

COMP 321: Design of Programming Language, Fall 2026

Homework 3: Variables, Functions, C Syntax

In this assignment, you will learn to implement operational semantics for variables and functions. We will call this language IMP (for imperative). Next, you will implement a translation from C-style syntax for variables to IMP.

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 hw03-handout.tar`.

To hand in this homework, upload `syntax.sml`, `step.sml`, `elab.sml`, `reverse.imp` and `reverse.impc` to your Google Drive handin folder in the `hw03` folder.

1 Variables and Functions

In this problem, you will add local variables and functions to the operational semantics for IMP. The support code includes the cases for the language constructs that you implemented last week, so you don't need to redo them. One difference from last week is that `step_statement` takes an additional argument, representing the program as a list of function declarations.

You can find the intended definition of the operational semantics for local variables and functions in the lecture notes for this week, linked from the Lectures page of the website <https://dlicata.wescreates.wesleyan.edu/teaching/pp1-26f/lect.html>. To make it possible to write more example programs, the language includes subtraction (`-`) and (rounding-down) integer division (`/`) in addition to addition. Warning: the parser does not do precedence of arithmetic operations and always right-associates, so you should fully parenthesize any expression involving division or subtraction.

1.1 Substitution

The first step in implementing variables is to implement substitution of a memory address for a variable.

Task 1 (15%). In `syntax.sml`, implement the substitution functions `subst_exp` and `subst_statement` for substituting into an expression and into a statement.

1.2 Representation of memory

Last week, you represented a memory as a list of address/value pairs. This week, to make it easier/faster to allocate memory for local variables, we will use a more realistic representation as an array A , paired with an integer $next$ identifying the next available memory location. An array is represented by the SML Vector type, which is documented at <https://smlfamily.github.io/Basis/vector.html>. For example, the list memory $[(0,211),(1,212),(2,321)]$ will be represented by an array with elements $[211,212,321]$ at indices $[0,1,2]$, paired with the number 3 indicating that the next available array entry is at position 3. The

domain of $(A, next)$ is $0, \dots, next - 1$. The semantics of the functions on memory will be slightly different than the list-based implementation we used last week.

We will need four functions on memories:

- The function `update_memory` takes a memory, an address, and a value and updates the memory to contain that value at that address. If $a \in \text{dom}(\mu)$, `update_memory( $\mu$ ,  $a$ ,  $v$ )` should behave like the list-based `update( $\mu$ ,  $a$ ,  $v$ )` function that we have discussed, replacing the value of a with v . If $a \notin \text{dom}(\mu)$, the `update_memory` should write v to address a if it is present in the underlying array of μ ; `update_memory` has unspecified behavior otherwise. Unlike the list-based `update( $\mu$ ,  $a$ ,  $v$ )`, the `update_memory` should not add a to the domain of μ if it is not already present.
- The function `lookup_memory` takes a memory and an address and returns the value at that address. If $a \in \text{dom}(\mu)$, `lookup_memory( $\mu$ ,  $a$ )` should behave like the list-based `lookup( $\mu$ ,  $a$ )` that we have discussed. If $a \notin \text{dom}(\mu)$, `lookup_memory` should return whatever value is in the underlying array of μ at position a if it exists, and has undefined behavior otherwise.
- The function `allocate` takes a memory μ returns the next available memory address a , along with a new memory μ' such that $a \in \text{dom}(\mu')$, with a containing some unspecified value v . The testing infrastructure will create an initial array of some length (`Actions.memsize` in `Top.sml`); if this size of memory becomes full, the behavior of `allocate` is unspecified.
- The function `deallocate` takes a memory μ and an address a and returns a new memory μ' such that a is not in $\text{dom}(\mu')$, with all addresses mapping to the same values. For this week, you can assume that variables are deallocated in stack order, so that `deallocate( $\mu$ ,  $a$ )` is only ever called with a being the most recently allocated variable.

Task 2 (15%). In `step.sml`, implement the `update_memory` and `lookup_memory` and `deallocate` and `allocate` functions.

1.3 Operational Semantics

Task 3 (15%). Implement the cases of `progress_statement` for local variables, function calls, and freeing memory, as well as while loops (since you didn't do that last week). You should follow the operational semantics from the lecture notes; you will need to extend the function rule given there to multi-input functions.

To test your operational semantics, there are a few basic tests that you can run with

```
Top.file_step "test-var1.imp";
Top.file_step "test-var2.imp";
Top.file_step "test-vardecl.imp";
Top.file_step "test-funcall.imp";
Top.file_step "test-funcall-args1.imp";
Top.file_step "test-funcall-args2.imp";
Top.file_step "test-funcall-args3.imp";
Top.file_step "test-funcall-args4.imp";
```

Notes on testing: The top-level runs the statement `main()`; so you might want to implement function calls with 0 arguments before function calls with arguments, then test the operational semantics for variable declarations, and then go back and implement function calls with arguments. If you want to see more of memory printed out, change the variable `print_beyond` in `Top.sml`, which controls how much unallocated memory *after* the top of the stack gets printed out (this is necessary because, with stack allocation, all memory should be freed by the end of your program). You should write additional tests for any aspects of the code that these programs do not test.

Task 4 (15%). To more fully test your operational semantics, in the file `reverse.imp`, write an IMP program to in-place reverse an array. The function `reverse` should take two inputs, the address of the 0th element of an array and the array's length. If you haven't seen in-place reversal before, the algorithm is to loop from 0 to the midpoint of the array, swapping the element at position i from the front of the array with the element at position i from the back of the array as you go. The `main()` function given in the support code stack-allocates an array `[41,42,43,44,45,46,47,48,49]`; after running `reverse` the memory should be `[49,48,47,46,45,44,43,42,41]`.

2 C-like Syntax

The IMP language has an assembly-language-like syntax for getting and setting memory, where `get` is used to explicitly get the contents of a memory address, and `a := v`; takes a memory *address* `a` and a value `v` and sets the address to that value. Since we substitute addresses for variables, a variable `x` by itself stands for an address, and `get(x)` stands for the contents of the variable. For example, to increment the value in `x`:

```
{var x;
  x := 0;
  x := (get(x) + 1);
}
```

To increment the value in whatever is pointed to by `y`:

```
{var x;
  x := 0;
  {var y;
    y := x;
    get(y) := (get(y) + 1);
  }
}
```

C uses a different syntax, where when a variable is used in an expression, it implicitly refers to the *value* stored in that variable, not the address of that local variable. The address operator `&` is used to refer to the variable's memory address itself. Additional levels of pointers beyond the local variable itself are still explicitly dereferenced with the `*` dereference operator, though. In the address operator `&e` and in an assignment statement `e = e2`;, the expression `e` must be something called an *lvalue*. An lvalue in C terminology is often referred to as “an expression whose value has a memory location”. This means the equivalent C program has a different number of dereferences (`*`) than an IMP program has `gets`. For example, the above examples are written

```
int x = 0;
x = x + 1;
```

and

```
int x = 0;
int* y = &x;
*y = *y + 1;
```

The intended reading of `x = x + 1;` is “after running this statement, the next time you use `x` in an expression to stand for the *value* in the memory address corresponding to `x`, the value will be the current value of `x` plus 1”. Similarly the intended reading of `*y = *y + 1;` is “after running this statement, the next time you run `*y`, the value will be the current value of `*y` plus 1.”

The file `csyntax.sml` contains an abstract syntax for C-like syntax, which is parsed by the parser in `parsec.sml`, and the module `TopC` contains the testing infrastructure for C-like syntax (e.g. `run TopC.file_step "filename.impc"`).

In this problem, your task is to write an *elaborator* that translates C-like syntax to the equivalent IMP syntax. That is, in `elab.sml` you will write functions `elab_exp` and `elab_statement` and `elab_decl` that take a C-like abstract syntax tree (`CSyntax.exp`, `CSyntax.statement`, `CSyntax.decl`) and output a IMP abstract syntax tree (`Syntax.exp`, `Syntax.statement`, `Syntax.decl`).¹ For this homework, your elaborator should allow variable names and dereferences of an arbitrary expression as lvalues (e.g. `x`, `*x`, `*(x+1)`).

Using your elaborator, will run the C-like syntax using the operational semantics you have already implemented by stepping its translation.

Task 5 (30%). Implement the elaboration in `elab.sml`. Many C-like syntax constructs are translated to the same construct in IMP; these cases are given in the support code (you can change them if you want to, but all of these are part of my solution). You need to do the cases for variables, address-of, and dereference expressions, and assignment and local variable declaration statements.

Task 6 (10%). In `reverse.impc`, write your array reverse code from Task 4 in C-like syntax. See `test-var.impc` `test-var1.impc` and `test-var2.impc` for examples of the new concrete syntax; everything else is written the same as in IMP.

¹Because we are working with two different SML types of abstract syntax trees in this problem, the support code does not “open” the syntax module like we usually do, so you need to refer to all datatype constructors and functions by their full module path, e.g. `CSyntax.Var` and `Syntax.Var`