

COMP 321: Design of Programming Language, Fall 2026

Homework 2: Operational Semantics

In this assignment, you will learn to implement operational semantics. The goal is both for you to understand how operational semantics are structured, and to understand how to implement them in SML.

To unzip the support code, you should be able to double-click to open the downloaded tar file on both Windows and Mac. If that fails, then do `tar -xf hw02-handout.tar`.

To hand in this homework, upload `step.sml` to your Google Drive handin folder.

1 Language Reference

The language will have the following constructs:

```
e ::= k | (e1 + e2) | get(e)
s ::= e1 := e2; | noop; | if e {s1} else {s2} | {s1 s2}
```

Everything except `if` and `{s1 s2}` was discussed in Tuesday’s lecture, while those two were part of Thursday’s lab problems. The conditional `if e {s1} else {s2}` runs the statement `s1` when `e` is nonzero and runs the statement `s2` when `e` is zero (i.e. nonzero numbers represent “true” and 0 represents “false”). `{s1 s2}` is the notation we will use for the sequencing of two statements, which means to run `s1` and then run `s2`. See the lecture notes and lab solutions from this week on the course website for more discussion of the language that you will implement. Here is a quick reference:

In abstract syntax, we have¹

$$e ::= \text{Num}(k) \mid \text{Plus}(e_1, e_2) \mid \text{Get}(e)$$
$$s ::= \text{Set}(e_1, e_2) \mid \text{Noop} \mid \text{If}(e, s_1, s_2) \mid \text{Sequence}(s_1, s_2)$$

These expressions/statements are done:

$$\frac{}{\text{Num}(k) \text{ done}} \quad \frac{}{\text{Noop done}}$$

Operations on memory:

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

¹Note that `Sequence` was called `Then` in lab, but it is renamed here to avoid confusing it with the “then” part of an `if` statement.

Rules for stepping expressions:

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

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

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

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

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

Rules for stepping statements:

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

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

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

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

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

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

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

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

2 Implementation

The support code for the homework is organized as follows:

- `input.sml`, `stream.sml`, `lexer.sml`, `parser.sml`, and `print.sml` implement user input, lexing, parsing, and printing. You probably don't need to look at these files at all, except `parser.sml` has the exact grammar for the concrete syntax.

- `syntax.sml` has the syntax datatypes. You will want to read this but don't need to change it.
- `top.sml` has the functions that run tests. You will want to read the signature, but don't need to read the implementation.
- `step.sml` is where you fill in your code.
- `test-X.imp` are test programs that can be run using your implementation.

Task 1 (90%). In `step.sml`, write four functions:

- The first function,

```
val lookup_memory : Syntax.memory * int -> int
```

implements the *lookup* function defined above. It takes a memory (represented as a list of address-value pairs) and an address and returns the corresponding value. You should raise the exception `StepError` if the address is not in the memory.

- The second function

```
val update_memory : Syntax.memory * int * int -> Syntax.memory
```

implements the *update* function defined above.

- The third function

```
val progress_exp : Syntax.memory * Syntax.exp -> Syntax.exp result
```

implements the e done and $\mu \mid e \mapsto e'$ rules listed above. The result type is defined as follows:

```
datatype 'a result = Stepped of 'a | Done
```

So an `exp result` is either `Done` or `Stepped(e')` where $e' : \text{exp}$. In particular, `progress_exp` should satisfy the following two specifications:

`progress_exp( $\mu, e$ )` returns `Done` iff e done, and returns `Stepped( $e'$ )` iff $\mu \mid e \mapsto e'$.

`progress_exp` should raise the exception `StepError` if neither of these applies.

- The fourth function

```
val progress_statement : Syntax.memory * Syntax.statement
  -> (Syntax.memory * Syntax.statement) result
```

implements the s done and $(\mu \mid s) \mapsto (\mu' \mid s')$ rules above. It should satisfy the following two specifications:

`progress_statement( $\mu, s$ )` returns Done iff s done, and returns `Stepped( $(\mu', s')$ )` iff $\mu \mid s \mapsto \mu' \mid s'$.

`progress_statement` should raise the exception `StepError` if neither of these applies.

If you want to test as you go, do things in the following order: the set statement, numeric constants, the plus expression, the sequence statement, the get expression, the if statement. The front end accepts a statement, not an expression, but once you have done the set statement, you can test any expression e by running the statement `0 := e;`. The front end always starts from the empty memory, so you will also need to implement the sequence statement before you can test the get expression, so that you can set a location and then get its value.

3 Testing

Every time you make an edit, you can compile the code by doing this in SMLNJ:

```
- CM.make("sources.cm");
```

Here is how to test. In `top.sml`, you will find 6 functions

```
(* read a single statement from what you type *)
val Top.run_print : unit -> unit (* just print the same expression back *)
val Top.run_step  : unit -> unit (* step until done, showing all steps of evaluation *)
val Top.run_eval  : unit -> unit (* step until done, show just the done state *)

(* read a single statement from a source file *)
val Top.file_print : string -> unit
val Top.file_step  : string -> unit
val Top.file_eval  : string -> unit
```

The `run` functions allow you to type in a single statement. The `file` functions allow you to read a single statement from a file like `test-blah.imp`.

For each of these, the `print` version just prints back the program you entered. This is useful if you just want to see if a new program parses. The `step` version steps a single statement until it is done using your `progress_statement` and shows each step of the operational semantics; this should be your primary way to test. The `eval` version runs the statement until it is done by repeatedly applying your `progress_statement`. It prints out only the final memory and statement. You can use this to quickly test if a program's behavior is what you expect.

For example,

```

- Top.file_step("test-plus.imp");
memory
statement  0 := 1 + 1;
Press return:
memory
statement  0 := 2;
Press return:
memory      0->2
statement  noop;
Press return:

- Top.run_step();
IMP0>0 := (1 + 1);
memory
statement  0 := 1 + 1;
Press return:
memory
statement  0 := 2;
Press return:
memory      0->2
statement  noop;
Press return:

```

Task 2 (10%). In a comment at the bottom of your `step.sml` explain what the final memory should be for each of the following statements. Then check that your implementation produces the correct answer on each file.

```

test-set.imp
test-set2.imp
test-set3.imp
test-plus.imp
test-plus2.imp
test-seq.imp
test-get.imp
test-get2.imp
test-get3.imp
test-if.imp
test-if2.imp
test-if3.imp

```