G53CMP – Coursework 1

Note: In lexical syntax and context free syntax terminals are in **bold** (not type-set in typewriter font)

# Task I.1

MiniTriangle is to be extended to include a repeat until loop.

e.g.

repeat

d := d – 1

until d == 0

## Grammar extension

### Lexical Syntax

Keyword -> … | **repeat** | **until**

### Context Free Syntax

*Command* -> … | **repeat** *Command* **until** *Expression*

### Abstract Syntax

Command -> … | **repeat** *Command* **until** *Expression*  CmdRepeat

## Modifications

### Token.hs

Here the 2 keywords are added to the token data structure:

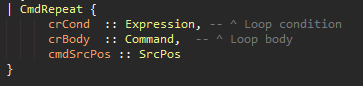


### Scanner.hs

The keywords are then added to the scanner:



### AST.hs



Following my expansion of the AST this code is added to the command data structure expand the abstract syntax tree.

### Parser.y

Terminal symbols were added in the parser.



As in the grammars the parser is modified to look for ‘repeat *command* until *expression’* which isadded in the command function.



### PPAST.hs



The pretty printer is expanded in a similar way to CmdWhile.

## Task I.2

Implement a conditional expression into MiniTriangle.

e.g. boolExp ? true : false

It should be right associative. So e1 ? e2 ? e3 : e4 :e5

Is parsed as e1 ? (e2 ? e3 : e4) : e5

## Grammar extension

### Lexical Syntax

Token -> … | ?

### Context Free Syntax

*Expression* -> … | *Expression* ? *Expression* **:** *Expression*

### Abstract syntax

*Expression* -> … | *Expression* ? *Expression* : *Expression* ExpCond

## Modifications

### Token.hs

The graphical tokens is expanded with:



We do not need to add ‘:’ as that is already implemented as the colon token.

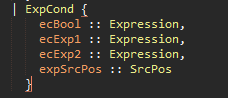
### Scanner.hs

The scanner is extended to include the new ‘?’ token.



### AST.hs

The data structure ‘Expression’ is extended for the conditional expression:



ecBool is the test condition, ecExp1 is the expression if the test condition it true and ecExp2 is the expression if the test condition is false.

### Parser.y

The conditional symbol is added in the terminals like so:

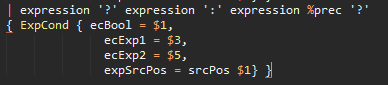


Again no need to add ‘:’ as that already in exists in the terminal symobls.

The precedence of the conditional expression is set for ‘?’ and ‘:’, the expression should right associative.

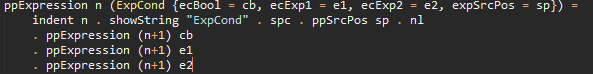


The parser is extended with the new expression definition:



### PPAST.hs

The pretty printer is modified so it can print out the conditional expression



# Task I.3

This task involved the implementation of elsif style if’s along with optional else as the original code did not support if…then without an else statement afterwards.

## Grammar extension

### Lexical Syntax

*Keyword* -> … | **elsif**

### Context Free Syntax

*Command* -> … | **if** *Expression* **then** *Command*

**elsif** *Expression* **then** *Command*

**else** *Command*

### Abstract Syntax Tree

*Command* -> … | **if** *Expression* **then** *Command*

**elsif** *Expression* **then** *Command*

**else** *Command* CmdIf

Else is optional (Maybe Command)

## Modifications

### Token.hs

The elsif keyword is added to the tokens:



### Scanner.hs



If it finds ‘elsif’ return the Elsif token.

### AST.hs

The data structure CmdIf is modified to include the elsif and optional else.

ciElse is modified from Command to Maybe Command, maybe command will return nothing if there is no else or will return a command.

ciElsIf is a list of (expression, command) tuples. It is a set because there can be any number of elsifs.

### 

### Parser.y

The elsif symbol is added to the terminal symbols:



The command function for if ... then is extended to include elsif and optional else, also 2 helper function are added to provide functionality for both features.



#### elsIf ::[(Expression, Command)]}



The function provides the pattern matching of elsifs and the functionality for multiple elsif’s. Returns a list of tuples (expression, command)

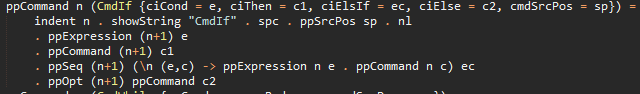
#### oElse :: Maybe Command



The oElse function returns nothing or a command

### PPAST.hs

The existing CmdIf pretty printer function has been expanded to include the elsifs and the optional else. The optional else is now using ppOpt which is an optional pretty printer function. To print the elsifs we use ppSeq which takes a list of items and applies a function to each list item, in our case we create a lambda expression to expand the tuple into expression, and command printed string.



# Task I.4

## Grammar extension

### Lexical Syntax

*Token* -> … | *CharacterLiteral*

### Context Free Syntax

*PrimaryExpression* -> … | *CharacterLiteral*

### Abstract Syntax Tree

*Expression* -> … | *CharacterLiteral* ExpLitChar

## Modifications

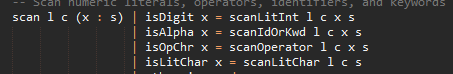
### Token.hs

The character literal non terminal is added to the tokens, with a Haskell type of ‘Char’:



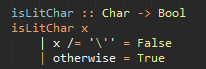
### Scanner.hs

The scanner pattern matches for a literal char, i.e. anything beginning with a ‘, then the rest of the string is passed to a function ‘scanLitChar’



A guard is added to the existing function to check if it begins with a single quote indicating a charater literal is to come.

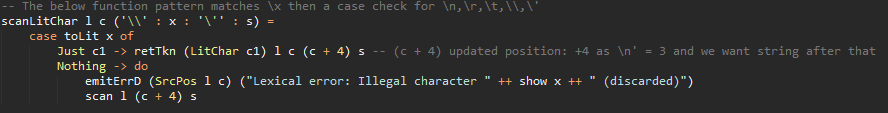
#### isLitChar :: Char -> Bool



The literal charater begins with a single quote, if the character input is not a single it returns false else returns true (meaning it’s a single quote).

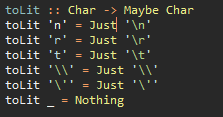
#### scanLitChar :: Int -> Int -> String -> D a

This function pattern matches the string that begins with \, and then checks if it’s one of the valid escaped characters.

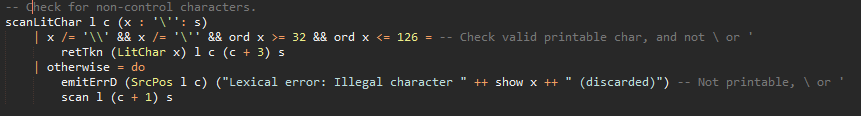


Checked if valid character literal with this Maybe function uses pattern matching and returns the literal character or nothing if it’s not a valid escaped character (doesn’t match above patterns):

#### toLit :: Char -> Maybe Char



The second pattern match for character literals check for a printable character, i.e. ‘a’. It runs a check to make sure the character is a printable character (or non-control character), the range is ASCII ‘ ‘ (space/code 32) to ASCII ‘~’ (tidle/126) it also checks that the character is not \ or ‘ as they would cause an error on the line, i.e. ‘’’ or ‘\’. If it fails the first guard it prints and error.



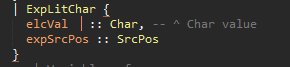
The last pattern match function is to prevent the pattern matches from being non exhaustive:



I.e. if the scanner finds a single quote but no ending single quote.

### AST.hs

The data structure Expression is extended with literal characters:



### Parser.y

The litchar symbol is added to the terminal symbols.



The pattern match for litchar is added to primary\_expression function:



I also created a function to display if an error occurs (not a litchar):



### PPAST.hs

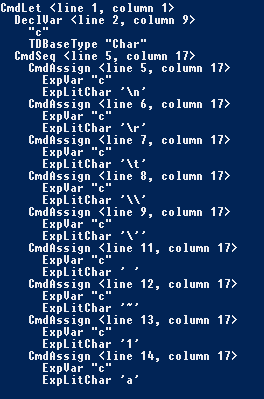
The pretty printer for expression is extended with:



# Testing

There are test files in MTTests that I have created to test the added functionality:

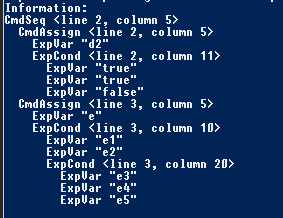
charLitTest – Testing of literal characters (task 4) - Pass



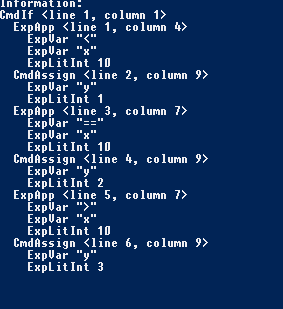
charLitTestErr – Testing errors for ‘\’ and ‘’’ (task 4) - Pass



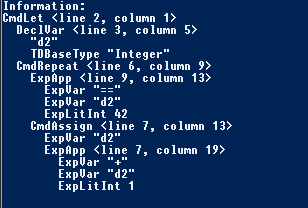
condExpTest.mt – Testing the conditional expression (task 2) - Pass



elsIfTest.mt – Testing the optional else and elsif (task 3) – Pass



repeatUntilTest.mt – Testing the repeat until loop (task 1) - Pass



All written test cases passed.