


## COMP 317: Semantics of Programming Languages

## Summary of Lectures 7 and 8



Note: the material covered in these two lectures is not covered in the [draft notes](#).

Listen to the lecture given on 16/2/16: 

The language SImpL has been kept as simple as possible so that we can concentrate on the basic principles of denotational semantics. The language has just one data type, integers, and the only flow-of-control constructs are sequential composition, conditionals, and while-loops. Real programming languages are more complex and offer many more features: these two lectures give a sketch of how these features can be given a denotational semantics. See the [past exam papers](#) for more examples.

Some constructs, such as for-loops and switch-statements, simply involve adding new syntactic constructs, and then adding corresponding clauses to the inductive definitions of the denotation functions. For example, for-loops can be added to the language by adding the following clause to the BNF definition of the language.

$$\langle \text{Program} \rangle ::= \text{for } (\langle \text{Program} \rangle; \langle \text{Boolean Expression} \rangle; \langle \text{Program} \rangle) \{ \langle \text{Program} \rangle \}$$

Since our inductive definitions follow the BNF definitions, if we want to define  $[[P]]$  for this new language of Programs, then we need to say what  $[[P]]$  is when  $P$  is a for-loop. So we might add a new clause to our inductive definition, for example

$$\begin{aligned} &\text{for Programs } P1, P2 \text{ and } P3, \text{ and all Boolean expressions } B, \\ &[[ \text{for}(P1; B; P2) \{ P3 \} ]] = [[ P1 \text{ while}(B) \{ P3 P2 \} ]]. \end{aligned}$$

This clause and the "old" clauses defining  $[[P]]$  together give a complete inductive definition of the semantics of the "new" language that includes for-loops.

Switch-statements can be added to SImpL in a similar way (see the [third tutorial problem-sheet](#)). Some constructs, on the other hand, can't be added so straightforwardly: the semantics of the new construct changes the semantics of the whole language, and we have to throw away the old semantics and start again from scratch. As an example, suppose we want to add post-increments (e.g., `'x++`) to the expressions of SImpL.

As an expression, the value of `'x++` is just the value of `'x`. But such expressions don't just give values; evaluating `'x++` has the side-effect of incrementing the value of `'x`. For example, if we run the program

```
'y := 2 * 'x++;
```

in a state where 'x has the value 1, then in the resulting state, both 'x and 'y have the value 2. Thus, when we evaluate an expression like  $2 * 'x++$ , we get *two* things: an integer value (in this example, 2); and a new state (in this example, the state where the value of 'x has been incremented to 2). The semantics of the language has to describe both the value we get by evaluating an expression, and the changes to the state that result as a side-effect of evaluating expressions.

Under the old rules, the denotation  $[[E]]$  of an expression  $E$  was a function from states to integers. This describes the value we get by evaluating an expression, but cannot say anything about any side-effects. In order to describe these side-effects, we need to get a state (the updated state) as output from this function. That is, in order to describe both the value and the side-effects of an expression, the denotation  $[[E]]$  should be a function that takes a state as input (the state in which we start to evaluate the expression), and gives as output an integer (the value of the expression) *and* a state (the state updated by any side-effects of  $E$ ).

This means that the denotation  $[[E]]$  in a language with post-increments is completely different from denotations in languages without post-increments: the type of the function is different, returning a pair (int and state) rather than just an int. When we add post-increments to SImpL, then, we have to throw away the old semantics, where  $[[E]]$  returns an integer, and replace it with a completely different semantics where  $[[E]]$  returns an integer and a state. As an example, if  $S('x) = 3$ , then  $[[2 * 'x++]](S)$  should be the pair  $(6, S['x \leftarrow 4])$ .

The syntax of expressions is now

$$\langle \text{Expression} \rangle ::= \langle \text{Numeral} \rangle \mid \langle \text{Variable} \rangle \mid \langle \text{Expression} \rangle + \langle \text{Expression} \rangle \mid \\ \langle \text{Expression} \rangle * \langle \text{Expression} \rangle \mid -\langle \text{Expression} \rangle \mid \langle \text{Variable} \rangle ++$$

The semantics of expressions is

for all States  $S$  and Numerals  $N$  and Variables  $V$  and Expressions  $E, E1, E2$

$$\begin{aligned} [[N]](S) &= ([N], S) \\ [[V]](S) &= (S(V), S) \\ [[-E]](S) &= (-n, S') && \text{where } (n, S') = [[E]](S) \\ [[E1+E2]](S) &= (n1+n2, S'') && \text{where } (n1, S') = [[E1]](S) \text{ and } (n2, S'') = [[E2]](S') \\ [[E1*E2]](S) &= (n1 \cdot n2, S'') && \text{where } (n1, S') = [[E1]](S) \text{ and } (n2, S'') = [[E2]](S') \\ [[V++]](S) &= (S(V), S[V \leftarrow S(V)+1]) \end{aligned}$$

Boolean expressions can contain expressions, for example  $'x++ < 100$ . This means that evaluating Boolean expressions also changes state as a side-effect, so for any Boolean expression  $B$ , its denotation  $[[B]]$  is a function that takes a state as input and returns not just a Boolean value, but also a state that records any side-effects of evaluating  $B$ . The inductive definition of  $[[B]]$  therefore also has to be completely re-done. One clause of this redefinition is:

for all States  $S$  and Expressions  $E1, E2$   
 $[[E1 < E2]](S) = (b, S'')$  where  $(n1, S') = [[E1]](S)$  and  $(n2, S'') = [[E2]](S)$   
 and  $b = \text{true}$  if  $n1 < n2$ , false otherwise

The remaining clauses are left as an exercise.

Programs contain expressions and Boolean expressions, so the semantics of programs also needs to be re-done. The semantics of skip and sequential composition remain unchanged, but the semantics of assignment (which contains an expression) becomes

for all States  $S$  and Variables  $V$  and Expressions  $E$   
 $[[V := E;]](S) = S[V \leftarrow n]$  where  $(n, S') = [[E]](S)$

The remaining clauses are left as an exercise.

## Key Points

- The language SImpL is different to the language of SImpL with post-increments (let's call this new language SImpL++)
- The semantics of SImpL++ is different to the semantics of SImpL: the denotation  $[[E]]$  of an expression  $E$  is a function from states to integer-state pairs; the denotation  $[[B]]$  of a Boolean expression  $B$  is a function from states to boolean-state pairs; the denotation  $[[P]]$  of a program  $P$  is still a function from states to states, but its definition has to take account of the differences in the semantics of expressions and Boolean expressions.

---

[Grant Malcolm](http://cgi.csc.liv.ac.uk/~grant/Teaching/COMP317/Lectures/107.html)