CSE 130: Programming Languages Sep. 27, 2000

Lecture Notes on Operational Semantics

Instructor: Daniele Micciancio

1 Operational semantics for a language of expressions

We have already studied how context free grammars can be used to formally specify the syntax of programming languages, and also to resolve certain semantic issues, like operator associativity and precedence, or the dangling-else problem. Of course, there is much more to be said about the semantics of computer programs. Today we start studying methods that can be used to formally specify the meaning of programming language constructs.

Consider a simple imperative programming language with a Pascal/C style "for" construct:

for v **from** E_1 **to** E_2 **do** S

with the intuitive meaning "execute statement S when variable v takes all values from E_1 to E_2 (inclusive). (Here E_1, E_2 are integer valued expressions.) Take for example the following program fragment:

```
y := 0;
for x from 1 to 5 do
    y := y+x;
```

The informal definition of the meaning of the "for" construct is enough to determine that at the end of the computation the value of y is 15.

Now consider the similar program:

```
y := 3;
for x from 1 to y do
    y := y+1;
```

What is the value of y at the end of the computation? There is not a unique answer to this question. If the expression $E_2 = y$ is evaluated only once and forall at the beginning of the for loop, the program terminate with $y = 6$, but if $E_2 = y$ is re-evaluated at each iteration the program does not even terminate! Both interpretations of the "for" instruction are reasonable and have actually been used in real programming languages (e.g. translate the above program in Pascal and C and try to run it. What is the result?). Deciding wheather the test expression should be evaluated only once,

or it should be re-evaluated at each iteration is indeed a design choice, and formal sematics can help in making these choices clear.

There are several methods to describe semantics. Today, we start with a method called *operational semantics*. The idea is to describe the meaning of a computer program showing how it is executed on an ideal computer.

Question: why to use an ideal computer instead of a real computer? There are several reasons:

- We want the specification of the semantics to be independent of the particular computer system that will be used to run the programs: e.g., a program written in Java should produce the same result no matter which computer is used to run it.
- Real computers can be extremely complex. Giving the semantics of a programming language in terms of a real computer would not add much to our understanding of the "meaning" of programs written in that language.

We first consider the simple language of expressions defined by the grammar:

$$\begin{aligned} E &::= T + E \mid T \\ T &::= F * T \mid F \\ F &::= \mathbf{num}(i) \mid (E) \end{aligned}$$

where $\mathbf{num}(i)$ is a token **num** with an attribute i representing an integer value. The computation associated to expression $5 * 7 + 8/(2 + 2)$ is

$$\begin{aligned} 5 * 7 + 8/(2 + 2) &\Rightarrow 35 + 8/(2 + 2) \\ &\Rightarrow 35 + 8/4 \\ &\Rightarrow 35 + 2 \\ &\Rightarrow 37 \end{aligned}$$

We can think this expression as being executed (evaluated) by an idealized computer whose internal state at any given instant is described by an expression. In general we can model a computational device using

- A set *States*: the set of all possible internal states of the the computer
- A binary relation $\Rightarrow$ over *States* that describe how the internal state of the computer changes from one step to the other
- An input function I that specify how the program P to be executed is mapped to the initial internal state $I(P)$ of the computer

- An output function O that specify the result of the computation.

We call the tuple $(States, \Rightarrow, I, O)$ a *transition system*. You can think the transition system as an infinite graph, with one node for every state, and arrows $s_1 \Rightarrow s_2$ between any pair of states (s_1, s_2) such that s_2 can be obtained from s_1 in one computational step. Given a program P we apply the input function I to obtain a node $s_1 = I(P)$ in the graph. We then follow the arrows $s_1 \Rightarrow s_2 \Rightarrow \cdots$ until we reach a node s_n from which the is no outgoing arrow. The output of the computation is $O(s_n)$.

In the previous example, *States* is the set of all expressions (or subexpressions) generated by the grammar, and we have an arrow $E_1 \Rightarrow E_2$ if expression E_2 can be obtained from expression E_1 performing a single arithmetic operation. The input function I is simply the identity function: given an expression E to be evaluated, the starting node is the one labeled with E. Following a path in the transition graph corresponds to performing arithmetic operations, one at a time. At some point, when all operations have been executed, we reach a node with a label $\mathbf{num}(i)$, and no other operation can be performed. The result of the computation is $O(\mathbf{num}(i)) = i$.

In general, for every node s there could be more than one outgoing arrow, and a choice should be made when selecting an execution path. For simplicity, in this course we will consider only *deterministic* transition systems, i.e., transition systems where for each node s there is at most one (other) node s' such that $s \Rightarrow s'$. If this is the case, given a program P (e.g., an expression), there is a unique sequence of states $s_1, s_2, \ldots$ such that

- $s_1 = I(P)$
- $s_i \Rightarrow s_{i+1}$ for all i.

The operational semantics of program P is defined as the sequence of states $s_1, s_2, \ldots$. (This sequence is also called the computation associated to program P). If the sequence is infinite, we say that P does not terminate, or loops. If the sequence $s_1, \ldots, s_n$ is finite, and cannot be further extended (i.e., there is no s such that $s_n \Rightarrow s$), then we say that the computation terminates and the final result is $O(s_n)$.

We still didn't specify how the relation $\Rightarrow$ is defined. In the case of the arithmetic expressions, we gave an informal definition saying that $E_1 \Rightarrow E_2$ if expression E_2 can be obtained from expression E_1 performing a single arithmetic operation. The relation can be formally specified by induction on the structure of the expressions E_1, E_2.

For example we have that

$$\mathbf{num}(i) + \mathbf{num}(j) \Rightarrow \mathbf{num}(i+j) \quad (1)$$

$$\mathbf{num}(i) * \mathbf{num}(j) \Rightarrow \mathbf{num}(i*j) \quad (2)$$

$$\mathbf{num}(i) - \mathbf{num}(j) \Rightarrow \mathbf{num}(i-j) \quad (3)$$

$$\mathbf{num}(i)/\mathbf{num}(j) \Rightarrow \mathbf{num}(i/j) \quad (4)$$
$$(\mathbf{num}(i)) \Rightarrow \mathbf{num}(i) \quad (5)$$

i.e., we can perform single arithmetic operations, or remove parenthesis from around a number in one computational step. (We disregard for the moment the fact that when dividing we should make sure that $j \neq 0$.) What if the expression is more complex? Say, the expression to be evaluated in $\mathbf{num}(3) + \mathbf{num}(7) * \mathbf{num}(5)$. This is an expression of the form $E + T$ where $E = \mathbf{num}(3)$ and $T = \mathbf{num}(7) * \mathbf{num}(5)$. The sum cannot be computed because one of the two operands (T) is not a simple number. What we should do in this case is to use the above transitions to get

$$\mathbf{num}(7) * \mathbf{num}(5) \Rightarrow \mathbf{num}(35)$$

and apply this transformation to a subexpression

$$\mathbf{num}(3) + \mathbf{num}(7) * \mathbf{num}(5) \Rightarrow \mathbf{num}(3) + \mathbf{num}(35).$$

In general we can say that if $T \Rightarrow T'$ then $\mathbf{num}(i) + T \Rightarrow \mathbf{num}(i) + T'$. This rule can be compactly represented as

$$\frac{T \Rightarrow T'}{\mathbf{num}(i) + T \Rightarrow \mathbf{num}(i) + T'}.$$

meaning that if the transition above the line is valid, then the transition below the line is also valid.

Other rules for the above expression language are

$$\frac{E \Rightarrow E'}{E + T \Rightarrow E' + T} \qquad \frac{E \Rightarrow E'}{E - T \Rightarrow E' - T}$$

$$\frac{T \Rightarrow T'}{T * F \Rightarrow T' * F} \qquad \frac{T \Rightarrow T'}{T/F \Rightarrow T' * F}$$

$$\frac{F \Rightarrow F'}{\mathbf{num}(i) * F \Rightarrow \mathbf{num}(i) * F'} \qquad \frac{F \Rightarrow F'}{\mathbf{num}(i)/F \Rightarrow \mathbf{num}(i)/F'}$$

$$\frac{E \Rightarrow E'}{(E) \Rightarrow (E')}$$

All these rules together can be used to transform any expression E generated by our grammar into an expression of the form $\mathbf{num}(i)$. In terms of the transition graph, this means that from any node labeled with an expression, we can reach (following the arrows) a node with label $\mathbf{num}(i)$. Notice that the transition system is deterministic, i.e., for any expression there is at most one rule that applies. Using this rules we can

give the following computation:

$$\begin{aligned}
\underline{5*7}+8/(2+2) &\Rightarrow 35+8/(\underline{2+2}) \\
&\Rightarrow 35+8/\underline{(4)} \\
&\Rightarrow 35+\underline{8/4} \\
&\Rightarrow \underline{35+2} \\
&\Rightarrow 37
\end{aligned}$$

where at each step we have underlined the subexpression to which an basic transformation has been applied. Notice that all underlined expressions matches the left hand side of one of the transformation (1),(2),(3),(4),(5).

We now consider boolean expressions, as generated by the grammar

$$\begin{aligned}
B &::= \mathbf{true} \mid \mathbf{false} \\
&\mid E = E \mid E > E \mid E < E
\end{aligned}$$

and extend the set *States* of the transition system to include also the "programs" generated by the grammar for B. The operational semantics of boolean expressions can be defined by the rules

$$\begin{aligned}
&\mathbf{num}(i) = \mathbf{num}(i) \Rightarrow \mathbf{true} && && (6)\\
&\mathbf{num}(i) = \mathbf{num}(j) \Rightarrow \mathbf{false} && (\text{if } i \neq j) && (7)\\
&\mathbf{num}(i) > \mathbf{num}(j) \Rightarrow \mathbf{true} && (\text{if } i > j) && (8)\\
&\mathbf{num}(i) > \mathbf{num}(j) \Rightarrow \mathbf{false} && (\text{if } i \leq j) && (9)\\
&\mathbf{num}(i) < \mathbf{num}(j) \Rightarrow \mathbf{true} && (\text{if } i < j) && (10)\\
&\mathbf{num}(i) < \mathbf{num}(j) \Rightarrow \mathbf{false} && (\text{if } i \geq j) && (11)
\end{aligned}$$

2 A simple imperative language

Now that we have defined the transition system for simple arithmetic and boolean expressions, let's move to an imperative programming language. Add the following production to the grammar

$$\begin{aligned}
S &::= \mathbf{var}(x) := E \mid [] \\
&\mid \mathbf{if}\ \ B\ \mathbf{then}\ S\ \mathbf{else}\ S \\
&\mid \mathbf{begin}\ \ L\ \mathbf{end} \\
&\mid \mathbf{while}\ B\ \mathbf{do}\ S \\
L &::= S \mid S; L
\end{aligned}$$

where the non-terminal symbols S and L represents statements and lists of statements respectively, and [] is the empty statements. Let's also augment the expressions

grammar with a production

$$E \ ::= \ \mathbf{var}(x)$$

to allow for expressions containing variables.

To define the operational semantics of this simple imperative language, it is not enough to extend the set *States* with the string generated by the grammars for S and L. In order to execute these programs we need to remember not only the program to be executed, but also the value of all variables used in the program. This mapping from variable names to values is called a "store", and is usually denoted with the letter σ (sigma). You can think of the store as a two column table, containing variable names in the left column and values (e.g., integer numbers in our example) in the right column. If a row contains variable x on the left and integer j on the right, than j is the value of variable x. It is convenient to represent stores as lists of pairs, instead of using tables, e.g., $\sigma = [x : 5, y : 6, z : 1]$ represent the store where x as value 5, y has value 6 and z has value 1. Notice: each variable name can appear at most once in the list. The store can also be represent as a function, e.g., $\sigma(x) = 5$, $\sigma(y) = 6$, etc. If a variable "w" is not defined, then $\sigma(w)$ outputs a special symbol $\perp$ that represents an undefined value or error.

So, we define *States* as the set of all pairs (P, σ) where P is a string generated by the grammar corresponding to a program or subprogram in our language, and σ is a store, i.e., σ is a function from a set *Names* of variable names, to a set *Values* of possible values for the variable. In our example *Values* is the set of the integer numbers plus the special symbol $\perp$ to denote undefined variables or error condition. (e.g., $\perp$ can also be used to define the result of a division by 0).

The transition system is a graph with a node corresponding to every pair (P, σ). Examples of transitions are

$$\begin{aligned} (\mathbf{if} \ \ \mathbf{true\ then}\ S_1\ \mathbf{else}\ S_2, \sigma) &\ \Rightarrow \ (S_1, \sigma) \\ (\mathbf{if} \ \ \mathbf{false\ then}\ S_1\ \mathbf{else}\ S_2, \sigma) &\ \Rightarrow \ (S_2, \sigma) \end{aligned}$$

i.e., if the test in a conditional is the constant true, execute the "then" statement, while if it is the constant false execute the "else" statement. Notice that in both cases the store σ does not change. If the test is a complex boolean expression, then first of all we have to evaluate the test:

$$\frac{(B, \sigma) \Rightarrow (B', \sigma)}{(\mathbf{if} \ \ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, \sigma) \Rightarrow (\mathbf{if} \ \ B'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, \sigma)}$$

All the transitions defined for the language of arithmetic and boolean expressions are still valid, if we add a "store" component to the state. In other words, if we defined rule $E \Rightarrow E'$ or $B \Rightarrow B'$ for the expression languages, then we define rules $(E, \sigma) \Rightarrow (E', \sigma)$ or $(B, \sigma) \Rightarrow (B', \sigma)$ for the new language with statements. We only

need a new transition for the expression introduced by the new rule $E ::= \mathbf{var}(x)$. The corresponding transition is

$$(\mathbf{var}(x), \sigma) \Rightarrow (\sigma(x), \sigma)$$

i.e., the result of evaluating expression $\mathbf{var}(x)$ when the store is σ, is the integer value $\sigma(x)$.

In all transitions that we have defined so far, the store never changes. The content of the store can be changed using the variable assignment operation:

$$(\mathbf{var}(x) := \mathbf{num}(j), \sigma) \quad \Rightarrow \quad ([], \sigma[j/x])$$

where $\sigma[j/x]$ is a notation for the store obtained from σ changing the value of x to j, i.e., $\sigma' = \sigma[j/x]$ corresponds to the function

$$\sigma'(y) = \begin{cases} j & \text{if "x" = "y"} \\ \sigma(y) & \text{otherwise} \end{cases}$$

(Remember that we use notation [] for the empty statement.)

We still need to define rules for the compound statement (begin ... end), and the while statement. Compound statements are easy:

$$(\mathbf{begin} \;\; S \;\; \mathbf{end}, \sigma) \Rightarrow (S, \sigma)$$

i.e., if the compound statement consists of a single statement, then we can remove the begin/end keywords. Other rules are

$$\frac{(S, \sigma) \Rightarrow (S', \sigma')}{(\mathbf{begin} \;\; S; L \;\; \mathbf{end}, \sigma) \Rightarrow (\mathbf{begin} \;\; S'; L \;\; \mathbf{end}, \sigma')}$$

$$(\mathbf{begin} \;\; []; L \;\; \mathbf{end}, \sigma) \Rightarrow (\mathbf{begin} \;\; L \;\; \mathbf{end}, \sigma)$$

i.e., if the compound statement contains more than one statement, then we should first execute the first statement. Once the first statement has been completely executed (i.e., it has been transformed to the empty statement []), then we should go on to the other statements in the list.

Finally we can use the semantic definition of the if/then/else and begin/end constructs do define the semantics of **while** as follows:

$$(\mathbf{while} \; B \; \mathbf{do} \; S, \sigma) \Rightarrow (\mathbf{if} \;\; B \; \mathbf{then \; begin} \;\; S; \mathbf{while} \; B \; \mathbf{do} \; S \mathbf{end \; else} \; [], \sigma)$$

For an example of use of this operational semantics to compute the meaning of a program in this language, see last section.

3 The FOR operation

We now show how the operational semantics method can be used to specify unambiguously the meaning of a new instruction. Extend our language with a FOR instruction:

$$S ::= \textbf{for var from } E \textbf{ to } E \textbf{ do} S$$

We have seen in the first section that there are several possible interpretation for this construct. In this section we give a possible semantics corresponding to the case where expressions are evaluated only once. The transition relation for the for instruction is defined as follows:

$$\frac{(E,\sigma) \Rightarrow (E',\sigma)}{(\textbf{for var}(x) \textbf{ from } E \textbf{ to } E_2 \textbf{ do } S,\sigma) \Rightarrow (\textbf{for var}(x) \textbf{ from } E' \textbf{ to } E_2 \textbf{ do } S,\sigma)}$$

$$\frac{(E,\sigma) \Rightarrow (E',\sigma)}{(\textbf{for var}(x) \textbf{ from num}(i) \textbf{ to } E \textbf{ do } S,\sigma) \Rightarrow (\textbf{for var}(x) \textbf{ from num}(i) \textbf{ to } E' \textbf{ do } S,\sigma)}$$

$$\begin{array}{l}(\textbf{for var}(x) \textbf{ from num}(i) \textbf{ to num}(j) \textbf{ do } S,\sigma) \\ \Rightarrow (\textbf{begin } \textbf{var}(x) := \textbf{num}(i); \textbf{while var}(x) < \textbf{num}(j) + 1 \textbf{ do} \\ \qquad\qquad \textbf{begin } S; \textbf{var}(x) := \textbf{var}(x) + 1 \textbf{ end end}, \sigma)\end{array}$$

The first rule says that we should first evaluate the first expression. The second rule says that once the first expression is evaluated, we should evaluate the second one. The third rule shows how the "for" instruction can be translated into a "while", once the expressions have been both evaluated.

Defining the other possible semantics for the "for" instructions if given as an exercise.

4 An example

We conclude with an example. Consider the program

```
begin  n := 0;
       s := 0;
       while n > 0  do
              begin  s := s+n;
                     n := n-1
              end
end
```

and let W be the program fragment

$$W = \textbf{while } n > 0 \textbf{ do begin } \; s := s + n; n := n - 1 \textbf{ end}$$

so that the entire program can be compactly represented as

$$P = \textbf{begin } \; n := 0; s := 0; W \textbf{ end}$$

The input function maps the program P to the initial state $I(P) = (P, \sigma)$, where $\sigma = [n : \bot, s : \bot]$ is the empty store, i.e., the store where all variables are undefined.

The operational semantics of P is:

$$\begin{aligned}
& (\textbf{begin } \; \underline{n := 3}; s := 0; W \textbf{ end}, [n : \bot, s : \bot]) \\
\Rightarrow \; & (\underline{\textbf{begin } \; []; s := 0; W \textbf{ end}}, [n : 3, s : \bot]) \\
\Rightarrow \; & (\textbf{begin } \; \underline{s := 0}; W \textbf{ end}, [n : 3, s : \bot]) \\
\Rightarrow \; & (\underline{\textbf{begin } \; []; W \textbf{ end}}, [n : 3, s : 0]) \\
\Rightarrow \; & (\underline{\textbf{begin } \; W \textbf{ end}}, [n : 3, s : 0]) \\
\Rightarrow \; & (W, [n : 3, s : 0]) = (\underline{\textbf{while } n > 0 \textbf{ do begin } \; s := s + n; n := n - 1 \textbf{ end}}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{if } \; \underline{n} > 0 \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } [], [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{if } \; \underline{3 > 0} \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } [], [n : 3, s : 0]) \\
\Rightarrow \; & (\underline{\textbf{if } \; \textbf{true} \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } []}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := \underline{s} + n; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := 0 + \underline{n}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := \underline{0 + 3}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; \underline{s := 3}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 3, s : 0]) \\
\Rightarrow \; & (\textbf{begin } \; \underline{\textbf{begin } \; []; n := n - 1 \textbf{ end}}; W \textbf{ end}, [n : 3, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \underline{\textbf{begin } \; n := n - 1 \textbf{ end}}; W \textbf{ end}, [n : 3, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; n := \underline{n} - 1; W \textbf{ end}, [n : 3, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; n := \underline{3 - 1}; W \textbf{ end}, [n : 3, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \underline{n := 2}; W \textbf{ end}, [n : 3, s : 3]) \\
\Rightarrow \; & (\underline{\textbf{begin } \; []; W \textbf{ end}}, [n : 2, s : 3]) \\
\Rightarrow \; & (\underline{\textbf{begin } \; W \textbf{ end}}, [n : 2, s : 3]) \\
\Rightarrow \; & (W, [n : 2, s : 3]) = (\underline{\textbf{while } n > 0 \textbf{ do begin } \; s := s + n; n := n - 1 \textbf{ end}}, [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{if } \; \underline{n} > 0 \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } [], [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{if } \; \underline{2 > 0} \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } [], [n : 2, s : 3]) \\
\Rightarrow \; & (\underline{\textbf{if } \; \textbf{true} \textbf{then begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end else } []}, [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := s + n; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := 3 + \underline{n}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; s := \underline{3 + 2}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 2, s : 3]) \\
\Rightarrow \; & (\textbf{begin } \; \textbf{begin } \; \underline{s := 5}; n := n - 1 \textbf{ end}; W \textbf{ end}, [n : 2, s : 3])
\end{aligned}$$

$\Rightarrow$ (**begin** **begin** $[]; n := n - 1$ **end**; W **end**, $[n : 2, s : 5]$)

$\Rightarrow$ (**begin** **begin** $n := n - 1$ **end**; W **end**, $[n : 2, s : 3]$)

$\Rightarrow$ (**begin** $n := \underline{n} - 1; W$ **end**, $[n : 2, s : 5]$)

$\Rightarrow$ (**begin** $n := \underline{2 - 1}; W$ **end**, $[n : 2, s : 5]$)

$\Rightarrow$ (**begin** $\underline{n := 1}; W$ **end**, $[n : 2, s : 5]$)

$\Rightarrow$ (**begin** $[]; W$ **end**, $[n : 1, s : 5]$)

$\Rightarrow$ (**begin** W **end**, $[n : 1, s : 5]$)

$\Rightarrow$ $(W, [n : 1, s : 5]) = ($**while** $n > 0$ **do begin** $s := s + n; n := n - 1$ **end**, $[n : 1, s : 5])$

$\Rightarrow$ (**if** $\underline{n} > 0$**then begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 1, s : 5])$

$\Rightarrow$ (**if** $\underline{1 > 0}$**then begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 1, s : 5])$

$\Rightarrow$ (**if** **truethen begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 1, s : 5])$

$\Rightarrow$ (**begin** **begin** $s := \underline{s} + n; n := n - 1$ **end**; W **end**, $[n : 1, s : 5])$

$\Rightarrow$ (**begin** **begin** $s := 5 + \underline{n}; n := n - 1$ **end**; W **end**, $[n : 1, s : 5])$

$\Rightarrow$ (**begin** **begin** $s := \underline{3 + 1}; n := n - 1$ **end**; W **end**, $[n : 1, s : 5])$

$\Rightarrow$ (**begin** **begin** $\underline{s := 6}; n := n - 1$ **end**; W **end**, $[n : 1, s : 5])$

$\Rightarrow$ (**begin** **begin** $[]; n := n - 1$ **end**; W **end**, $[n : 1, s : 6])$

$\Rightarrow$ (**begin** **begin** $n := n - 1$ **end**; W **end**, $[n : 1, s : 6])$

$\Rightarrow$ (**begin** $n := \underline{n} - 1; W$ **end**, $[n : 1, s : 6])$

$\Rightarrow$ (**begin** $n := \underline{1 - 1}; W$ **end**, $[n : 1, s : 6])$

$\Rightarrow$ (**begin** $\underline{n := 0}; W$ **end**, $[n : 1, s : 6])$

$\Rightarrow$ (**begin** $[]; W$ **end**, $[n : 0, s : 6])$

$\Rightarrow$ (**begin** W **end**, $[n : 0, s : 6])$

$\Rightarrow$ $(W, [n : 0, s : 6]) = ($**while** $n > 0$ **do begin** $s := s + n; n := n - 1$ **end**, $[n : 0, s : 6])$

$\Rightarrow$ (**if** $\underline{n} > 0$**then begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 0, s : 6])$

$\Rightarrow$ (**if** $\underline{0 > 0}$**then begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 0, s : 6])$

$\Rightarrow$ (**if** **falsethen begin** **begin** $s := s + n; n := n - 1$ **end**; W **end else** $[], [n : 0, s : 6])$

$\Rightarrow$ $([], [n : 0, s : 6])$

and the computation terminates with final store $[n : 0, s : 6]$. If the output function is $O(P, \sigma) = \sigma(s)$ (i.e., the value of s at the end of the computation), the final result is $O([], [n : 0, s : 6]) = 6$.

The above example show how, although the operational semantics has the advantage of formally and unambiguously specifying the meaning of a program, reasoning about a program using the operational semantics can be quite lengthy and cumbersome. In the next lecture we will start studying a different method do define the semantics of computer programs (called *axiomatic semantics*) that is more suited to prove properties about computer programs.