Complier of Minila

Lecture Note 5

Topics

- Quick reminder of (part of) the previous two lectures
- Compilation of expressions
- Compilation of statements
- Compiler

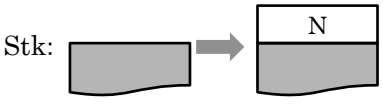
Quick Reminder (1)

- A typical Minila program is in the form:
  S_1 S_2 … 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), v(*i*) (a variable, where *i* = 1,2,…)
  - E_a op E_b, where op is ++, --, **, //, %%, ===, !==, <<, >>, &&, or ||.

Quick Reminder (2)

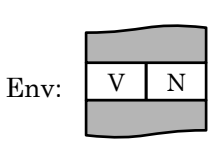
Let PC be a program counter, Stk a stack and Env an environment.

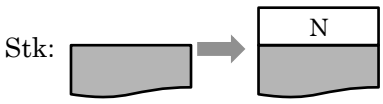
- push(*N*):
  - Push *N* onto Stk and set PC to PC+1.



PC: X ➡ X+1

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



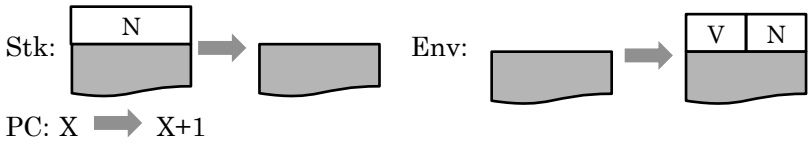


PC: X ➡ X+1

Quick Reminder (3)

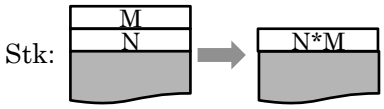
Let PC be a program counter, Stk a stack and Env an environment.

- 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 PC+1.



PC: X ➡ X+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.



PC: X ➡ X+1

Quick Reminder (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 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 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 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 PC+1.

Quick Reminder (5)

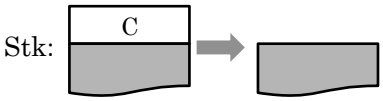
- lessThan:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if N is less than M and push 0 onto Stk otherwise, and set PC to PC+1.
- greaterThan:
  - Let M,N be the two natural numbers from the top of Stk. Pop Stk twice, push 1 onto Stk if N is greater than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

Quick Reminder (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 1 onto Stk otherwise , and set PC to PC+1.

Quick Reminder (7)

- 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.



PC: X ➡ **if** C = 0 **then** X+1 **else** X+N

- quit:
  - Terminate the execution.

Quick Reminder (8)

- The compiler translates Minila programs into lists of instructions.
- When the compiler takes the program:

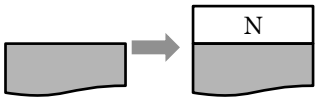
```
v(0) := 1 ;
v(1) := 1 ;
while v(1) << 10 || v(1) === 10 do
      v(0) := v(0) ** v(1) ;
      v(1) := v(1) ++ 1 ;
od
```

- it generates the list of instructions:

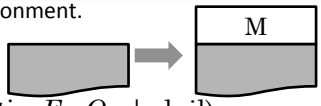
```
push(1) | store(v(0)) | push(1) | store(v(1)) |
load(v(1)) | push(10) | lessThan | load(v(1)) | push(10) |
equal | or | jumpOnCond(2) | jump(10) |
load(v(0)) | load(v(1)) | multiply | store(v(0)) |
load(v(1)) | push(1) | add | store(v(1)) |
bjump(17) | quit | clnil
```

Compilation of Expressions (1)

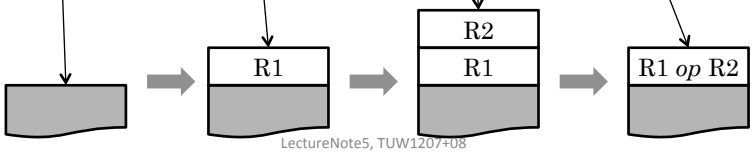
- A natural number N:
  eq genForExp(N) = push(N) | clnil .
- A variable V:
  eq genForExp(V) = load(V) | clnil .
- E1 *op* E2:
  eq genForExp(E1 *op* E2)
  = genForExp(E1) @ genForExp(E2) @
  (*instructionForOp* | clnil) .



Let M be the natural number corresponding to V in the environment.



Let R1 and R2 be the results of evaluating E1 and E2.



Compilation of Expressions (2)

- E1 ** E2:
  eq genForExp(E1 ** E2)
  = genForExp(E1) @ genForExp(E2) @ (multiply | clnil) .
- E1 ++ E2:
  eq genForExp(E1 ++ E2)
  = genForExp(E1) @ genForExp(E2) @ (add | clnil) .
- E1 -- E2:
  eq genForExp(E1 -- E2)
  = genForExp(E1) @ genForExp(E2) @ (minus | clnil) .
- E1 << E2:
  eq genForExp(E1 << E2)
  = genForExp(E1) @ genForExp(E2) @ (lessThan | clnil) .
- E1 >> E2:
  eq genForExp(E1 >> E2)
  = genForExp(E1) @ genForExp(E2) @ (greaterThan | clnil) .

Compilation of Expressions (2)

- E1 === E2:
  eq genForExp(E1 === E2)
  = genForExp(E1) @ genForExp(E2) @ (equal | clnil) .
- E1 !== E2:
  eq genForExp(E1 !== E2)
  = genForExp(E1) @ genForExp(E2) @ (notEqual | clnil) .
- E1 && E2:
  eq genForExp(E1 && E2)
  = genForExp(E1) @ genForExp(E2) @ (and | clnil) .
- E1 || E2:
  eq genForExp(E1 || E2)
  = genForExp(E1) @ genForExp(E2) @ (or | clnil) .

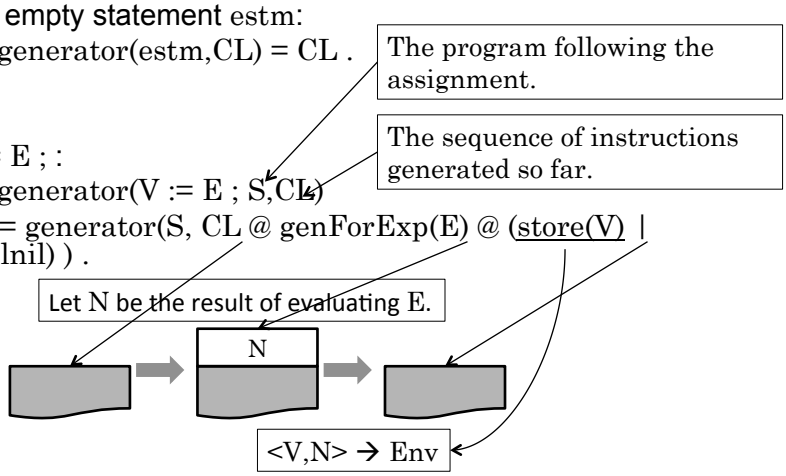
Compilation of Expressions (3)

- 3 ++ 4 :
  push(3) | push(4) | add | clnil
- v(0) ++ 4 :
  load(v(0)) | push(4) | add | clnil
- 3 ++ 4 ** 5 :
  push(3) | push(4) | push(5) | multiply | add | clnil
- v(0) ++ v(1) << 10 || v(0) ++ v(1) === 10 :
  load(v(0)) | load(v(1)) | add | push(10) | lessThan |
  load(v(0)) | load(v(1)) | add | push(10) | equal |
  or | clnil

Compilation of Statements (1)

- The empty statement estm:
  eq generator(estm,CL) = CL .

- V := E ; :
  eq generator(V := E ; S,CL)
  = generator(S, CL @ genForExp(E) @ (store(V) | clnil)) .



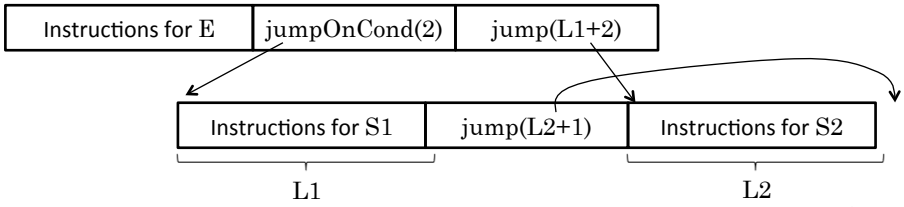
Compilation of Statements (2)

- Examples
  - v(0) := 1 ; :
    push(1) | store(v(0)) | clnil
  - v(0) := v(0) ++ 4 ; :
    load(v(0)) | push(4) | add | store(v(0)) | clnil
  - v(0) := 3 ++ 4 ** 5 ; :
    push(3) | push(4) | push(5) | multiply | add | store(v(0)) | clnil
  - v(0) := v(0) ++ v(1) << 10 | | v(0) ++ v(1) === 10 ;:
    load(v(0)) | load(v(1)) | add |(push(10) | lessThan |
    load(v(0)) | load(v(1)) | add | push(10) | equal |
    or | store(v(0)) | clnil

Compilation of Statements (3)

- if E then S1 else S2 fi :

```
eq generator(if E then S1 else S2 fi S,CL)
    = generator(S, CL @ genForExp(E)
                     @ (jumpOnCond(2) |
                        jump(len(generator(S1,clnil)) + 2) | clnil)
                     @ generator(S1,clnil)
                     @ (jump(len(generator(S2,clnil)) + 1) | clnil)
                     @ generator(S2,clnil) ) .
```



Compilation of Statements (4)

- if 1 then estm else estm fi :
  push(1) | jumpOnCond(2) | jump(2) | jump(1) | clnil

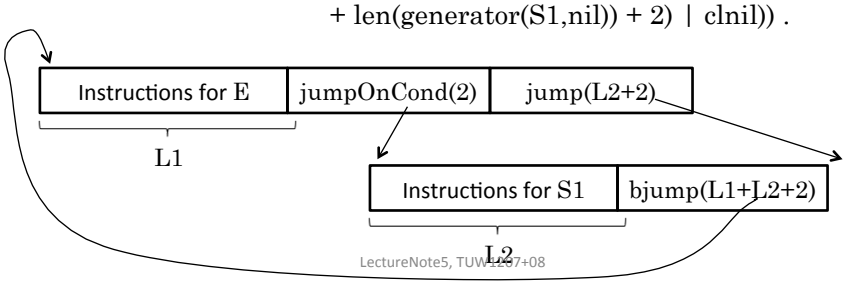
- if v(0) then v(0) := 0 ; else v(0) := 1 ; fi :
  load(v(0)) | jumpOnCond(2) | jump(4) |
  push(0) | store(v(0)) | jump(3) |
  push(1) | store(v(0)) | clnil

- if v(0) << v(1) ++ 1 then v(0) := v(0) ++ 1 ; else estm fi :
  load(v(0)) | load(v(1)) | push(1) | add | lessThan |
  jumpOnCond(2) | jump(6) |
  load(v(0)) | push(1) | add | store(v(0)) |
  jump(1) | clnil

Compilation of Statements (5)

- while E do S1 od :

```
eq generator(while E do S1 od S,CL)
    = generator(S,
        CL @ genForExp(E)
            @ (jumpOnCond(2) |
                jump(len(generator(S1,nil)) + 2) | clnil)
            @ generator(S1,clnil)
            @ (bjump(len(genForExp(E))
                        + len(generator(S1,nil)) + 2) | clnil)) .
```



Compilation of Statements (6)

- while 0 do estm od :
  push(0) | jumpOnCond(2) | jump(2) | bjump(3) | clnil

- while v(0) << 10 do v(1) := v(1) ++ v(0) ; od :
  load(v(0)) | push(10) | lessThan |
  jumpOnCond(2) | jump(6) |
  load(v(1)) | load(v(0)) | add | store(v(1)) | bjump(9) | clnil

- while v(0) << 9 | | v(0) === 10 do v(1) := v(1) ++ v(0) ; od :
  load(v(0)) | push(9) | lessThan |
  load(v(0)) | push(10) | equal | or |
  jumpOnCond(2) | jump(6) |
  load(v(1)) | load(v(0)) | add | store(v(1)) | bjump(13) | clnil

Compilation of Statements (7)

- When the compiler 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
```

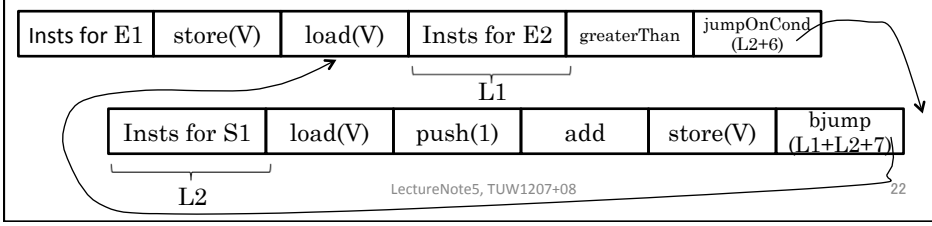
- it generates the list of instructions:

```
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
```


Compilation of Statements (8)

- for V E1 E2 do S1 od :

```
eq generator(for V E1 E2 do S1 od S,CL)
    = generator(S, CL @ genForExp(E1)
                   @ (store(V) | load(V) | clnil)
                   @ genForExp(E2)
                   @ (greaterThan |
                     jumpOnCond(len(generator(S1,clnil)) + 6) | clnil)
                   @ generator(S1,clnil)
                   @ (load(V) | push(1) | add | store(V) |
                     bjump(len(genForExp(E2)) +
                         len(generator(S1,clnil)) + 7) | clnil) ) .
```



Compilation of Statements (9)

- for v(0) 0 1 do estm od :
  push(0) | store(v(0)) |
  load(v(0)) | push(1) | greaterThan | jumpOnCond(6) |
  load(v(0)) | push(1) | add | store(v(0)) | bjump(8) | clnil

- for v(0) 0 10 do v(1) := v(1) ++ v(0) ; od :
  push(0) | store(v(0)) |
  load(v(0)) | push(10) | greaterThan | jumpOnCond(10) |
  load(v(1)) | load(v(0)) | add | store(v(1)) |
  load(v(0)) | push(1) | add | store(v(0)) | bjump(12) | clnil

- for v(0) v(1) (v(2) ++ v(1)) do v(3) := v(3) ** v(0) ; od :
  load(v(1)) | store(v(0)) |
  load(v(0)) | load(v(2)) | load(v(1)) | add | greaterThan |
  jumpOnCond(10) |
  load(v(3)) | load(v(0)) | multiply | store(v(3)) |
  load(v(0)) | push(1) | add | store(v(0)) | bjump(14) | clnil

Compilation of Statements (10)

- When the compiler takes the program:

```
v(0) := 1 ;
for v(1) 1 10 do
   v(0) := v(1) ** v(0) ;
od
```

- it generates the list of instructions:

```
push(1) | store(v(0)) |
push(1) | store(v(1)) | load(v(1)) | push(10) |
greaterThan | jumpOnCond(10) |
load(v(1)) | load(v(0)) | multiply | store(v(0)) |
load(v(1)) | push(1) | add | store(v(1)) |
bjump(12) | quit | clnil
```

Compiler

Function compile:

```
op compile : Stm -> Clist
eq compile(S:Stm)
    = generator(S,cnil) @ (quit | cnil) .
```


Exercises

1. Define a compiler of the calculator in Exercise 1 of Lecture 3 for the virtual machine in Exercise 2 in Lecture 4.