

COMP 321 Lecture 04: Operational Semantics Lab

Dan Licata

Fall 2026

1 Language Quick Reference

For this lab, we will start from the operational semantics we did on Monday. The lecture notes from Monday's class linked from the Lectures page explain these concepts.

For the concrete syntax, we have

```
e ::= k | (e1 + e2) | get(e)
s ::= e1 := e2; | noop
```

In abstract syntax, we have

```
e ::= Num(k) | Plus(e1, e2) | Get(e)
s ::= Set(e1, e2) | Noop
```

Which expressions/statements are done:

$\overline{\text{Num}(k) \text{ done}}$ $\overline{\text{Noop done}}$

Operations on memory:

$lookup([(a_1, v_1), \dots, (a_n, v_n)], a) = v_i \text{ if } a = a_i$

$insert([(a_1, v_1), \dots, (a_i, v_i), \dots, (a_n, v_n)], a_i, v) = [(a_1, v_1), \dots, (a_i, v), \dots, (a_n, v_n)]$

$insert([(a_1, v_1), \dots, (a_n, v_n)], a, v) = [(a_1, v_1), \dots, (a_n, v_n), (a, v)] \quad \text{if } \forall i. a \neq a_i$

Stepping expressions:

$$\frac{}{\mu \mid \text{Plus}(\text{Num}(k_1), \text{Num}(k_2)) \mapsto \text{Num}(k_1 + k_2)} \text{Plus-instr}$$

$$\frac{\mu \mid e_1 \mapsto e'_1}{\mu \mid \text{Plus}(e_1, e_2) \mapsto \text{Plus}(e'_1, e_2)} \text{Plus-search-1}$$

$$\frac{\mu \mid e_2 \mapsto e'_2 \quad e_1 \text{ done}}{\mu \mid \text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)} \text{Plus-search-2}$$

$$\frac{\mu \mid e \mapsto e'}{\mu \mid \text{Get}(e) \mapsto \text{Get}(e')} \text{Get-search}$$

$$\frac{}{\mu \mid \text{Get}(\text{Num}(k)) \mapsto \text{Num}(\text{lookup}(\mu, k))} \text{Get-instr}$$

Stepping statements:

$$\frac{\mu \mid e_1 \mapsto e'_1}{\mu \mid \text{Set}(e_1, e_2) \mapsto \mu \mid \text{Set}(e'_1, e_2)} \text{Set-search-1}$$

$$\frac{\mu \mid e_2 \mapsto e'_2 \quad e_1 \text{ done}}{\mu \mid \text{Set}(e_1, e_2) \mapsto \mu \mid \text{Set}(e_1, e'_2)} \text{Set-search-2}$$

$$\frac{}{(\mu \mid \text{Set}(\text{Num}(a), \text{Num}(v))) \mapsto (\text{insert}(\mu, a, v) \mid \text{Noop})} \text{Set-instr}$$

We say that an expression e_1 steps in many steps to another expression e_n if there is a sequence of expressions

$$e_1 \mapsto e_2 \mapsto e_3 \mapsto \dots \mapsto e_n$$

where for each i , $\mu \mid e_i \mapsto e_{i+1}$. We write $\mu \text{mid } e_1 \mapsto^* e_n$ for this.

Similarly, we say that a machine state $\mu_1 \mid s_1$ steps in many steps to another machine state $\mu_n \mid s_n$ if there is a sequence of machine states

$$(\mu_1 \mid s_1) \mapsto (\mu_2 \mid s_2) \mapsto \dots \mapsto (\mu_n \mid s_n)$$

where for each i , $\mu_i \mid s_i \mapsto \mu_{i+1} \mid s_{i+1}$. We write $(\mu_1 \mid s_1) \mapsto^* (\mu_n \mid s_n)$ for this.

2 Examples of Stepping

Let μ_0 be the memory

17	18	3	21	12
0	1	2	3	4

i.e. the list [(0,17),(1,18),(2,3),(3,21),(4,12)]

TASK. For each of the following expressions/statements, parse the concrete syntax to an abstract syntax tree, and then give derivations stepping that tree in many steps until you reach an expression that is done. For each expression, start from the memory μ_0 .

- `get (0)`

$$\frac{}{\mu_0 \mid \text{Get}(\text{Num}(0)) \mapsto 17} \text{Get-instr}$$

Note that we don't explicitly show a step for $\text{lookup}(\mu_0, 0) = 17$ because *lookup* is a *meta-level function*, i.e. a function defined in the metalanguage (math/SML) that we are using to define the imperative programming language that we are studying in this problem.

- `get (1+1)`

There are two steps. The first step is

$$\frac{\frac{}{\mu_0 \mid \text{Plus}(\text{Num}(1), \text{Num}(1)) \mapsto \text{Num}(2)} \text{Plus-instr}}{\mu_0 \mid \text{Get}(\text{Plus}(\text{Num}(1), \text{Num}(1))) \mapsto \text{Get}(\text{Num}(2))} \text{Get-search}$$

Note that this derivation has height 2 because we need to search inside the outer `Get` to find an instruction to perform. The second step is

$$\frac{}{\mu_0 \mid \text{Get}(\text{Num}(2)) \mapsto \text{Num}(3)} \text{Get-instr}$$

Together, these two steps show that

$$\mu_0 \mid \text{Get}(\text{Plus}(\text{Num}(1), \text{Num}(1))) \mapsto^* (\text{Num}(3))$$

- `get (get (2) )`

There are two steps. The first step is

$$\frac{\frac{}{\mu_0 \mid \text{Get}(\text{Num}(2)) \mapsto \text{Num}(3)} \text{Get-instr}}{\mu_0 \mid \text{Get}(\text{Get}(\text{Num}(2))) \mapsto \text{Get}(\text{Num}(3))} \text{Get-search}$$

The second step is

$$\frac{}{\mu_0 \mid \text{Get}(\text{Num}(3)) \mapsto \text{Num}(21)} \text{Get-instr}$$

Together, these two steps show that

$$\mu_0 \mid \text{Get}(\text{Get}(\text{Num}(2))) \mapsto^* (\text{Num}(21))$$

TASK. Give a derivation stepping the statement

$$(\mu_0 \mid \text{Set}(\text{Num}(4), \text{Num}(19)))$$

to its next step.

$$\frac{}{\mu_0 \mid \text{Set}(\text{Num}(4), \text{Num}(19)) \mapsto [(0, 17), (1, 18), (2, 3), (3, 21), (4, 19)] \mid \text{Noop}} \text{Set-instr}$$

3 Conditionals

Next, you will add conditionals to the language. In concrete syntax, we will write

```
s ::= ... | if e {s1} else {s2}
```

This means that the statements include everything they included before, plus an `if` statement. Since we don't have booleans, treat 0 as false and any non-zero integer as true.

TASK. What subtrees should the abstract syntax tree $\text{If}(\dots)$ have?

The if-then-else has three subpieces, the conditional test e , the then branch statement s_1 and the else branch statement s_2 . So we have three subtrees $\text{If}(e, s_1, s_2)$.

TASK. Give operational semantics rules for stepping If .

$$\frac{\mu \mid e \mapsto e'}{\mu \mid \text{If}(e, s_1, s_2) \mapsto \mu \mid \text{If}(e', s_1, s_2)} \text{ If-search}$$

$$\frac{k \neq 0}{\mu \mid \text{If}(\text{Num}(k), s_1, s_2) \mapsto \mu \mid s_1} \text{ If-true}$$

$$\frac{}{\mu \mid \text{If}(\text{Num}(0), s_1, s_2) \mapsto \mu \mid s_2} \text{ If-false}$$

Several students wanted to use $e \mapsto^* 0$ as a premise of If-false and similarly for If-true . While this has the right behavior, it smuggles many steps of work for e into a single step of work for If . The version with a search rule is preferable because one step of work for e becomes one step of work for the If .

TASK. Consider the following program:

```
if get(4) {
  5 := 12;
}
else {
  5 := 13;
}
```

Give an initial memory μ_1 and a derivation stepping (in many steps) this program to a memory containing (5, 12). Next, give an initial memory μ_2 and a derivation stepping (in many steps) this program to a memory containing (5, 13).

Leaving off the Num 's to make it fit on the page, the first step is

$$\frac{[(4, 1)] \mid \text{Get}((4)) \mapsto (1)}{[(4, 1)] \mid \text{If}(\text{Get}((4)), \text{Set}((5), (12)), \text{Set}((5), (13))) \mapsto [(4, 1)] \mid \text{If}(\text{Num}(1), \text{Set}((5), (12)), \text{Set}((5), (13)))} \text{ If-search}$$

The next step is

$$\frac{}{[(4, 1)] \mid \text{If}(\text{Num}(1), \text{Set}((5), (12)), \text{Set}((5), (13))) \mapsto [(4, 1)] \mid \text{Set}((5), (12))} \text{If-true}$$

The final step is

$$\frac{}{[(4, 1)] \mid \text{Set}((5), (12)) \mapsto [(4, 1), (5, 12)] \mid \text{Noop}} \text{Set-instr}$$

For the next part, use the memory $[(4, 0)]$ and the derivations are analogous but with `If-false`.

4 Sequencing

Next, you will add the sequencing of two statements to the language. In concrete syntax we might write

`s ::= ... | s1 s2`

to mean “run `s1` and then run `s2`”. For example,

`4 := 19;`
`5 := get(4) + 1;`

means “set location 4 to 19, then set location 5 to the value in location 4 plus 1”.

In abstract syntax, we will make a tree node

$$s ::= \dots \mid \text{Sequence}(s_1, s_2)$$

TASK. Give operational semantics rules for stepping the statement $\text{Sequence}(s_1, s_2)$.

$$\frac{\mu \mid s_1 \mapsto \mu' \mid s'_1}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu' \mid \text{Sequence}(s'_1, s_2)} \text{Sequence-1}$$

$$\frac{s_1 \text{ done}}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu \mid s_2} \text{Sequence-2}$$

TASK. Parse the program `4 := 19; 5 := get(4) + 1;` and give a derivation of it stepping in many steps until it is done.

First

$$\frac{\frac{}{[] \mid \text{Set}(4, 19) \mapsto [(4, 19)] \mid \text{Noop}} \text{Set-instr}}{[] \mid \text{Sequence}(\text{Set}(4, 19), \text{Set}(5, \text{Plus}(\text{Get}(4), 1))) \mapsto [(4, 19)] \mid \text{Sequence}(\text{Noop}, \text{Set}(5, \text{Plus}(\text{Get}(4), 1)))} \text{Sequenc}$$

Then

$$\frac{[(4, 19)] \mid \text{Sequence}(\text{Noop}, \text{Set}(5, \text{Plus}(\text{Get}(4), 1))) \mapsto [(4, 19)] \mid \text{Set}(5, \text{Plus}(\text{Get}(4), 1))}{\text{Sequence-2}}$$

Then $[(4, 19)] \mid \text{Set}(5, \text{Plus}(\text{Get}(4), 1))$ steps in many steps to $[(4, 19), (5, 20)] \mid \text{Noop}$ by `Set-search-2` and `Plus-search-1` and `Get-instr`, then `Set-search-2` and `Plus-instr`, and then `Set-instr`.

5 Loops

Next, you will add a while loop `while e {s}`, which means “run `s` repeatedly as long as `e`” is true (nonzero).

TASK. Give operational semantics for `While( $e, s$ )`.

One easy way to do this is to use `Sequence` and `If`:

$$\frac{}{\mu \mid \text{While}(e, s) \mapsto \mu \mid \text{If}(e, \text{Sequence}(s, \text{While}(e, s)), \text{Noop})} \text{While-expand}$$

The following almost works

$$\frac{\mu \mid e \mapsto e'}{\mu \mid \text{While}(e, s) \mapsto \mu \mid \text{While}(e', s)} \text{While-search}$$

$$\frac{}{\mu \mid \text{While}(\text{Num}(0), s) \mapsto \mu \mid \text{Noop}} \text{While-false}$$

$$\frac{k \neq 0}{\mu \mid \text{While}(\text{Num}(k), s) \mapsto \mu \mid \text{Sequence}(s, \text{While}(\text{Num}(k), s))} \text{While-true}$$

The issue is that in `While-true`, by the time we have stepped `e` to `Num( $k$ )`, we’ve forgotten the original test expression `e`, and only have its current value `Num( $k$ )`. Another solution is to modify the syntax of `While` to remember this information.

TASK. Give a derivation stepping the program

```
4 := 2; while get(4) {4 := get(4) - 1;}
```

until it is done (you can assume rules for `-` that are analogous to `+`).

This one is too long to fit horizontally, so here are instructions for making derivations of each step:

```
[ ] | 4 := 2; while get(4) {4 := get(4) - 1;}
|-> [(4,2)] | Noop; while get(4) {4 := get(4) - 1;} by Sequence-1
because [ ] | 4:=2 |-> [(4,2)] | Noop by Set-instr
```

```
[(4,2)] | Noop; while get(4) {4 := get(4) - 1;}
|-> [(4,2)] | while get(4) {4 := get(4) - 1;}
by Sequence-2
```

```
[(4,2)] | while get(4) {4 := get(4) - 1;}
|-> [(4,2)] | if get(4) {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
by While-expand
```

```
[(4,2)] | if get(4) {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
|-> [(4,2)] | if 2 {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
```

```

by If-search because get-Instr [(4,2)] get(4) |-> 2 by get-Instr

  [(4,2)] | if 2 {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
|-> [(4,2)] | {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}}
by If-true

  [(4,2)] | {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}}
|-> [(4,2)] | {4 := 2 - 1; while get(4) {4 := get(4) - 1;}}
by Sequence-1 and Set-2 and Minus-search-1 and Get-instr

  [(4,2)] | {4 := 2 - 1; while get(4) {4 := get(4) - 1;}}
|-> [(4,2)] | {4 := 1 ; while get(4) {4 := get(4) - 1;}}
by Sequence-1 and Set-2 and Minus-search-1
because [(4,2)] 2 - 1 |-> 1 by Minus-instr

  [(4,2)] | {4 := 1 ; while get(4) {4 := get(4) - 1;}}
|-> [(4,1)] | {Noop ; while get(4) {4 := get(4) - 1;}}
because [(4,2)] 4 := 1 |-> [(4,1)] Noop by Set-instr

  [(4,2)] | {4 := 1 ; while get(4) {4 := get(4) - 1;}}
|-> [(4,1)] | {Noop ; while get(4) {4 := get(4) - 1;}}

  [(4,2)] | {4 := 1 ; while get(4) {4 := get(4) - 1;}}
|-> [(4,1)] | while get(4) {4 := get(4) - 1;}
by Sequence-2

... repeat the last 8 steps in the [(4,1)] memory ...

  [(4,9)] | while get(4) {4 := get(4) - 1;}
|-> [(4,9)] | if get(4) {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
by While-expand

  [(4,0)] | if get(4) {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
|-> [(4,0)] | if 0 {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
by If-search because get-Instr [(4,0)] get(4) |-> 0 by get-Instr

  [(4,0)] | if 2 {4 := get(4) - 1; while get(4) {4 := get(4) - 1;}} else {}
|-> [(4,0)] | noop;
by If-false

```

6 Bonus Problem: Variables and Functions

TASK. Add local variables to the language, by giving syntax, operational semantics, and some examples.

TASK. Add functions to the language, by giving syntax, operational semantics, and some examples.

7 Bonus Bonus Problem: Progress

If you finish all that, see me for a theorem to prove about the operational semantics.