UNIVERSITY OF NOTTINGHAM

G53CMP Compilers

Henrik Nilsson

Coursework Part I

Tai Nguyen Bui (txn02u)

30 October 2013

Task I.1

The MiniTriangle was extended with a repeat-loop using the syntax 'repeat *cmd* until *boolExp'* to construct it. The repeat-loop allows the command *cmd* to be repeated at least once until *boolExp* is evaluated.

First of all, the MiniTriangle Lexical Syntax (Token.hs) was extended with the new Keywords 'Repeat' and 'Until'. Then, the keywords 'repeat' and 'until' were added to the MiniTriangle Lexical analyser (Scanner.hs) as follow:

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

After extending the grammars, a new constructor was added in the Abstract Syntax Tree of the MiniTriangle (AST.hs) for the Repeat-loop function.

```
| CmdRepeat {
        crBody :: Command,
        crUntil :: Expression,
        cmdSrcPos :: SrcPos
  }
```

Two new terminal symbols were created in the MiniTriangle parser (Parser.y):

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

In order to declare a new command to let the parser interprets a repeat-loop

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

Finally, the Simple pretty printer for AST (PPAST.hs) was extended to be able to print the new command

```
ppCommand n (CmdRepeat {crBody = c, crUntil = c1, cmdSrcPos = sp }) =
        indent n . showString "CmdRepeat" . spc . ppSrcPos sp . nl
        . ppCommand (n+1) c
        . ppExpression (n+1) c1
```

Task I.2

The MiniTriangle is to be extended with a new C/Java-style conditional expression that has the syntax *boolExp* ? *exp1* : *exp2.* The conditional expression evaluates *exp1* if *boolExp* is true, otherwise evaluates *exp2.* Additionally, the precedence of this expression is the lowest of all operators and it is right associative.

First of all, the MiniTriangle Lexical Syntax (Token.hs) was extended with the new graphical token ‘CJCond’ to represent the ‘?’ of the expression. ‘:’ was not added since it already existed with the graphical token name ‘Colon’. In the MiniTriangle Lexical analyser (Scanner.hs) the operator “?” was matched with the token CJCond in the following way:

```
mkOpOrSpecial ”:” = Colon
mkOpOrSpecial “?” = CJCond
```

“:” operator already existed.
After extending the grammars, a new expression was added in the Abstract Syntax Tree of the MiniTriangle (AST.hs) for the C/Java conditional, since it was specified to not use ExpApp for the conditional expression.

```
|ExpcjCond {
        ecjCond ::Expression,
        ecjTrue :: Expression,
        ecjFalse :: Expression,
        expSrcPos :: SrcPos
}
```

One new terminal symbols was created in the MiniTriangle parser (Parser.y):

```
‘?’ {(CJCond, $$)}
‘:’{(Colon, $$)}     was already created
```

Then, the precedence of the expression was modified, adding

```
%right ‘?’ ‘:’
```

To the list of precedence that already exists. Additionally, the MiniTriangle parser was extended with a new expression definition to interpret a C/Java conditional expression.

```
| expression ‘?’ expression ‘:’ expression %prec ‘?’
        { ExpcjCond { ecjCond = $1,
                      ecjTrue   = $3,
                      ecjFalse  = $5,
                      expSrcPos = srcPos $1 } }
```

In order to print the new C/Java conditional expression, the Simple pretty printer for AST (PPAST.hs) was extended with the following code

```
ppExpression n (ExpcjCond {ecjCond = c, ecjTrue = t, ecjFalse = f, expSrcPos = sp}) =
   indent n . showString "ExpcjCond" . spc . ppSrcPos sp . nl
   . ppExpression (n+1) c
   . ppExpression (n+1) t
   . ppExpression (n+1) f
```

Task I.3

The initial MiniTriangle if-command was limited since the else-branch was not optional. Additionally, there was not a possibility of implementing one or more Ada-style "elseif… then …" after the then-branch and before the else-branch.

Initially, as recommended in the coursework's specifications, the optional else was implemented and then the elseif.
The MiniTriangle lexical syntax file (Token.hs) was extended with the new keyword 'ElseIf', needed to implement an elseif option in the if-command. Moreover, In the MiniTriangle Lexical analyser (Scanner.hs) the keyword 'elseif' was added to the code as follows:

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

Then, the command CmdIf of the MiniTriangle Abstract Syntax Tree (AST.hs) was modified to meet the new requirements.

```
| CmdIf {
          ciCond  :: Expression,
          ciThen  :: Command,
          ciElseIf  :: [(Expression, Command)],
          ciElse    :: Maybe Command,
          cmdSrcPos :: SrcPos
}
```

ciElseIf has the type of an array of tuples in order to be able to have more than one elseif in an if-command. Additionally, ciElse has the type Maybe Command due to the fact that an Else is optional, so then, could be 'Nothing' or a Command.

In the MiniTriangle parser the terminal symbol ELSEIF was added

```
ELSEIF {(ElseIf, $$) }
```

Furthermore, the if-command was modified. Two helper functions where added in order to deal with the elseif and optional else.

```
  | IF expression THEN command elseIf optionalElse
      { CmdIf {ciCond = $2, ciThen = $4, ciElseIf = $5, ciElse = $6, cmdSrcPos = $1} }
```

The helper function elseIf is a recursive function that returns a sequence of tuples (expression, command) where the length depends on the number of elseif input.

```
elseIf     :: { [( Expression, Command)]}
elseIf     : { [] }
          | ELSEIF expression THEN command elseIf
         { ($2, $4) : $5 }
```

Addionally, the optionalElse function returns either nothing if no else was input or a command otherwise.

```
optionalElse :: {Maybe Command}
optionalElse : { Nothing }
           | ELSE command
            { Just $2 }
```

The Simple pretty printer for AST (PPAST.hs) was modified to be able to print the new modifications of the if-command

```
ppCommand n (CmdIf {ciCond = e, ciThen = c1, ciElseIf = ecseq,  ciElse = c3, cmdSrcPos = sp})\
 =
   indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
   . ppExpression (n+1) e
   . ppCommand (n+1) c1
   . ppSeq (n+1) (\n (e,c) -> ppExpression n e . ppCommand n c) ecseq
   . ppOpt (n+1) ppCommand c3
```

Task I.4

This task was to extend the MiniTriangle with character literals.
Lexical symbols (Token.hs) were extended with a new token, in order to be able to work with character literals.

```
| LitChar {liChar :: Char }
```

A new expression was added in the Abstract Syntax Tree (AST.hs) of the MiniTriangle

```
  -- | Literal character
  |  ExpLitChar {
       eliChar   :: Char,
       expSrcPos :: SrcPos
     }
```

Then, the terminal symbols of the MiniTriangle Parser (Parser.y) had to be extended to be able to use literal characters. Therefore, the following line was added:

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

Additionally, a primary expression was added to the code

```
  | LITCHAR
      { ExpLitChar { eliChar = tspLIChar $1, expSrcPos = tspSrcPos $1} }
```

and projection function was implemented to detect if the input is a valid Literal character. This function analyses the pairs of token and source position.

```
tspLIChar :: (Token, SrcPos) -> Char
tspLIChar (LitChar {liChar = c}, _) = c
tspLIChar _ = parserErr "tspLIChar" "Not a LitChar"
```

The simple pretty printing for AST (PPAST.hs) was extended with a new pretty printing for Literal character expressions.

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

Testing

Test files were created in MTTests folder in order to tests task one, two and three.