G53 CMP Report

Abstract

The report aims to explain the solutions to the coursework one of the G53 CMP module

Task I.1 (Weight 15%) Extend MiniTriangle with a repeat-loop. Infor- mally, the loop construct has the following syntax:

```
        repeat
            cmd
        until
            boolExp
```

The semantics is that the command cmd is repeated (at least once) until boolExp evaluates to true.

Solution:

Modified Code:

Add repeat and until as new tokens:

In token.hs:

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

I*n AST.hs*

Add the AST of repeat and until keyword

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

data Command =
      ....
  | CmdRepeat {
      crBody    :: Command,
      crCond    :: Expression,
    cmdSrcPos :: SrcPos
  }
```

In Parser.y

Add repeat and until to tokens:

```
%token
 ...
 REPEAT  { (Repeat, $$) }
 UNTIL   { (Until, $$) }
```

Add repeat and until to command to make them as a part of AST:

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

In PPAST.hs

To make repeat command be printable:

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

Modified syntax:

MiniTriangle Lexical Syntax :
Keyword -> begin | const | do | else | end | if | in | let | then | var | while | repeat | until

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

MiniTriangle Abstract Syntax:

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 Command until Expression	CmdRepeat

Task 1.2:

Extend MiniTriangle with C/Java-style conditional expressions. Informally, the conditional expression should have the following syntax:

boolExp ? exp_1 : exp_2

Solution:

Modified to the code:

In tokens.hs:

```
data Token
  ...
  | Question  -- ^ \"?\"
```

In AST.hs:

```
data Expression
....
 -- | Ternary operator application
```

```
  | ExpCon {
    ecCond    :: Expression,
    ecFstEx   :: Expression,
    ecSndEx   :: Expression,
    expSrcPos :: SrcPos
    }
```

In Scanner.hs:

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

In Parser.y

```
%token
  .......
    '?'     { (Question, $$) }

%right '?'
%right ':'

expression :: { Expression }
expression
    :.....
    | ternary_expression { $1 }
ternary_expression :: { Expression }
    : expression '?' expression ':' expression
    { ExpCon {ecCond = $1, ecFstEx = $3, ecSndEx = $5, expSrcPos = srcPos $1}}
```

In PPAST.hs

```
ppExpression n (ExpCon {ecCond = ex1, ecFstEx = ex2, ecSndEx = ex3, expSrcPos = sp}) =
    indent n . showString "ExpCon" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) ex1. ppExpression (n+1) ex2. ppExpression (n+1) ex3
```

Modified to the syntax:

MiniTriangle Lexical Syntax:

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

MiniTriangle Context-Free Syntax:

```
Expression -> PrimaryExpression
          | Expression ? Expression : Expression
          | Expression BinaryOperator Expression
```

MiniTriangle Abstract Syntax:

Expression -> <u>IntegerLiteral</u>   ExpLitInt

```
  | Name                                      ExpVar

 | Expression ? Expression : Expression       ExpCond


 | Expression ( Expression* )                 ExpApp
```

Task 1.3:

Extend the syntax of MiniTriangle if-command
so that:

- the else-branch is optional
- zero or more Ada-style “elsif . . . then . . . ” are allowed after the then- branch but before the (now optional) else-branch.

Solution:

Modified to the code:

In tokens.hs

```
data Token
  ...
| Else  -- ^ \"else\"
| Elsif -- ^ \"elsif\"
```

In AST.hs

```
data Command
....
-- | Conditional command
   | CmdIf {
           ciCond    :: Expression, -- ^ Condition
           ciThen    :: Command,           -- ^ Then-branch
           ciElseBranch    :: Maybe Command,              -- ^ Else-branch
        cmdSrcPos :: SrcPos
     }
   -- | Conditional command else
   | CmdElse {
        ceElse    :: Command,             -- ^ Then-branch
           cmdSrcPos :: SrcPos
     }
   -- | Conditional command elsif
   | CmdElsif {
        ceiCond    :: Expression,          -- ^ Condition
        ceiThen    :: Command,              -- ^ Then-branch
        ceiElsif   :: Maybe Command,       -- ^ Elsif-branch
           cmdSrcPos :: SrcPos
     }
```

In Scanner.hs

```
mkIdOrKwd :: String -> Token
........
mkIdOrKwd "elsif" = Elsif
```

In Parser.y

```
command
    :........
    | IF expression THEN command elseBranch
        { CmdIf {ciCond = $2, ciThen = $4, ciElseBranch = $5, cmdSrcPos = $1} }
elseBranch :: { Maybe Command }
elseBranch
         : {Nothing}
         | ELSE command
             { Just CmdElse { ceElse = $2, cmdSrcPos = $1} }
         | ELSIF expression THEN command elseBranch
             { Just CmdElsif {ceiCond = $2, ceiThen = $4, ceiElsif = $5, cmdSrcPos = $1} }
```

In PPAST

```
ppCommand n (CmdElsif {ceiCond = e, ceiThen = c1, ceiElsif = c2,cmdSrcPos = sp}) =
    indent n
    . ppExpression (n+1) e
    . ppCommand (n+1) c1
    . ppOpt n ppCommand c2
```

Modified to the syntax:

MiniTriangle Lexical Syntax :

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

MiniTriangle Context-Free Syntax:

A new non-terminal elseBranch was added to help extend the if-command.

```
Command -> VarExpression := Expression

          | VarExpression ( Expressions )

          | if Expression then Command elseBranch

          | while Expression do Command
          | repeat Command until Expression
          | let Declarations in Command

          | begin Commands end

elseBranch -> e

          | else Command
```

```
        | elsif Expression then Command elseBranch
```

MiniTriangle Abstract Syntax:

```
Command -> Expression := Expression                          CmdAssign
        | Expression ( Expression*)                          CmdCall
        | begin Command* end                                 CmdSeq
        | if Expression then Command                         CmdIf

        ( elsif Expression then Command )* ( else Command | e)

        | while Expression do Command                        CmdWhile
        | repeat Command until Expression                    CmdRepeat
```

Task 1.4:

Extend the MiniTriangle with character literals as described by the following productions: Character-Literal → ′ (Graphic | Character-Escape) ′ Graphic → any non-control character except ′ and \ Character-Escape → \(n|r|t|\|′)

Solution:

Modified to the code:

In tokens.hs

```
data Token
 ...
| LitChar{charVal :: Char}
```

In AST.hs

```
data Expression
.....
 -- | Charactor integer                                  -- task 1.4
   | ExpLitChar {
          echarVal   ::  Char,              -- ^ Charactor value
          expSrcPos :: SrcPos
}
```

In Scanner.hs

```
isGraphicChar :: Char -> Bool
isGraphicChar c = char >= 32 && char <= 126 && char /= 92
        where char = ord c

isEscapeChar :: Char -> Bool                                 -- task 1.4
isEscapeChar c = c == '\\' || c == '\'' || c == 'n' || c == 'r' || c == 't'
```

```
printEChar :: Char -> Char                                    -- task 1.4
printEChar c | c == '\\'        = '\\'
             | c == '\''        = '\''
             | c == 'n'         = '\n'
             | c == 'r'         = '\r'
             | c == 't'         = '\t'
             | otherwise        = error "Print Escape Charactor Error"
scanELitChar l c x s = retTkn(LitChar (printEChar x)) l c (c+4) s

scanGLitChar l c x s = retTkn(LitChar (x)) l c (c+3) s
scanner cont = P $ scan
    where
          .....
          scan l c ('\'' : '\'' : s)                  = error "Not a valid Character"
    scan l c ('\'' : '\\' : x : '\'' : s) | isEscapeChar x = scanELitChar l c x s
                                          | otherwise = error "Not a valid Character"

    scan l c ('\'' :  x : '\'' : s)       | isGraphicChar x =  scanGLitChar l c x s
                                          | otherwise = error "Not a valid Character"
```

In Parser.y

```
primary_expression :: { Expression }
    : .....
    | LITCHAR                                                 -- task 1.4
            { ExpLitChar {echarVal = tspCIVal $1, expSrcPos = tspSrcPos $1} }
```

In PPAST.hs

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

Modified to the syntax:

MiniTriangle Lexical Syntax:

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

CharactorLiteral -> ‘(Graphic | Character- Escape)’
Graphics -> any non-control character except ’ and \

Character-Escape -> \ (n | r | t | \ | ’)

MiniTriangle Context-Free Syntax:

PrimaryExpression -> IntegerLiteral
| CharactorLiteral

| VarExpression
| UnaryOperator PrimaryExpression | (Expression)

MiniTriangle Abstract Syntax:

Expression ->

IntegerLiteral	ExpLitInt
| CharactorLiteral	ExpLitChar
| Name	ExpVar
| Expression ? Expression : Expression	ExpCond
| Expression (Expression*)	ExpApp