module IR3 where

import Data.Int     ( Int16 )
import CS350.Unique
import CS350.Renderable  ( Renderable(..) )
import HERA.Assembly     ( AssemblyProgram, AInstruction(..) )
import HERA.Base         ( Register(..), Condition(None, Nz), rt )
import Data.List    ( intercalate, nub )
import Data.Maybe   ( isNothing )
import Text.Printf  ( printf )

-- This variant of the language treats variables as mutable.

type VarName = String

-- Arithmetic expressions
data Exp
  = VarE VarName         -- string representing an object-language variable
  | ConstE Int16         -- a constant Int16 value
  | AddE Exp Exp         -- sum of two expressions
  | MulE Exp Exp         -- product of two expressions
  | NegE Exp             -- negation of an expression
  deriving (Show, Eq)

-- Abstract syntax of commands
data Command
  = Skip                       -- skip
  | Assign VarName Exp         -- X := e
  | Seq Command Command        -- c1 ; c2
  | IfNZ Exp Command Command   -- if (e /= 0) then cmd1 else cmd2
  | WhileNZ Exp Command        -- while (e /= 0) do cmd
  deriving (Show, Eq)

exampleBranch :: Command
exampleBranch =
  let x1 = "X1"
      x2 = "X2"
      vx1 = VarE x1
      vx2 = VarE x2
  in
    Seq (Assign x1 (AddE vx1 vx2))
      (Seq (IfNZ vx2
            (Assign x1 (AddE vx1 (ConstE 1)))
            (Assign x2 vx1))
            (Assign x2 (MulE vx2 vx1)))

{-
  X2 := X1 + X2;
  IFNZ X2 THEN
    X1 := X1 + 1
  ELSE
    X2 := X1
  X2 := X2 * X1
-}
```hs
IR3.hs
73: WhileNZ X1 DO
74:   X2 := X2 * X1;
75:   X1 := X1 + (-1);
76: DONE
77: -}
78: factorial :: Command
79: factorial =
80: let x = "X1"
81:   ans = "X2"
82: in
83: Seq (Assign x (ConstE 6))
84:   (Seq (Assign ans (ConstE 1))
85:     (WhileNZ (VarE x)
86:       (Seq (Assign ans (MulE (VarE ans) (VarE x)))
87:         (Assign x (AddE (VarE x) (ConstE (-1)))))))
88:
89: -------------------------------------
90: -- Intermediate representation
91:
92: type Label = Unique
93:
94: -- operands
95: data Operand
96:   = Id Unique
97:   | Const Int16
98:   deriving (Show, Eq)
99:
100: -- binary operations
101: data BinaryOp
102:   = Add
103:   | Mul
104:   deriving (Show, Eq)
105:
106: -- comparisons
107: data CmpOp
108:   = Equal
109:   | LessThan
110:   deriving (Show, Eq)
111:
112: -- instructions
113: -- note that there is no nesting of operations!
114: data Instruction
115:   = Let Unique BinaryOp Operand Operand
116:   | Load Unique VarName
117:   | Store VarName Operand
118:   | Compare Unique CmpOp Operand Operand
119:   deriving (Show, Eq)
120:
121: -- Block terminator
122: data Terminator
123:   = Return
124:   | Branch Label          -- unconditional branch
125:   | CondBr Operand Label Label -- conditional branch
126:
127: -- Basic blocks
128: data Block = Bl { instructions :: [Instruction]
129:                     , terminator   :: Terminator
130:                     }
131:
132: -- Control flow graph: a pair of an entry block and a set of labeled blocks
133: data Program = Cfg Block [(Label, Block)]
134:
135: -------------------------------
136: -- Pretty-printing
137:
138: instance Renderable Operand where
139:   render (Id u)    = render u
140:   render (Const c) = render c
141:
142: instance Renderable BinaryOp where
143:   render Add = "add"
144:   render Mul = "mul"
```
instance Renderable CmpOp where
  render Equal = "equal"
  render LessThan = "lessthan"

instance Renderable Instruction where
  render (Let u bop op1 op2) = printf "let %s = %s %s %s"
    (render u) (render bop) (render op1) (render op2)
  render (Load u x) = printf "let %s = load %s"
    (render u) ("var" ++ x)
  render (Store x op) = printf "let _ = store %s %s %s"
    (render op) ("var" ++ x)
  render (Compare u cmpop op1 op2) = printf "let %s = icmp %s %s %s"
    (render u) (render cmpop) (render op1) (render op2)

instance Renderable Terminator where
  render Return = "  ret ()"
  render (Branch lbl) = printf "  br %s" (render lbl)
  render (CondBr op lbl1 lbl2) = printf "  cbr %s %s %s"
    (render op) (render lbl1) (render lbl2)

instance Renderable Block where
  render (Bl { instructions = insns, terminator = term }) =
    intercalate " in
" (map render insns) ++
    (if length insns > 0 then " in
" else ") ++
    render term

instance Renderable Program where
  render (Cfg entry blocks) =
    printf "let program () = 
%s
in entry ()" $
    printf "let rec entry () =
%s" (render entry) ++ 

    intercalate "\n\n" (map (
        printf "and %s () = 
%s" (render lbl) (render block)) blocks) ++

-- Compilation

-- A stream is a sequence of elements *in reverse order*
data Stream = MkStr [Element]

{- During generation, we typically emit code so that it is in
  _reverse_ order when the stream is viewed as a list. That is,
  instructions closer to the head of the list are to be executed
  later in the program. That is because cons is more efficient than
  append.

  To help make code generation easier, we define snoc (reverse cons)
  and reverse append, which let us write code sequences in their
  natural order.

-}
IR3.hs

216: MkStr x <> y = MkStr (y : x)
217: -}
218:
219: -- Turn a single instruction into a stream
220: inst :: Instruction -> Stream
221: inst = MkStr . ([:]) . I
222:
223: -- Turn a terminator into a stream
224: term :: Terminator -> Stream
225: term = MkStr . ([:]) . T
226:
227: -- Turn a label into a stream
228: label :: Label -> Stream
229: label = MkStr . ([:]) . L
230:
231: -- An empty stream
232: emptyStream = MkStr []
233:
234: {- Convert an instruction stream into a control flow graph.
235: Assumes that the instructions are in ‘reverse’ order of execution.
236: -}
237: buildCfg :: Stream -> Program
238: buildCfg (MkStr code) =
239:   let (insns, m_term, blks) = go [] Nothing [] code in
240:     case m_term of
241:       Nothing -> error "buildCfg: entry block has no terminator"
242:       Just term -> Cfg (Bl { instructions = insns, terminator = term }) blks
243:     where
244:       go insns m_term blks [] = (insns, m_term, blks)
245:       go insns m_term blks (L l : rest)
246:         | Just term <- m_term
247:         = go [] Nothing ((l, Bl { instructions = insns, terminator = term }) : blks) rest
248:         | null insns -- this happens for a dummy (redundant) label
249:         = go [] Nothing blks rest
250:         | otherwise
251:         = error (printf "buildCfg: block labeled %s has no terminator" (render l))
252:       go insns m_term blks (T t : rest)
253:         | null insns && isNothing m_term
254:         = go [] (Just t) blks rest
255:         | otherwise
256:         = error (printf "buildCfg: block is missing label: %s" (unlines (map render insns)))
257:       go insns m_term blks (I i : rest)
258:         = go (i : insns) m_term blks rest
259:     compileBop :: BinaryOp -> Exp -> Exp -> UniqueM (Stream, Operand)
260:     compileBop bop e1 e2 = do
261:       (is1, ret1) <- compileExp e1
262:       (is2, ret2) <- compileExp e2
263:       ret <- newUnique "tmp"
264:       pure (ins1 ++ ins2 ++ inst (Let ret bop ret1 ret2), Id ret)
265:     compileExp :: Exp -> UniqueM (Stream, Operand)
266:     compileExp (VarE x) = do ret <- newUnique "tmp"
267:       pure (inst (Load ret x), Id ret)
268:     compileExp (ConstE c) = pure (emptyStream, Const c)
269:     compileExp (AddE e1 e2) = compileBop Add e1 e2
270:     compileExp (MulE e1 e2) = compileBop Mul e1 e2
271:     compileExp (NegE e1) = compileBop Mul e1 (-1)
272:     compileCmd :: Command -> UniqueM Stream
273:     compileCmd Skip = pure emptyStream
274:     compileCmd (Assign v e) = do
275:       (is, op) <- compileExp e
276:       pure (is ++ inst (Store v op))
277:     compileCmd (Seq c1 c2) = do
278:       str1 <- compileCmd c1
279:       str2 <- compileCmd c2
280:       pure (str1 ++ inst str2)
281:     compileCmd (IfNZ e c1 c2) = do
IR3.hs

```haskell
288:   (is, result) <- compileExp e
289:   c1_insns <- compileCmd c1
290:   c2_insns <- compileCmd c2
291:   guard <- newUnique "guard"
292:   nz_branch <- newUnique "nz"
293:   z_branch <- newUnique "z"
294:   merge <- newUnique "merge"
295:
296:   pure (is +++
297:       -- Compute the guard result
298:       inst (Compare guard Equal result (Const 0)) +++
299:       term (CondBr (Id guard) z_branch nz_branch) +++
300:
301:       -- guard is non-zero
302:       label nz_branch +++
303:       c1_insns +++
304:       term (Branch merge) +++
305:
306:       -- guard is zero
307:       label z_branch +++
308:       c2_insns +++
309:       term (Branch merge) +++
310:
311:       label merge)
312:
313: compileCmd (WhileNZ e c) = do
314:   (is, result) <- compileExp e
315:   c_insns <- compileCmd c
316:   guard <- newUnique "guard"
317:   entry <- newUnique "entry"
318:   body  <- newUnique "body"
319:   exit  <- newUnique "exit"
320:
321:   pure (term (Branch entry) +++
322:       label entry +++
323:       is +++
324:       inst (Compare guard Equal result (Const 0)) +++
325:       term (CondBr (Id guard) exit body) +++
326:       label body +++
327:       c_insns +++
328:       term (Branch entry) +++
329:       label exit)
330:
331: compile :: Command -> UniqueM Program
332: compile cmd = do
333:   str <- compileCmd cmd
334:   pure (buildCfg (str +++ term Return))
335:
336: --------------------------------------------------------
337: -- Compilation to HERA
338:  
339: {- Strategy:
340: - each named variable becomes a named data location in the data segment
341: - each temporary corresponds to a memory location, where the temporary number
342: is the address of the memory location.
343: - No function calls here, so we don't have to worry about interactions with, e.g.,
344: the function stack.
345: -}
346: 
347: compileProgram :: Program -> AssemblyProgram
348: compileProgram (Cfg entry labeled_blocks) =
349:   data_insns ++
350:   compileBlock entry ++
351:   concatMap compileLabeledBlock labeled_blocks
352: where
353: all_variables = collectVariables (entry : map snd labeled_blocks)
354: data_insns = concatMap mk_data_insns all_variables
355: mk_data_insns var_name = [ ADlabel var_name
356: , AInteger 0 ]
357:```
IR3.hs

360: compileLabeledBlock :: (Label, Block) -> AssemblyProgram
361: compileLabeledBlock (lbl, block)
362:  = [ALabel (uniqueString lbl)] ++
363:       compileBlock block
364:
365: compileBlock :: Block -> AssemblyProgram
366: compileBlock (Bl { instructions = insns, terminator = term })
367:  = concatMap compileInstruction insns ++
368:       compileTerminator term
369:
370: compileInstruction :: Instruction -> AssemblyProgram
371: compileInstruction (Let u bop op1 op2)
372:  = getOperand R1 op1 ++
373:     getOperand R2 op2 ++
374:     bopInstruction bop R1 R1 R2 ++
375:     [ ASet R2 (fromIntegral $ uniqueNumber u)
376:       , AStore R1 0 R2 ]
377: compileInstruction (Load u var_name)
378:  = [ ASetl R1 var_name
379:       , ALoad R1 0 R1
380:       , ASet R2 (fromIntegral $ uniqueNumber u)
381:       , AStore R1 0 R2 ]
382: compileInstruction (Store var_name op)
383:  = getOperand R1 op ++
384:     [ ASet1 R2 var_name
385:       , AStore R1 0 R2 ]
386: compileInstruction (Compare u cmpop op1 op2)
387:  = getOperand R1 op1 ++
388:     getOperand R2 op2 ++
389:     cmpInstruction cmpop R1 R1 R2 ++
390:     [ ASet R2 (fromIntegral $ uniqueNumber u)
391:       , AStore R1 0 R2 ]
392:
393: compileTerminator :: Terminator -> AssemblyProgram
394: compileTerminator Return = [AHalt]
395: compileTerminator (Branch lbl) = [ABr None (uniqueString lbl)]
396: compileTerminator (CondBr op lbl1 lbl2)
397:  = getOperand R1 op ++
398:     [ AFlags R1
399:     , ABr Nz (uniqueString lbl1)
400:     , ABr None (uniqueString lbl2) ]
401:
402: getOperand :: Register -> Operand -> AssemblyProgram
403: getOperand reg (Id u) = [ ASet rt (fromIntegral $ uniqueNumber u)
404:     , ALoad reg 0 rt ]
405: getOperand reg (Const i) = [ ASet reg (fromIntegral i) ]
406:
407: bopInstruction :: BinaryOp -> Register -> Register -> Register -> AssemblyProgram
408: bopInstruction Add rd ra rb = [AAdd rd ra rb]
409: bopInstruction Mul rd ra rb = [AMul rd ra rb]
410:
411: cmpInstruction :: CmpOp -> Register -> Register -> Register -> AssemblyProgram
412: cmpInstruction Equal   rd ra rb = [ ACmp ra rb
413:     , ASavef rd
414:     , ASetu rd rd
415:     , ASetu l l]
416: cmpInstruction LessThan rd ra rb = [ ACmp ra rb
417:     , ASavef rd
418:     , ASetu l l
419:     , ASetu m m]
420:
421: collectVariables :: [Block] -> [String]
422: collectVariables blocks = nub $ concatMap collect1 blocks
423:   where
424:     collect1 (Bl { instructions = insns }) = concatMap collect_insn insns
425:     collect_insn (Load _ var_name)  = [var_name]
426:     collect_insn (Store var_name _) = [var_name]
427:     collect_insn _                 = []