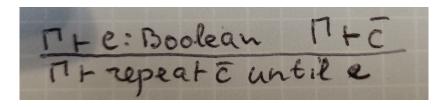
Contents

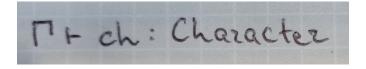
1	Tas	sk 1
	1.1	Repeat until
	1.2	Character literal
	1.2	If-else extended
	1.4	
	1.4	Conditional expression
2	Tas	sk 2
	2.1	Repeat until
		2.1.1 MTIR
		2.1.2 TypeChecker
		2.1.3 PPMTIR
	2.2	
		2.2.1 Type
		2.2.2 TypeChecker
	2.3	
	2.0	2.3.1 MTIR
		2.3.2 PPMTIR
		2.3.3 TypeChecker
	9.4	v -
	2.4	Conditional expression
		2.4.1 MTIR
		2.4.2 PPMTIR
		2.4.3 TypeChecker

1 Task 1

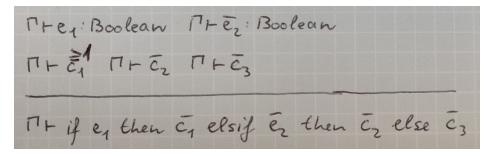
1.1 Repeat until



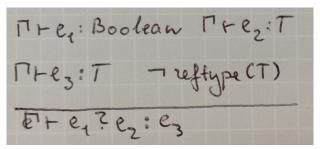
1.2 Character literal



1.3 If-else extended



1.4 Conditional expression



2 Task 2

2.1 Repeat until

2.1.1 MTIR

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

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

c' <- chkCmd 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 $AST \rightarrow MTIR$

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"),
                                              ESVLbl "neg"),
         ("neg",
                    Arr [Integer] Integer,
         ("<",
                    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"),
("!",
                                         ESVLbl "not"),
           Arr [Boolean] Boolean,
                                         ESVLbl "getint"),
("getint", Arr [Snk Integer] Void,
                                          ESVLbl "putint"),
("putint", Arr [Integer] Void,
                                           ESVLbl "skip")]
("skip",
           Arr [] Void,
```

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.

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:

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 = 1, A.ecFalse = r, A.expSrcPos = sp }) = do
    c' <- chkTpExp env c Boolean
    (t1, 1') <- infNonRefTpExp env l
    (tr, r') <- infNonRefTpExp env r
    require (t1 == tr) sp $ errMsg tl tr
    -- Make sure both types are same and not reference
    return (t1, ExpCond { ecCond = c', ecTrue = 1', ecFalse = r', expType = tl, expSrcPos = sp })
    where
        errMsg tl tr = "Expected: " ++ (show tl) ++ " and " ++ (show tr) ++ " types to match"</pre>
```

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.