Program Syntax; Operational Semantics

CS 536: Science of Programming, Fall 2019

9/6 p.1

A. Why

- Our simple programming language is a model for the kind of constructs seen in actual languages.
- Step-by-step program evaluation involves a sequence of program / state snapshots. [9/1 typo]

B. Outcomes

At the end of today, you should be able to

- Describe he syntax of our simple deterministic programming language.
- Translate programs in our language to and from C / C++ / Java.
- Use operational semantics to describe step-by-step execution of programs in our language.

C. Our Simple Programming Language

- As mentioned before, we're going to use the simplest programming language we can get away with. This is because having fewer language constructs makes it easier to analyze how a language works (E.g., we'll ignore declarations.)
- **Notation:** We'll typically use e, e' for expressions, B for boolean expressions, S for statements.
- Our initial programming language has five kinds of statements. (We'll add more as we go along.)
- **No-op** statement
  - The no-op statement does nothing. Syntax: **`skip`**
- **Assignment** statement
  - **Syntax**: $v := e$ or $b[e_1][\ldots][e_n] := e$
  - For an n-dimensional array, n indexes must appear. (We can't slice arrays as in C.)
- **Sequence** statement
  - **Syntax**: S **`;`** S' (semicolon is a separator)
  - S' can be a sequence, so you can get longer sequences like S_1 **`;`** S_2 **`;`** S_3. Longer sequences get read left-to-right.
- **Conditional** statement
  - **Syntax 1**: **`if`** B **`then`** S_1 **`else`** S_2 **`fi`**
  - **Syntax 2**: **`if`** B **`then`** S_1 **`fi`** is an abbreviation for **`if`** B **`then`** S_1 **`else skip fi`**.
- **Iterative** statement
  - **Syntax**: **`while`** B **`do`** S_1 **`od`**
  - We'll omit **`for`** loop and **`do-while`** loops, since we can fake them using **`while`** loops.
- **Programs**: A program is just a statement (typically a sequence statement).

Example 1: A Sample Program:

- The program below calculates powers of 2. When it finishes, $(n < 0 \rightarrow y = 1) \wedge (n \geq 0 \rightarrow x = n \wedge y = 2^n)$.

```
if n < 0 then
        y := 1
else
        x := 0;
        y := 1;
        while x < n
        do
            x := x+1;
            y := y+y
        od
fi
```

- When we discuss the semantics and correctness of a program, we'll have to look at not only a whole program statement but also all its embedded sub-statements. ("All embedded sub-statements" here meaning the statement, its sub-statements, its sub-sub-statements, etc.) The program in Example 1 contains 10 embedded sub-statements. In no particular order:
  1. `y := 1` (in the true branch of the **`if-else`**)
  2. `x := 0`
  3. `y := 1` (in the false branch of the **`if-else`**)
  4. `x := x+1`
  5. `y := y+y`
  6. `x := x+1; y := y+y`
  7. **`while`** `x < n` **`do`** `x := x+1; y := y+y` **`od`**
  8. `x := 0; y := 1` (reading sequences left to right)
  9. `x := 0; y := 1;` **`while`** `x < n` **`do`** `x := x+1; y := y+y` **`od`**
  10. (The entire statement): **`if`** `n < 0` **`then`** `y := 1` **`else`** *S* **`fi`**, where *S* is statement 9 above.

D. Relationship to Actual Languages

- Converting between a typical language like C or C++ or Java and our programming language is pretty straightforward. The only real issue is that since our language doesn't include assignment expressions, they have to be rewritten as assignment statements.
- In C, C++, and Java,
  - As statements, `z++` and `++z` are identical.
  - As an expression, `++z` does `z=z+1` and yields the new value of `z`.
  - As an expression, `z++` does `z=z+1` but yields the old value of `z`.

Example 2

- The C statement: `x = a * ++z;` is equivalent to our `z := z+1; x := a * z`

Example 3

- The C statement: `x = a * z++;` is equivalent to our `temp := z; z := z+1; x := a * temp` [this how compilers treat this C code]. Another equivalent: `x := a * z; z := z+1`

Example 4

- The C loop statement: `while (--x >= 0) z*=x;` is equivalent to our

  `x := x-1; **while** x ≥ 0 **do** z := z*x; x := x-1 **od**`

- The decrement of `x` before the loop is for the first execution of `--x >= 0` (it has to be done before the **`while`** test). The decrement of `x` at the end of the loop body is for all the other executions of `--x >= 0`, where we jump to the top of the loop, decrement `x`, and test against 0.

Example 5

- The C loop: `while (x-- > 0) z*=x;`
- Our equivalent: **`while`** `x > 0` **`do`** `x := x-1; z := z*x` **`od;`** `x := x-1`
  - The decrement of `x` after the **`do`** is for all the `x-- > 0` tests that evaluate to true.
  - The decrement of `x` after the **`od`** is for the `x-- > 0` test that evaluates to false.

Example 6

- In C: `p = 1; for (x = 2 ; x < n ; ++x) p = p * x;` /* Calculates `p = n!` for `n ≥ 0` */
  - Our equivalent: `p := 1; x := 2; while x < n **do** p = p * x; x := x+1 **od**`
- In C, with a `for` loop, the increment / decrement clause gets done at the end of the loop body, as a statement, before jumping up to do the test, so this C program is equivalent to other C programs
  - `p = 1; x = 2; while (x < n) {p = p * x; ++x;)`
  - `p = 1; x = 2; while (x < n) {p = p * x; x++;)`
  - `p = 1; x = 2; while (x < n) p = p * x++;` // Can't use `++x` here

Notes on Translations

- There can be more than one possible translation, especially for complicated programs.
- For the loop translation examples above, if you really have trouble believing the given equivalences, the easiest way to convince yourself that they work is to write them up as C programs and run them. Either trace their execution or toss in a bunch of print statements to show you how the variables change.

E. Operational Semantics of Programs

- To model how our programs work, let's look at an **operational** semantics: We'll model execution as a sequence of "configurations" — snapshots of the program and memory state over time. The semantics rules describe how step-by-step execution of the program changes memory. When the program is complete, we have the final memory state of execution.
- **Definition**: A **configuration** $\langle S, \sigma \rangle$ is an ordered pair of a program and state.

- **Definition**: $\langle S,\sigma\rangle \to \langle S_1,\sigma_1\rangle$ means that **executing** S in state σ **for one step** yields $\langle S_1,\sigma_1\rangle$. We say S_1 is the **continuation** of S. The exact change indicated by the arrow depends on the program.
  - **Example 1**: $\langle \texttt{x := x+1; y := x}, \{\texttt{x}=5\}\rangle \to \langle \texttt{y := x}, \{\texttt{x}=6\}\rangle$ because execution of the first assignment changes the value of `x` and leaves us with one more assignment to execute.
- **Definition**: **Execution of *S* starting in state σ ends in** (**or terminates in**) **state** τ if there is a sequence of executions steps $\langle S,\sigma\rangle \to \ldots \to \langle E,\tau\rangle$ where E is the **empty program**, the program that indicates there is nothing left to execute. (Think of it as like an empty string.)
  - **Example 2**: $\langle \texttt{x := x+1; y := x; z := y+3}, \{\texttt{x}=5\}\rangle \to \langle \texttt{y := x; z := y+3}, \{\texttt{x}=6\}\rangle$ $\to \langle \texttt{z := y+3}, \{\texttt{x}=6, \texttt{y}=6\}\rangle \to \langle E, \{\texttt{x}=6, \texttt{y}=6, \texttt{z}=9\}\rangle$, so execution of our initial three-assignment program starting in the state where `x` is 5 ends with a state where `x` and `y` are 6 and `z` is 9.
- **Note**: "**Converges to**" is equivalent to "**Terminates in**". If we're not particularly interested in what the terminating state is, we may leave it out: "(Starting) in σ, *S converges*" or "*S terminates in* τ when run in σ".

F. Operational Semantics Rules

- There is an operational semantics rule for each kind of statement. The **`skip`** and assignment statements complete in one step; the sequence and conditional statements require multiple steps; the iterative statement may complete in any number of steps or loop forever.

Skip Statement

- The **`skip`** statement completes execution and does nothing to the state.
- $\langle \texttt{skip}, \sigma\rangle \to \langle E, \sigma\rangle$

Simple Assignment Statement

- To execute $v := e$ in state σ, update σ so that the value of v is the value of e in σ.
  - $\langle v := e, \sigma\rangle \to \langle E, \sigma[v \mapsto \sigma(e)]\rangle$.
- **Example 3**: $\langle \texttt{x := x+1}, \sigma\rangle = \langle E, \sigma[\texttt{x} \mapsto \sigma(\texttt{x+1})]\rangle = \langle E, \sigma[\texttt{x} \mapsto \sigma(\texttt{x})+1]\rangle$.
- **Example 4**: $\langle \texttt{x := 2*x*x + 5*x + 6}, \sigma\rangle = \langle E, \sigma[\texttt{x} \mapsto \alpha]\rangle$ where $\alpha = \sigma(\texttt{2*x*x + 5*x + 6}) = 2\beta^2 + 5\beta + 6$ where $\beta = \sigma(\texttt{x})$. For complicated expressions, it can be helpful to introduce new symbols like α and β, but it's also correct to write

  $\langle E, \sigma[\texttt{x} \mapsto \sigma(\texttt{2*x*x + 5*x + 6})]\rangle$

  $= \langle E, \sigma[\texttt{x} \mapsto 2\times\sigma(\texttt{x})\times\sigma(\texttt{x}) + 5\times\sigma(\texttt{x}) + 6)]\rangle$

  $= \langle E, \sigma[\texttt{x} \mapsto 2\sigma(\texttt{x})^2 + 5\,\sigma(\texttt{x}) + 6)]\rangle$

Array Element Assignment Statements

- To execute $\texttt{b}[e_1] := e$ in σ, evaluate the index e_1 to get some value α; update the function denoted by `b` at index α with the value of e.

- $\langle \texttt{b}[e_1] := e, \sigma \rangle \rightarrow \langle E, \sigma[\texttt{b}[\alpha] \mapsto \beta] \rangle$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$.

- **Example 5**: If $\sigma(\texttt{x}) = 8$, then $\langle \texttt{b}[\texttt{x+1}] := \texttt{x*5}, \sigma \rangle = \langle E, \sigma[\texttt{b}[\alpha] \mapsto \beta] \rangle = \langle E, \sigma[\texttt{b}[9] \mapsto 40] \rangle$ where $\alpha = \sigma(\texttt{x+1}) = \sigma(\texttt{x})+1 = 8+1 = 9$ and $\beta = \sigma(\texttt{x*5}) = \sigma(\texttt{x})*5 = 8*5 = 40$.
- The multi-dimensional versions of array assignment are similar; we'll omit them.

Conditional Statements

- For an **`if-then-else`** statement, our one step is to evaluate the test and jump to the beginning of the appropriate branch.
  - If $\sigma(B) = \text{T}$ then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$
  - If $\sigma(B) = \text{F}$ then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$
- For an **`if-then`** statement, the missing **`else`** clause defaults to **`else skip`**, so our continuation is never empty. (Though, granted, the execution of $\langle \textbf{skip}, \sigma \rangle$ is pretty trivial.)
  - If $\sigma(B) = \text{T}$ then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$
  - If $\sigma(B) = \text{F}$ then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ fi}, \sigma \rangle = \langle \textbf{if } B \textbf{ then } S_1 \textbf{ else skip fi}, \sigma \rangle \rightarrow \langle \textbf{skip}, \sigma \rangle$
- **Example 6**: Let $S \equiv$ **`if`** `x>0` **`then`** `y:=0` **`fi`**. (Don't forget the implicit **`else skip`** here.)
  - Then $\langle S, \sigma[\texttt{x} \mapsto 5] \rangle \rightarrow \langle \texttt{y} := 0, \sigma[\texttt{x} \mapsto 5] \rangle \rightarrow \langle E, \sigma[\texttt{x} \mapsto 5][\texttt{y} \mapsto 0] \rangle$.
    - (The first arrow is for the **`if-then`**; the second is for the assignment to `y`.)
  - Similarly, $\langle S, \sigma[\texttt{x} \mapsto -1] \rangle \rightarrow \langle \textbf{skip}, \sigma[\texttt{x} \mapsto -1] \rangle \rightarrow \langle E, \sigma[\texttt{x} \mapsto -1] \rangle$.
    - (The first arrow is for the **`if-else`**; the second is for the **`skip`**.)

Iterative Statements

- For a **`while`** statement, our one step is to evaluate the test and jump either to the end of the loop or to the beginning of the loop body. (After the body we'll continue by jumping to the top of the loop.) Let $W \equiv \textbf{while } B \textbf{ do } S \textbf{ od}$; then
  - If $\sigma(B) = \text{F}$, then $\langle W, \sigma \rangle \rightarrow \langle E, \sigma \rangle$
  - If $\sigma(B) = \text{T}$, then $\langle W, \sigma \rangle \rightarrow \langle S \textbf{ ; } W, \sigma \rangle$
  - This last case is the only operational semantics rule that produces a continuation that's textually larger than the starting statement (which is why we can get infinite loops).
- We'll look at a detailed example of a **`while`** loop in a bit.

Sequence Statements

- To execute a sequence $S_1 \textbf{ ; } S_2$, for one step, we execute S_1 for one step. This one step may or may not complete all of S_1. If it does, then we continue by executing S_2. If one step of execution takes S_1 to some statement T_1, then we continue by executing $T_1 \textbf{ ; } S_2$.
  - If $\langle S_1, \sigma \rangle \rightarrow \langle E, \sigma_1 \rangle$ then $\langle S_1 \textbf{ ; } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma_1 \rangle$
  - If $\langle S_1, \sigma \rangle \rightarrow \langle T_1, \sigma_1 \rangle$ then $\langle S_1 \textbf{ ; } S_2, \sigma \rangle \rightarrow \langle T_1 \textbf{ ; } S_2, \sigma_1 \rangle$

- The rule for executing sequences is somewhat different from the other execution rules because it is a "rule of inference" — if some program executes this way, then our program will execute that way. All the other execution rules are axiomatic — they do not involve a recursive use of the "→" relation.
- **Example 7**: Since $\langle \mathtt{x} := \mathtt{y}, \sigma \rangle \to \langle E, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{y})] \rangle$, we know
  - $\langle \mathtt{x} := \mathtt{y}\ \mathtt{;}\ \mathtt{y} := 2, \sigma \rangle \to \langle \mathtt{y} := 2, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{y})] \rangle$
  - (Note execution isn't complete; we'd need to continue with … $\to \langle E, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{y})][\mathtt{y} \mapsto 2] \rangle$.)

G. Examples of Loops

- Let's look at a couple of examples of loop execution to see how their semantics can be worked out. As part of that, we'll find that it involves writing a fair amount of formal manipulation using our notations. We'll do something about that in the following section.
- **Example 8**: Let $W \equiv$ **while** $\mathtt{x} \geq 0$ **do** $\mathtt{x} := \mathtt{x}-1$ **od**, then (in lots of detail)

$\langle W, \sigma[\mathtt{x} \mapsto 1] \rangle$

$\to \langle \mathtt{x} := \mathtt{x}-1\ \mathtt{;}\ W, \sigma[\mathtt{x} \mapsto 1] \rangle$	(Since $\sigma[\mathtt{x} \mapsto 1](\mathtt{x} \geq 0) = \mathrm{T}$)
$\to \langle W, \sigma[\mathtt{x} \mapsto 0] \rangle$	(Evaluate assignment of x) $\quad \sigma[\mathrm{x} \mapsto 1][\mathrm{x} \mapsto 0] = \sigma[\mathrm{x} \mapsto 0]$
$\to \langle \mathtt{x} := \mathtt{x}-1\ \mathtt{;}\ W, \sigma[\mathtt{x} \mapsto 0] \rangle$	(Since $\sigma[\mathtt{x} \mapsto 0](\mathtt{x} \geq 0) = \mathrm{T}$)
$\to \langle W, \sigma[\mathtt{x} \mapsto -1] \rangle$	(Evaluate assignment of x)
$\to \langle E, \sigma[\mathtt{x} \mapsto -1] \rangle$	(Since $\sigma[\mathtt{x} \mapsto -1](\mathtt{x} \geq 0) = \mathrm{F}$)

- **Example 9**: Let $S \equiv \mathtt{s} := 0\ \mathtt{;}\ \mathtt{i} := 0\ \mathtt{;}\ W$, where $W \equiv$ **while** $\mathtt{i} < \mathtt{n}$ **do** S_1 **od**, where $S_1 \equiv \mathtt{s} := \mathtt{s}+\mathtt{i}+1\ \mathtt{;}\ \mathtt{i} := \mathtt{i}+1$. Let $\sigma(\mathrm{n}) = 2$, then (in lots of detail)

$\langle S, \sigma \rangle = \langle \mathtt{s} := 0\ \mathtt{;}\ \mathtt{i} := 0\ \mathtt{;}\ W, \sigma \rangle$

$\to \langle \mathtt{i} := 0\ \mathtt{;}\ W, \sigma[\mathtt{s} \mapsto 0] \rangle$	(Evaluate first statement of conditional)
$\to \langle W, \sigma_0 \rangle$	(Where $\sigma_0 = \sigma[\mathtt{s} \mapsto 0][\mathtt{i} \mapsto 0]$)
$\to \langle S_1\ \mathtt{;}\ W, \sigma_0 \rangle$	(Since $\sigma_0(\mathtt{i} < \mathtt{n}) = \mathrm{T}$)
$= \langle \mathtt{s} := \mathtt{s}+\mathtt{i}+1\ \mathtt{;}\ \mathtt{i} := \mathtt{i}+1\ \mathtt{;}\ W, \sigma_0 \rangle$	(By defn of S_1 — note this step is "=", not "→")
$\to \langle \mathtt{i} := \mathtt{i}+1\ \mathtt{;}\ W, \sigma_0[\mathtt{s} \mapsto 1] \rangle$	(Evaluate assignment of s)
$\to \langle W, \sigma_1 \rangle$	(Let $\sigma_1 = \sigma_0[\mathtt{s} \mapsto 1][\mathtt{i} \mapsto 1]$ and evaluate asgt of i)
$\to \langle S_1\ \mathtt{;}\ W, \sigma_1 \rangle$	(Since $\sigma_1(\mathtt{i} < \mathtt{n}) = \mathrm{T}$)
$\to \langle \mathtt{i} := \mathtt{i}+1\ \mathtt{;}\ W, \sigma_1[\mathtt{s} \mapsto 3] \rangle$	(By defn of S_1)
$\to \langle W, \sigma_2 \rangle$	(Where $\sigma_2 = \sigma_1[\mathtt{s} \mapsto 3][\mathtt{i} \mapsto 2]$)
$\to \langle E, \sigma_2 \rangle$	(Since $\sigma_2(\mathtt{i} < \mathtt{n}) = \mathrm{F}$)

H. Using Multi-Step Execution to Abbreviate Executions

- With long executions, we often summarize multiple steps of execution to concentrate on the most interesting configurations.
- **Definition**: We say $\langle S_0, \sigma_0\rangle$ **evaluates to** $\langle S_n, \sigma_n\rangle$ in ***n* steps** and write $\langle S_0, \sigma_0\rangle \to^n \langle S_n, \sigma_n\rangle$ if there are $n+1$ configurations $\langle S_0, \sigma_0\rangle$, ..., $\langle S_{n-1}, \sigma_{n-1}\rangle$ such that $\langle S_0, \sigma_0\rangle \to \langle S_1, \sigma_1\rangle \to \ldots \langle S_{n-1}, \sigma_{n-1}\rangle \to \langle S_n, \sigma_n\rangle$. For $n = 0$ we define $\langle S_0, \sigma_0\rangle \to^0 \langle S_0, \sigma_0\rangle$.
- **Definition**: We say $\langle S_0, \sigma_0\rangle$ **evaluates to** $\langle T, \tau\rangle$ and write $\langle S_0, \sigma_0\rangle \to^* \langle T, \tau\rangle$ if $\langle S_0, \sigma_0\rangle \to^n \langle T, \tau\rangle$ for some n.
  - An equivalent way to say all of this is that $\to^n$ is the n-fold composition of $\to$ and $\to^*$ is the reflexive transitive closure of $\to$.
- **Definition**: Execution of S starting in σ **terminates in** (**the final state**) τ if $\langle S, \sigma\rangle \to^* \langle E, \tau\rangle$. Execution of S starting in σ **terminates** if it terminates in some τ.
- **Example 10**: Let S be `s := 0 ; i := n+1 ;` W, where $W \equiv$ **`while`** `i > 0` **`do`** S_1 **`od`** and $S_1 \equiv$ `i := i-1; s := s+i`. Since we're going to execute the body S multiple times, it will be helpful to look at a generic execution of S in an arbitrary state $\tau[\texttt{i} \mapsto \alpha][\texttt{s} \mapsto \beta]$. We find

$$\begin{aligned}\langle S_1, \tau[\texttt{i} \mapsto \alpha][\texttt{s} \mapsto \beta]\rangle &= \langle \texttt{i} := \texttt{i}-1; \texttt{s} := \texttt{s}+\texttt{i}, \tau[\texttt{i} \mapsto \alpha][\texttt{s} \mapsto \beta]\rangle \\ &\to \langle \texttt{s} := \texttt{s}+\texttt{i}, \tau[\texttt{s} \mapsto \beta][\texttt{i} \mapsto \alpha-1]\rangle \\ &\to \langle E, \tau[\texttt{i} \mapsto \alpha-1][\texttt{s} \mapsto \beta+\alpha-1]\rangle\end{aligned}$$

  - Combining the two steps gives us $\langle S_1, \tau[\texttt{i} \mapsto \alpha][\texttt{s} \mapsto \beta]\rangle \to^2 \langle E, \tau[\texttt{i} \mapsto \alpha-1][\texttt{s} \mapsto \beta+\alpha-1]\rangle$. Adding the test at the top of the loop (for when $\alpha > 0$) lets us give behavior of an arbitrary loop iteration:

$$\langle W, \tau[\texttt{i} \mapsto \alpha][\texttt{s} \mapsto \beta]\rangle \to^3 \langle W, \tau[\texttt{i} \mapsto \alpha-1][\texttt{s} \mapsto \beta+\alpha-1]\rangle$$

  - Let's evaluate S when n is 2; if $\sigma(\texttt{n}) = 2$, then

$$\begin{aligned}\langle S, \sigma\rangle &= \langle \texttt{s} := 0 \texttt{ ; i} := \texttt{n+1 ; } W, \sigma\rangle \\ &\to^2 \langle W, \sigma[\texttt{s} \mapsto 0][\texttt{i} \mapsto 3]\rangle && \text{(After loop initialization)} \\ &\to^3 \langle W, \sigma[\texttt{i} \mapsto 2][\texttt{s} \mapsto 2]\rangle && \text{(After one iteration of the loop)} \\ &\to^3 \langle W, \sigma[\texttt{i} \mapsto 1][\texttt{s} \mapsto 3]\rangle && \text{(After two iterations)} \\ &\to^3 \langle W, \sigma[\texttt{i} \mapsto 0][\texttt{s} \mapsto 3]\rangle && \text{(After three iterations)} \\ &\to \langle E, \sigma[\texttt{i} \mapsto 0][\texttt{s} \mapsto 3]\rangle && \text{(And now we stop, since i > 0 is false)}\end{aligned}$$

Program Syntax; Operational Semantics

CS 536: Science of Programming, Fall 2019

A. Why

- Our simple programming language is a model for the kind of constructs seen in actual languages.
- Step-by-step program evaluation can be described using a sequence of program / state snapshots.

B. Outcomes

At the end of today, you should be able to

- Read and write simple programs in our programming language.
- Translate simple programs in our language to and from C / C++ / Java.
- Describe the step-by-step execution of a program in our language by giving its operational semantics.

C. Problems

Part I: Program Syntax

1. In our simple language, **`if`** `x < 0` **`then`** `x := 0` **`fi`** is (syntactically) equivalent to what other statement?
2. How are **`if`** B **`then`** S_1 **`else`** S_2 **`fi`** and **`if`** B **`then`** e_1 **`else`** e_2 **`fi`** different?

For Questions 3 – 8, translate the given C / C++ / Java program fragments into our simple programming language.

3. `++x; if (x < y) { x = y = y+1; }`
4. `y = z * ++x; z = z+x;`
5. `y = z * x++; z = z+x;`
6. `x = z = 0; while (x++ < n) z = z+x;`
7. `z = 1; for (x = n ; x >= 1 ; --x) z = z * x;`
8. `x = 0; while (x++ <= n) { y = (++x)*y;}`

Part II: Operational Semantics

9. Evaluate each of the following configurations to completion. If there are multiple steps, show each step individually.
   a. $\langle \texttt{x := x+1}, \{\texttt{x} = 5\} \rangle$
   b. $\langle \texttt{y := 2*x}, \{\texttt{x} = 6\} \rangle$
   c. $\langle \texttt{x := x+1}, \sigma \rangle$ (Your answer will be symbolic — you'll need to include $\sigma(\texttt{x})$.)
   d. $\langle \texttt{x := x+1; y := 2*x}, \{\texttt{x} = 5\} \rangle$

10. Let $S \equiv$ **if** x > 0 **then** x := x+1 **else** y := 2*x **fi**.
    a. Let $\sigma(\mathbf{x}) = 8$, evaluate $\langle S, \sigma \rangle$ to completion, showing the individual steps. Give the final state.
    b. Repeat, if $\sigma(\mathbf{x}) = 0$.
    c. Repeat, if we don't know what $\sigma(\mathbf{x})$ is. (Your answer will be symbolic.)

11. Let $S \equiv$ **if** x > 0 **then** x := x/z **fi**. Evaluate S (starting) in σ, for each the σ below:
    a. $\sigma = \{\mathbf{x} = 8, \mathbf{z} = 3\}$ (and don't forget, integer division truncates)
    b. $\sigma = \{\mathbf{x} = -2, \mathbf{z} = 3\}$

12. Let $W \equiv$ **while** x < 3 **do** S **od** where $S \equiv$ x := x+1; y := y*x.
    a. Show what evaluation of the body S in an arbitrary state τ does.
    b. Use your answer from part a to evaluate W in σ where $\sigma \vDash \mathbf{x} = 4 \wedge \mathbf{y} = 1$.
    c. Repeat part b where $\sigma \vDash \mathbf{x} = 1 \wedge \mathbf{y} = 1$.

CS 536: Solution to Activity 5 (Program Syntax; Operational Semantics)

Part I: Syntax

1. `if x<0 then x:=0 else skip fi`

2. **if** B **then** S_1 **else** S_2 **fi** is a statement; its evaluation can change the state.
   if B **then** e_1 **else** e_2 **fi** is an expression; its evaluation produces a value.

3. `x:=x+1; if x<y then y:=y+1; x:=y fi`

4. `x:=x+1; y:=z*x; z:=z+x`

5. `y:=z*x; x:=x+1; z:=z+x`

6. `z:=0; x:=z; while x < n do x:=x+1; z:=z+x od; x:=x+1`

7. `z:=1; x:=n; while x >= 1 do z:=z*x; x:=x-1 od`

8. In the solution below, the increment of x after the **od** is for the x++ of the test that breaks out of the loop. For the body of the loop, the first increment of x is for the x++ in x++ <= n after testing x <= n. The immediately following increment of x is for the ++x in y = (++x)*y because the increment occurs before calculating y = x*y. You could certainly combine the two x := x+1 to just one x := x+2.

   `x := 0; while x<= n do x := x+1; x := x+1; y := x*y od; x := x+1`

Part II: Operational Semantics

9. (Calculate meanings of programs)

   a. $\langle x := x+1, \{x=5\}\rangle \to \langle E, \tau\rangle$ where $\tau = \{x=5\}[x \mapsto \{x=5\}(x+1)]$
   $= \{x=5\}[x \mapsto 6] = \{x=6\}$.

   b. $\langle y := 2*x, \{x=6\}\rangle \to \langle E, \tau\rangle$ where $\tau = \{x=6\}[y \mapsto \{x=6\}(2*x)]$
   $= \{x=6\}[y \mapsto 12] = \{x=6, y=12\}$

   c. $\langle x := x+1, \sigma\rangle \to \langle E, \sigma[x \mapsto \sigma(x+1)]\rangle = \langle E, \sigma[x \mapsto \sigma(x)+1]\rangle$

   d. $\langle x := x+1; y := 2*x, \{x=5\}\rangle$
   $\to \langle y := 2*x, \{x=5\}[x \mapsto \alpha]\rangle$ where $\alpha = \{x=5\}(x+1) = 6$
   $= \langle y := 2*x, \{x=5\}[x \mapsto 6]\rangle$
   $= \langle y := 2*x, \{x=6\}\rangle$

$\rightarrow \langle E, \{\mathtt{x}=6\}[\mathtt{y} \mapsto \beta]\rangle$ where $\beta = \{\mathtt{x}=6\}(2\mathtt{*x}) = 12$

$= \langle E, \{\mathtt{x}=6, \mathtt{y}=12\}\rangle$

10. Let $S \equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x}:=\mathtt{x}+1$ **else** $\mathtt{y}:=2\mathtt{*x}$ **fi**.

    a. If $\sigma(\mathtt{x}) = 8$, then $\sigma(\mathtt{x}>0) = \mathrm{T}$,

    so $\langle S, \sigma\rangle \rightarrow \langle \mathtt{x}:=\mathtt{x}+1, \sigma\rangle \rightarrow \langle E, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}+1)]\rangle = \langle E, \sigma[\mathtt{x} \mapsto 9]\rangle$.

    b. If $\sigma(\mathtt{x}) = 0$, then $\sigma(\mathtt{x}>0) = \mathrm{F}$,

    so $\langle S, \sigma\rangle \rightarrow \langle \mathtt{y}:=2\mathtt{*x}, \sigma\rangle \rightarrow \langle E, \sigma[\mathtt{y} \mapsto \sigma(2\mathtt{*x})]\rangle = \langle E, \sigma[\mathtt{y} \mapsto 0]\rangle$

    c. If $\sigma(\mathtt{x})>0$ then $\langle S, \sigma\rangle \rightarrow \langle \mathtt{x}:=\mathtt{x}+1, \sigma\rangle = \langle E, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x})+1]\rangle$.

    If $\sigma(\mathtt{x}) \le 0$ then $\langle S, \sigma\rangle \rightarrow \langle \mathtt{y}:=2\mathtt{*x}, \sigma\rangle = \langle E, \sigma[\mathtt{y} \mapsto 2\times\sigma(\mathtt{x})]\rangle$.

11. Let $S \equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x}:=\mathtt{x/z}$ **fi** $\equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x}:=\mathtt{x/z}$ **else skip fi**

    a. If $\sigma=\{\mathtt{x}=8, \mathtt{z}=3\}$, then $\sigma(\mathtt{x}>0) = \mathrm{T}$ so $\langle S,\sigma\rangle \rightarrow \langle \mathtt{x}:=\mathtt{x/z}, \sigma\rangle \rightarrow \langle E, \sigma[\mathtt{x} \mapsto \alpha]\rangle$ where $\alpha = \sigma(\mathtt{x/z}) = \sigma[\mathtt{x} \mapsto 8/3] = \sigma[\mathtt{x} \mapsto 2]$, since integer division truncates.

    b. If $\sigma=\{\mathtt{x}=-2, \mathtt{z}=3\}$ then $\sigma(\mathtt{x}>0) = \mathrm{F}$, so $\langle S,\sigma\rangle \rightarrow \langle$**skip**$,\sigma\rangle \rightarrow \langle E, \sigma\rangle$.

12. Let $W \equiv$ **while** $\mathtt{x}<3$ **do** S **od** where $S \equiv \mathtt{x}:=\mathtt{x}+1\mathtt{;}\ \mathtt{y}:=\mathtt{y}\mathtt{*}\mathtt{x}$.

    a. For arbitrary τ, $\langle S,\tau\rangle \rightarrow \langle \mathtt{x}:=\mathtt{x}+1\mathtt{;}\ \mathtt{y}:=\mathtt{y}\mathtt{*}\mathtt{x},\tau\rangle \rightarrow \langle \mathtt{y}:=\mathtt{y}\mathtt{*}\mathtt{x},\tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1]\rangle$

    $\rightarrow \langle E, \tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1][\mathtt{y} \mapsto \alpha]\rangle$ where $\alpha=\tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1](\mathtt{y}\mathtt{*}\mathtt{x}) = \tau(\mathtt{y})\times(\tau(\mathtt{x})+1)$.

    b. If $\sigma \vDash \mathtt{x}=4 \wedge \mathtt{y}=1$, then $\sigma(\mathtt{x}<3) = \mathrm{F}$ so $\langle W,\sigma\rangle \rightarrow \langle E,\sigma\rangle$.

c. If $\sigma \vDash \mathtt{x}=1 \wedge \mathtt{y}=1$, then $\sigma(\mathtt{x}<3) = \mathrm{T}$ so we have at least one iteration to do.

Let $\sigma_0 = \sigma$, let $\sigma_1 = \sigma_0(\mathtt{y})\times(\sigma_0(\mathtt{x})+1)$, and let $\sigma_2 = \sigma_1(\mathtt{y})\times(\sigma_1(\mathtt{x})+1)$. Then

$\sigma_0 = \sigma[\mathtt{x} \mapsto 1][\mathtt{y} \mapsto 1]$

$\sigma_1 = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x})+1][\mathtt{y} \mapsto \sigma_0(\mathtt{y})\times(\sigma_0(\mathtt{x})+1)] = \sigma[\mathtt{x} \mapsto 2][\mathtt{y} \mapsto 2]$

$\sigma_2 = \sigma_1[\mathtt{x} \mapsto 2+1][\mathtt{y} \mapsto 2\times(2+1)] = \sigma[\mathtt{x} \mapsto 3][\mathtt{y} \mapsto 6]$

Since σ_0 and $\sigma_1 \vDash \mathtt{x} < 3$ but $\sigma_2 \vDash \mathtt{x} \ge 3$, we have

$\langle W, \sigma\rangle \rightarrow \langle S\mathtt{;}\ W, \sigma_0\rangle \rightarrow^* \langle W, \sigma_1\rangle = \langle S\mathtt{;}\ W, \sigma_1\rangle \rightarrow^* \langle W, \sigma_2\rangle \rightarrow \langle E, \sigma_2\rangle$, so σ_2 is the final state.

