UNIVERSITY OF NOTTINGHAM

G53CMP Compilers

Module Convenor: Henrik Nilsson

Coursework Part 2

Tai Nguyen Bui (txn02u)

29th November 2013

Task II.1

In order to be able to extend Minitriangle type system which covers the following language extensions: Character literals, if-command with elsif and optional else, conditional expression and repeat-until loop, the following rules were formulated.

Repeat-until loop rule

$$\frac{\Gamma \vdash c \quad \Gamma \vdash e{:}Boolean}{\Gamma \vdash repeat\ c\ until\ e} \text{ (T-REPEAT)}$$

Conditional expression rule will not accept arguments with reference type, following the strategy for implementing the type system.

$$\frac{\Gamma \vdash e{:}Boolean \quad \Gamma \vdash t : T \quad \Gamma \vdash f : T \quad \neg reftype(T)}{\Gamma \vdash e\ ?\ t :\ f} \text{ (T-COND)}$$

If-command rule was modified to extend it with elseif and optional else.

$$\frac{\Gamma \vdash e_1{:}Boolean \quad \Gamma \vdash c_1 : T \quad \Gamma \vdash \overline{e_2} : Boolean \quad \Gamma \vdash \overline{c_2} : \overline{T} \quad \Gamma \vdash c_3 : T}{\Gamma \vdash if\ \ e_1\ \ then\ \ c_1\ elsif\ \ \overline{e_2}\ \ then\ \overline{c_2}\ else\ c_3} \text{ (T-COND)}$$

$$\frac{\Gamma \vdash e_1{:}Boolean \quad \Gamma \vdash c_1 : T \quad \Gamma \vdash \overline{e_2} : Boolean \quad \Gamma \vdash \overline{c_2} : \overline{T}}{\Gamma \vdash if\ \ e_1\ \ then\ \ c_1\ elsif\ \ \overline{e_2}\ \ then\ \overline{c_2}} \text{ (T-COND)}$$

Character literals rule

$$\Gamma \vdash c :\ Character \text{ (T-LITCHAR)}$$

Task II.2

After the rules for the extensions and additions were designed, then the type checker was to be modified in order to be able to handle the language extensions.

The modifications for the new type, Character, were as follow:

I added to the Type.hs the new type. Additionally, for this type Eq and Show instances were extended.

```
Type
          | **Char**            -- ^ The Character type
Eq instance
        **Char**  ==  **Char**  =  True
Show instance
        showsPrec  _  **Char**  = showString "Character"
```

The MiniTriangle Initial environment was also extended with the new Type. ‘getchr’ has a write-only variable reference and allows the MiniTriangle to ask the user for an input.

```
    [("Boolean", Boolean),
     ("Integer", Integer),
     **("Char", Char)]**
.
.
  ("getint",  Arr [Snk Integer] Void,       ESVLbl "getint"),
     ("putint",  Arr [Integer] Void,          ESVLbl "putint"),
     **("getchr", Arr [Snk Char] Void,          ESVLbl "getchar"),**
     **("putchr", Arr [Char] Void,             ESVLbl "putchar"),**
     ("skip",    Arr [] Void,                 ESVLbl "skip")]
```

The MiniTriangle internal representation and pretty printing function were already implemented for the type Character.

The MiniTriangle Internal Representation was extended with the repeat-until loop command, conditional expression and new if-command as follow:

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

ciCondThens is a sequence of (Expression, Command), it will at least have 1 pair. Maybe Command is used to allow an optional else, either Nothing or Just command.

```
  -- | Conditional command - elsif + optional else
  | CmdIf {
      ciCondThens :: [(Expression,
                  Command)],     -- ^ Condition
      ciMbElse    :: Maybe Command, -- ^ Else-branch
      cmdSrcPos :: SrcPos
   }

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

Additionally, the pretty printer function was modified in order to be able to print the language extensions with the following code:

Repeat-until

```
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
```

If-command with optional else and elsif

```
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
```

Conditional expression

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

Finally, the MiniTriangle Type Checker was extended to check that the types of the new extensions are correct.

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

-- T-REPEAT ckecks the command and the type of the expression
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})

-- T-COND (This code checks that the expressions does not have reference type)
infTpExp env (A.ExpCond {A.ecCond = e, A.ecTrue = et, A.ecFalse = ef, A.expSrcP\
os = sp}) = do
   (t, e')   <- infNonRefTpExp env e
   (t', et') <- infNonRefTpExp env et
   (t'', ef') <- infNonRefTpExp env ef
   require (t' == t'') (srcPos e) ("Conditional expression arguments type are \
not valid, they may be references type")
   return (t', ExpCond {ecCond = e', ecTrue = et', ecFalse = ef', expType = t'\
, expSrcPos = sp})

-- T-IF (helper functions were added to process facilitates the type checking)
chkCmd env (A.CmdIf {A.ciCondThens = ecs, A.ciMbElse = mc2,
              A.cmdSrcPos=sp}) = do
   ecs' <- mapM (elseif env) ecs                -- env |- (es : Boolean, cs)
   mc2' <- optElse env mc2                      -- env |- mc2
   return (CmdIf {ciCondThens = ecs', ciMbElse = mc2', cmdSrcPos = sp})

elsif :: Env -> (A.Expression, ACommand) -> D (Expression, Command)
elsif env (e, c) = do
         e' <- chkTpExp env e Boolean
         c' <- chkCmd env c
         return (e',c')

optElse :: Env -> Maybe A.Command -> D (Maybe Command)
optElse e (Nothing) = return (Nothing)
optElse e (Just c) = do
          cs <- chkCmd e c
          return (Just cs)
```

Task II.3

In this task, some TAM programs were implemented to be able to understand the Triangle Abstract Machine, and the TAM code language.

A new module “MyTAMCode.hs” was created in the source tree. TAMCode and TAMInterpreter modules were imported.

For task a, a TAM program that read a number from the user and then print the numbers from 1 to the number given was implemented. Moreover, another program was created to calculate the factorial of a given number.

```
print1ton = [GETINT,
        LOAD (SB 0),
        LOADL 0,
        GTR,
        JUMPIFZ "negInput",
        LOADL 1,
        Label "loop",
        LOAD (ST (-1)),
        PUTINT,
        LOADL 1,
        ADD,
        LOAD (SB 0),
        LOAD (SB 1),
        EQL,
        JUMPIFZ "loop",
        JUMP "end",
        Label "negInput",
        LOADL 1,
        LSS,
        Label "end",
        PUTINT,
        HALT]
```

```
fac =   [GETINT,
      LOAD (SB 0),
      LOADL (0),
      GTR,
      JUMPIFZ "negInput",
      LOAD (SB 0),
      Label "loop",
      LOADL 1,
      SUB,
      LOAD (SB 1),
      JUMPIFZ "end",
      LOAD (SB 0),
      LOAD (SB 1),
      MUL,
      STORE (SB 0),
      LOAD (SB 1),
      JUMPIFNZ "loop",
      Label "negInput",
      LOADL 0,
      LOAD (SB 0),
      EQL,
      Label "end",
      ADD,
      PUTINT,
      HALT]
```

The MiniTriangle standard library was extended with two new procedures, getchr to be able to read a single character from terminal, and also putchr to write a single character to the terminal. Modifications in the MiniTriangle environment were not needed, as they were already done in task 1

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

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

Task II.4

The last task of the coursework was to extend the code generator in order to be able to generate the code correctly for the new language extensions. As it already was modified for the character literals, only the rest of the extensions were added.

Repeat-until command was easily extended in the code generator, as it was similar to the while-loop

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

If-command was added to the code generator, two helper functions were used to evaluate the possible elsif and elses of the command. MapM_ function was used to loop through the sequence of elseifs.

```
execute majl env n (CmdIf {ciCondThens = (e,c):ecs, ciMbElse = mc2}) = do
  lblOver <- newName
  lblElseIf <- newName
  evaluate majl env e
  emit (JUMPIFZ "lblElseIf")
  execute majl env n c
  emit (JUMP lblOver)
  emit (Label lblElseIf)
  mapM_ (elseifHelper majl env n lblOver) ecs
  elseHelper majl env n lblOver mc2
  emit (Label lblOver)
```

```
elseHelper :: Int -> CGEnv -> MTInt -> Name -> Maybe Command -> TAMCG ()
elseHelper majl env n l (Just c) = do
                    execute majl env n c
                    emit (JUMP l)
elseHelper majl env n l (Nothing) = return ()
```

elseifHelper in the next page

```
elseifHelper :: Int -> CGEnv -> MTInt -> Name -> (Expression, Command) -> TAMCG
()

elseifHelper majl env n l (e, c) = do
                    lblTempElse <- newName
                    evaluate majl env e
                    emit (JUMPIFZ lblTempElse)
                    execute majl env n c
                    emit (JUMP l)
                    emit (Label lblTempElse)
```

The first helper function executes the possible command, and then jumps to the end of the program. The elseif helper on the other hand evaluates whether the expression is true or false, if it is true, then executes the command given and jumps to the end of the whole program, otherwise it does not execute anything and continues with the next elseif or else if exists.

The code added for the Conditional expression in the code generator was:

```
evaluate majl env (ExpCond {ecCond = e, ecTrue = et, ecFalse = ef, expType = t}\
) = do
   lblTrue    <- newName
   lblFalse   <- newName
   lblOver    <- newName
   evaluate majl env e
   emit (JUMPIFNZ lblTrue)
   emit (Label lblTrue)
   evaluate majl env et
   emit (JUMP lblOver)
   emit (Label lblFalse)
   evaluate majl env ef
   emit (Label lblOver)
```

Finally, the Char type was added to the sizeOf function of the code Generator.