Interpreter and Virtual Machine of Minila

Lecture Note 4

Topics

- Quick reminder of the previous lecture
- Interpretation of expressions
- Interpretation of statements
- Interpreter
- Instructions
- Virtual machine

Quick Reminder (1)

- A typical Minila program is in the form:
  $S_1\ S_2\ \ldots\ S_n$, where each S_i is as follows:
  - `estm`
  - `X := E ;`
  - `if E then` S_a `else` S_b `fi`
  - `while E do S od`
  - `for X` E_a E_b `do S od`
- Expressions are as follows:
  - N (a natural number), $\mathrm{v}(i)$ (a variable, where $i = 1,2,\ldots$)
  - E_a op E_b, where op is `++`, `--`, `**`, `//`, `%%`, `===`, `!==`, `<<`, `>>`, `&&`, or `||`.

Quick Reminder (2)

- The interpreter interprets Minila programs using an environment.
- Returns the environment used when a program terminates.
- When the interpreter takes the program:

```
v(0) := 1 ;
v(1) := 1 ;
while v(1) << 10 || v(1) === 10 do
      v(0) := v(0) ** v(1) ;
      v(1) := v(1) ++ 1 ;
od
```

it returns the environment:

```
((< v(0) , 3628800 >) | ((< v(1) , 11 >) | empty))
```

Quick Reminder (3)

- The virtual machine executes lists of instructions using an environment and a stack.
- Stacks (which are implemented as lists) are used to evaluate expressions.
- When it takes the list of instructions:

```
push(1) | store(v(0)) | push(1) | store(v(1)) |
load(v(1)) | push(10) | lessThan | load(v(1)) | push(10) |
equal | or | jumpOnCond(2) | jump(10) |
load(v(0)) | load(v(1)) | multiply | store(v(0)) |
load(v(1)) | push(1) | add | store(v(1)) |
bjump(17) | quit | clnil
```

- it returns the environment:

```
((< v(0) , 3628800 >) | ((< v(1) , 11 >) | empty))
```


Interpretation of Expressions (1)

- Function that interprets an expression with an environment:

```
op evalExp : Exp Env&Err -> Nat&Err
```

- In case that it takes errEnv,

```
eq evalExp(E,errEnv) = errNat .
```

- A natural number `N`:

```
eq evalExp(N,EV) = N .
```

- A variable `V`:

```
eq evalExp(V,EV) = lookup(V,EV) .
```

- `E1 ++ E2`:

```
eq evalExp(E1 ++ E2,EV)
     = evalExp(E1,EV) + evalExp(E2,EV) .
```

Interpretation of Expressions (2)

- `E1 -- E2:`
```
eq evalExp(E1 -- E2,EV)
     = sd(evalExp(E1,EV),evalExp(E2,EV)) .
```
- `E1 ** E2:`
```
eq evalExp(E1 ** E2,EV)
      = evalExp(E1,EV) * evalExp(E2,EV) .
```
- `E1 // E2:`
```
eq evalExp(E1 // E2,EV)
      = evalExp(E1,EV) quo evalExp(E2,EV) .
```
- `E1 %% E2:`
```
eq evalExp(E1 %% E2,EV)
      = evalExp(E1,EV) rem evalExp(E2,EV) .
```


Interpretation of Expressions (3)

- `E1 === E2:`
```
eq evalExp(E1 === E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if evalExp(E1,EV) == evalExp(E2,EV) then 1 else 0
  fi)
        fi .
```
- `E1 !== E2:`
```
eq evalExp(E1 !== E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if evalExp(E1,EV) == evalExp(E2,EV) then 0 else 1
  fi)
        fi .
```

Interpretation of Expressions (4)

- E1 << E2:

```
eq evalExp(E1 << E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if evalExp(E1,EV) < evalExp(E2,EV)
              then 1 else 0 fi)
       fi .
```

- E1 >> E2:

```
eq evalExp(E1 >> E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if evalExp(E1,EV) > evalExp(E2,EV)
              then 1 else 0 fi)
        fi .
```


Interpretation of Expressions (5)

- E1 && E2:

```
eq evalExp(E1 && E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if (evalExp(E1,EV) == 0) or (evalExp(E2,EV) ==
   0)
                 then 0 else 1 fi ) fi .
```

- E1 || E2:

```
eq evalExp(E1 || E2,EV)
     = if evalExp(E1,EV) == errNat or evalExp(E2,EV) == errNat
        then errNat
        else (if (evalExp(E1,EV) == 0) and (evalExp(E2,EV) ==
   0)
                then 0 else 1 fi ) fi .
```

Interpretation of Statements (1)

- Function that interprets a statement with an environment:

```
op eval : Stm Env&Err -> Env&Err
```

- In case that it takes errEnv,

```
eq eval(S,errEnv) = errEnv .
```

- `estm:`

```
eq eval(estm,EV) = EV .
```


Interpretation of Statements (2)

- `V := E ;:`

```
eq eval(V := E ; S,EV)
   = eval(S,
         evalAssign(V,evalExp(E,EV),EV)) .
eq evalAssign(V,errNat,EV) = errEnv .
eq evalAssign(V,N,errEnv) = errEnv .
eq evalAssign(V,errNat,errEnv) = errEnv .
eq evalAssign(V,N,EV) = update(V,N,EV) .
```

Interpretation of Statements (3)

- `if E then S1 else S2 fi:`

```
eq eval(if E then S1 else S2 fi S,EV)
     = eval(S,evalIf(evalExp(E,EV),S1,S2,EV)) .
eq evalIf(errNat,S1,S2,EV) = errEnv .
eq evalIf(N,S1,S2,errEnv) = errEnv .
eq evalIf(errNat,S1,S2,errEnv) = errEnv .
eq evalIf(N,S1,S2,EV)
     = if N == 0 then eval(S2,EV)
                 else eval(S1,EV) fi .
```


Interpretation of Statements (4)

- `while E do S1 od:`

```
eq eval(while E do S1 od S,EV)
     = eval(S,evalWhile(E,S1,EV)) .
eq evalWhile(E,S,errEnv) = errEnv .
eq evalWhile(E,S,EV)
     = if evalExp(E,EV) == errNat
        then errEnv
        else (if evalExp(E,EV) == 0 then EV
                 else evalWhile(E,S,eval(S,EV)) fi)
        fi .
```

Interpretation of Statements (5)

- for V E1 E2 do S od:

```
eq eval(for V E1 E2 do S1 od S,EV)
     = eval(S,evalFor(V,E2,S1,evalExp(E1,EV),EV)) .
eq evalFor(V,E,S,errNat,EV) = errEnv .
eq evalFor(V,E,S,N,errEnv) = errEnv .
eq evalFor(V,E,S,errNat,errEnv) = errEnv .
eq evalFor(V,E,S,N,EV)
     = if evalExp(V,update(V,N,EV)) == errNat
            or evalExp(E,update(V,N,EV)) == errNat
        then errEnv
        else (if evalExp(V,update(V,N,EV)) > evalExp(E,update(V,N,EV))
                then update(V,N,EV)
                else (if eval(S,update(V,N,EV)) == errEnv then errEnv
                        else evalFor(V,E,S,
                                lookup(V,eval(S,update(V,N,EV))) + 1,
                                eval(S,update(V,N,EV))) fi ) fi ) fi .
```


Interpreter

- Function interpret:

```
op interpret : Stm -> Env&Err
eq interpret(S:Stm) = eval(S,empEnv).
```

Instructions (1)

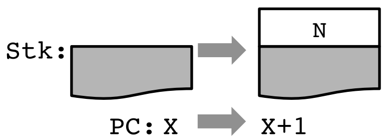
- Instructions are defined in module COMMAND:

```
mod! COMMAND principal-sort Command {
  pr(NAT) pr(VAR)
  [Command ErrCommand < Command&Err]
  op errCommand : -> ErrCommand {constr}
  op push : Nat -> Command        op load : Var -> Command
  op store : Var -> Command       op multiply : -> Command
  op divide : -> Command          op mod : -> Command
  op add : -> Command             op minus : -> Command
  op lessThan : -> Command        op greaterThan : -> Command
  op equal : -> Command           op notEqual : -> Command
  op and : -> Command             op or : -> Command
  op jump : Nat -> Command        op bjump : Nat -> Command
  op jumpOnCond : Nat -> Command
  op quit : -> Command
}
```

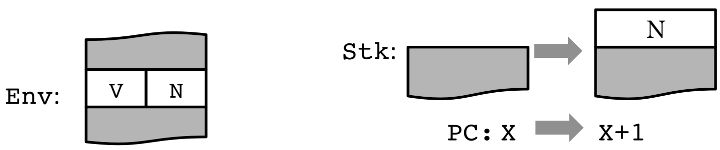

Instructions (2)

Let PC be a program counter, Stk a stack and Env an environment.

- `push(`*N*`)`:
  - Push N onto Stk and set PC to $\mathrm{PC}{+}1$.



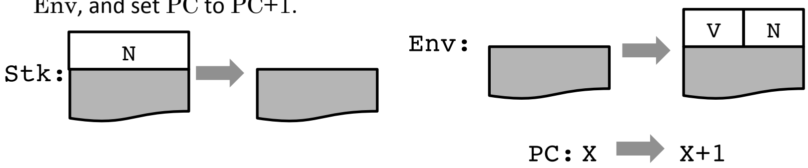
- `load(`*V*`)`:
  - Find *N* corresponding to *V* in Env, push *N* onto Stk, and set PC to PC+1.



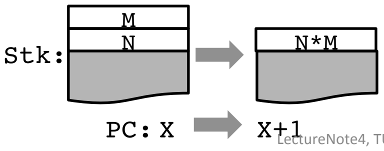
Instructions (3)

Let `PC` be a program counter, `Stk` a stack and `Env` an environment.

- `store(V):`
  - Let N be the natural number at the top of Stk. Pop Stk, register V and N into Env, and set PC to $\mathrm{PC}{+}1$.



- `multiply:`
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push *N***M* onto Stk, and set PC to PC+1.



Instructions (4)

- `divide`:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push the quotient of dividing N by M onto Stk, and set PC to $\mathrm{PC}{+}1$.
- `mod`:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push the remainder of dividing N by M onto Stk, and set PC to $\mathrm{PC}{+}1$.
- `add`:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push $N{+}M$ onto Stk, and set PC to $\mathrm{PC}{+}1$.
- `minus`:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push the absolute value of the difference between N and M onto Stk, and set PC to $\mathrm{PC}{+}1$.

Instructions (5)

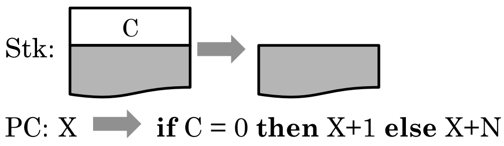
- `lessThan`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if *N* is less than *M* and push 0 onto Stk otherwise, and set PC to PC+1.
- `greaterThan`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if *N* is greater than *M* and push 0 onto Stk otherwise, and set PC to PC+1.
- `equal`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if *N* equals *M* and push 0 onto Stk otherwise, and set PC to PC+1.

Instructions (6)

- `notEqual`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 0 onto Stk if *N* equals *M* and push 1 onto Stk otherwise, and set PC to PC+1.
- `or`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 0 onto Stk if both *N* and *M* are 0 and push 1 onto Stk otherwise , and set PC to PC+1.
- `and`:
  - Let *M*,*N* be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if both *N* and *M* are not 0 and push 0 onto Stk otherwise , and set PC to PC+1.

Instructions (6)

- `jump` *N*:
  - Set PC to PC+*N*.
- `bjump` *N*:
  - Set PC to PC – *N*.
- `jumpOnCond` *N*:
  - Let *C* be the natural number at the top of Stk. Pop Stk and set PC to PC+*N* if *C* is not 0 and set PC to PC+1 otherwise.



- `quit`:
  - Terminate the execution.

Virtual Machine (1)

- Module `CLIST`:

```
mod! CLIST {
  pr(LIST(M <= TRIV-ERR2COMMAND)
      * {sort List -> CList, op nil ->
  clnil} )
}
```

```
view TRIV-ERR2COMMAND from TRIV-ERR to COMMAND {
  sort Elt -> Command,
  sort Err -> ErrCommand,
  sort Elt&Err -> Command&Err,
  op err -> errCommand
}
```

Virtual Machine (2)

- Module VM:

```
mod! VM {
  pr(CLIST)
  pr(ENV)
  pr(STACK)
  op vm : CList -> Env&Err
  op exec : CList Nat Stack&Err Env&Err -> Env&Err
  op exec2 : Command&Err CList Nat
                 Stack&Err Env&Err -> Env&Err
  … }
```

The 2nd arg of exec plays the program counter, and so does the 3rd arg of exec2.

Virtual Machine (3)

- Function vm:

```
eq vm(CL) = exec(CL,0,empstk,empEnv) .
```

- Function exec:

```
eq exec(CL,PC,errStack,Env) = errEnv .
eq exec(CL,PC,Stk,errEnv) = errEnv .
eq exec(CL,PC,errStack,errEnv) = errEnv .
eq exec(CL,PC,Stk,Env)
     = exec2(nth(CL,PC),CL,PC,Stk,Env) .
```

  - In case that it takes errStack or errEnv, it returns errEnv.
  - It takes an instruction sequence CL, a program counter PC, a stack Stk and an environment Env, and calls exec2 with the instruction pointed to by PC in CL, CL, PC, Stk and Env.

Virtual Machine (4)

- Function exec2:

```
eq exec2(errCommand,CL,PC,S&E,E&E) = errEnv .
eq exec2(C&E,CL,PC,errStack,E&E) = errEnv .
eq exec2(C&E,CL,PC,S&E,errEnv) = errEnv .
eq exec2(AnInstruction,CL,PC,Stk,Env)
      = exec(CL,PC',Stk',Env') .
```

  - In case that it takes errCommand, errStack or errEnv, it returns errEnv.
  - PC', Stk' and Env' are the program counter, the stack and the environment after the execution of *AnInstruction*.

Virtual Machine (5)

- push(N):

```
eq exec2(push(N),CL,PC,Stk,Env)
     = exec(CL,PC + 1,N | Stk,Env) .
```

- load(v):

```
eq exec2(load(V),CL,PC,Stk,Env)
     = exec(CL,PC + 1,lookup(V,Env) | Stk,Env) .
```

- store(V):

```
eq exec2(store(V),CL,PC,empstk,Env) = errEnv .
eq exec2(store(V),CL,PC,N | Stk,Env)
     = exec(CL,PC + 1,Stk,update(V,N,Env)) .
```

Virtual Machine (6)

- multiply:

```
eq exec2(multiply,CL,PC,empstk,Env) = errEnv .
eq exec2(multiply,CL,PC,N1 | empstk,Env) = errEnv .
eq exec2(multiply,CL,PC,N2 | N1 | Stk,Env)
     = exec(CL,PC + 1,N1 * N2 | Stk,Env) .
```

- divide:

```
eq exec2(divide,CL,PC,empstk,Env) = errEnv .
eq exec2(divide,CL,PC,N1 | empstk,Env) = errEnv .
eq exec2(divide,CL,PC,N2 | N1 | Stk,Env)
     = exec(CL,PC + 1,N1 quo N2 | Stk,Env) .
```

✓exec2 is defined for mod, add and minus in a similar way.

Virtual Machine (7)

- lessThan:

```
eq exec2(lessThan,CL,PC,empstk,Env) = errEnv .
eq exec2(lessThan,CL,PC,N1 | empstk,Env) = errEnv .
eq exec2(lessThan,CL,PC,N2 | N1 | Stk,Env)
     = if N1 < N2 then exec(CL,PC + 1,1 | Stk,Env)
                   else exec(CL,PC + 1,0 | Stk,Env) fi.
```

- and:

```
eq exec2(and,CL,PC,empstk,Env) = errEnv .
eq exec2(and,CL,PC,N1 | empstk,Env) = errEnv .
eq exec2(and,CL,PC,N2 | N1 | Stk,Env)
     = if N1 == 0 or N2 == 0 then exec(CL,PC + 1,0 | Stk,Env)
                             else exec(CL,PC + 1,1 | Stk,Env) fi.
```

✓exec2 is defined for greaterThan, equal, notEqual and or in a similar way.

Virtual Machine (8)

- `jump(N):`

```
eq exec2(jump(N),CL,PC,Stk,Env)
     = exec(CL,PC + N,Stk,Env) .
```

- `bjump(N):`

```
eq exec2(bjump(N),CL,PC,Stk,Env)
     = exec(CL,sd(PC,N),Stk,Env) .
```

- `jumpOnCond(N):`

```
eq exec2(jumpOnCond(N),CL,PC,empstk,Env) = errEnv .
eq exec2(jumpOnCond(N),CL,PC,N1 | Stk,Env)
     = if N1 == 0 then exec(CL,PC + 1,Stk,Env)
                  else exec(CL,PC + N,Stk,Env) fi .
```

- `quit:`

```
eq exec2(quit,CL,PC,Stk,Env) = Env .
```


Exercises

1. Implement an interpreter of the calculator in Exercise 1 of Lecture 3.
2. Implement a virtual machine for the calculator in Exercise 1 of Lecture 3.