TaskI.1

Repeat-loop is obviously a command. Therefore, similar to other commands, like While-loop or If-statement, we should add new production to the non-terminal Command in the Context-Free Syntax and define the new command in the Abstract Syntax.

Firstly, Repeat and Until are terminals. Therefore, in 'Token.hs', 'Repeat' and 'Until' should be added as Keywords:

```
    | Repeat      -- ^ \"repeat\"
    | Until       -- ^ \"until\"
```

Then, in 'Scanner.hs', we need scanner to identify these two keywords in the program, and transfer them from a literal string to a token:

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

In 'AST.hs', we extend the abstract syntax for the syntactic category Command:

```
    -- | Repeat-loop
    | CmdRepeat {
          crCmd     :: Command,       -- ^ Loop-command
          crCond    :: Expression,    -- ^ Loop-condition
          cmdSrcPos :: SrcPos
      }
```

'crCmd' stands for the command needs to be executed in the Repeat-loop,
'crCond' stands for the condition needs to be judged for executing the Repeat-loop,

In 'Parser.y', we define the Repeat-loop in the compiler's context-free syntax:

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

Register $1 stores "REPEAT";
Register $2 stores command;
Register $3 stores "UNTIL";
Rsgister $4 stores expression

In 'PPAST.hs', we add a new pattern to 'ppCommand' function to print the new Repeat-loop command:

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

TaskI.2

(boolExp ? exp1 : exp2) represents conditional command (If-statement) in an expression manner. Therefore, new production should be added to Expression non-terminal in both abstract syntax and context-free syntax.

Firstly, '?' and ':' are terminals. Therefore, in 'Token.hs', they should be added as graphical tokens (':' has already been added):

```
    | QMark     -- ^ \"?\"
```

Then, in 'Scanner.hs', we need scanner to identify these two graphical tokens in the program, and transfer them from a literal string to a token:

```
        mkOpOrSpecial "?"  = QMark
```

In 'AST.hs', we extend the abstract syntax for the syntactic category Expression:

```
    -- | Function "a ? b : c" represents conditional command but in an expression manner
    | ExpCond {
          ecCond    :: Expression,      -- ^ Condition
          ecThen    :: Expression,      -- ^ Then-branch
          ecElse    :: Expression,      -- ^ Else-branch
          expSrcPos :: SrcPos
      }
```

If 'ecCond' evaluates to be true, then 'ecThen' will be evaluated for the whole conditional expression. Otherwise, 'ecElse' will be evaluated

In 'Parser.y', we define the conditional expression in the compiler's context-free syntax:

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

Register $1 stores expression

Register $2 stores '?'

Register $3 stores expression

Rsgister $4 stores ':'

Register $5 stores expression

In 'PPAST.hs', we add a new pattern to 'ppExpression' function to print the new conditional expression:

```
ppExpression n (ExpCond {ecCond = e, ecThen = e1, ecElse = e2, expSrcPos = sp}) =
    indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppExpression (n+1) e1
    . ppExpression (n+1) e2
```

TaskI.3

For this task, we need to rewrite the original conditional command to fit the new feature, and a new non-terminal should be introduced:

Command → **if** *Expression* **then** *Command ElseBranch*

ElseBranch → **else** *Command*

| **elsif** *Expression* **then** *Command ElseBranch*

| **ε**

Explanation:

- If else-branch is not optional and there is no elsif-branch, *Elsebranch* should be interpreted as *Elsebranch* → **else** *Command*
- If else-branch is optional and there is no elsif-branch, *Elsebranch* should be interpreted as **ε**
- If else-branch is not optional and there is/are elsif-branch(es), the first *Elsebranch* should be interpreted as *Elsebranch* → **elsif** *Expression* **then** *Command Elsebranch*. The last *Elsebranch* should be interpreted as *Elsebranch* → **else** *Command*. All other *Elsebranch* should be interpreted as *Elsebranch* → **elsif** *Expression* **then** *Command Elsebranch*.
- If else-branch is optional and there is/are elsif-branch(es), the first *Elsebranch* should be interpreted as *Elsebranch* → **elsif** *Expression* **then** *Command Elsebranch*. All other *Elsebranch* should be interpreted as *Elsebranch* → **elsif** *Expression* **then** *Command Elsebranch*.

Firstly, Elsif is a terminal. Therefore, in 'Token.hs', 'Elsif' should be added as Keywords:

```
    | Elsif      -- ^ \"elsif\"
```

Then, in 'Scanner.hs', we need scanner to identify this keyword in the program, and transfer it from a literal string to a token:

```
        mkIdOrKwd "elsif"  = Elsif
```

In 'AST.hs', we modify the abstract syntax of the original conditional command:

```
    -- | Conditional command
    | CmdIf {
          ciCond    :: Expression,       -- ^ Condition
          ciThen    :: Command,          -- ^ Then-branch
          ciElse    :: ElseBranch,       -- ^ Else-branch
          cmdSrcPos :: SrcPos
      }
```

Then add a new non-terminal to the compiler's abstract-syntax:

```haskell
data ElseBranch
    -- | Conditional command with else ending
    = EBsingle {
        ebS         :: Command,         -- ^ Else-branch
        ebSrcPos    :: SrcPos
    }
    -- | Conditional command with elsif extension
    | EBelsif {
        ebCond      :: Expression,      -- ^ Condition
        ebThen      :: Command,         -- ^ Then-branch
        ebElse      :: ElseBranch,      -- ^ Else-branch
        ebSrcPos    :: SrcPos
    }
    -- | Conditional command without else ending
    | EBepsilon

instance HasSrcPos ElseBranch where
    srcPos = ebSrcPos
```

In 'Parser.y', we modify the original conditional command in the compiler's context-free syntax:

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

Then, we add the new non-terminal to it:

```
elsebranch :: { ElseBranch }
elsebranch
    : ELSE command
        { EBsingle {ebS = $2, ebSrcPos = $1} }
    | ELSIF expression THEN command elsebranch
        { EBelsif {ebCond = $2, ebThen = $4, ebElse = $5, ebSrcPos = $1} }
    | { EBepsilon }
```

In 'PPAST.hs', we modify the original pattern of conditional command in 'ppCommand' function:

```haskell
ppCommand n (CmdIf {ciCond = e, ciThen = c1, ciElse = eb, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c1
    . ppElseBranch (n) eb
```

Then, we add the new function 'ppElsebranch' to print the new non-terminal:

```haskell
------------------------------------------------------------------------------
-- Pretty printing of elsebranch
------------------------------------------------------------------------------

ppElseBranch :: Int -> ElseBranch -> ShowS
ppElseBranch n (EBsingle {ebS = c, ebSrcPos = sp} ) =
    ppCommand (n+1) c
ppElseBranch n (EBelsif {ebCond = e, ebThen = c, ebElse = eb, ebSrcPos = sp} ) =
    ppExpression (n+1) e
    . ppCommand (n+1) c
    . ppElseBranch (n) eb
ppElseBranch n (EBepsilon) =
    indent 0
```

TaskI.4

Task4 aims to make the compiler to identify character literals.

Firstly, in 'Token.hs', we add a new token to represent the character literals:

```
    | LitChar{lcVal :: Char}          -- ^ Character Literals
```

In 'Scanner.hs', because we need the scanner to successfully identify character literals, the work is huge. I will explain each new piece added to the 'Scanner.hs':

```
        scan l c (x : s) | isDigit x = scanLitInt l c x s
                         | x == '\'' = scanLitChar l c s
```

- Character literals must be started with a single quotation mark, wherever we find one, we can deploy the new function 'scanLitChar' to check whether it is a valid character literals.

```
        -- scanLitChar :: Int -> Int -> String -> D a
        scanLitChar l c (x : '\'' : s) = if ((fromEnum x >= 32) && (fromEnum x <= 126) && (x /= '\'') && (x /= '\\'))
                                         then retTkn (LitChar x) l c (c + 3) s
                                         else do
                                             emitErrD (SrcPos l c)
                                                 ("Lexical error: Illegal \
                                                 \character "
                                                 ++ show x
                                                 ++ " (discarded)")
```

- This is the scanLitChar which identifies character literals without escape character. It starts with a single quotation and ends with a single quotation.

```
        scanLitChar l c ('\\' : x : '\'' : s) = if x == 'n' || x == 'r' || x == 't' || x == '\\' || x == '\''
                                                then retTkn (LitChar (remakeEscape x)) l c (c + 4) s
                                                else do
                                                    emitErrD (SrcPos l c)
                                                        ("Lexical error: Illegal \
                                                        \character "
                                                        ++ show x
                                                        ++ " (discarded)")
                                                    scan l (c + 4) s
```

- This is the scanLitChar which identifies character literals with escape character. It starts with a single quotation and a slash, and ends with a single quotation.

```
        -- remakeEscape :: Char -> Char
        remakeEscape x | x == 'n'  = '\n'
                       | x == 'r'  = '\r'
                       | x == 't'  = '\t'
                       | x == '\\' = '\\'
                       | x == '\'' = '\''
```

- remakeEscape is a function making the testing character back to a escapesequence

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

- This is the scanLitChar which identifies invalid character literals.

In ‘AST.hs’, we extend the character literals as a new expression:

```
        -- | Literal character
        | ExpLitChar {
              elcVal    :: Char,             -- ^ Character value
              expSrcPos :: SrcPos
          }
```

Same in ‘Parser.y’, we extend the primary expression:

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

And add the projection functions:

```
tspLCVal :: (Token,SrcPos) -> Char
tspLCVal (LitChar {lcVal = n}, _) = n
tspLCVal _ = parserErr "tspLCVal" "Not a LitChar"
```

In ‘PPAST.hs’, we add a new pattern to ‘ppExpression’ function to print character literals:

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

Final Abstract Syntax:

Program → *Command*   Program

Command → *Expression* **:=** *Expression*   CmdAssign
| *Expression* (*Expression* *)   CmdCall
| **begin** *Command* * **end**   CmdSeq
| **if** *Expression* **then** *Command* *ElseBranch*   CmdIf
| **while** *Expression* **do** *Command*   CmdWhile
| **let** *Declaration* * **in** *Command*   CmdLet
| **repeat** *Command* **until** *Expression*   CmdRepeat

ElseBranch → **else** *Command*   EBsingle
| **elsif** *Expression* **then** *Command* *ElseBranch*   EBelsif
| **ε**   EBepsilon

Expression → <u>*IntegerLiteral*</u>   ExpLitInt
| <u>*CharacterLiteral*</u>   ExpLitChar
| <u>*Name*</u>   ExpVar
| *Expression* (*Expression* *)   ExpApp
| *Expression* **?** *Expression* **:** *Expression*   ExpCond

Declaration → **const** <u>*Name*</u> **:** *TypeDenoter* **=** *Expression*   DeclConst
| **var** <u>*Name*</u> **:** *TypeDenoter* (**:=** *Expression* | **ε**)   DeclVar

TypeDenoter → <u>*Name*</u>   TDBaseTyp