

COMP 321 Lecture 03: Operational Semantics

Dan Licata

Fall 2026

Last time, we saw how to turn a string into an abstract syntax tree by lexing/tokenizing and parsing. In these classes, we will look at how to define what it means to run a program.

The syntax tree representation is useful for this because, in a string, the subexpressions of an expression are *next to* the expression, while in a tree, the subexpressions are *subtrees*. For example, in the string " (1+2) " the operands of the + are before and after the +, while in the string `get (2)` the operand to `get` is after it. In contrast, in an abstract syntax tree like `Plus( $e_1, e_2$ )` or `Get( $e$ )`, the operands are subtrees of the expression — the parser does the work of grouping the operands as subtrees. When we define what it means to run a program, we will often recursively run these subtrees.

1 Expressions

1.1 Syntax

Last time, we introduced the following language constructs:

- A numeric constant `k` is a natural number (maybe capped at $2^{32} - 1$ or $2^{64} - 1$ to fit into a single unsigned machine integer). The corresponding abstract syntax is `Num( $k$ )`.
- The concrete syntax `( $e_1$  +  $e_2$ )` adds e_1 and e_2 . The corresponding abstract syntax is `Plus( $e_1, e_2$ )`.
- The concrete syntax `get ( $e$ )` gets the value at the memory location specified by the value of e . The corresponding abstract syntax is `Get( $e$ )`.

We can give a grammar for the abstract syntax trees of this language:

$$e ::= \text{Num}(k) \mid \text{Plus}(e_1, e_2) \mid \text{Get}(e)$$

This grammar specifies what are called *expressions*, which is terminology for the parts of a program that have a *value* or *answer*. For example, the value of $(1 + (2 + 3))$ is 6, while the value of `get (2)` is a memory that maps address 2 to 17 is the number 17.

1.2 Operational Semantics

Our job now is to define what it means to run such a program. We will do this via three predicates/relations:

1. e done is a predicate which means that the expression e is a value.
2. $e \mapsto e'$ (read “ e steps to e' ”) is a binary relation on two expressions, specifying that the expression e becomes the expression e' after one individual step of work.
3. $e \mapsto^* e'$ (read “ e steps to e' in many steps”) is a binary relation on two expressions, specifying that the expression e steps in a chain of zero or more individual steps to e' .

Here are some examples for how these should be defined:

- `Num(3)` is done
- `Plus(Num(2), Num(3))` is not done.
- `Plus(Num(2), Num(3))` steps to `Num(5)`. We write

$$\text{Plus}(\text{Num}(2), \text{Num}(3)) \mapsto \text{Num}(5)$$

for this.

- `Plus(Num(1), Plus(Num(2), Num(3)))` steps to `Plus(Num(1), Num(5))`.
- `Plus(Num(1), Plus(Num(2), Num(3)))` steps to `Num(6)` in many steps. We write

$$\text{Plus}(\text{Num}(1), \text{Plus}(\text{Num}(2), \text{Num}(3))) \mapsto^* \text{Num}(6)$$

for this.

These predicates are defined by the following rules. First, we say that `Num( $k$ )` is done, and no other expressions are done.

For stepping, the most basic rule is

`Plus-instr` For all k_1 and k_2 , $\text{Plus}(\text{Num}(k_1), \text{Num}(k_2)) \mapsto \text{Num}(k_1 + k_2)$

For example, this rule can be used to conclude that $\text{Plus}(\text{Num}(2), \text{Num}(3)) \mapsto \text{Num}(5)$ by taking k_1 to be 2 and k_2 to be 3.¹

However, this rule does not fully define what it means to evaluate $\text{Plus}(e_1, e_2)$ to a value, because e_1 and e_2 might not yet be values of the form $\text{Num}(k)$. For example, consider $\text{Plus}(\text{Num}(1), \text{Plus}(\text{Num}(2), \text{Num}(3)))$. So we also need rules that evaluate the subtrees. First,

Plus-search-1 For all expressions e_1 and e'_1 and e_2 , if $e_1 \mapsto e'_1$ then $\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e'_1, e_2)$.

According to this rule, we can conclude that $\text{Plus}(\text{Plus}(\text{Num}(2), \text{Num}(3)), \text{Num}(1))$ steps to $\text{Plus}(\text{Num}(5), \text{Num}(1))$ by taking e_1 to be $\text{Plus}(\text{Num}(2), \text{Num}(3))$ and e'_1 to be $\text{Num}(5)$ and e_2 to be $\text{Num}(1)$, with the premise $e_1 \mapsto e'_1$ holding by the rule **Plus-instr**.

However, we're still not quite done, because e_2 also might not yet be done, as in $\text{Plus}(1, \text{Plus}(\text{Num}(2), \text{Num}(3)))$. So we need

Plus-search-2 For all expressions e_1 and e_2 and e'_2 , if $e_2 \mapsto e'_2$ then $\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)$.

According to this rule, we can conclude that $\text{Plus}(\text{Num}(1), \text{Plus}(\text{Num}(2), \text{Num}(3)))$ steps to $\text{Plus}(\text{Num}(1), \text{Num}(5))$ by taking e_2 to be $\text{Plus}(\text{Num}(2), \text{Num}(3))$ and e'_2 to be $\text{Num}(5)$ and e_1 to be $\text{Num}(1)$, with the premise $e_2 \mapsto e'_2$ holding by the rule **Plus-instr**.

1.2.1 Determinism

The rules **Plus-search-1** and **Plus-search-2** overlap, in that an expression like

$$\text{Plus}(\text{Plus}(\text{Num}(1), \text{Num}(2)), \text{Plus}(\text{Num}(3), \text{Num}(4)))$$

can step both by **Plus-search-1** to

$$\text{Plus}(\text{Num}(3), \text{Plus}(\text{Num}(3), \text{Num}(4)))$$

¹One might reasonably object that this rule doesn't really explain anything, since it uses a prior notion of $+$ to explain the **Plus** of this new programming language. The idea with a rule like this is that we are assuming that our mathematical metalanguage in which we are defining this stepping relation already has a notion of addition; if not, we could build it from some more primitive ingredients, like Peano arithmetic. On the implementation side, we are assuming that a computer's processor already knows how to add (unsigned) machine integers.

and by `Plus-search-2` to

`Plus(Plus(Num(1), Num(2)), Num(7))`

We generally expect a program to have a single behavior, not more than one. Having one behavior is called *determinism*, while having more than one is called *non-determinism*.² For now, we will study deterministic languages where all programs have at most one next step; later on, we will look at *concurrent/multi-threaded* languages that are intentionally non-deterministic.

To make the above rules deterministic, we need to pick one of `Plus-search-1` and `Plus-search-2` to take priority over the other. Picking `Plus-search-1` is called *left-to-right evaluation order*, while picking `Plus-search-2` is called *right-to-left evaluation order*. We'll pick left-to-right, though the choice is somewhat arbitrary. We can implement this choice by guarding `Plus-search-2` with a condition that says that `Plus-search-1` does not apply, checking that e_1 is done so does not take a step:

`Plus-search-2` For all expressions e_1 and e_2 and e'_2 , if $e_2 \mapsto e'_2$ **and** e_1 done then $\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)$.

Finally $e_1 \mapsto^* e_n$ is true if there is a sequence of steps $e_1 \mapsto e_2 \mapsto \dots \mapsto e_{n-1} \mapsto e_n$ where $e_i \mapsto e_{i+1}$ according to the above definition.

1.3 Inference Rules

Throughout this semester, we will use a terser notation for the concepts introduced above called *inference rules*. An inference rule notates a statement like “for all a,b,c if A and B then C” by writing

$$\frac{A \quad B}{C}$$

A and B are called the *premises* and C the *conclusion*. A proof or *derivation* can be notated by stacking inference rules in a tree.

²In this case, continuing to step will reconcile the differences, with both sides eventually producing `Num(10)`. If any two next steps can be reconciled, the language has a property called *confluence*, which means that there is only the *appearance* of non-determinism.

Concretely, the above rules are written as follows:

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

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

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

We can write a proof/derivation by stacking rules until each branch of the trees gets to a rule with no premises, which is called an *axiom*. For example

$$\frac{\frac{}{\text{Plus}(\text{Num}(3), \text{Num}(4)) \mapsto \text{Num}(7)} \text{ Plus-instr} \quad \frac{}{\text{Num}(3) \text{ done}}}{\text{Plus}(\text{Num}(3), \text{Plus}(\text{Num}(3), \text{Num}(4))) \mapsto \text{Plus}(\text{Num}(3), \text{Num}(7))} \text{ Plus-search-2}$$

2 Expressions with Memory

So far, we have defined what it means to run a program that uses Plus and Num; we still have to give operational semantics rules for Get.

Since the e in $\text{Get}(e)$ might not be a value, we should have a search rule that runs it:

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

Finally, we come to the case for

$$\frac{}{\text{Get}(\text{Num}(k)) \mapsto \text{the value in the memory at address } k} \text{ Get-instr}$$

To make this precise, we need to know what the current values of memory are. This requires changing the whole operational semantics from a binary relation on two expressions to a three-place relation on a memory and two expressions. We will write this relation as

$$\mu \mid e \mapsto e'$$

and read it as “in memory μ , e steps to e' ”.

How should we represent the memory μ ? We need it to determine a function from addresses to values (in this case, numbers), so it could be a function $\text{unsigned int} \rightarrow \text{unsigned int}$. However, a computer’s memory is finite (e.g. your

computer has 16GB of RAM), so the memory can't truly be a function with infinite domain. We can represent a finite-domained function as a list of tuples like

$$[(a_1, v_1), (a_2, v_2), (a_3, v_3), (a_4, v_4), \dots, (a_n, v_n)]$$

We define

$$lookup(\mu, a) = v_i \text{ if } a = a_i$$

to be the value at address a in the memory. We will write a memory whose addresses are $0, \dots, n-1$ as an array

v_0	v_1	v_2	$\dots$	v_n
0	1	2	$\dots$	n

Using a memory, the above stepping rule for Get becomes

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

We also need to change all of the previous rules to carry the memory through to the place where it is used

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

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

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

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

Example. Suppose μ is the memory $[(2, 100)]$ and the program is $\text{get}(1+1)$. The first step it takes is

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

Then this steps as follows

$$\frac{}{\mu \mid \text{Get}(\text{Num}(2)) \mapsto \text{Num}(100)} \text{Get-instr}$$

So overall we have

$$\mu \mid \text{Get}(\text{Plus}(\text{Num}(1), \text{Num}(1))) \mapsto \text{Get}(\text{Num}(2)) \mapsto^* \text{Num}(100)$$

3 Statements

C-like languages (C, C++, Java, C#, ...) make a distinction between expressions, which have values, and *statements*, which are run for their effects (writing to memory, printing, sending data over the network, displaying GUI elements, ...). We will now extend the syntax of the language with some statements, by adding a second type of program to the grammar for programs.

For the concrete syntax, we have

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

For now, the only statement is an *assignment* $e1 := e2;$, which means to change the memory address $e1$ so that it contains the value of $e2$. Next time, we will add a bunch of other kinds of statements, like conditionals (`if`) and loops (`while`, `for`). Notice that the syntax of statements refers to the syntax of expressions.

In abstract syntax, we have

$$\begin{aligned} e &::= \text{Num}(k) \mid \text{Plus}(e_1, e_2) \mid \text{Get}(e) \\ s &::= \text{Set}(e_1, e_2) \end{aligned}$$

and one can extend the parser from Homework 1 to parse statements as well, turning $e1 := e2$ into $\text{Set}(e_1, e_2)$.

The operational semantics will now require two single-step relations,

$$\mu \mid e \mapsto e'$$

as before and

$$(\mu \mid s) \mapsto (\mu' \mid s')$$

for stepping a statement.

As above, the first relation evaluates an expression in a memory without changing the memory.³

The second relation steps a *machine state*, written $\mu \mid s$ where μ is a memory and s is a statement, to another machine state.⁴ Read this as “in the memory μ , the statement s steps to the statement s' , changing the contents of memory to μ' ”.

³In C-like languages, expressions actually can change memory, but for now we will keep things simple and ignore this.

⁴There is no particular reason for the notation $(\mu \mid s)$ — formally you could write (μ, s) for this pair, but $,$ gets used for lots of other things, so we will avoid it here. This use of $\mid$ has nothing to do with its use in grammars for syntax.

First, we have some search rules

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

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

Notice that the relation for stepping statements uses the relation for stepping expressions in the premise. Because expressions can't change memory (for now), the search rules always output the same memory μ that they started with. *Set-search-2* determinizes the case when both e_1 and e_2 can take a step in the same way as above.

Finally, we get to the case for $\text{Set}(\text{Num}(a), \text{Num}(v))$, which is supposed to set the contents of address a to be v . To do this, we will need an operation that changes memory:

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

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

This is basically a dictionary insert for a dictionary represented as a list of key-value pairs.

Using *insert*, the idea is to step an assignment like this

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

However, we also need to produce a “next statement” as output. To do this, we can add a statement *Noop* (read “no-op” for “no operation”), which is an expression that “does nothing”. For example, a processor running *Noop* would be *idle* for a timestep. We use *Noop* to represent a finished program:

$$\overline{\text{Noop done}}$$

So the rule becomes

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