

Syntax and Semantics of Programs

CS 536 Lecture 5, Wed Jan 25, 2012

A. Why

- Our simple programming language is a model for the kind of constructs seen in actual languages.

B. Outcomes

At the end of today, you should

- Know the basic syntax of our simple deterministic programming language.
- Know the basic semantics of our programming language.

C. Our Simple Programming Language

- As mentioned before, we're going to use the simplest programming language we can get away with, at first. This is because to analyze how a language works, it's nicest to have very few kinds of constructions in the language. (E.g., we'll ignore declarations.)
- Our initial programming language has five kinds of statements.
 1. ***No-op*** statement
 - The no-op statement does nothing. Syntax: **skip**
 2. ***Assignment*** statement
 - Syntax $v := e$ where v is a variable and e is an expression.
 - v can be a simple variable or an array indexing expression.
 3. ***Sequence*** statement
 - Syntax: $S_1 ; S_2$ where here and below, S_1 and S_2 are statements. Semicolon is a separator.
 - Longer sequences are read left-to-right: $S_1 ; S_2 ; S_3$, for example.
 4. ***Conditional*** statement
 - Syntax 1: **if B then S_1 else S_2 fi** where B is a boolean expression.
 - Syntax 2: **if B then S_1 fi** is an abbreviation for **if B then S_1 else skip fi**.
 5. ***Iterative*** statement
 - Syntax: **while B do S_1 od** where B is a boolean expression and S_1 is a statement.
 - We have no **for** loop or **do-while** loop but can fake them using **while** loops.
- A program is just a statement (typically a sequence statement).

- **Sample Program:**

```

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

```

- The program contains 10 sub-statements (including itself):
 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` (probably should read sequences left to right)
 9. `x := 0; y := 1; while x < n do x := x+1; y := y+y od`
 10. `if n < 0 then y := 1 else S fi`, where S is statement 9 above.
- The program calculates powers of 2:
 - When it finishes, $(n < 0 \rightarrow y = 1) \wedge (n \geq 0 \rightarrow x = n \wedge y = 2^n)$ is satisfied.

D. Relationship to Actual languages

- Converting between a typical language like C or C++ or Java and our programming language is pretty straightforward.
- Since our language doesn't include assignment expressions, they have to be rewritten as assignment statements.

- **Example 1**

- In C: `while (--x >= 0) z*=x;`
- Our equivalent:

```

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`.

- **Example 2**

- In C: `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 at the beginning of the loop body is for all the $x-- \geq 0$ tests that evaluate to true.
- The decrement of x after the loop is for the $x-- \geq 0$ test that evaluates to false.
- Someone asked a question about timing of updates in C:
 - $x = a*b*z++$; is equivalent to $temp = z$; $z = z+1$; $x = a*b*temp$;
 - $++z$ is equivalent to $z=z+1$ (as an expression)

E. The Meaning of Programs as State Transformers

- **Programs are state transformers:** We evaluate programs in order to transform the memory state. Here are the intuitive descriptions:
 - **skip** does the null transformation. (It doesn't actually modify the state.)
 - **Assignment 1:** $v := e$ does a state update on v ; if σ is the current state, the assignment changes the state to $\sigma[v \mapsto \sigma(e)]$.
 - **Example:** If we execute $x := 0$ in state σ , the resulting state is $\sigma[x \mapsto 0]$.
 - **Example:** If $\sigma(z) = 3$, then if we execute $z := z+1$ in state σ , the resulting state is $\sigma[z \mapsto 4]$.
 - **Example:** If we execute $y := x+1$ in state $\sigma[x \mapsto 0]$, the result is $\sigma[x \mapsto 0][y \mapsto 1]$, since the value of $x+1$ (i.e., $\sigma[x \mapsto 0](x+1)$ equals 1).
 - **Assignment 2:** $b[e_1] := e_2$ does a state update on the array value of b : If σ is the current state and α and β are the values of e_1 and e_2 respectively in σ , then the assignment $b[e_1] := e_2$ changes σ to $\sigma[b[\alpha] \mapsto \beta]$. (Or equivalently, it changes the state to $\sigma[b \mapsto \sigma(b)[\alpha \mapsto \beta]]$.)
 - **Sequences** do a series of transformations, one after another.
 - **Example:** If we execute $x := 0$; $y := x+1$ in state σ , we are in state $\sigma[x \mapsto 0]$ after the first statement and we end up in state $\sigma[x \mapsto 0][y \mapsto 1]$.
 - **Conditional and iterative statements** are similar: Their tests don't alter the state; they alter control. Their bodies (the true/false branches and loop bodies) can alter the state.
 - **Example:** If $\sigma(x)$ is less than 0, then if we execute


```
if  $x < 0$  then  $x := 0$ ;  $y := x+1$  else skip fi
```

 in state σ , we execute $x := 0$; $y := x+1$ in state σ and end in state $\sigma[x \mapsto 0][y \mapsto 1]$. On the other hand, if $\sigma(x)$ is nonnegative, then we execute **skip** in state σ and finish in state σ .

Formal Definition of $M(S, \sigma)$, the Meaning of a Statement in a State

- **Definition:** If we execute statement S starting in state σ and end up in state τ , we'll say that τ is **the meaning of S in σ** . **Notation:** $M(S, \sigma) = \tau$.
 - If S goes into an infinite loop, then $M(S, \sigma)$ is undefined.
- The definition of $M(S, \sigma)$ is recursive in the structure of S (we use the meanings of the component sub-statements of S to get the meaning of S).
- The base cases are the statements without sub-statements:
- **Skip:** The skip statement does nothing to the state.
 - $M(\text{skip}, \sigma) = \sigma$.
- **Assignment 1:** For simple variables, update the state so that the left-hand-side variable is bound to the value of the right-hand-side expression.
 - $M(v := e, \sigma) = \sigma[v \mapsto \sigma(e)]$
 - **Example:** $M(x := x+1, \sigma) = \sigma[x \mapsto \sigma(x+1)] = \sigma[x \mapsto \sigma(x)+1]$.
 - For complicated expressions, it can be helpful to introduce new symbols.
 - **Example:** $M(x := 2*x*x + 5*x + 6, \sigma) = \sigma[x \mapsto \alpha]$ where $\alpha = \sigma(2*x*x + 5*x + 6) = 2\beta^2 + 5\beta + 6$ where $\beta = \sigma(x)$.
- **Assignment 2:** For array updates, update the state so that the array element indicated on the left-hand-side of the assignment has the value of the right-hand-side expression:
 - $M(b[e_1] := e_2, \sigma) = \sigma[b[\alpha] \mapsto \beta] = \sigma[b \mapsto \sigma(b)[\alpha \mapsto \beta]]$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e_2)$.
 - **Example:** If $\sigma(x) = 8$, then $M(b[x+1] := x*5, \sigma) = \sigma[b[9] \mapsto 40] = \sigma[b \mapsto \sigma(b)[9 \mapsto 40]]$.
- The recursive cases of $M(S, \sigma)$ are sequence, conditional, and iterative statements:
- **Sequence:** Evaluate the first statement in the current state and then execute the second statement in the state that results.
 - $M(S_1 ; S_2, \sigma) = M(S_2, M(S_1, \sigma))$ [note the reversal in subscripts]
 - **Example:**

$$\begin{aligned} &M(x := x+1 ; y := y*x, \sigma) \\ &= M(y := y*x, M(x := x+1, \sigma)) \\ &= M(y := y*x, \sigma[x \mapsto \beta+1]) \quad \text{where } \beta = \sigma(x) \\ &= \sigma[x \mapsto \beta+1][y \mapsto \delta] \quad \text{where } \delta = \sigma[x \mapsto \beta+1](y*x) = \sigma(y) \times (\beta+1) \end{aligned}$$
- **Conditional 1:** If the current state satisfies the test, evaluate the true branch, otherwise evaluate the false branch.
 - $M(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) = M(S_1, \sigma)$ if $\sigma \models B$; it = $M(S_2, \sigma)$ if $\sigma \models \neg B$.
 - **Example:** $M(\text{if } y \text{ then } x := x+1 \text{ else } z := x+2 \text{ fi}, \sigma)$
 $= \sigma[x \mapsto \sigma(x)+1]$ if $\sigma(y) = T$; if $\sigma(y) = F$, it = $\sigma[z \mapsto \sigma(x)+2]$
- **Conditional 2:** Since an **if-then** statement is an abbreviation for an **if-else-skip**, the meaning of an **if-then** is a special case of an **if-else**:

- $M(\mathbf{if\ } B \ \mathbf{then\ } S_1 \ \mathbf{fi}, \sigma) = M(\mathbf{if\ } B \ \mathbf{then\ } S_1 \ \mathbf{else\ skip\ fi}, \sigma)$
 $= M(S_1, \sigma)$ if $\sigma \models B$; it $= M(\mathbf{skip}, \sigma) = \sigma$ if $\sigma \models \neg B$.
- **Iterative:** If its test is true, a while loop behaves like **skip** (it finishes without changing the state). If its test is false, a while loop behaves like an evaluation of its loop body followed by an evaluation of the loop.
 - if $\sigma \models \neg B$, then $M(\mathbf{while\ } B \ \mathbf{do\ } S \ \mathbf{od}, \sigma) = \sigma$.
 - If $\sigma \models B$, then $M(\mathbf{while\ } B \ \mathbf{do\ } S \ \mathbf{od}, \sigma) = M(S; \mathbf{while\ } B \ \mathbf{do\ } S \ \mathbf{od}, \sigma)$
 $= M(\mathbf{while\ } B \ \mathbf{do\ } S \ \mathbf{od}, M(S, \sigma))$.
- **Loop Example 1:** Let $W \equiv \mathbf{while\ } x < n \ \mathbf{do\ } S_1 \ \mathbf{od}$, where $S_1 \equiv x := x+1; y := y+y$.
 - Let
 - $\tau_0 = \{x = 0, n = 2, y = 1\}$
 - $\tau_1 = M(S, \tau_0) = \{x = 1, n = 2, y = 2\}$
 - $\tau_2 = M(S, \tau_1) = \{x = 2, n = 2, y = 4\}$, and
 - $\tau_3 = M(S, \tau_2) = \{x = 3, n = 2, y = 8\}$.
 - Then
 - $M(W, \tau_0) = M(W, M(S, \tau_0)) = M(W, \tau_1)$, since $\tau_0 \models x < n$
 - $M(W, \tau_1) = M(W, M(S, \tau_1)) = M(W, \tau_2)$, since $\tau_1 \models x < n$
 - $M(W, \tau_2) = M(W, M(S, \tau_2)) = M(W, \tau_3)$, since $\tau_2 \models x < n$
 - $M(W, \tau_3) = \tau_3$, since $\tau_3 \not\models x < n$
- **Loop Example 2:** Let $W \equiv \mathbf{while\ } x \neq n \ \mathbf{do\ } x := x-1 \ \mathbf{od}$.
 - Let
 - $\tau_0 = \{x = -1, n = 0\}$
 - $\tau_1 = \{x = -2, n = 0\}$
 - $\tau_2 = \{x = -3, n = 0\}$
 -
 - $\tau_j = \{x = -j-1, n = 0\}$
 - Then
 - $M(W, \tau_0) = M(W, M(S, \tau_0)) = M(W, \tau_1)$, since $\tau_0 \models x \neq n$
 - $M(W, \tau_1) = M(W, M(S, \tau_1)) = M(W, \tau_2)$, since $\tau_1 \models x \neq n$
 - ...
 - $M(W, \tau_j) = M(W, M(S, \tau_j)) = M(W, \tau_{j+1})$, since $\tau_j \models x \neq n$
 - ...
 - and all of these are undefined.