Task II.1

Before extending real code, we first extend the MiniTriangle type system for the four new constructs we added in CW1.

- **Repeat-until-loops:**

$$\frac{\Gamma \vdash e : Boolean \quad \Gamma \vdash c}{\Gamma \vdash repeat\ c\ until\ e}$$

- **Conditional expressions:**

$$\frac{\Gamma \vdash e_1 : Boolean \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash (e_1\ ?\ e_2\ :\ e_3) : T}$$

- **Extended syntax for the if-command:**

$$\frac{\Gamma \vdash e_1 : Boolean \quad \Gamma \vdash \bar{e}_2 : Boolean \quad \Gamma \vdash c_1 \quad \Gamma \vdash \bar{c}_2 \quad \Gamma \vdash c_3^{\leq 1}}{\Gamma \vdash if\ e_1\ then\ c_1\ elsif\ \bar{e}_2\ then\ \bar{c}_2\ else\ c_3^{\leq 1}}$$

- **Character literals:**

$$\Gamma \vdash c : Character$$

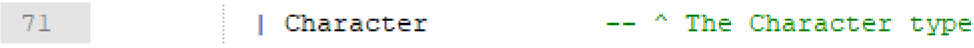
TaskII.2

After extending the type system, we now need to modify real code to enable checker to analyse context grammar after parser's lexical analysis.

I implemented the four constructs one at a time in the following order:

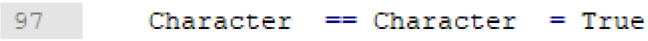
1. **Character Literals:**

- *Type.hs:*

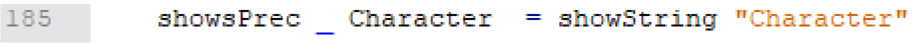


```
    | Character          -- ^ The Character type
```

Adding Character as a new type in Type representation.



```
    Character  == Character  = True
```



```
    showsPrec _ Character  = showString "Character"
```

At the same time, extended Eq and Show instance for this type.

- *MTStdEnv.hs:*

```
            ("Character", Character)]
```

Add type Character to [Types:] in MiniTriangle initial environment.

```
            ("getchr",  Arr [Snk Character] Void,       ESVLbl "getchr"),
            ("putchr",  Arr [Character] Void,           ESVLbl "putchr"),
```

Add ‘getchr’ and ‘putchr’ to [Procedures:] in MiniTriangle initial environment.

‘getchr’ reads user input from terminal and has a write-only reference.

In MTIR.hs and PPMTIR.hs, Character has already been implemented. LibMT.hs will be modified in TaskII.3 to agree with the code we modified here.

- *TypeChecker.hs:*

```
-- T-LITCHR
infTpExp env e@(A.ExpLitChr {A.elcVal = c, A.expSrcPos = sp}) = do
    c' <- toMTChr c sp
    return (Character,                            -- env |- n : Character
            ExpLitChr {elcVal = c', expType = Character, expSrcPos = sp})
```

Check type of the expression.

Return the expected type and expression in Monad Type D.

2. **Repeat-until-loops:**

- *MTIR.hs:*

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

Add repeat-until-loops to Command in MiniTriangle internal representation.

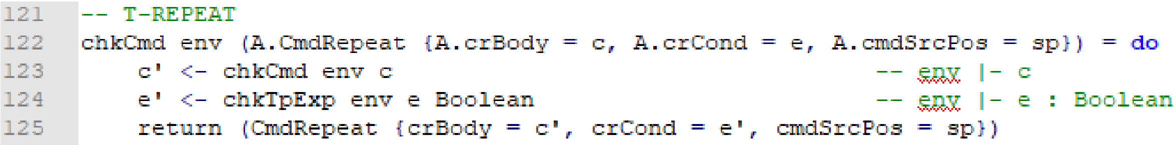
In MTIR, Command has the same structure as in AST.

- *PPMTIR.hs:*

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

Same to the pretty printer for MTIR.

- *TypeChecker.hs:*



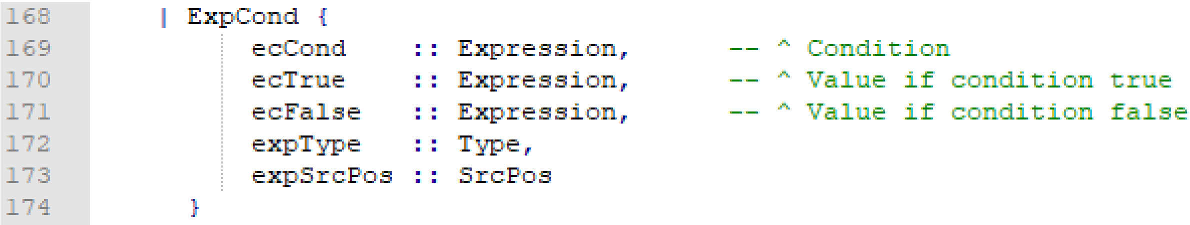
```
-- T-REPEAT
chkCmd env (A.CmdRepeat {A.crBody = c, A.crCond = e, A.cmdSrcPos = sp}) = do
    c' <- chkCmd env c                                  -- env |- c
    e' <- chkTpExp env e Boolean                        -- env |- e : Boolean
    return (CmdRepeat {crBody = c', crCond = e', cmdSrcPos = sp})
```

Checks command and type of expression (Loop-condition should be a Boolean).

Transfer from AST Type to Monad Type D and return them for further manipulation.

3. Conditional Expressions:

- *MTIR.hs:*

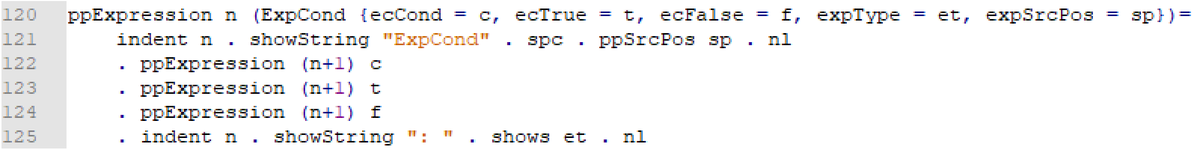


```
    | ExpCond {
          ecCond    :: Expression,      -- ^ Condition
          ecTrue    :: Expression,      -- ^ Value if condition true
          ecFalse   :: Expression,      -- ^ Value if condition false
          expType   :: Type,
          expSrcPos :: SrcPos
      }
```

Add conditional expression to Expression in MiniTriangle internal representation.

There is an expected return type in conditional expression (different from AST).

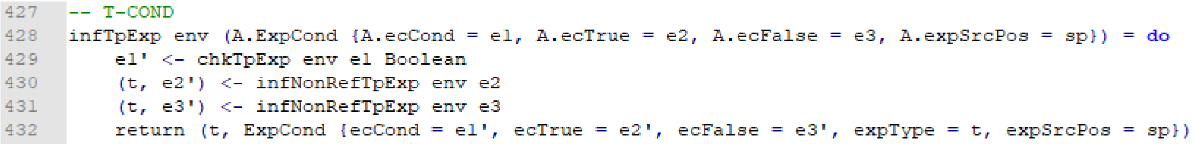
- *PPMTIR.hs:*



```
ppExpression n (ExpCond {ecCond = c, ecTrue = t, ecFalse = f, expType = et, expSrcPos = sp})=
    indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
    . ppExpression (n+1) c
    . ppExpression (n+1) t
    . ppExpression (n+1) f
    . indent n . showString ": " . shows et . nl
```

Same to the pretty printer for MTIR.

- *TypeChecker.hs:*



```
-- T-COND
infTpExp env (A.ExpCond {A.ecCond = e1, A.ecTrue = e2, A.ecFalse = e3, A.expSrcPos = sp}) = do
    e1' <- chkTpExp env e1 Boolean
    (t, e2') <- infNonRefTpExp env e2
    (t, e3') <- infNonRefTpExp env e3
    return (t, ExpCond {ecCond = e1', ecTrue = e2', ecFalse = e3', expType = t, expSrcPos = sp})
```

Since conditional expression is an expression, it should have a return type.

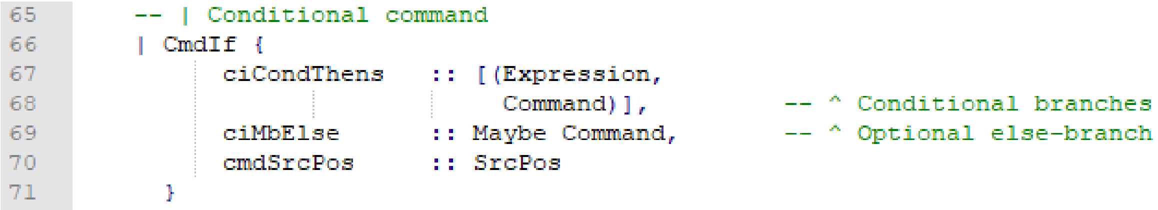
However, e2 and e3 allow to have variables in them, which are reference types.

Therefore, if we need checker to do contextual analysis, we first need to use infNonRefTpExp function to resolve the type of e2 and e3 before checker's contextual analysis.

When we use infNonRefTpExp function, we assume that e2 and e3 have the same type. Thus, the whole conditional expression will only have one kind of return type.

4. Extension for if-then-else-commands:

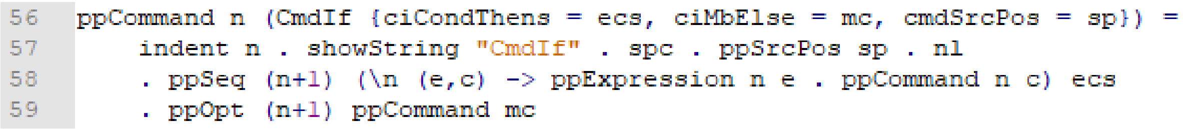
- *MTIR.hs:*



```
    -- | Conditional command
    | CmdIf {
          ciCondThens   :: [(Expression,
                              Command)],        -- ^ Conditional branches
          ciMbElse      :: Maybe Command,       -- ^ Optional else-branch
          cmdSrcPos     :: SrcPos
      }
```

Extend if-then-else commands to Command in MiniTriangle internal representation (same as AST).

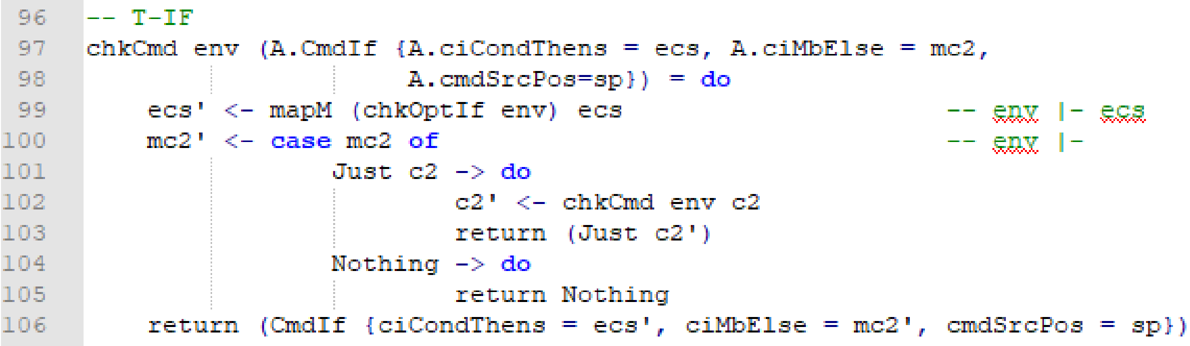
- *PPMTIR.hs:*



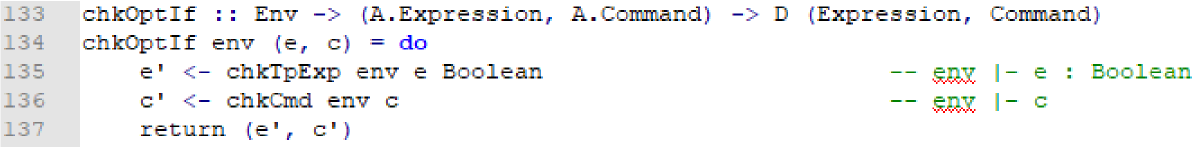
```
ppCommand n (CmdIf {ciCondThens = ecs, ciMbElse = mc, cmdSrcPos = sp}) =
    indent n . showString "CmdIf" . spc . ppSrcPos sp . nl
    . ppSeq (n+1) (\n (e,c) -> ppExpression n e . ppCommand n c) ecs
    . ppOpt (n+1) ppCommand mc
```

Same to the pretty printer for MTIR (same as PPAST).

- *TypeChecker.hs:*



```
-- T-IF
chkCmd env (A.CmdIf {A.ciCondThens = ecs, A.ciMbElse = mc2,
                     A.cmdSrcPos=sp}) = do
    ecs' <- mapM (chkOptIf env) ecs                    -- env |- ecs
    mc2' <- case mc2 of                                -- env |-
                Just c2 -> do
                        c2' <- chkCmd env c2
                        return (Just c2')
                Nothing -> do
                        return Nothing
    return (CmdIf {ciCondThens = ecs', ciMbElse = mc2', cmdSrcPos = sp})
```



```
chkOptIf :: Env -> (A.Expression, A.Command) -> D (Expression, Command)
chkOptIf env (e, c) = do
    e' <- chkTpExp env e Boolean                     -- env |- e : Boolean
    c' <- chkCmd env c                               -- env |- c
    return (e', c')
```

ecs represents a list of tuples, consist of an expression and a command (ecs : [(e, c)]).

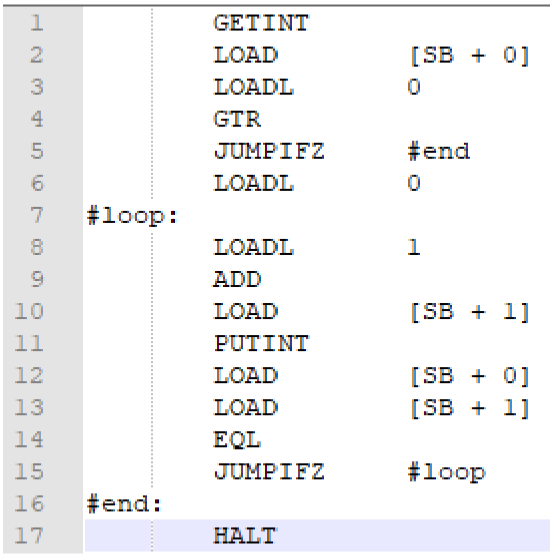
mc2 represents a command wrapped with a Maybe type. Maybe type allows the else-branch can be missing.

chkOptIf is a helper function which checks a single tuple consisted of an expression (should be a Boolean) and a command and transfer them into Monad Type D and return the new tuple.

MapM function enables chkOptIf to operate on every element of ecs list.

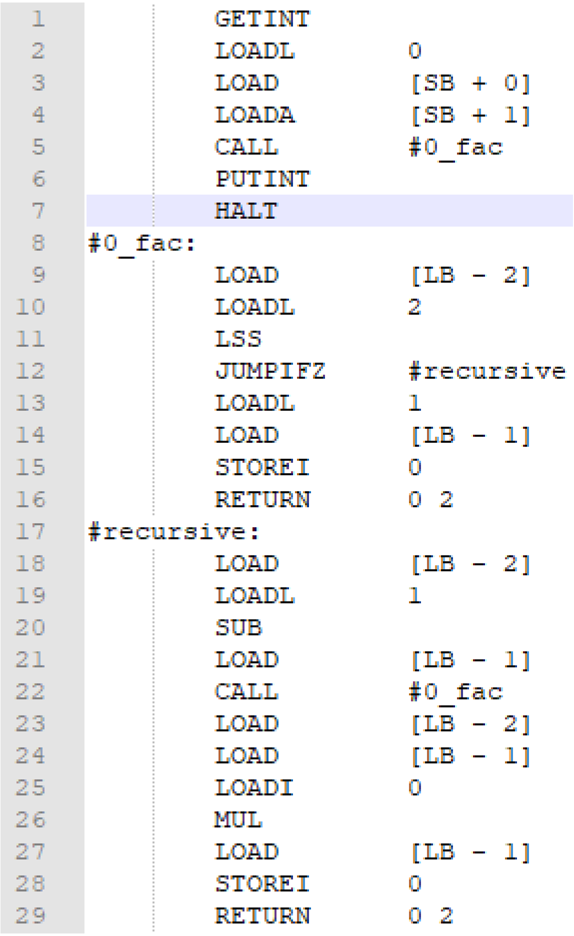
TaskII.3

- *myTamCode3a:*



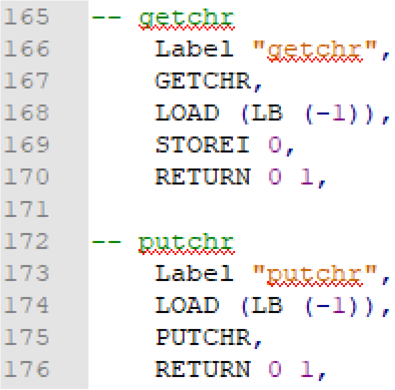
```
        GETINT
        LOAD        [SB + 0]
        LOADL       0
        GTR
        JUMPIFZ     #end
        LOADL       0
#loop:
        LOADL       1
        ADD
        LOAD        [SB + 1]
        PUTINT
        LOAD        [SB + 0]
        LOAD        [SB + 1]
        EQL
        JUMPIFZ     #loop
#end:
        HALT
```

- *myTamCode3b:*



```
        GETINT
        LOADL       0
        LOAD        [SB + 0]
        LOADA       [SB + 1]
        CALL        #0_fac
        PUTINT
        HALT
#0_fac:
        LOAD        [LB - 2]
        LOADL       2
        LSS
        JUMPIFZ     #recursive
        LOADL       1
        LOAD        [LB - 1]
        STOREI      0
        RETURN      0 2
#recursive:
        LOAD        [LB - 2]
        LOADL       1
        SUB
        LOAD        [LB - 1]
        CALL        #0_fac
        LOAD        [LB - 2]
        LOAD        [LB - 1]
        LOADI       0
        MUL
        LOAD        [LB - 1]
        STOREI      0
        RETURN      0 2
```

- *LibMT.hs:*



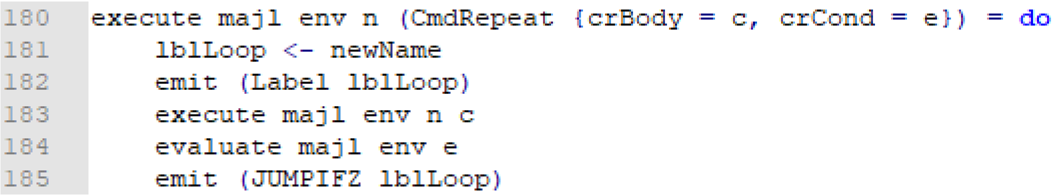
```
-- getchr
    Label "getchr",
    GETCHR,
    LOAD (LB (-1)),
    STOREI 0,
    RETURN 0 1,

-- putchr
    Label "putchr",
    LOAD (LB (-1)),
    PUTCHR,
    RETURN 0 1,
```

Add ‘getchr’ and ‘putchr’ to agree with codes we modified in TaskII.2.

TaskII.4

- *Repeat-until-loop:*

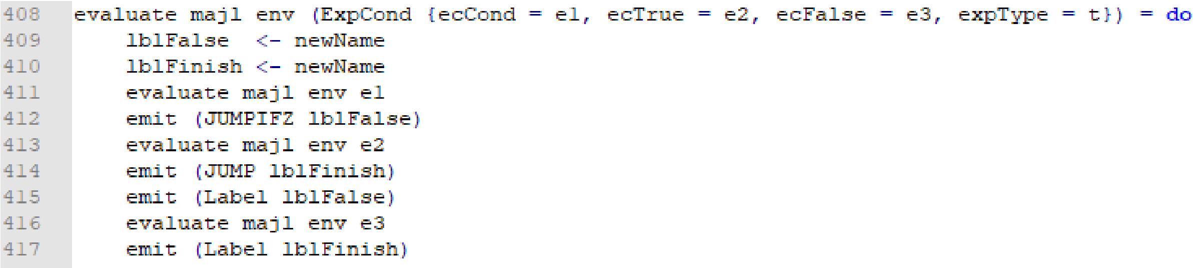


```
execute majl env n (CmdRepeat {crBody = c, crCond = e}) = do
    lblLoop <- newName
    emit (Label lblLoop)
    execute majl env n c
    evaluate majl env e
    emit (JUMPIFZ lblLoop)
```

Repeat-until-loop is similar with While-loop.

The only difference is the Repeat-until-loop will always run for at least one time. Therefore, there is no need to check loop-condition at first.

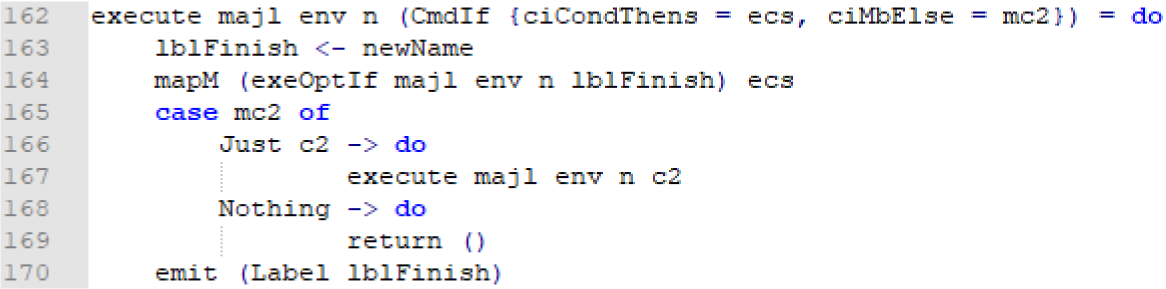
- *Conditional Expressions:*



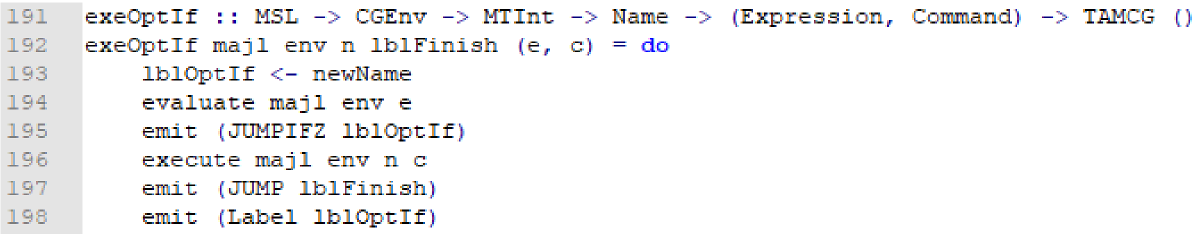
```
evaluate majl env (ExpCond {ecCond = e1, ecTrue = e2, ecFalse = e3, expType = t}) = do
    lblFalse  <- newName
    lblFinish <- newName
    evaluate majl env e1
    emit (JUMPIFZ lblFalse)
    evaluate majl env e2
    emit (JUMP lblFinish)
    emit (Label lblFalse)
    evaluate majl env e3
    emit (Label lblFinish)
```

Conditional Expression is similar with simple If-Command. Since in the generator, there is no worry about types. Just add expected return type and appropriate function (‘evaluate’ not ‘execute’) to it.

- *If-then-else-command:*



```
execute majl env n (CmdIf {ciCondThens = ecs, ciMbElse = mc2}) = do
    lblFinish <- newName
    mapM (exeOptIf majl env n lblFinish) ecs
    case mc2 of
        Just c2 -> do
                execute majl env n c2
        Nothing -> do
                return ()
    emit (Label lblFinish)
```



```
exeOptIf :: MSL -> CGEnv -> MTInt -> Name -> (Expression, Command) -> TAMCG ()
exeOptIf majl env n lblFinish (e, c) = do
    lblOptIf <- newName
    evaluate majl env e
    emit (JUMPIFZ lblOptIf)
    execute majl env n c
    emit (JUMP lblFinish)
    emit (Label lblOptIf)
```

Similar with what I have done for the Typechecker.hs.

exeOptIf helper function can generate code for every if-then branch and elsif-then branch.

Wrapped with Maybe type allows else-branch to be optional.