Contents

1 Task 1

1.1 Repeat until

$$\frac{\Gamma \vdash e : \text{Boolean} \qquad \Gamma \vdash \bar{c}}{\Gamma \vdash \text{repeat } \bar{c} \text{ until } e}$$

1.2 Character literal

$$\Gamma \vdash ch : \text{Character}$$

1.3 If-else extended

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \text{Boolean} \qquad \Gamma \vdash \bar{e}_2 : \text{Boolean} \\ \Gamma \vdash \bar{c}_1^{\geq 1} \qquad \Gamma \vdash \bar{c}_2 \qquad \Gamma \vdash \bar{c}_3\end{array}}{\Gamma \vdash \text{if } e_1 \text{ then } \bar{c}_1 \text{ elsif } \bar{e}_2 \text{ then } \bar{c}_2 \text{ else } \bar{c}_3}$$

1.4 Conditional expression

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \text{Boolean} \qquad \Gamma \vdash e_2 : T \\ \Gamma \vdash e_3 : T \qquad \neg\, \text{reftype}(T)\end{array}}{\Gamma \vdash e_1 \,?\, e_2 : e_3}$$

2 Task 2

2.1 Repeat until

2.1.1 MTIR

Update the MiniTriangle Internal Representation inside, so we can stored typed version

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

2.1.2 TypeChecker

Add a pattern match for type checking AST `CmdRepeat` data type

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

2.1.3 PPMTIR

Now need a way to print the typed repeat command. We do this by adding a `CmdRepeat` pattern match to `ppCommand`

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

2.2 Character literal

2.2.1 Type

Firstly we add `Character` to Type data type

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

Next inside `instance Eq Type where` we add an equality operator pattern for it.

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

Finally, we add Character pattern match to `instance Show Type where`

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

2.2.2 TypeChecker

We add a `ExpLitChar` pattern match to `infTpExp`. The only thing we do here is convert the character value to `MTChar` and transform $AST \rightarrow MTIR$

```
-- T-CHAR
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})
```

2.2.3 MTStdEnv

We also need to update our standard environment to contain characters. Do this by updating first list argument in `mtStdEnv` function

```
mtStdEnv :: Env
mtStdEnv =
    mkTopLvlEnv
        [("Boolean",   Boolean),
         ("Integer",   Integer),
         ("Character", Character)]
        [("false",   Boolean, ESVBool False),
         ("true",    Boolean, ESVBool True),
         ("minint",  Integer, ESVInt (minBound :: MTInt)),
         ("maxint",  Integer, ESVInt (maxBound :: MTInt)),
         ("+",       Arr [Integer, Integer] Integer, ESVLbl "add"),
         ("-",       Arr [Integer, Integer] Integer, ESVLbl "sub"),
         ("*",       Arr [Integer, Integer] Integer, ESVLbl "mul"),
         ("/",       Arr [Integer, Integer] Integer, ESVLbl "div"),
         ("^",       Arr [Integer, Integer] Integer, ESVLbl "pow"),
         ("neg",     Arr [Integer] Integer,          ESVLbl "neg"),
         ("<",       Arr [Integer, Integer] Boolean, ESVLbl "lt"),
         ("<=",      Arr [Integer, Integer] Boolean, ESVLbl "le"),
         ("==",      Arr [Integer, Integer] Boolean, ESVLbl "eq"),
```

```
        ("!=",      Arr [Integer, Integer] Boolean, ESVLbl "ne"),
        (">=",      Arr [Integer, Integer] Boolean, ESVLbl "ge"),
        (">",       Arr [Integer, Integer] Boolean, ESVLbl "gt"),
        ("&&",      Arr [Boolean, Boolean] Boolean, ESVLbl "and"),
        ("||",      Arr [Boolean, Boolean] Boolean, ESVLbl "or"),
        ("!",       Arr [Boolean] Boolean,          ESVLbl "not"),
        ("getint",  Arr [Snk Integer] Void,         ESVLbl "getint"),
        ("putint",  Arr [Integer] Void,             ESVLbl "putint"),
        ("skip",    Arr [] Void,                    ESVLbl "skip")]
```

2.3 If-else extended

2.3.1 MTIR

Firstly, we update the internal representation to allow multiple *elsif* and optional *else* branches. Do this by modifying `CmdIf` inside `Command` data type.

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

2.3.2 PPMTIR

Now we need to update the pretty print function, so the new syntax can be seen

```
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
```

2.3.3 TypeChecker

Next we have to update the command type checking. Do this by updating `chkCmd` function with:

```
-- T-IF
chkCmd env (A.CmdIf {A.ciCondThens = ifs, A.ciMbElse = mbElse,
                     A.cmdSrcPos=sp}) = do
    ifs' <- mapM chkThen ifs
    mbElse' <- case mbElse of
        Nothing -> return Nothing
        Just c -> fmap Just $ chkCmd env c
    return (CmdIf {ciCondThens = ifs', ciMbElse = mbElse', cmdSrcPos = sp})
    where
        chkThen (e, c) = do
            e' <- chkTpExp env e Boolean
            c' <- chkCmd env c
            return (e', c')
```

In the first step we have to go trough the list of our if branches checking each one. We use mapM here in order to make sure list is wrapped in a single monad, rather than having a list of monads. Next we check the optional else branch. To do this properly, we need to make sure that it always returns `D Maybe` type. If no branch, we just use return function to wrap our Maybe type. If there is a command, we run a check and then map inner contents with a `Just` type

2.4 Conditional expression

2.4.1 MTIR

Firstly add a new data type to Expresion.

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

2.4.2 PPMTIR

Next we add a way to print the expression to the screen

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

2.4.3 TypeChecker

Now we can add a typechecking pattern to `infTpExp`

```
-- T-COND
infTpExp env (A.ExpCond { A.ecCond = c, A.ecTrue = l, A.ecFalse = r, A.expSrcPos = sp }) = do
    c' <- chkTpExp env c Boolean
    (tl, l') <- infNonRefTpExp env l
    (tr, r') <- infNonRefTpExp env r
    require (tl == tr) sp $ errMsg tl tr
    -- Make sure both types are same and not reference
    return (tl, ExpCond { ecCond = c', ecTrue = l', ecFalse = r', expType = tl, expSrcPos = sp })
    where
        errMsg tl tr = "Expected: " ++ (show tl) ++ " and " ++ (show tr) ++ " types to match"
```

We make sure that both types are the same as well as both non-references. Because both are the same we just assign left expression type as the main expression type.

3 Task 3

3.1 Section a

```
        LOADLB      0 2
        LOADL       1
        STORE       [SB + 0]
        LOADA       [SB + 1]
        CALL        getint
        JUMP        #1
#0:
        LOAD        [SB + 0]
        CALL        putint
        LOAD        [SB + 0]
        LOADL       1
        ADD
        STORE       [SB + 0]
#1:
        LOAD        [SB + 0]
        LOAD        [SB + 1]
        GTR
        JUMPIFZ     #0
        POP         0 2
        HALT
```

3.2 Section b

```
        LOADL       0
        LOADA       [SB + 0]
        CALL        getint
        LOAD        [SB + 0]
        LOADL       1
        CALL        #0_f
        CALL        putint
        POP         0 1
        HALT
#0_f:
        LOAD        [LB - 2]
        LOADL       0
        GTR
        JUMPIFNZ    #2
        LOAD        [LB - 1]
        JUMP        #1
#2:
        LOAD        [LB - 2]
        LOADL       1
        SUB
        LOAD        [LB - 1]
        LOAD        [LB - 2]
        MUL
        CALL        #0_f
#1:
        RETURN      1 2
```

3.3 Section c

3.3.1 LibMt

Add new commands to `libMt`.

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

```
    RETURN 0 1,

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

3.3.2 MTStdEnv

Add two new entries to `mtStdEnv`

```
("getchar", Arr [Snk Character] Void,       ESVLbl "getchar"),
("putchar", Arr [Character] Void,           ESVLbl "putchar"),
```

4 Task 4

4.1 If-else extended

We import `import Data.Maybe (isJust)` to use for checking else branch. The main goal is to modify execute command inside `CodeGenerator.hs`. But before we do that, let's define a helper function for getting a label name of an if branch.

```
-- el - else branch label
-- Used for generating a if/elsif labels
-- if no more elsif commands left, we jump to else label
getNextLabel [] el = return el
getNextLabel _  el = newName
```

This function is passed two arguments, where first one is an arary of if branch tupples `(Expression, Commands)` and the second is the name of else branch label. If we still have any if branches left, we create a new label, otherwise it should jump to else label. Next we update `execute` command with `CmdIf` pattern.

```
execute majl env n (CmdIf {ciCondThens = ct, ciMbElse = mbE }) = do
    lblIf <- newName
    lblOver <- newName
    -- This make sures that last elsif branch jumps to over label
    -- if there is no else branch
    lblElse <- if isJust mbE then newName else return lblOver
    exec ct lblIf lblElse lblOver
    case mbE of
        Nothing -> return ()
        Just c -> do
            emit (Label lblElse)
            execute majl env n c
    emit (Label lblOver)
    where
        -- cl - Current label
        -- el - Else label
        -- ol - Over label
        -- If we have no more elsif, we jump to else label
        -- After each branch commands jump to over label
        exec [] cl el ol = return ()

        exec ((e, c):ifs) cl el ol = do
        nl <- getNextLabel ifs el
        emit (Label cl)
        evaluate majl env e
        emit (JUMPIFZ nl)
        execute majl env n c
        emit (JUMP ol)
        exec ifs nl el ol
```

First we generate all needed labels `lblElse`, **lblIf**, **lblOver**, then we use `exec` function to recursivelly go trough all of the if branches. Once that is do, we check whether else branch has any commands, if not no work is executed.

4.2 Repeat-until

We update `execute` function with the following pattern match:

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

As can be seen, it's almost identical to the `CmdWhile` pattern, except, there initial jump to loop condition check.

4.3 Conditional expression

We update `execute` function with the following pattern match:

```
evaluate majl env (ExpCond { ecCond = c, ecTrue = t, ecFalse = f }) = do
    done <- newName
    fLabel <- newName
    evaluate majl env c
    emit (JUMPIFZ fLabel)
    evaluate majl env t
    emit (JUMP done)
    emit (Label fLabel)
    evaluate majl env f
    emit (Label done)
```

Works by first evaluating the condition, then if it returns 0 (false) we jump to execute the false variant, if not execute the true variant and jump to done label.