# Contents

# 1 Task 1

## 1.1 Extending token

First step is to extend the `Token` data type found in *Token.hs* file.

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

## 1.2 Adding keywords to the scanner

Second step is to have new keywords be picked up by our scanner, to do this we have to update `mkIdOrKwd` function in *Scanner.hs* file.

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

## 1.3 Adding new command to AST

We now have defined the token as well as we are able to recognize it using the scanner. The next step is to be able to represent it in out abstract syntax tree. To do so, we update `Command` data type inside `AST.hs`.

```
-- repeat until
| CmdRepeat {
      crCmd     :: Command,         -- ^ Action
      crCond    :: Expression,      -- ^ Condition
      cmdSrcPos :: SrcPos
  }
```

## 1.4 Keyword parser recognision

The next step is to update our parser to support this new syntax (all modification will be done to `Parser.y` file). First we modify the `token` to contain our new defined syntax

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

Next step is to add our command definition to the `command` function:

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

At this point our command will be picked up by the parser and the relative AST will be generated.

## 1.5 Pretty print

To allow debugging of the new syntax by printing, we have to add our command defined in *AST.hs* to *PPAST.hs* file. We add one more pattern match to `ppCommand`

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

We take our indentation level, the command and output stringified result. Show command name and print out inner command as well as expression at further indentation.

# 2 Task 2

## 2.1 Token

First step is to add question mark to `Token` data type in *Token.hs*

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

## 2.2 AST

Next step is to update AST by adding new expression to `Expression` data type.

```
-- | Conditional expression
| ExpCond {
    ecCond      :: Expression,      -- ^ Condition
    ecLeft      :: Expression,      -- ^ Return value when evaluation is true
    ecRight     :: Expression,      -- ^ Return value when evaluation is false
    expSrcPos   :: SrcPos
  }
```

This allows to store our condition, left and right choises in AST

## 2.3 Scanner

We also need to make sure that question mark (?) is picked up as a proper token. Do this by adding another pattern match for scan function

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

## 2.4 Parser

Next we need to make sure that parser picks up this new syntax by first adding new token:

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

Now we can use it to define the expression in the parser

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

It can now be picked up by the parser. We save first expression as condition, second expression as left evaluation and third epxression as right evaluation.

## 2.5 Pretty print

Last step is to make sure we can debug by logging out the parsed expression. Add new pattern match for our newly defined `ExpCond` type

```
pExpression n (ExpCond {ecCond = c, ecLeft = l, ecRight = r, expSrcPos = sp}) =
    indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) c
    . ppExpression (n+1) l
    . ppExpression (n+1) r
```

This will print the name and postion of our type as well as on the rest of new lines with further indentation: condition on first line, left evaluation on second and the right evaluation on third line.

# 3 Task 3

## 3.1 AST

First step is to make the else branch optional by modifying the `ciElse` type signiture to:

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

This also serves additional funcitonality of being to set `ciElse` to another `CmdIf` as it's a command. This will be usefull for *elsif* command

## 3.2 PPAST

Now that the branch is optional it needs to be printed differently. We modify `ppCommand` pattern match for *CmdIf*:

```
ppCommand n (CmdIf {ciCond = e, ciThen = c1, ciElse = c2, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) e
    . ppCommand (n+1) c1
    . ppOpt (n+1) ppCommand c2
```

We are now using `ppOpt (n+1) ppCommand c2` instead of `. ppCommand (n+1) c2`

## 3.3 Token

Next step is to modify `Token` data type for matching *elsif*

```
| ElseIf    -- ^ \"elsif\"
```

## 3.4 Scanner

Next step is to update the scanner to pick up our new syntax. Can do that by adding extra pattern match to `mkIdOrKwd` function

```
mkIdOrKwd "elsif"  = ElseIf
```

## 3.5 Parser

Now we need the parser to be able to recognise our new syntax. Begin by adding new token type

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

Next we have to modify current command parsing for if statements, we change it to

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

Where `ifElseBranch` is defined as:

```
ifElseBranch :: { Maybe Command }
ifElseBranch
    : ELSEIF expression THEN command ifElseBranch
      { Just (CmdIf {ciCond = $2, ciThen = $4, ciElse = $5, cmdSrcPos = $1})   }
    | ELSE command { Just $2 }
    | {- empty -} { Nothing }
```

We use `Maybe` data type here in order to allow for optional branching. This uses the idea that any elseif statement is essentially an if statement inside else statement. For example:

```
if (cond) {} elsif (cond2) {}
// Is essentially the same as
if (cond) {} else { if(cond2) {} }
```

`ifElseBranch` has to be made into a seperate function, else you would be able to do `elsif (cond)` without prior if statement

# 4 Task 4

## 4.1 AST

We begin by adding a new expression to `Expression` data type store our character:

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

## 4.2 Token

Next we have to add new token to `Token` data type:

```
| LitChar {liChar :: Char}          -- ^ Character literals
```

This allows us to store the actual character value inside the type

## 4.3 Scanner

We begin by adding some utility functions:

```
isGrChr :: Char -> Bool
isGrChr '\'' = False
isGrChr '\\' = False
isGrChr '\n' = False
isGrChr '\r' = False
isGrChr '\t' = False
isGrChr _    = True

canEscape :: Char -> Bool
canEscape 'n'  = True
canEscape 'r'  = True
canEscape 't'  = True
canEscape '\\' = True
canEscape '\'' = True
canEscape _    = False
```

`isGrChr` allows us to check if character is graphical and `canEscape` allows to validate whether a character should be escaped or not. Next we add another guard to scan command to check whether we are about to scan a character:

```
| x == '\'' = scanLitChar l c x s
```

In here we call a `scanLitChar` function, which is defined as:

```
scanLitChar l c x s
    -- Assume that less than 2 length means missing quote
    | length s < 2 = failD (SrcPos l c) "Character not closed"
    -- Single character character have to be graphical
    | sc == '\'' && isGrChr fc = retTkn (LitChar fc) l c (c + 3) (drop 2 s)
    | canEscape sc &&
        length ss >= 1 && -- Sanity check, so we don't throw during runtime
        head ss == '\'' = retTkn (escape sc) l c (c + 4) (drop 3 s)
    | otherwise    = failD (SrcPos l c) "Could not parse the character"
    where
        (fc:sc:ss) = s
        escape 'n'  = LitChar '\n'
        escape 'r'  = LitChar '\r'
        escape 't'  = LitChar '\t'
        escape '\\' = LitChar '\\'
        escape '\'' = LitChar '\''
        escape c    = LitChar c
```

We begin by checking that the string is long enough to form a character literal with `length s < 2` guard. Then we take first and second character in `(fc:sc:ss) = s`, this allows to check for the simplest character, which is just a single graphical character. We check that via `sc == '\'` and `isGrChr fc` if it is, we say that next character will be 3 steps ahead (we pass our character and second quote). And drop the first two symbols from the string we got. Next we have to check whether we encountered and escape character by checking if our second character can be escaped. If it is, we check whether the closing quote exists by first validating length of rest of the string. This allows us to confirm that it is valid. The last step is to escape the character and pass it as a character literal. If anything goes wrong, we fail with a message "Could not parse the character".

## 4.4 Parser

Begin with updating our tokens by adding:

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

This allows the parser to recognise it. Next we update `primary_expressions` function by adding new pattern

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

`tspLCVal` is defined as

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

This allows to extract exact character value from the token.