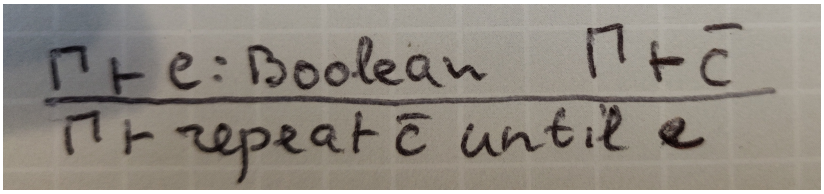
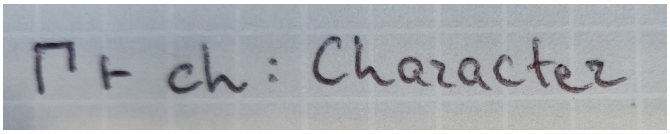
Contents

1 Task 1

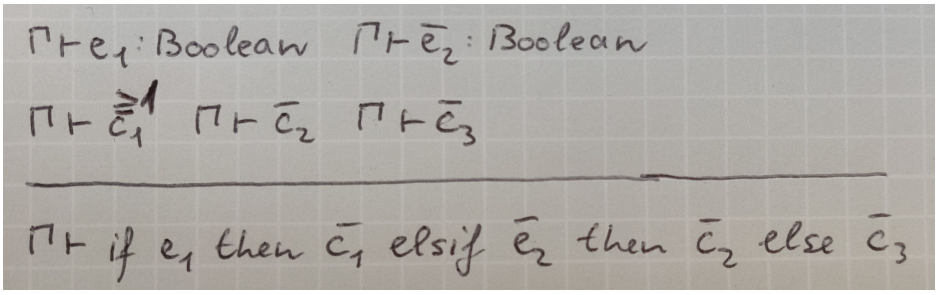
1.1 Repeat until



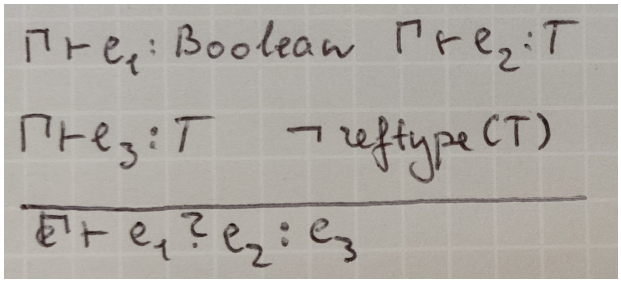
1.2 Character literal



1.3 If-else extended



1.4 Conditional



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})
```

2.3 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