The G53CMP - CW2 Report

Task 2.1

Brief

The following new constructs to the MiniTriangle language was added

- repeat until loop
- conditional expressions
- character literals
- extended if command with else if and else branch

Repeat until loop

T-REPEAT was added to Commands of typing relations

Γ |- c   Γ |- e : Boolean

———————————— T-REPEAT

Γ |- repeat c until e

Conditional expressions

T-CON was added to Expressions of typing relations

Γ |- e1 : Boolean   Γ |- e2 : T   Γ |-e3 : T

————————————————————————T-CON

Γ |- e1 ? e2 : e3 : T

Character literals

Add Character to Type

Type	→		*Types:*
		`Void`	*The empty type (procedure return type)*
	|	`Boolean`	*The Boolean type*
	|	`Integer`	*The Integer type*
	|	`Src` *Type*	*Read-only variable reference (source)*
	|	`Snk` *Type*	*Write-only variable reference (sink)*
	|	`Ref` *Type*	*Variable reference*
	|	$Type^* \rightarrow Type$	*Type of procedures and functions (arrow)*
	|	Character	The Character type

T-CON was added to Expressions of typing relations

Γ |- n : Character (T-LITCHAR)

Extended if command

T-CON was added to Command of typing relations

Γ |- e : Boolean   Γ |- c1   (Γ |- es : Boolean Γ |- cs) (Γ |- c2)

-- T-IF

 Γ |- if e then c1 (elsif es then cs) (else c2)

Task 2.2

Brief

After adding the 4 commands and expressions into the initial abstract syntax AST, the type checker checks the types and translate the initial AST into a version of AST into MTIR. There are some additions done by me in these five files Type.hs, MTStdEvd.hs, MTIR.hs, PPMIT.hs and TypeChecker.hs.

Repeat until loop

In MTIR.hs, I added the repeat command .

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

In PPMTIR.hs, The pretty printing function of repeat command was added

```
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
```

In the TypeChecker.hs, added the following code to check the type of repeat command.

```
-- 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})
```

Character literals

In Type.hs, add the new type Char

```
data Type = SomeType          -- ^ Some unknown type
          | Void         -- ^ The empty type (return type of procedures)
          | Boolean      -- ^ The Boolean type
          | Integer      -- ^ The Integer type
          | Char         -- ^ The Character type
          | Src Type          -- ^ Read-only variable reference (source)
          | Snk Type          -- ^ Write-only variable reference (sink)
          | Ref Type          -- ^ Variable reference
          | Ary Type MTInt  -- ^ Array; fields repr. element type and size
          | Arr [Type] Type -- ^ Type of procedures and functions (arrow).
                -- The fields represent the argument types and
                                -- the result type. The latter is 'Void' for
```

The instance Type, add Char to it to show we have a type Char

```
instance Eq Type where
    SomeType    == _           = True
    _           == SomeType    = True
    Void        == Void        = True
    Boolean     == Boolean     = True
    Integer     == Integer     = True
    Char        == Char        = True
    Src t1      == Src t2      = t1 == t2
    Snk t1      == Snk t2      = t1 == t2
    Ref t1      == Ref t2      = t1 == t2
    Ary t1 s1   == Ary t2 s2   = t1 == t2 && s1 == s2
    Arr ts1 t1 == Arr ts2 t2 = ts1 == ts2 && t1 == t2
    _           == _           = False
```

In MTStdEnv.hs, add a new (“char” char) to add character to the Mini Triangle environment

```
mtStdEnv :: Env
mtStdEnv =
    mkTopLvlEnv
        [("Boolean", Boolean),
         ("Integer", Integer),
         ("Char", Char)]
```

In TypeChecker.hs

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

Conditional expressions

In MTIR.hs, we need to add the conditional expressions type which show the type in expression.

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

In PPMTIR.hs, added the prett printing function of conditional expression

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

In TypeChecker.hs , add the code to check the type of conditional expression. We should check the first expression with chkTpExp env ce Bollean, then check the true expression with t1, and false t2. If t1 and t2 are the same type, then we do return else output an error message.

```
-- Cond
infTpExp env (A.ExpCond {A.ecCond = ce, A.ecTrue = te, A.ecFalse = fe, A.expSrcPos = sp}) = do
    ce' <- chkTpExp env ce Boolean
    (t1, te') <- infTpExp env te
    (t2, fe') <- infTpExp env fe
    if t1 <: t2 then do                         -- s <: Source _
        return (t1, ExpCond {ecCond = ce', ecTrue = te', ecFalse = fe', expType = t1, expSrcPos = sp})
     else do                            -- s /<: Source _
        emitErrD sp ("Expected type \"" ++ show t2 ++ "\", got \"" ++ show t1 ++ "\"")
        return (t1, ExpCond {ecCond = ce', ecTrue = te', ecFalse = fe', expType = t2, expSrcPos = sp})
```

Extended If

In MTIRS.hs, add the if command in according to AST

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

In PPMTIR.hs, we extended the pretty printing function of if command in according to PPAST

```
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
```

In TypeChecker.hs, we check the type of if command. First we need to know whether it has an else branch using case. ecs is a monad type of list of (expression, command) pairs. To deal with the list of monad we use mapM to match every item a function. I defined a function called checkEach, which has a type (a -> m b) that deal with a pair (e, c) int D (e', c') .

```
-- T-IF
chkCmd env (A.CmdIf {A.ciCondThens = ecs, A.ciMbElse = mc2,
                     A.cmdSrcPos=sp}) = do
    case mc2 of
        Just c  -> do  ecs'  <- mapM chkEach ecs
                       c'    <- chkCmd env c
                       return (CmdIf {ciCondThens = ecs', ciMbElse = Just c', cmdSrcPos = sp})
        Nothing -> do  ecs'  <- mapM chkEach ecs
                       return (CmdIf {ciCondThens = ecs', ciMbElse = Nothing, cmdSrcPos = sp})
      where chkEach (e, c) = do  e' <- chkTpExp env e Boolean
                                 c' <- chkCmd env c
                                 return (e', c')
```

Task 2.3

(a) Brief: we need to use a loop and mak sure there is no number printed if n < 1. First we nned to read an integer by using GETINT and then use address [LB 0] and [LB 1] to store the input number n and the current loop number m. The m should be checked to make sure it is not less than 1 every loop before printed and the print number should be (n - (m - 1)). After that n and (m - 1) are stored to the LB 0] and [LB 1]. n never changes, but m will be subtracted 1 for every loop until the number is less than 1.

```
-- task 3
printN = [Label "printN",
          GETINT,
          LOAD (LB 0),
          Label "loop",
          LOAD (LB 0),
          LOAD (LB 1),
          LOADL 1,
          LSS,
          JUMPIFNZ "atLeastOne",
          LOAD (LB 1),
          LOADL 1,
          SUB,
          SUB,
          PUTINT,
          LOADL 1,
          SUB,
          JUMP "loop",
          Label "atLeastOne"]
```

(b) To implement a recursieve TAM function fac to comput factorial n, first we need to read an integer from user using GETINT. The base case is returning 1 for any given argument not more than 1. The inductive case: fac(n) = n*fac(n-1) if n is greater than 1. I use CALL and return to

```
-- use recursive
printFac =
        [GETINT,
         CALL "printFac",
         PUTINT,
         HALT,
         Label "printFac",
         LOAD (LB (-1)),
         LOADL 1,
         LSS,
         JUMPIFNZ "atLeastOne",
         LOAD (LB (-1)),
         LOAD (LB (-1)),
         LOADL 1,
         SUB,
         CALL "printFac",
         MUL,
         RETURN 1 1,
         Label "atLeastOne",
         LOADL 1,
         RETURN 1 1]
```

call the function and return a value.

(c) In LibMT.hs the getchr and putchr are defined followed by getint and put int

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

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

In MTStdEnv.hs, add getchr and pur to the environment

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

Task 2.4

Conditional Expression

It is very similar to the original if command.

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

Repeat until loop

Because repeat command means repeat the body at least once until the condition is true. Execute c and then evaluate e, if the condition is true then we stop and finish, then it will do the loop again.

```
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)
```

Extended if

The we need to use case to check whether mc has is void or not. The ms is monad type list so mapM should be used be used to match the monad elements. And function executeEach was defined to help to execute each of the list.

```
execute majl env n (CmdIf {ciCondThens = ecs, ciMbElse = mc}) = do
    lblOver <- newName
    case mc of
        Just c   -> do  mapM_  (executeEach lblOver) ecs
                        execute majl env n c
                        emit (Label lblOver)
        Nothing -> do   mapM_  (executeEach lblOver) ecs
                        emit (Label lblOver)
     where executeEach  lbl (e', c') = do lblElse <- newName
                                          evaluate majl env e'
                                          emit (JUMPIFZ lblElse)
                                          execute majl env n c'
                                          emit (JUMP lbl)
                                          emit (Label lblElse)
```