

COMP 321 Lecture 5/6: Variables and Functions

Dan Licata

Fall 2026

See the Lab solutions linked from last Wednesday's class for the material on sequencing, conditionals, and while loops that we covered at the beginning of Monday's class.

In these classes, we will look at how to formalize variables and functions in our operational semantics.

1 Variables

In C, what can you do with a local variable? You can get the value of the variable (x), you can get the address of the variable ($\&x$), and you can assign a new value to the variable ($x = 1$;). To support these operations, we will take the point of view that

in an imperative language, a variable is a placeholder for a memory address that is chosen by the language implementation

For example, consider the statement

```
var x;  
x := get(y) + 1;  
z := get(x);
```

where y and z are some other variables that are in scope. This declares a new local variable x , sets it to be the current value of y plus 1, and then assigns the value of x to the other variable z . The corresponding program in C notation is

```
int x;  
x = y + 1;  
z = x;
```

This statement is morally the same as

```
19 := get(y) + 1;
z := get(19);
```

as long as memory address 19 is not used elsewhere in the program as long as x is in scope. Of course, we could just as well have chosen

```
20 := get(y) + 1;
z := get(20);
```

or any other unused memory address. If we choose an address that is already in use, then what are supposed to be two different variables will get collapsed into one. E.g.

```
20 := get(10) + 1;
30 := get(20);
```

means the same thing as the program above if the rest of the program uses address 10 for y and address 30 for z , but

```
10 := get(10) + 1;
30 := get(10);
```

(using address 10 for both y and x) does not mean the same thing because it also increments y , while the original statement leaves y unchanged.

From this point of view, the main use of variables is to make program more readable (since humans are better at reading variable names than memory addresses), and to make program more *portable* — if you hard-code a specific memory address like 19 or 20, you can only run the program if that address is available. Memory is given to your program by the operating system, and the exact addresses given to the program will change from run to run, depending on what memory is in use by other programs. So we would need some translation so that that memory address 0 in the program means the first address given to your program, not the literal virtual memory address 0, at which point the “addresses” are basically variables.

1.1 Scope

In concrete syntax, we will use the new statement

```
s ::= ... | {var x; s}
```

to declare a variable x in the scope of the statement s . (Think `{int x; s}` but we have not introduced types yet.) This means the variable x can be used in s , and

cannot be used after the closing curly brace after s . We call this a *local variable in s* . For this week, we will not yet check that variables are only used when they are in scope, but the operational semantics will leave the use of a variable that is out of scope undefined. In abstract syntax we write $\text{LocalVar}(x.s)$, where the $.$ indicates that x is in scope in s .

We use a variable by the new expression

```
e ::= ... | x
```

that allows a variable to be used as an expression. In abstract syntax, on paper we will write x as an expression, though in SML code it will be represented by $\text{Var}(x)$, where x is a variable name.

For example,

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

is allowed because x is in scope when it is used, but

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

is not allowed.

One question that comes up is what happens if you reuse a variable, which is called *shadowing*. For example, after running the code

```
{ var x;
  x := 1;
  {var x;
    x := 2;
  }
  y := get(x)
}
```

should y be 2 or 1? In most languages, a variable name refers to the closest declaration above the use of that variable, so the inner $x := 2$ assigns to the inner declaration of x — which is a different memory address than the outer declaration — leaving the outer declaration of x unchanged, and so y is 1.

1.2 Substitution

To define the operational semantics, we will use a meta-operation (meaning, an operation in the mathematical language that we are using to study programming languages, not in the language we are studying) to replace a variable with a memory address. This operation is called *substitution*, and written $e[a/x]$ where e is an expression, a is a memory address $\text{Num}(k)$, and x is a variable name. This is defined by recursion on the abstract syntax tree of expressions. Suppose the grammar for expressions is

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

Then substitution is defined as follows:

$$\begin{aligned} x[a/x] &= a \\ y[a/x] &= y \text{ if } y \neq x \\ \text{Num}(k)[a/x] &= \text{Num}(k) \\ \text{Get}(e)[a/x] &= \text{Get}(e[a/x]) \\ \text{Plus}(e_1, e_2)[a/x] &= \text{Plus}(e_1[a/x], e_2[a/x]) \end{aligned}$$

That is, to “find and replace all x ’s with a ”, if a variable is x it gets replace, if it isn’t it stays the same, and otherwise you find and replace all x ’s with a in all subtrees of the abstract syntax tree. Similarly, for statements

$$s ::= \text{Set}(e_1, e_2) \mid \text{Sequence}(s_1, s_2) \mid \text{If}(e, s_1, s_2) \mid \text{While}(e, s) \mid \text{LocalVar}(x.s)$$

we define

$$\begin{aligned} \text{Set}(e_1, e_2)[a/x] &= \text{Set}(e_1[a/x], e_2[a/x]) \\ \text{Sequence}(e_1, e_2)[a/x] &= \text{Sequence}(e_1[a/x], e_2[a/x]) \\ \text{If}(e, s_1, s_2)[a/x] &= \text{If}(e[a/x], s_1[a/x], s_2[a/x]) \\ \text{While}(e, s)[a/x] &= \text{While}(e[a/x], s[a/x]) \\ \text{LocalVar}(y.s)[a/x] &= \text{LocalVar}(y.s[a/x]) \text{ if } y \neq x \\ \text{LocalVar}(y.s)[a/x] &= \text{LocalVar}(y.s) \text{ if } y = x \end{aligned}$$

Mostly this just applies the substitution to all subtrees, except when we get to substituting into a declaration of a different local variable, we need to make sure that we handle shadowing appropriately. This can be done by stopping the substitution if the local variable is the same name as the one being substituted for (because in that case, the inner declaration of x takes precedence, and there is no name for the x being substituted for that is in scope in s).

1.3 Operational Semantics

Last time, we represented a memory $\mu = [(a_1, v_1), \dots, (a_n, v_n)]$ by a list of pairs of an address and a value, where each pair represents the fact that the address a_i contains that value v_i . We write $\text{dom}(\mu)$ for the domain of μ , i.e. $[a_1, \dots, a_n]$. The operation $\text{update}(\mu, a, v)$ changes address a to contain v if a is already in μ , or inserts (a, v) if not. To implement the idea that a local variable is a placeholder for a memory location chosen by the language, we can say

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{LocalVar}(x.s) \mapsto \text{update}(\mu, a, 0) \mid s[\text{Num}(a)/x]} \text{Declare}$$

That is, we choose a new memory address a that is not already in the memory, and substitute a for the variable x . For example, according to this rule

```
{ var x; x := get(10) + 1; 30 := get(x); }
|->
20 := get(10) + 1; 30 := get(20);
```

if we chose the address 20 and substitute it for x . (It also steps to $40 := \text{get}(10) + 1; 30 := \text{get}(40)$ or any other address you want, as long as it is not already in use..)

Because of this operational semantics rule for local variable declarations, all variables should be replaced with locations before we would attempt to step them, so the operational semantics should never attempt to step a variable. (An exception is if a variable is used outside of its scope, but we will check for that next week.)

1.4 Warning: Comparison with C

Note that the syntax we are using for variables is different than C syntax. In C, using a variable in an expression (e.g. $x+1$) implicitly gets the *value* stored in the location corresponding to x , which we will write explicitly as $\text{get}(x)$. To avoid implicitly getting the value, you use the “address of” operator $\&$ x , which we will write as just x . The advantage of the C syntax is that it makes what is probably the more common case (getting the value stored in a local variable) shorter to write than the less common case (the address of a local variable). However, it is more difficult to give an operational semantics for the C-style syntax, so we will save that for the homework.

2 Initialization

In the operational semantics above, we used $\text{update}(\mu, a, 0)$ to *initialize* the location a to 0. Initialization is important if we write a program like

```

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

```

that uses `get(x)` without first setting `x` to contain some value. Here are some ways that a language can handle initialization:

- Initialize all new locations to some default value like 0. This is safe and predictable, but potentially time-consuming, especially if one includes initializing all elements of an array.
- Require an initial value to be given in the variable declaration (`{var x := e; s}`). Same tradeoffs as the previous, but does not require all types to have a default value; the programmer must supply one.
- Separate variable declaration and initialization, but require a variable to be set to before its value is gotten. This is more flexible and faster than the previous, because if a program was going to assign to a variable at some point before using it, the program does not require a separate initialization. However, it can sometimes be difficult to tell if a variable has been set. While some assignments are obvious — `x := e` clearly sets `x` — some are hard to see — `get(y) := e` only sets `x` if the address for `x` has been stored in `y`.
- Separate variable declaration and initialization, and do not check that a variable has been set before it is gotten; return whatever happens to be in the memory. This is what C does, and is fastest but also more dangerous, in terms of security. If the memory address used for the variable has been used previously by the program or another program, it might contain some private data (a password, ...), and a get-before-set would be able to accidentally read that private data, even though it is not supposed to. This can lead to security bugs where programs to leak private information. For example, this can happen when a program intentionally prints out the value stored in a variable, but in some paths through the code that variable is mistakenly not initialized, and some private data happens to be stored in the address that is used, because some other part of the program was manipulating that private data without printing it.

One way to model this is with a non-deterministic rule

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{LocalVar}(x.s) \mapsto \text{update}(\mu, a, v) \mid s[\text{Num}(a)/x]} \text{Declare-any}$$

which allows the memory to be initialized to any value you want. Then an implementation can choose “whatever is currently in that address” for v and have a valid transition according to this rule (but this strategy will not implement all valid transitions according to this rule).

3 Memory Management

Consider a program

```
{ var x;  
  x := 10;  
}  
{ var y;  
  y := 20;  
}  
{ var z;  
  z := 20;  
}
```

According to the above operational semantics, we might end with the memory $[(0, 10), (1, 20), (2, 30)]$. If there were n local variables, the memory would have size n . This is because once we allocate an address and update the memory with some value for that address, we never remove the address from the memory. This will cause programs to eventually run out of memory.

Memory management is how programming languages enable programmers to reuse memory. There are two styles:

- automatic memory management, mainly called *garbage collection*. In this style, the operational semantics model stays the same as above, where memory is never de-allocated. However, behind the scenes, the programming language implementation uses a component called a garbage collector to detect and free up memory that will never again be used by the program. This is easier on the programmer (you often don’t need to think about it), and safer (memory that the program will use never accidentally gets deleted), but garbage collection takes time while the program is running, which can be unacceptable in some circumstances.
- manual memory management. The program must explicitly note when memory should be freed, or the programming language has rules that memory is freed at certain points. This is harder on the programmer, more error-prone, but can be faster.

Most higher-level languages these days are garbage-collected, while C is not. When we get to Rust later in the course, it is an interesting example of a language that attempts to solve the safety problems with manual memory management without the time cost of a traditional garbage collector.

To add manual memory management to the language, we introduce a new statement

$s ::= \dots \mid \text{free}(e)$

with the following rules

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

$$\frac{}{\mu \mid \text{Free}(\text{Num}(k)) \mapsto \text{deallocate}(\mu, k) \mid \text{Noop}} \text{Free-instr}$$

where $\text{deallocate}(\mu, a)$ deletes $(a, -)$ from the list:

$$\text{deallocate}([\dots, (a_{i-1}, v_{i-1}), (a_i, v_i), (a_{i+1}, v_{i+1}) \dots], a_i) = [\dots, (a_{i-1}, v_{i-1}), (a_{i+1}, v_{i+1}) \dots]$$

Using this, we can implement *stack allocation of local variables* as in C. This means that when a local variable's scope ends, the memory address used for that local variable is freed.

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{LocalVar}(x.s) \mapsto \text{update}(\mu, a, v) \mid (\text{Sequence}(s[\text{Num}(a)/x], \text{Free}(\text{Num}(a))))} \text{Declare-any-stack}$$

With this rule, the above example at the beginning of this section can re-use the same 1 address at a time for x and y and z , and will finish with the empty memory.

3.1 Use after free bugs

If $\text{free}(a)$ is supposed to “free up” or “deallocate” the memory at address a , what happens when you do something like $\text{free}(20); s; y := \text{get}(20); ?$

The rule for getting the contents of memory

$$\frac{(a, v) \in \mu}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto v} \text{Get-in}$$

means that the operational semantics of $\text{Get}(\text{Num}(a))$ is not defined if you get an address that is not in μ . This can be defined in a few ways:

- The programming language raises some kind of error when this actually happens while running the program. This is safer and more predictable, but slower because each memory access has to check if the memory is currently allocated.
- The programming language attempts to prevent this code from compiling, by somehow tracking frees and gets and ensuring that you do not get a location that has been freed. This is again difficult with pointers. For example, the program

```
{var y;
  {var x;
    y := x;
  }
z := get (get (y) );
}
```

doesn't look like it uses the memory allocated for `x` after it has gone out of scope, but it does by storing that address in a different variable `y`, and then dereferencing that pointer outside of the scope of `x`.

- The `get` steps to whatever happens to be in that memory location, e.g. the value that was there before the location was freed, or whatever some other part of the program wrote to that location in between the `free` and the `get`. This is what C does. It's faster but can lead to the same kind of security bugs as using uninitialized memory, in addition to hard-to-track-down correctness bugs if the program does this by accident, because whatever value happens to be in that memory flows through the program for a while before it causes something to go wrong. One way to model this is with a non-deterministic rule

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto v} \text{Get-notin}$$

which says that `get (a)` can return any value it wants when `a` is not in the memory.

4 Functions

Finally, we will add functions to the language. In concrete syntax, we write a function declaration as

```
f (var x) {
  s
}
```

where f is the function name, x is the variable for the argument, and s is the body. A program is a sequence of function declarations.¹ For this week, functions will not return a value, so think of them as like `void f(int x) { s }` in C. We then add a *statement* $f(e)$ representing a function call. In the homework, you will add multi-argument functions, written

```
f (var x1, var x2, ..., var xn) {
  s
}
```

with function calls written $f(e_1, \dots, e_n)$.

For example, here is a program consisting of two functions:

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

main() {
  {var y;
   y := 27;
   f(y);
  }
}
;
```

By convention, stepping a program starts with running the function call `main()`. After running $f(y)$, the local variable `y` contains 28, because f takes the memory address of `y`, and adds 1 to the value stored there.

In abstract syntax, for unary functions, we say that a program is a list of declarations, where a function declaration has the function name, the argument variable, and the body. Then we add a function call statement that has the function name and the argument expression.

$$\begin{aligned}
 p &::= [] \mid f, p \\
 f &::= \text{Func}(f, x.s) \\
 s &::= \dots \mid \text{Call}(f, e)
 \end{aligned}$$

¹To make the parser easier, in the homework code, this sequence of function declarations must end with a semicolon after the last one.

The operational semantics for Call is

$$\frac{\text{Func}(f, x.s) \in p}{\mu \mid \text{Call}(f, e) \mapsto \mu \mid \text{LocalVar}(x, \text{Sequence}(\text{Set}(x, e), s))} \text{Call}$$

That is, find the function declaration for f , and then step to a statement that allocates a local variable for the argument, sets the variable to the argument, and then runs the body. Having already established the operational semantics for local variables, we can reuse them here to simplify the definition of functions (compared to if we wrote out all of the rules needed to step $\{\text{var } x; x := e; s\}$ all over again).