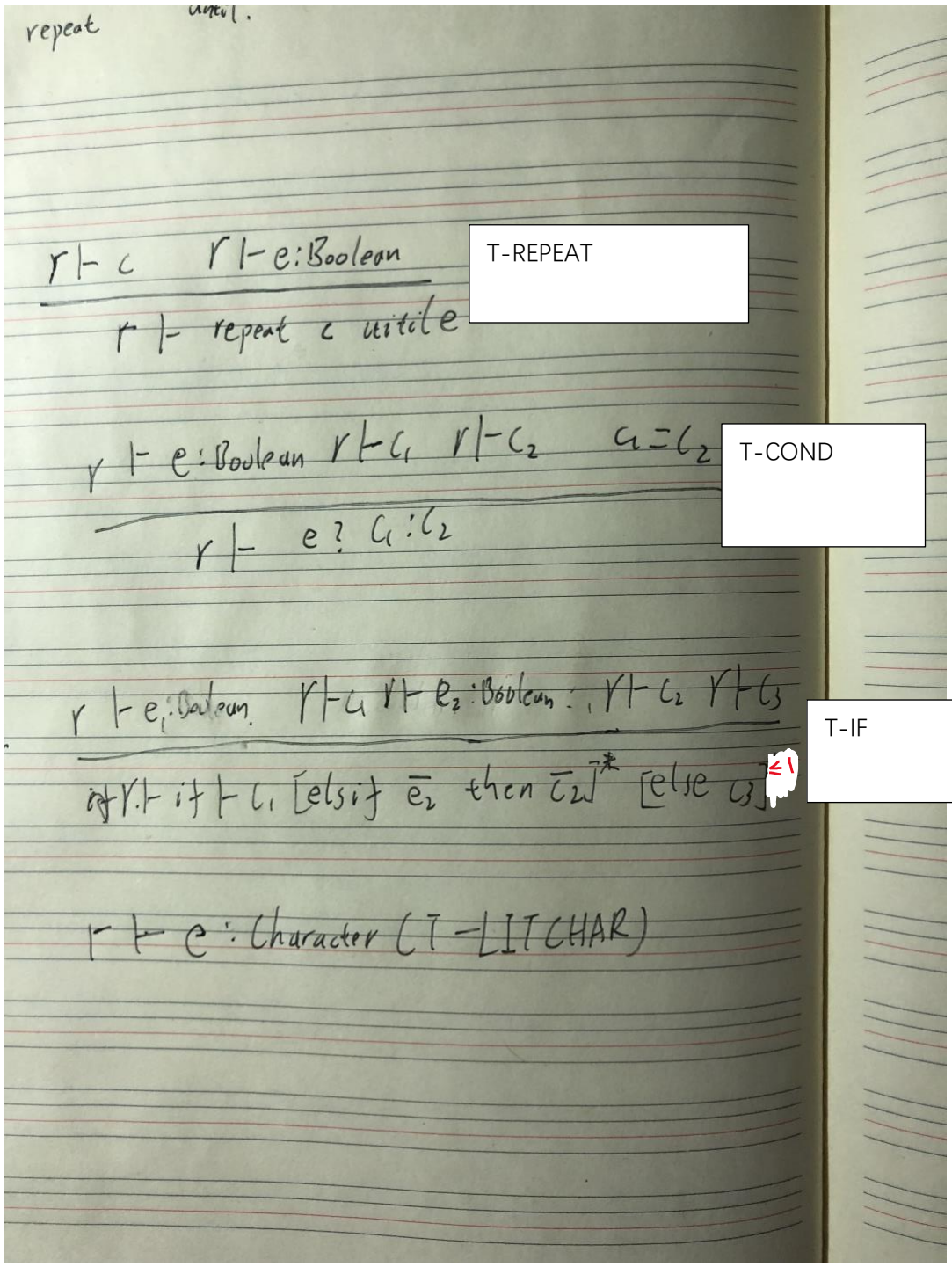
Compiler Course Work Part II Report

Task II-1:



Task II-2:
Character:
Add Character type in Type.hs

```
data Type = SomeType            -- ^ Some unknown type
          | Void                -- ^ The empty type (return type of proc
          | Boolean             -- ^ The Boolean type
          | Integer             -- ^ The Integer type
          | Character           -- ^ The Character literals type
```

Modify MTStdEnv.hs

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

Modify TypeChecker.hs

```
340   infTpExp :: Env -> A.Expression -> D (Type, Expression)
341   -- T-LITINT
342   infTpExp env e@(A.ExpLitInt {A.eliVal = n, A.expSrcPos = sp}) = do
343       n' <- toMTInt n sp
344       return (Integer,                          -- env |- n : Integer
345               ExpLitInt {eliVal = n', expType = Integer, expSrcPos = sp})
346   -- T-LITCHR
347   infTpExp env e@(A.ExpLitChr {A.elcVal = c, A.expSrcPos = sp}) = do
348       c' <- toMTChr c sp
349       return (Character,                            -- env |- n : Integer
350               ExpLitChr {elcVal = c', expType = Character, expSrcPos = sp})
```

Repeat until loop:
Modify MTIR.hs

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

Modify 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
```

These parts is pretty similar to the AST.hs and PPAST.hs
Modify TypeChecker.hs add chkCmd

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

Cond Expression:
Modify MTIR.hs:

```
168     -- | Conditional expression
169     | ExpCond {
170         ecCond    :: Expression,      -- ^ Condition
171         ecTrue    :: Expression,      -- ^ Value if condition true
172         ecFalse   :: Expression,      -- ^ Value if condition false
173         expType   :: Type,
174         expSrcPos :: SrcPos
175       }
```

Kind of different from AST.hs Expression should have a type.
Modify PPMTIR.hs:

```
120 ppExpression n (ExpCond {ecCond = e1, ecTrue = e2, ecFalse = e3, expType = t, expSrcPos = sp})=
121     indent n . showString "ExpCond" . spc . ppSrcPos sp . nl
122     . ppExpression (n+1) e1
123     . ppExpression (n+1) e2
124     . ppExpression (n+1) e3
125     . indent n . showString ": " . shows t . nl
```

Modify 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
    (t1, e2') <- infTpExp env e2
    (t2, e3') <- infTpExp env e3
    if t1 == t2 then do
        return (t1, ExpCond {ecCond = e1', ecTrue = e2', ecFalse = e3', expType = t1, expSrcPos = sp})
    else do
        emitErrD sp ("Two type should be the same")
        return (SomeType, ExpCond {ecCond = e1', ecTrue = e2', ecFalse = e3', expType = SomeType, expSrcPos = sp
```

Should check if two value has same type

Extended if-then-else-commands:
MTIR.hs

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

PPMTIR.hs:

```haskell
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
```

TypeChecker.hs:

```haskell
execute majl env n (CmdIf {ciCondThens = ecs, ciMbElse = c1}) = do
    lblOver <- newName
    executeIfSeq majl env n ecs lblOver
    case c1 of
        Just ec -> do
            execute majl env n ec
            emit (Label lblOver)
        Nothing -> emit (Label lblOver)
```

```haskell
126 chkIfCmd :: Env -> (A.Expression, A.Command) -> D (Expression, Command)
127 chkIfCmd env (e,c) = do
128     e' <- chkTpExp env e Boolean
129     c' <- chkCmd env c
130     return (e', c')
```

Create a function to check each pair of expression and command

Task II-3:

(a)MyTAMCode-3a.tam

```
    LOADL       1
    GETINT
    #loop:
    LOADL       0
    LOAD        [SB + 1]
    LSS
    JUMPIFZ     #halt
    LOAD        [SB + 0]
    PUTINT
    LOAD        [SB + 0]
    LOADL       1
    ADD
    STORE       [SB + 0]
    LOAD        [SB + 1]
    LOADL       1
    SUB
    STORE       [SB + 1]
    JUMP        #loop
    #halt:
HALT
```

Result:

(b)MyTAMCode-3b.tam

```
        GETINT
        CALL #fac
        MUL
        PUTINT
        HALT
#fac:
        LOAD [LB - 1]
        LOADL 1
        SUB
        LOAD [LB + 3]
        JUMPIFZ #basecase
        CALL #fac
        MUL
        RETURN 1 0
#basecase:
        POP 0 1
        LOADL 1
        RETURN 1 0
```

Result:

```
Likes-MacBook:HMTC-SrcPartII like$ ./hmtc --run MyTAMCode-3b.tam
Enter integer:
4
24
TAM Halted!
Likes-MacBook:HMTC-SrcPartII like$ ./hmtc --run MyTAMCode-3b.tam
Enter integer:
5
120
TAM Halted!
Likes-MacBook:HMTC-SrcPartII like$ ./hmtc --run MyTAMCode-3b.tam
Enter integer:
6
720
TAM Halted!
Likes-MacBook:HMTC-SrcPartII like$ ./hmtc --run MyTAMCode-3b.tam
Enter integer:
8
40320
TAM Halted!
```

(c) LibMT.hs:

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

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

-- skip
```

MTStdEnv.hs:

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

The parameters of RETURN m n here is keep m elements in front of stack and n elements in bottom of local stack base.
The getchr should LOAD (LB (-1)) and STOREI 0 because it load an address of a value, then this function get the address and store value to that address, then use RETURN 0 1 to remove the address been used.

Task II-4: modify codeGenerator .hs
Character type:

```
505    sizeOf :: Type -> MTInt
506    sizeOf SomeType  = cgErr "sizeOf" sizeOfErrMsgSomeType
507    sizeOf Void      = 0
508    sizeOf Boolean   = 1
509    sizeOf Integer   = 1
510    sizeOf Character = 1
```

Repeat until loop:

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

The logic of repeat until loop and while loop is oppsite, so here use JUMPIFZ but while loop should use JUMPIFNZ

Condition expression:

```
400 evaluate majl env (ExpCond {ecCond = c, ecTrue = ct, ecFalse = cf, expType = t}) = do
401     lblElse <- newName
402     lblOver <- newName
403     evaluate majl env c
404     emit (JUMPIFZ lblElse)
405     evaluate majl env ct
406     emit (JUMP lblOver)
407     emit (Label lblElse)
408     evaluate majl env cf
409     emit (Label lblOver)
```

Extended if-then-else-commands:
Two part:

```
152  execute majl env n (CmdIf {ciCondThens = ecs, ciMbElse = c1}) = do
153      lblOver <- newName
154      executeIfSeq majl env n ecs lblOver
155      case c1 of
156          Just ec -> execute majl env n ec
157      emit (Label lblOver)
```

```
executeIfSeq :: MSL -> CGEnv -> MTInt -> [(Expression, Command)] -> Name -> TAMCG ()
executeIfSeq majl env n [] over    = return ()
executeIfSeq majl env n ((e,c):cs) over = do
    lblCond <- newName
    evaluate majl env e
    emit (JUMPIFZ lblCond)
    execute majl env n c
    emit (JUMP over)
    emit (Label lblCond)
    executeIfSeq majl env n cs over
```

Should have a new function for recursively generate the if-elsif sequence