Alessio Cauteruccio
psyagca
4287379

G53CMP CW1

Task 1.1

Extend MiniTriangle with a repeat-loop. Informally, the loop construct has the following syntax:

repeat
cmd
until
boolExp

Token.hs ->

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

Scanner.hs ->

```
-- | MiniTriangle scanner.
scanner :: ((Token, SrcPos) -> P a) -> P a
….
      mkIdOrKwd "repeat"  = Repeat
      mkIdOrKwd "until"   = Until
```

AST.hs ->

```
-- | Repeat-Until-command
   | CmdRep {
       crComm    :: Command,        -- ^ Repeat-condition
       cuExpr    :: Expression,     -- ^ Until-expression
       cmdSrcPos :: SrcPos
    }

Parser.y -> command :: { Command }
….
        | REPEAT command UNTIL expression
                { CmdRep {crComm = $2, cuExpr = $4, cmdSrcPos = $1} }
```

PPAST.hs ->

```
ppCommand n (CmdRep {crComm = c, cuExpr = e, cmdSrcPos = sp}) =
   indent n . showString "CmdRep" . spc . ppSrcPos sp . nl
   . ppCommand (n+1) c
   . ppExpression (n+1) e
```

Alessio Cauteruccio
psyagca
4287379

Lexical Syntax Extension

Program -> *(Token | Separator)*
Token -> *Keyword | Identifier | IntegerLiteral | Operator*
| , | ; | : | := | = | (|) | eot
Keyword -> *begin | const | do | else | end | if | in*
| let | then | var | while | ***repeat*** *|* ***until***
Identifier -> *Letter | Identifier Letter | Identifier Digit*
except Keyword
IntegerLiteral -> *Digit | IntegerLiteral Digit*
Operator -> ^ | * | / | + | - | < | <= | == | *!*= | >= | > | && | || | *!*
Letter -> *A | B | . . . | Z | a | b | . . . | z*
Digit -> *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
Separator -> *Comment | space | eol*
Comment -> *// (any character except eol) eol*

Context-Free Syntax

Program -> *Command*
Commands -> *Command*
| Command ; Commands
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***
Expressions -> *Expression*
| Expression , Expressions
Expression -> *PrimaryExpression*
| Expression BinaryOperator Expression
PrimaryExpression -> IntegerLiteral
| VarExpression
| UnaryOperator PrimaryExpression
| (Expression)
VarExpression -> Identifier
BinaryOperator -> ^ | * | / | + | - | < | <= | == | *!*= | >= | > | && | ||
UnaryOperator -> - | !
Declarations -> *Declaration*
| Declaration ; Declarations
Declaration -> *const Identifier : TypeDenoter = Expression*
| var Identifier : TypeDenoter
| var Identifier : TypeDenoter := Expression
TypeDenoter -> *Identifier*

Alessio Cauteruccio
psyagca
4287379

Abstract Syntax

Program	*-> Command Program*	
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	**CmdRep**
Expression	*-> IntegerLiteral*	ExpLitInt
	| Name	ExpVar
	| Expression (Expression)	ExpApp
Declaration	*-> const Name : TypeDenoter = Expression*	DeclConst
	| var Name : TypeDenoter (:= Expression | ϱ)	DeclVar
TypeDenoter	*-> Name*	TDBaseType

Task 1.2

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

boolExp ? exp1 : exp2

Token.hs ->

```
data Token
   -- Graphical tokens
   ...
   | Question  -- ^ \"?\" T1.2
```

Scanner.hs ->

```
scanner :: ((Token, SrcPos) -> P a) -> P a
scanner cont = P $ scan
   where
        mkOpOrSpecial :: String -> Token
        mkOpOrSpecial "?"  = Question --T1.2
```

AST.hs ->

```
data Expression
   ...
   -- | Conditional Expression T1.2
   | ExpCond {
        eaBool    :: Expression,
        eaFirst   :: Expression,
        eaSecond  :: Expression,
```

Alessio Cauteruccio
psyagca
4287379

```
        expSrcPos :: SrcPos
    }
```

Parser.y ->

```
%token
   ...
   '?'        { (Question, $$) } --T1.2
…
%right ':' --T1.2
...
expression :: { Expression }
expression
   : primary_expression
   ...
   -- T1.2
   | expression '?' expression ':' expression
      { ExpCond {eaBool    = $1,
                 eaFirst   = $3,
                 eaSecond  = $5,
                 expSrcPos = srcPos $1} }
```

PPAST.hs ->

```
ppExpression :: Int -> Expression -> ShowS
...
-- T1.2
ppExpression n (ExpCond {eaBool = b, eaFirst = f, eaSecond = s, expSrcPos = sp}) =
   indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
   . ppExpression (n+1) b
   . ppExpression (n+1) f
   . ppExpression (n+1) s
```

Lexical Syntax Extension

Program -> *(Token | Separator)*
Token -> *Keyword | Identifier | IntegerLiteral | Operator*
| , | ; | : | := | = | (|) | ***?*** *| eot*
Keyword -> *begin | const | do | else | end | if | in*
| let | then | var | while | repeat | until
Identifier -> *Letter | Identifier Letter | Identifier Digit*
except Keyword
IntegerLiteral -> *Digit | IntegerLiteral Digit*
Operator -> ^ | * | / | + | - | < | <= | == | *!=* | >= | > | && | || | *!*
Letter -> *A | B | . . . | Z | a | b | . . . | z*
Digit -> *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
Separator -> *Comment | space | eol*
Comment -> *// (any character except eol) eol*

Alessio Cauteruccio
psyagca
4287379

Context-Free Syntax

Program *-> Command*
Commands *-> Command*
| Command ; Commands
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
Expressions *-> Expression*
| Expression , Expressions
Expression *-> PrimaryExpression*
| Expression BinaryOperator Expression
*| **Expression '?' Expression ':' Expression***
PrimaryExpression -> IntegerLiteral
| VarExpression
| UnaryOperator PrimaryExpression
| (Expression)
VarExpression -> Identifier
BinaryOperator -> ^ | * | / | + | - | < | <= | == | *!*= | >= | > | && | ||
UnaryOperator -> - | *!*
Declarations *-> Declaration*
| Declaration ; Declarations
Declaration *-> const Identifier : TypeDenoter = Expression*
| var Identifier : TypeDenoter
| var Identifier : TypeDenoter := Expression
TypeDenoter *-> Identifier*

Abstract Syntax

Program	*-> Command Program*	
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	CmdRep
Expression	*-> IntegerLiteral*	ExpLitInt
	| Name	ExpVar
	| Expression (Expression)	ExpApp
	*| **Expression '?' Expression ':' Expression***	**ExpCond**
Declaration	*-> const Name : TypeDenoter = Expression*	DeclConst

Alessio Cauteruccio
psyagca
4287379

| *var Name : TypeDenoter*
(:= Expression | ϱ) DeclVar
TypeDenoter -> Name TDBaseType

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.*

Token.hs ->

```
data Token
...
   -- Keywords
   ...
   | Elsif     -- ^ \"elsif\"  T1.3
```

Scanner.hs ->

```
scanner :: ((Token, SrcPos) -> P a) -> P a
scanner cont = P $ scan
   where
      ...
      mkIdOrKwd :: String -> Token
      ...
      mkIdOrKwd "elsif"   = Elsif --T1.3
```

AST.hs ->

```
module AST (
   ...
   ElsifCommand (..),  -- Not abstract. Instances: HasSrcPos. T1.3
...
data Command
   ...
   -- | Conditional command T1.3
   | CmdIf {
        ciCond    :: Expression,      -- ^ Condition
        ciThen    :: ElsifCommand,        -- ^ Then-branch
        cmdSrcPos :: SrcPos
     }
…
data ElsifCommand --T1.3
   = Cmd { --Command
```

Alessio Cauteruccio
psyagca
4287379

```
        cmd :: Command,
        elCmdSrcPos :: SrcPos
     }
   | ElCmd { --Cmd-else-cmd
        elfCmd :: Command,
        elsCmd :: Command,
        elCmdSrcPos :: SrcPos
     }
   | ElsifCmd { --cmd-elsif-exp-then-elsifcmd
        eifCmd :: Command,
        eifExp :: Expression,
        einCmd :: ElsifCommand,
        elCmdSrcPos :: SrcPos
     }
```

Parser.y ->

```
%token
          ...
          ELSIF       { (Elsif, $$) }  --T1.3
…
command :: { Command }
command…
          | IF expression THEN elsifCommand --T1.3
          { CmdIf {ciCond = $2, ciThen = $4, cmdSrcPos = $1} }
…
--T1.3
elsifCommand :: { elsifCommand }
elsifCommand
   : command
      { Cmd {cmd = $1, elCmdSrcPos = srcPos $1} }
   | command ELSE command
      { ElCmd {elfCmd = $1, elsCmd = $3, elCmdSrcPos = srcPos $1} }
         | command ELSIF expression THEN elsifCommand
      { ElsifCmd {eifCmd = $1, eifExp = $3, einCmd = $5, elCmdSrcPos = srcPos $1 } }
```

PPAST.hs ->

```
ppCommand :: Int -> Command -> ShowS
...
--T1.3
ppCommand n (CmdIf {ciCond = e, ciThen = c1, cmdSrcPos = sp}) =
   indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
```

Alessio Cauteruccio
psyagca
4287379

```
   . ppExpression (n+1) e
   . ppElsifCommand (n+1) c1
…
-- Pretty Printing of elsif T1.3
ppElsifCommand :: Int -> ElsifCommand -> ShowS
ppElsifCommand n (Cmd {cmd = c, elCmdSrcPos = sp}) =
   indent n . showString "Cmd" . spc . ppSrcPos sp . nl
   . ppCommand (n+1) c
ppElsifCommand n (ElCmd {elfCmd = f, elsCmd = s, elCmdSrcPos = sp}) =
   indent n . showString "ElCmd" . spc . ppSrcPos sp . nl
   . ppCommand (n+1) f
   . ppCommand (n+1) s
ppElsifCommand n (ElsifCmd {eifCmd = c, eifExp = e, einCmd = ec, elCmdSrcPos = sp}) =
   indent n . showString "ElsifCmd" . spc . ppSrcPos sp . nl
   . ppCommand (n+1) c
   . ppExpression (n+1) e
   . ppElsifCommand (n+1) ec
```

Lexical Syntax Extension

Program -> *(Token | Separator)*
Token -> *Keyword | Identifier | IntegerLiteral | Operator*
| , | ; | : | := | = | (|) | ? | eot
Keyword -> *begin | const | do | else | end | if | in*
| let | then | var | while | repeat | until | ***elsif***
Identifier -> *Letter | Identifier Letter | Identifier Digit*
except Keyword
IntegerLiteral -> *Digit | IntegerLiteral Digit*
Operator -> ^ | * | / | + | - | < | <= | == | *!=* | >= | > | && | || | *!*
Letter -> *A | B | . . . | Z | a | b | . . . | z*
Digit -> *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
Separator -> *Comment | space | eol*
Comment -> *// (any character except eol) eol*

Context-Free Syntax

Program -> *Command*
Commands -> *Command*
| Command ; Commands
Command -> *VarExpression := Expression*
| VarExpression (Expressions)
| if Expression then ElsifCommand
| while Expression do Command
| let Declarations in Command
| begin Commands end
| repeat Command until Expression

Alessio Cauteruccio
psyagca
4287379

ElsifCommand	***-> Command***
	| Command else Command
	| Command elsif Expression then ElsifCommand
Expressions	*-> Expression*
	| Expression , Expressions
Expression	*-> PrimaryExpression*
	| Expression BinaryOperator Expression
	| Expression '?' Expression ':' Expression
PrimaryExpression	*-> IntegerLiteral*
	| VarExpression
	| UnaryOperator PrimaryExpression
	| (Expression)
VarExpression	*-> Identifier*
BinaryOperator	*-> ^ | * | / | + | - | < | <= | == | != | >= | > | && | ||*
UnaryOperator	*-> - | !*
Declarations	*-> Declaration*
	| Declaration ; Declarations
Declaration	*-> const Identifier : TypeDenoter = Expression*
	| var Identifier : TypeDenoter
	| var Identifier : TypeDenoter := Expression
TypeDenoter	*-> Identifier*

Abstract Syntax

Program	*-> Command Program*	
Command	*-> Expression := Expression*	CmdAssign
	| Expression (Expression)	CmdCall
	| begin Command end	CmdSeq
	| if Expression then ElsifCommand	**CmdIf**
	| while Expression do Command	CmdWhile
	| let Declaration in Command	CmdLet
	| repeat Command until Expression	CmdRep
ElsifCommand	**-> Command**	**Cmd**
	| Command else Command	**ElCmd**
	| Command elsif Expression then ElsifCommand	**ElsifCmd**
Expression	*-> IntegerLiteral*	ExpLitInt
	| Name	ExpVar
	| Expression (Expression)	ExpApp
	| Expression '?' Expression ':' Expression	ExpCond
Declaration	*-> const Name : TypeDenoter = Expression*	DeclConst
	| var Name : TypeDenoter	
	(:= Expression | ϱ)	DeclVar
TypeDenoter	*-> Name*	TDBaseType

Alessio Cauteruccio
psyagca
4287379

Task 1.4

Extend 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 | \ | ’)

Token.hs ->

```
data Token
   ...
   -- Tokens with variable spellings
   ...
   | CharLit {chVal :: Char}          -- ^ Character Literals T1.4
```

Scanner.hs ->

```
scanner :: ((Token, SrcPos) -> P a) -> P a
scanner cont = P $ scan
   where
      ...
      -- Scan Character Literals T1.4
      scan l c ('\'' : x : '\'' : s) = scanCharLit l c x s
      scan l c ('\'' : '\\' : x :'\'' : s) = retTkn (CharLit (convertEscCh x)) l c (c+4) s
      ...
      -- T1.4
      scanCharLit l c x s | (x /= '\'') && (x /= '\\') = retTkn (CharLit x) l c (c+3) s
                          | otherwise = do
                                  emitErrD (SrcPos l c)
                                        ("Lexical error: Illegal \
                                         \character "
                                         ++ show x
                                         ++ " (discarded)")
                                  scan l (c + 1) s
      convertEscCh x | x == 'n' = '\n'
                     | x == 'r' = '\r'
                     | x == 't' = '\t'
                     | x == '\\' = '\\'
                     | x == '\'' = '\''
```

AST.hs ->

```
data Expression
   ...
   -- | Character Literal T1.4
   | ExpCharLit {
        eChLit    :: Char,
        expSrcPos :: SrcPos
    }
```

Alessio Cauteruccio
psyagca
4287379

Parser.y ->

```
%token
   ...
   CHARLIT     { (CharLit {}, _) } --T1.4
…
primary_expression :: { Expression }
   : ...
   | CHARLIT --T1.4
      { ExpCharLit {eChLit = tspChVal $1 , expSrcPos = tspSrcPos $1} }
…
-- T1.4
tspChVal :: (Token,SrcPos) -> Char
tspChVal (CharLit {chVal = c}, _) = c
tspChVal _ = parserErr "tspChVal" "Not a Character Literal"
```

PPAST ->

```
ppExpression :: Int -> Expression -> ShowS
-- T1.4
ppExpression n (ExpCharLit {eChLit = c}) =
   indent n . showString "ExpCharLit". spc . shows c . nl
```

Lexical Syntax Extension

Program	*-> (Token | Separator)*
Token	*-> Keyword | Identifier | IntegerLiteral | Operator*
	| , | ; | : | := | = | (|) | ? | eot
Keyword	*-> begin | const | do | else | end | if | in*
	| let | then | var | while | repeat | until | elsif
Identifier	*-> Letter | Identifier Letter | Identifier Digit*
	except Keyword
IntegerLiteral	*-> Digit | IntegerLiteral Digit*
CharacterLiteral	***-> ' (Graphic | CharEscape) '***
Graphic	***-> any non-control character except ' and ***
CharEscape	***-> \ (n | r | t | \ | ')***
Operator	*-> ^ | * | / | + | - | < | <= | == | != | >= | > | && | || | !*
Letter	*-> A | B | . . . | Z | a | b | . . . | z*
Digit	*-> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
Separator	*-> Comment | space | eol*
Comment	*-> // (any character except eol) eol*

Context-Free Syntax

Program	*-> Command*
Commands	*-> Command*
	| Command ; Commands
Command	*-> VarExpression := Expression*

Alessio Cauteruccio
psyagca
4287379

	| VarExpression (Expressions)
	| if Expression then ElsifCommand
	| while Expression do Command
	| let Declarations in Command
	| begin Commands end
	| repeat Command until Expression
ElsifCommand	*-> Command*
	| Command else Command
	| Command elsif Expression then ElsifCommand
Expressions	*-> Expression*
	| Expression , Expressions
Expression	*-> PrimaryExpression*
	| Expression BinaryOperator Expression
	| Expression '?' Expression ':' Expression
PrimaryExpression	*-> IntegerLiteral*
	| VarExpression
	| UnaryOperator PrimaryExpression
	| (Expression)
	| CharLit
VarExpression	*-> Identifier*
BinaryOperator	*-> ^ | * | / | + | - | < | <= | == | != | >= | > | && | ||*
UnaryOperator	*-> - | !*
Declarations	*-> Declaration*
	| Declaration ; Declarations
Declaration	*-> const Identifier : TypeDenoter = Expression*
	| var Identifier : TypeDenoter
	| var Identifier : TypeDenoter := Expression
TypeDenoter	*-> Identifier*

Abstract Syntax

Program	*-> Command Program*	
Command	*-> Expression := Expression*	CmdAssign
	| Expression (Expression)	CmdCall
	| begin Command end	CmdSeq
	| if Expression then ElsifCommand	CmdIf
	| while Expression do Command	CmdWhile
	| let Declaration in Command	CmdLet
	| repeat Command until Expression	CmdRep
ElsifCommand	-> Command	Cmd
	| Command else Command	ElCmd
	| Command elsif Expression then ElsifCommand	ElsifCmd
Expression	*-> IntegerLiteral*	ExpLitInt
	| Name	ExpVar
	| Expression (Expression)	ExpApp
	| Expression '?' Expression ':' Expression	ExpCond
	| CharLit	**ExpCharLit**

Alessio Cauteruccio
psyagca
4287379

Declaration	*-> const Name : TypeDenoter = Expression*	DeclConst
	| var Name : TypeDenoter	
	(:= Expression | ϱ *)*	DeclVar
TypeDenoter	*-> Name*	TDBaseType