody = $2, cmdSrcPos = $1} }
  ...
```

5. Add pretty printing *CmdRepeat* in file *PPAST.hs*:

```
  ppCommand :: Int -> Command -> ShowS
  ...
   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
   ...
```

Task 1.2

In this task, the *Minitriangle Lexical Syntax* has been modified as follows:

```
  ...
  Token -> Keyword | Identifier | IntegerLiteral | Operator| , | ; | : | := | = | ( | ) | ?
  |eot
  ...
```

The *MiniTriangle Context-Free Syntax* has been modified as follows:

```
  ...
  Expression -> PrimaryExpression
     | Expression BinaryOperator Expression
```

```
        | Expression ? Expression : Expression
    ...
```

The *MiniTriangle Abstract Syntax* has been modified as follows:

```
    ...
    Expression -> IntegerLiteral ExpLitInt
        | Name ExpVar
        | Expression ( Expression_ ) ExpApp
        | Expression ? Expression : Expression ExpCon
    ...
```

According to these changes we can modified the codes like the contents illustrated below:

1. Add new token *'?'* in file *Token.hs*:

```
    data Token
        -- Graphical tokens
        ...
        | Condition -- ^ \"?\"
        ...
```

2. Add new operator *'?'* in file *Scanner.hs*:

```
    mkOpOrSpecial :: String -> Token
    ...
    mkOpOrSpecial "?" =    Condition
    ...
```

3. Add new expression *ExpCon* in file *AST.hs*:

```
    data Expression
        ...
        | ExpCond {
                  ecCond       :: Expression,          -- ^ Condition
                  ecTrue       :: Expression,          -- ^ Value if condition true
                  ecFalse      :: Expression,          -- ^ Value if condition false
                  expSrcPos :: SrcPos
        }
        ...
```

4. Add '?' to tokens and *'?'* and *':'* to expressions in file *Parser.y*:

```
    %token
    ...
    '?' { (Condition, $$) }
    ...
    %right '?' ':'
    ...
    expression :: { Expression }
        ...
        | expression '?' expression ':' expression
             { ExpCond {ecCond      = $1,
                                  ecTrue      = $3,
                                  ecFalse     = $5,
                                  expSrcPos = srcPos $1} }
            ...
```

5. Add pretty printing *ExpCon* in file *PPAST.hs*:

```
    ppExpression :: Int -> Expression -> ShowS
       ...
       ppExpression n (ExpCond {ecCond = c, ecTrue = t, ecFalse = f, expSrcPos
    = sp})=
            indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
            . ppExpression (n+1) c
            . ppExpression (n+1) t
            . ppExpression (n+1) f
       ...
```

Task 1.3

In this task, the *Minitriangle Lexical Syntax* has been modified as follows:

```
   ...
   Keyword -> begin | const | do | else | elsif |end | if | in | let | then | var | while |
   repeat | until
   ...
```

The *MiniTriangle Context-Free Syntax* has been modified as follows:

```
   ...
   Command -> VarExpression := Expression
      | VarExpression ( Expressions )
      | if Expression then Command elsifs optelse
          else Command
      | while Expression do Command
      | let Declarations in Command
      | begin Commands end
      | repeat Commands until Expression
   ...
```

The *MiniTriangle Abstract Syntax* has been modified as follows:

```
   ...
      Command -> Expression := Expression CmdAssign
         | Expression ( Expression* ) CmdCall
         | begin Command* end CmdSeq
         | if Expression then Command
             (elsif Expression then Command)* else Command CmdIf
         | while Expression do Command CmdWhile
         | let Declaration* in Command CmdLet
         | repeat Commands until Expression CmdRepeat
```

...

According to these changes we can modified the codes like the contents illustrated below:

1. Add new token *Elsif* in file *Token.hs*:
```
data Token
  ...
  | Elsif -- ^ \"elsif\"
  ...
```
2. Add new keyword *elsif* in file *Scanner.hs*:
```
mkIdOrKwd :: String -> Token
  ...
  mkIdOrKwd "elsif" = Elsif
  ...
```
3. Update *if* in file *AST.hs*:
```
data Command
  ...
  -- | Conditional command
  | CmdIf {
          ciCondThens :: [(Expression, Command)],    -- ^ Conditional
     branches
          ciMbElse     :: Maybe Command, -- ^ Optional else-branch
          cmdSrcPos    :: SrcPos
     }
  ...
```
4. Add *'ELSIF'* to token, add *optelse* and *elsifs* command and update *if* command in file *Parser.y*:
```
%token
...
ELSIF          { (Elsif, $$) }
...
command :: { Command }
...
     | IF expression THEN command elsifs optelse
          { CmdIf {ciCondThens = ($2,$4) : $5, ciMbElse = $6, cmdSrcPos =
$1} }
...
     optelse :: { Maybe Command }
     optelse : {- epsilon -}
                    { Nothing }
                  | ELSE command
                    { Just $2 }
```

```
        elsifs :: { [(Expression, Command)] }
        elsifs : {- epsilon -}
                      { [] }
               | ELSIF expression THEN command elsifs
               { ($2,$4) : $5 }
   ...
```

5. Update pretty printing *CmdIf* in file *PPAST.hs*:

```
ppCommand :: Int -> Command -> ShowS
  ...
  ppCommand n (CmdIf {ciCondThens = ecs, ciMbElse = mc, cmdSrcPos = sp}) =
      indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
      . ppSeq (n+1) (\n (e,c) -> ppExpression n e . ppCommand n c) ecs
      . ppOpt (n+1) ppCommand mc
  ...
```

Task 1.4

In this task, the *Minitriangle Lexical Syntax* has been modified as follows:

```
...
Token -> Keyword | Identifier | CharacterLiteral | IntegerLiteral | Operator| , | ;
| : | := | = | ( | ) | ? | eot
...
```

The *MiniTriangle Context-Free Syntax* has been modified as follows:

```
...
PrimaryExpression -> IntegerLiteral
   | CharacterLiteral
   | VarExpression
   | UnaryOperator PrimaryExpression
   | (Expression)
...
```

The *MiniTriangle Abstract Syntax* has been modified as follows:

```
...
Expression -> IntegerLiteral    ExpLitInt
   | CharacterLiteral    ExpLitChr
   | Name ExpVar
   | Expression ? Expression : Expression    ExpCon
   | Expression ( Expression* ) ExpApp
...
```

According to these changes we can modified the codes like the contents illustrated below:

1. Add new token *LitChr* in file *Token.hs*:

```
data Token
   --Tokens with variable spellings
   ...
   | LitChr {lcVal :: Char}              -- ^ Character literals
   ...
```

2. Add *literal characters* scanner in file *Scanner.hs*:

```
scanner :: ((Token, SrcPos) -> P a) -> P a
   ...
   -- Scan character literals
   scan l c ('\'' : s) = scanLitChr l c s
   ...
   scanLitChr l c ('\\' : x : '\'' : s) =
        case encodeEsc x of
             Just e    -> retTkn (LitChr e) l c (c + 4) s
             Nothing -> do
                  emitErrD (SrcPos l c)
                                 ("Lexical error: Illegal escaped character "
                                 ++ show x ++ " in character literal (discarded)")
                  scan l (c + 4) s
   scanLitChr l c (x : '\'' : s)
         | x >= ' ' && x <= '~' && x /= '\'' && x /= '\\' =
             retTkn (LitChr x) l c (c + 3) s
         | otherwise = do
             emitErrD (SrcPos l c)
                     ("Lexical error: Illegal character "
                     ++ show x ++ " in character literal (discarded)")
             scan l (c + 3) s
   scanLitChr l c s = do
             emitErrD (SrcPos l c)
                     ("Lexical error: Malformed character literal \\(discarded)")
             scan l (c + 1) s

   encodeEsc 'n'    = Just '\n'
   encodeEsc 'r'    = Just '\r'
   encodeEsc 't'    = Just '\t'
   encodeEsc '\\' = Just '\\'
   encodeEsc '\'' = Just '\''
   encodeEsc _       = Nothing
   ...
```

3. Add new expression *literal characters* in file *AST.hs*:

```
data Expression
   ...
   -- | Literal character
```

```
        | ExpLitChr {
                elcVal        :: Char,                -- ^ Character value
                expSrcPos :: SrcPos
            }
    ...
```

4. Add expression *LITCHAR* in file *Parser.y*:

```
primary_expression :: { Expression }
    ...
    | LITCHR
    { ExpLitChr {elcVal = tspLCVal $1, expSrcPos = tspSrcPos $1} }
    ...
```

5. Add pretty printing *ExpLitChr* in file *PPAST.hs*:

```
ppExpression :: Int -> Expression -> ShowS
    ...
    ppExpression n (ExpLitChr {elcVal = v}) =
    indent n . showString "ExpLitChr". spc . shows v . nl
    ...
```