semantics

April 3, 2022

1 Licenciatura em Ciências da Computação

1.1 Semântica de Linguagens de Programação 2021/2022

1.1.1 A70373 Alexandre Rodrigues Baldé

1.1.2 Conteúdo:

1.2 Tipos de dados básicos, exemplos de programas While

[ ]:
```
:ext InstanceSigs
:ext BangPatterns
```

[ ]:
```
import qualified Data.Map   as M
import qualified Data.Maybe as Maybe

import           Data.List (intercalate)

-- For Natural Semantics
import qualified Control.Monad.State.Strict as St
import           Control.Monad              (when)
```

[ ]:
```
type Z = Integer

type Var = String

data Aexp
    = Num Z
    | Var Var
    | Aexp `Plus` Aexp
    | Aexp `Mul` Aexp
```

```haskell
    | Aexp `Minus` Aexp
    deriving Eq

instance Show Aexp where
    show :: Aexp -> String
    show aexp = case aexp of
        Num z -> show z
        Var v -> v
        a1 `Plus` a2 -> show a1 ++ " + " ++ show a2
        a1 `Mul` a2 -> show a1 ++ " * " ++ show a2
        a1 `Minus` a2 -> show a1 ++ " - " ++ show a2
```

[ ]:
```haskell
data Bexp
    = T
    | F
    | Aexp `Eq` Aexp
    | Aexp `Le` Aexp
    | Aexp `Lt` Aexp
    | Aexp `Ge` Aexp
    | Not Bexp
    | Bexp `And` Bexp
    | Bexp `Or` Bexp
    deriving Eq

instance Show Bexp where
    show :: Bexp -> String
    show bexp = "(" ++ helper bexp ++ ")"
        where
            helper bexp = case bexp of
                T -> "true"
                F -> "false"
                ae `Eq` ae' -> show ae ++ " == " ++ show ae'
                ae `Le` ae' -> show ae ++ " <= " ++ show ae'
                ae `Lt` ae' -> show ae ++ " < " ++ show ae'
                ae `Ge` ae' -> show ae ++ " >= " ++ show ae'
                Not be -> "not (" ++ show be ++ ")"
                be `And` be' -> show be ++ " && " ++ show be'
                be `Or` be' -> show be ++ " || " ++ show be'
```

[ ]:
```haskell
type State = M.Map Var Z

getSt :: State -> Var -> Z
getSt st var = Maybe.fromMaybe 0 (M.lookup var st)

-- Alínea b) ii)
stUpdate :: State -> Var -> Z -> State
stUpdate st var v = M.insert var v st
```

```haskell
showState :: State -> String
showState st =
    let pairs = M.toList st
        helper :: (Var, Z) -> String
        helper (var, num) = var ++ " := " ++ show num
    in intercalate "; " $ map helper pairs
```

[ ]:
```haskell
subAexp :: Var -> Aexp -> Aexp -> Aexp
subAexp y a0 a@(Num _) = a
subAexp y a0 (Var x)
   | x == y = a0
   | otherwise = Var x
subAexp y a0 (Plus a1 a2) = Plus (subAexp y a0 a1) (subAexp y a0 a2)
subAexp y a0 (Mul a1 a2) = Mul (subAexp y a0 a1) (subAexp y a0 a2)
subAexp y a0 (Minus a1 a2) = Minus (subAexp y a0 a1) (subAexp y a0 a2)

subBexp :: Var -> Aexp -> Bexp -> Bexp
subBexp y a0 a = case a of
    T -> T
    F -> F
    (Eq a1 a2) -> Eq (subAexp y a0 a1) (subAexp y a0 a2)
    (Le a1 a2) -> Le (subAexp y a0 a1) (subAexp y a0 a2)
    (Lt a1 a2) -> Lt (subAexp y a0 a1) (subAexp y a0 a2)
    (Ge a1 a2) -> Ge (subAexp y a0 a1) (subAexp y a0 a2)
    (Not b1) -> Not (subBexp y a0 b1)
    (And b1 b2) -> And (subBexp y a0 b1) (subBexp y a0 b2)
    (Or b1 b2) -> Or (subBexp y a0 b1) (subBexp y a0 b2)

arithEval :: Aexp -> (State -> Z)
arithEval a st = case a of
    Num n -> n
    Var x -> getSt st x
    Plus a1 a2 -> arithEval a1 st + arithEval a2 st
    Mul a1 a2 -> arithEval a1 st * arithEval a2 st
    Minus a1 a2 -> arithEval a1 st - arithEval a2 st

boolEval :: Bexp -> (State -> Bool)
boolEval b st = case b of
    T -> True
    F -> False
    Eq a1 a2  -> arithEval a1 st == arithEval a2 st
    Le a1 a2  -> arithEval a1 st <= arithEval a2 st
    Lt a1 a2  -> arithEval a1 st < arithEval a2 st
    Ge a1 a2  -> arithEval a1 st >= arithEval a2 st
    Not b1    -> not (boolEval b1 st)
    And b1 b2 -> boolEval b1 st && boolEval b2 st
```

```
    Or b1 b2  -> boolEval b1 st || boolEval b2 st
```

1.2.1 State Helpers

[ ]:
```
-- Igual ao tipo State, mas com instância de Show legível.
newtype State' = St {
    getState :: State
    } deriving (Eq)

instance Show State' where
    show = showState . getState
```

[ ]:
```
data Stm
    = Var `Assign` Aexp
    | Skip
    | Stm `Comp` Stm
    | IfThenElse Bexp Stm Stm
    | WhileDo Bexp Stm
    deriving (Eq)

instance Show Stm where
    show :: Stm -> String
    show stm = case stm of
        Assign v aexp          -> v ++ " := " ++ show aexp
        Skip                   -> show "skip"
        stm1 `Comp` stm2       -> show stm1 ++ "; " ++ show stm2
        IfThenElse b stm1 stm2 -> "if " ++ show b ++ " then " ++ show stm1 ++ "␣
↪else " ++ show stm2
        WhileDo b stm          -> "while " ++ show b ++ " do " ++ show stm
```

1.2.2 Example programs

Retirados da ficha 1.

[ ]:
```
ex1State :: State
ex1State = M.fromList [("n", 6), ("x", 3), ("y", 2)]

swap :: Stm
swap = Comp (Comp c1 c2) c3
    where
        c1 = Assign "n" (Var "x")
        c2 = Assign "x" (Var "y")
        c3 = Assign "y" (Var "n")

minProg :: Stm
```

```haskell
minProg = IfThenElse b1 if2 if3
    where b1 = Lt (Var "x") (Var "y")
          b2 = Lt (Var "x") (Var "z")
          b3 = Lt (Var "y") (Var "z")
          if3 = IfThenElse b3 (Assign "m" (Var "y")) (Assign "m" (Var "z"))
          if2 = IfThenElse b2 (Assign "m" (Var "x")) (Assign "m" (Var "z"))

expProg :: Stm
expProg = Comp assgn while
    where
        assgn = Assign "r" (Num 1)
        while = WhileDo bexp whileStm
        bexp = Var "y" `Ge` Num 1
        whileStm = Comp
            (Assign "r" (Mul (Var "r") (Var "x")))
            (Assign "y" (Minus (Var "y") (Num 1)))

fact :: Stm
fact = Comp assgn while
    where
        assgn = Assign "f" (Num 1)
        while = WhileDo bexp whileStm
        bexp = Var "n" `Ge` Num 1
        whileStm = Comp
            (Assign "f" (Mul (Var "f") (Var "n")))
            (Assign "n" (Minus (Var "n") (Num 1)))
```

1.3 Natural Semantics

[ ]:
```haskell
evalNS :: State -> Stm -> State
evalNS st stm = St.execState (helper stm) st
    where
        helper :: Stm -> St.State State ()
        helper (var `Assign` a) = do
            st <- St.get
            let n = arithEval a st
            St.modify (\s -> stUpdate s var n)
        helper Skip = do
            return ()
        helper (c1 `Comp` c2) = do
            s1 <- helper c1
            helper c2
        helper (IfThenElse b c1 c2) = do
            s <- St.get
            if boolEval b s
                then helper c1
                else helper c2
```

```haskell
        helper (WhileDo b c) = do
            s <- St.get
            when (boolEval b s) $ do
                    s' <- helper c
                    helper (WhileDo b c)
```

[ ]:
```haskell
evalNS ex1State fact
```

fromList [("f",720),("n",0),("x",3),("y",2)]

[ ]:
```haskell
evalNS ex1State expProg
```

fromList [("n",6),("r",9),("x",3),("y",0)]

Structural Operational Semantics

[ ]:
```haskell
-- Uma transição em semântica operacional estrutural:
-- * ou dá origem a um estado (Left)
-- * ou dá origem a um comando intermédio, juntamente com um novo estado (Right)
stepSOS :: State -> Stm -> Either State (Stm, State)
stepSOS st stm = case stm of
    x `Assign` ae ->
        let n = arithEval ae st
        in Left $ stUpdate st x n
    Skip -> Left st
    stm1 `Comp` stm2 -> case stepSOS st stm1 of
        Left st' -> Right (stm2, st')
        Right (stm1', st') -> Right (stm1' `Comp` stm2, st')
    IfThenElse be stm1 stm2 -> if boolEval be st
        then Right (stm1, st)
        else Right (stm2, st)
    WhileDo be stm' -> Right (IfThenElse be (stm' `Comp` WhileDo be stm') Skip,␣
↪st)

-- Devolve a transição após o número de passos de execução pedido, se for␣
↪possível,
-- numa lista com todas as transições que lhe precederam.
nstepsSOS :: State -> Stm -> Integer -> [Either State' (Stm, State')]
nstepsSOS st stm n
    | n <= 0 = []
    | otherwise = case stepSOS st stm of
        Left s -> [Left $ St s]
        Right (stm', st') -> Right (stm', St st') : nstepsSOS st' stm' (n - 1)

-- Imprimir uma configuração numa string legível
-- e.g. mapM_ (putStr . helperSOS) $ nstepsSOS ex1State fact 10
```

```haskell
helperSOS :: Either State' (Stm, State') -> String
helperSOS (Left st) = "Left: " ++ show st ++ ";\n"
helperSOS (Right (stm, st)) =
    "Right: " ++
    show stm ++ ";\n" ++
    show st ++ ";\n"

evalSOS :: State -> Stm -> State'
evalSOS st stm = St $ helperSOS (Right (stm, st))
    where helperSOS :: Either State (Stm, State) -> State
          helperSOS (Left st) = st
          helperSOS (Right (stm', st')) =
              case stepSOS st' stm' of
                  Left st''           -> st''
                  Right (stm'', st'') -> helperSOS (Right (stm'', st''))
```

[ ]:
```haskell
evalSOS ex1State fact
```

```
f := 720; n := 0; x := 3; y := 2
```

[ ]:
```haskell
evalSOS ex1State expProg
```

```
n := 6; r := 9; x := 3; y := 0
```

1.4 Abstract Machine 1

[ ]:
```haskell
-- Mapping from variable names to positions in memory.
-- Used during "compilation" of While code to AM1 bytecode.
type Env = M.Map Var Z

getEnv :: Env -> Var -> Z
getEnv e var = e M.! var

type NextAddr = Z

data EnvStateAM1 = EnvSt {
    getEnvSt :: !Env,
    getNxtAdr :: !NextAddr
    } deriving (Eq, Show)
```

[ ]:
```haskell
getEnv :: Env -> Var -> Z
getEnv e var = e M.! var

data AM1Instr
    = PUSH Z
    | ADD
```

```
    | MULT
    | SUB
    | TRUE
    | FALSE
    | EQUAL
    | LE
    | GE
    | LTHAN
    | AND
    | OR
    | NEG
    | PUT Z
    | GET Z
    | NOOP
    | BRANCH AM1Code AM1Code
    | LOOP AM1Code AM1Code
    deriving (Eq, Show)

type AM1Code = [AM1Instr]
```

[ ]:

```
aexpToAM1Code :: EnvStateAM1 -> Aexp -> (AM1Code, EnvStateAM1)
aexpToAM1Code m@(EnvSt e nxtAdr) a = case a of
    Num n -> ([PUSH n], m)
    Var var -> case M.lookup var e of
        Nothing  -> ([GET nxtAdr], EnvSt (M.insert var nxtAdr e) (nxtAdr + 1))
        Just adr -> ([GET adr], m)
    ae `Plus` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [ADD]], m'')
    ae `Mul` ae' ->
        let (code, m') = aexpToAM1Code m ae

            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [MULT]], m'')
    ae `Minus` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [SUB]], m'')

bexpToAM1Code :: EnvStateAM1 -> Bexp -> (AM1Code, EnvStateAM1)
bexpToAM1Code m@(EnvSt e nxtAdr) b = case b of
    T -> ([TRUE], m)
    F -> ([FALSE], m)
    ae `Eq` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
```

```
        in (concat [code', code, [EQUAL]], m'')
    ae `Le` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [LE]], m'')
    ae `Lt` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [LTHAN]], m'')
    ae `Ge` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [GE]], m'')
    Not be ->
        let (code, m') = bexpToAM1Code m be
        in ( NEG : code, m')
    be `And` be' ->
        let (code, m') = bexpToAM1Code m be
            (code', m'') = bexpToAM1Code m' be'
        in (concat [code', code, [AND]], m'')
    be `Or` be' ->
        let (code, m') = bexpToAM1Code m be
            (code', m'') = bexpToAM1Code m' be'
        in (concat [code', code, [OR]], m'')
```

[ ]:

```
whileToAM1 :: Stm -> (AM1Code, EnvStateAM1)
whileToAM1 stm = St.runState (helper stm) (EnvSt M.empty 0)
    where
        helper :: Stm -> St.State EnvStateAM1 AM1Code
        helper (var `Assign` aexp) = do
            envSt <- St.get
            let (code, EnvSt e nxtAdr) = aexpToAM1Code envSt aexp
            case M.lookup var e of
                Nothing -> do
                    let newEnv = EnvSt (M.insert var nxtAdr e) (nxtAdr + 1)
                    St.put newEnv
                    return $ code ++ [PUT nxtAdr]
                Just n  -> do
                    St.put $ EnvSt e nxtAdr
                    return $ code ++ [PUT n]
        helper Skip = return [NOOP]
        helper (c1 `Comp` c2) = do
            code1 <- helper c1
            code2 <- helper c2
            return $ code1 ++ code2
        helper (IfThenElse b c1 c2) = do
            memSt <- St.get
```

```haskell
            let (predCode, memSt') = bexpToAM1Code memSt b
            St.put memSt'
            thenCode <- helper c1
            elseCode <- helper c2
            return $ predCode ++ [BRANCH thenCode elseCode]
        helper (WhileDo b c) = do
            memSt <- St.get
            let (predCode, memSt') = bexpToAM1Code memSt b
            St.put memSt'
            loopCode <- helper c
            return [LOOP predCode loopCode]
```

[ ]:
```haskell
type Stack = [Either Z Bool]

-- Mapping from address positions to the values they contain.
-- Should look like:
-- 0 -> n_1
-- 1 -> n_2
-- 2 -> n_3,
-- ...
-- k -> n_k
-- and so on, where n_i are integers.
type Memory = M.Map Z Z

type AM1Config = (AM1Code, Stack, Memory)
```

[ ]:
```haskell
stepAM1 :: AM1Config -> AM1Config
stepAM1 conf@([], stack, mem) = conf
stepAM1 (c : cs, stack, mem) = case c of
    PUSH n -> (cs, Left n : stack, mem)
    ADD -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Left (z1 + z2) : stack, mem)
        _ -> error "ADD: invalid stack for operation!"
    MULT -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Left (z1 * z2) : stack, mem)
        _ -> error "MULT: invalid stack for operation!"
    SUB -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Left (z1 - z2) : stack, mem)
        _ -> error "SUB: invalid stack for operation!"
    TRUE -> (cs, Right True : stack, mem)
    FALSE -> (cs, Right False : stack, mem)
    EQUAL -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Right (z1 == z2) : stack, mem)
```

```
        _ -> error "EQUAL: invalid stack for operation!"
    LE -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Right (z1 <= z2) : stack, mem)
        _ -> error "LE: invalid stack for operation!"
    GE -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Right (z1 >= z2) : stack, mem)
        _ -> error "GE: invalid stack for operation!"
    LTHAN -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Right (z1 < z2) : stack, mem)
        _ -> error "LTHAN: invalid stack for operation!"
    AND -> case stack of
        Right b1 : Right b2 : stack' ->
            (cs, Right (b1 && b2) : stack, mem)
        _ -> error "AND: invalid stack for operation!"
    OR -> case stack of
        Right b1 : Right b2 : stack' ->
            (cs, Right (b1 || b2) : stack, mem)
        _ -> error "OR: invalid stack for operation!"
    NEG -> case stack of
        Right b1 : stack' ->
            (cs, Right (not b1) : stack, mem)
        _ -> error "NEG: invalid stack for operation!"
    PUT n -> case stack of
        Left z : stack' -> (cs, stack', M.insert n z mem)
        _ -> error "PUT: invalid stack for operation"
    GET n -> (cs, Left (Maybe.fromJust $ M.lookup n mem): stack, mem)
    NOOP -> (cs, stack, mem)
    BRANCH ins ins' -> case stack of
      Right b : stack' ->
          let instr = if b then ins else ins'
          in (instr, stack', mem)
      _   -> error "BRANCH: invalid stack for operator!"

    LOOP ins ins' -> (ins ++ [BRANCH (ins' ++ [LOOP ins ins']) [NOOP]] ++ cs,
↪stack, mem)
```

[ ]:
```
initConfigAM1 :: State -> Stm -> (AM1Config, Env)
initConfigAM1 initSt stm =
    let code :: AM1Code
        envSt :: EnvStateAM1
        (code, envSt) = whileToAM1 stm

        environ = getEnvSt envSt
```

```haskell
        memory :: Memory
        memory = M.fromList [(getEnv environ variable, getSt initSt variable) |␣
↪variable <- M.keys environ]
    in ((code, [], memory), environ)

-- Dado um estado inicial e um comando da linguagem while, simula a sua execução
-- na máquina abstrata AM1.
-- Devolve as variáveis usadas no programa, e os valores que estavam nas␣
↪respetivas
-- posições de memória aquando da terminação da execução.
-- Pode não terminar! (Halting problem).
runStmInAM1 :: State -> Stm -> M.Map Var Z
runStmInAM1 initSt stm =
    let (init@(initCode, initStack, initMemory), environ) = initConfigAM1␣
↪initSt stm
        run :: AM1Config -> AM1Config
        run !cfg =
            let cfg' = stepAM1 cfg
            in if cfg' == cfg then cfg else run cfg'
        (finalCode, finalStack, finalMemory) = run init

        varsToValues = M.fromList [(var, finalMemory M.! (environ M.! var)) |␣
↪var <- M.keys environ]

    in varsToValues
```

[ ]:
```haskell
runStmInAM1 ex1State minProg
runStmInAM1 ex1State swap
runStmInAM1 ex1State expProg
runStmInAM1 ex1State fact
```

```
fromList [("m",0),("x",3),("y",2),("z",0)]


fromList [("n",3),("x",2),("y",3)]


fromList [("r",9),("x",3),("y",0)]


fromList [("f",720),("n",0)]
```

1.5 Abstract Machine 2

[ ]:
```haskell
-- Mapping from variable names to positions in memory.
-- Used during "compilation" of While code to AM1 bytecode.
type Env = M.Map Var Z
```

```haskell
type NextAddr = Z

-- Program counter associated with each instruction.
-- Must be positive, starts at 1, each instruction has a unique PC value,
-- and strictly increases by 1 unit with every atomic instruction.
type ProgramCounter = Z

type Stack = [Either Z Bool]

-- Mapping from address positions to the values they contain.
-- Should look like:
-- 0 -> n_1
-- 1 -> n_2
-- 2 -> n_3,
-- ...
-- k -> n_k
-- and so on, where n_i are integers.
type Memory = M.Map Z Z
```

[ ]:
```haskell
data AM2Instr
    = PUSH Z
    | ADD
    | MULT
    | SUB
    | TRUE
    | FALSE
    | EQUAL
    | LE
    | GE
    | LTHAN
    | AND
    | OR
    | NEG
    | PUT Z
    | GET Z
    | NOOP
    | LABEL ProgramCounter
    | JUMP ProgramCounter
    | JUMPFALSE ProgramCounter
    deriving (Eq, Show)

type AM2Code = [AM2Instr]
```

[ ]:
```haskell
type AM2Config = (ProgramCounter, AM2Code, Stack, Memory)

type AM2AnnotatedProgram = M.Map ProgramCounter AM2Instr
```

[ ]:
```haskell
data EnvStateAM2 = EnvSt2 {
    getEnvSt  :: !Env,
    getNxtAdr :: !NextAddr,
    getInstrs :: AM2AnnotatedProgram,
    getNxtPC  :: ProgramCounter
    } deriving (Eq)

instance Show EnvStateAM2 where
    show (EnvSt2 env nxtAdr instrs nxtPc) =
        "env: " ++ show env ++ "\n" ++
        "next address: " ++ show nxtAdr ++ "\n" ++
        "instructions (with pc): " ++ show instrs ++ "\n" ++
        "next program counter (pc): " ++ show nxtPc ++ "\n"
```

1.5.1 Tradução de expressões aritméticas e booleanas para bytecode AM2

[ ]:
```haskell
copyPasteHelper ae ae' instr = do
    -- Careful with the order with which this is done - whichever is done first
    -- puts its code on the stack first, so the second operand has to go first.
    code' <- aexpToAM2Code ae'
    code <- aexpToAM2Code ae
    St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ nxtAdr␣
 ↪(M.insert nxtPC instr instrs) (nxtPC + 1))
    return $ concat [code', code, [instr]]

aexpToAM2Code :: Aexp -> St.State EnvStateAM2 AM2Code
aexpToAM2Code a = case a of
    Num n -> do
        let instr = PUSH n
        St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
 ↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
        return [instr]
    Var var -> do
        EnvSt2 environ nxtAdr instrs nxtPC <- St.get
        case M.lookup var environ of
            Nothing  -> do
                let instr = GET nxtAdr
                St.put $ EnvSt2 (M.insert var nxtAdr environ) (nxtAdr + 1) (M.
 ↪insert nxtPC instr instrs) (nxtPC + 1)
                return [instr]
            Just adr -> do
                let instr = GET adr
                St.put $ EnvSt2 environ nxtAdr (M.insert nxtPC instr instrs)␣
 ↪(nxtPC + 1)
                return [instr]
```

```haskell
    ae `Plus` ae' -> copyPasteHelper ae ae' ADD
    ae `Mul` ae' -> copyPasteHelper ae ae' MULT
    ae `Minus` ae' -> copyPasteHelper ae ae' SUB

bexpToAM2Code :: Bexp -> St.State EnvStateAM2 AM2Code
bexpToAM2Code b = case b of
    T -> do
        let instr = TRUE
        St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
        return [instr]
    F -> do
        let instr = FALSE
        St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
        return [instr]
    ae `Eq` ae' -> copyPasteHelper ae ae' EQUAL
    ae `Le` ae' -> copyPasteHelper ae ae' LE
    ae `Lt` ae' -> copyPasteHelper ae ae' LTHAN
    ae `Ge` ae' -> copyPasteHelper ae ae' GE
    Not be -> do
        code <- bexpToAM2Code be
        let instr = NEG
        St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
        return $ code ++ [instr]
    be `And` be' -> copyPasteHelper2 be be' AND
    be `Or` be' -> copyPasteHelper2 be be' OR
    where
        copyPasteHelper2 be be' instr = do
            -- Careful with the order with which this is done - whichever is␣
↪done first
            -- puts its code on the stack first, so the second operand has to␣
↪go first.
            code' <- bexpToAM2Code be'
            code <- bexpToAM2Code be
            St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
            return $ concat [code', code, [instr]]
```

[ ]:
```haskell
:ext FlexibleContexts
```

[ ]:
```haskell
whileToAM2 :: Stm -> (AM2Code, EnvStateAM2)
whileToAM2 stm = St.runState (helper stm) (EnvSt2 M.empty 0 M.empty 1)
    where
        incrCounter = do
```

```haskell
            EnvSt2 e nA is nxtPC <- St.get
            St.put $ EnvSt2 e nA is $ nxtPC + 1
            return nxtPC

        helper :: Stm -> St.State EnvStateAM2 AM2Code
        helper (var `Assign` aexp) = do
            code <- aexpToAM2Code aexp
            EnvSt2 environ nxtAdr instrs nxtPC <- St.get
            case M.lookup var environ of
                Nothing -> do
                    let instr = PUT nxtAdr
                    St.put $ EnvSt2 (M.insert var nxtAdr environ) (nxtAdr + 1)␣
↪(M.insert nxtPC instr instrs) (nxtPC + 1)
                    return $ code ++ [instr]
                Just n  -> do
                    let instr = PUT n
                    St.put $ EnvSt2 environ nxtAdr (M.insert nxtPC instr␣
↪instrs) (nxtPC + 1)
                    return $ code ++ [instr]
        helper Skip = do
            let instr = NOOP
            St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 environ␣
↪nxtAdr (M.insert nxtPC instr instrs) (nxtPC + 1))
            return [instr]
        helper (c1 `Comp` c2) = do
            code1 <- helper c1
            code2 <- helper c2
            return $ code1 ++ code2
        -- O código máquina gerado para o comando IfThenElse e para o comando␣
↪WhileDo
        -- é complexo porque deve primeiro gerar o código dos subcomandos e␣
↪predicados,
        -- e só depois colocar as instruções de salto e labels - cujo program␣
↪counter
        -- terá de ser guardado antes da tradução dos subcomandos.
        --
        -- Ver incrCounter.
        helper (IfThenElse b c1 c2) = do
            predCode <- bexpToAM2Code b
            jzProgCounter <- incrCounter
            thenCode <- helper c1
            elseProgCounter <- incrCounter
            elseCode <- helper c2
            afterIfProgCounter <- incrCounter
            let ifJump     = JUMPFALSE elseProgCounter
                elseLabel  = LABEL elseProgCounter
```

```haskell
                jumpToRest = JUMP afterIfProgCounter
                restLabel  = LABEL afterIfProgCounter
            EnvSt2 environ nxtAdr instrs _ <- St.get
            let jumps = M.fromList [(jzProgCounter, ifJump), (elseProgCounter,␣
↪jumpToRest), (afterIfProgCounter, restLabel)]
            -- Incrementa-se o contador de código devido ao LABEL final, que␣
↪apontará
            -- para o código depois do IfThenElse, se existir.
            St.put $ EnvSt2 environ nxtAdr (instrs `M.union` jumps)␣
↪(afterIfProgCounter + 1)
            return $ predCode ++ [ifJump] ++ thenCode ++ [jumpToRest] ++␣
↪elseCode ++ [restLabel]
        helper (WhileDo b c) = do
            boolTestCounter <- incrCounter
            predCode <- bexpToAM2Code b
            jzProgCounter <- incrCounter
            loopCode <- helper c
            jumpCounter <- incrCounter
            afterWhileCounter <- incrCounter
            let whileLabel = LABEL boolTestCounter
                whileJump  = JUMPFALSE afterWhileCounter
                loopJump   = JUMP boolTestCounter
                restLabel  = LABEL afterWhileCounter
            EnvSt2 environ nxtAdr instrs _ <- St.get
            let jumps = M.fromList [(boolTestCounter, whileLabel),␣
↪(jzProgCounter, whileJump), (jumpCounter, loopJump), (afterWhileCounter,␣
↪restLabel)]
            St.put $ EnvSt2 environ nxtAdr (instrs `M.union` jumps)␣
↪(afterWhileCounter + 1)

            return $ [whileLabel] ++ predCode ++ [whileJump] ++ loopCode ++␣
↪[loopJump] ++ [restLabel]
```

[ ]:
```haskell
-- Given an AM2 configuration, execute a single instruction
-- and transition into the next configuration.
--
-- Requires
stepAM2 :: AM2Config -> AM2AnnotatedProgram -> AM2Config
stepAM2 conf@(_, [], stack, mem) _ = conf
stepAM2 (pc, c : cs, stack, mem) ann ={-
    trace (
        "code: " ++ show (c : cs) ++ "\n" ++
        "stack: " ++ show stack
    ) $ -}case c of
    PUSH n -> (pc', cs, Left n : stack, mem)
    ADD -> case stack of
```

```haskell
        Left z1 : Left z2 : stack' ->
            (pc', cs, Left (z1 + z2) : stack, mem)
        _ -> error "ADD: invalid stack for operation!"
    MULT -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Left (z1 * z2) : stack, mem)
        _ -> error "MULT: invalid stack for operation!"
    SUB -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Left (z1 - z2) : stack, mem)
        _ -> error "SUB: invalid stack for operation!"
    TRUE -> (pc', cs, Right True : stack, mem)
    FALSE -> (pc', cs, Right False : stack, mem)
    EQUAL -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Right (z1 == z2) : stack, mem)
        _ -> error "EQUAL: invalid stack for operation!"
    LE -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Right (z1 <= z2) : stack, mem)
        _ -> error "LE: invalid stack for operation!"
    GE -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Right (z1 >= z2) : stack, mem)
        _ -> error "GE: invalid stack for operation!"
    LTHAN -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Right (z1 < z2) : stack, mem)
        _ -> error "LTHAN: invalid stack for operation!"
    AND -> case stack of
        Right b1 : Right b2 : stack' ->
            (pc', cs, Right (b1 && b2) : stack, mem)
        _ -> error "AND: invalid stack for operation!"
    OR -> case stack of
        Right b1 : Right b2 : stack' ->
            (pc', cs, Right (b1 || b2) : stack, mem)
        _ -> error "OR: invalid stack for operation!"
    NEG -> case stack of
        Right b1 : stack' ->
            (pc', cs, Right (not b1) : stack, mem)
        _ -> error "NEG: invalid stack for operation!"
    PUT n -> case stack of
        Left z : stack' -> (pc', cs, stack', M.insert n z mem)
        _ -> error "PUT: invalid stack for operation"
    GET n -> (pc', cs, Left (Maybe.fromJust $ M.lookup n mem): stack, mem)
    NOOP -> (pc', cs, stack, mem)
    LABEL lab -> (pc', cs, stack, mem)
```

```haskell
    JUMP lab -> case M.lookup lab ann of
        Nothing    -> error "JUMP: invalid label!"
        Just instr ->
            let instrs = M.elems $ M.dropWhileAntitone (<= lab) ann
            in {-trace ("instrs: " ++ show (instr : instrs) ++ "\nmem: " ++␣
↪show mem)-} (lab, instr : instrs, stack, mem)
    JUMPFALSE lab -> case stack of
        Right b : stack' -> if b
                then (pc', cs, stack', mem)
                else case M.lookup lab ann of
                    Nothing    -> error "JUMPFALSE: invalid label!"
                    Just instr ->
                        let instrs = M.elems $ M.dropWhileAntitone (<= lab) ann
                        in (lab, instr : instrs, stack', mem)
        _               -> error "JUMPFALSE: invalid stack for operation"

    where
        pc' = pc + 1
```

[ ]:
```haskell
-- A configuração inicial de um programa para AM2 precisa vir acompanhada de
-- um Map com a associação entre cada instrução e o seu program counter,
-- porque no caso das instruções de salto em que é possível "regredir" no
-- programa, usar só uma lista para instruções não o permitirá.
initConfigAM2 :: State -> Stm -> (AM2Config, Env, AM2AnnotatedProgram)
initConfigAM2 initSt stm =
    let code :: AM2Code
        envSt :: EnvStateAM2
        (code, envSt) = whileToAM2 stm

        environ = getEnvSt envSt
        annotatedByteCode = getInstrs envSt--M.fromList $ zip (M.keys .␣
↪getInstrs $ envSt) code

        memory :: Memory
        memory = M.fromList [(getEnv environ variable, getSt initSt variable) |␣
↪variable <- M.keys environ]
    in ((1, code, [], memory), environ, annotatedByteCode)

-- Dado um estado inicial e um comando da linguagem while, simula a sua execução
-- na máquina abstrata AM2.
-- Devolve as variáveis usadas no programa, e os valores que estavam nas␣
↪respetivas
-- posições de memória aquando da terminação da execução.
-- Pode não terminar! (Halting problem).
runStmInAM2 :: State -> Stm -> M.Map Var Z
runStmInAM2 initSt stm =
```

```haskell
    let (init@(initPC, initCode, initStack, initMemory), environ, annotated) =␣
↪initConfigAM2 initSt stm
        program_length = M.size annotated
        run :: AM2Config -> AM2Config
        run !cfg =
            let cfg'@(pc, code, stack, memory) = stepAM2 cfg annotated
            -- Here cfg' needs to be the final configuration, and not cfg.
            -- Causes hard-to-diagnose bugs.
            in if fromInteger pc == (program_length + 1) then cfg' else run cfg'
        (finalPC, finalCode, finalStack, finalMemory) = run init

        varsToValues = M.fromList [(var, finalMemory M.! (environ M.! var)) |␣
↪var <- M.keys environ]

    in varsToValues
```

[ ]:
```haskell
runStmInAM2 ex1State expProg
runStmInAM2 ex1State fact
```

```
fromList [("r",9),("x",3),("y",0)]
```

```
fromList [("f",720),("n",0)]
```