Program Syntax; Operational Semantics

CS 536: Science of Programming, Spring 2023

2023-01-25 pp.3,5,6,7

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.

B. Outcomes

At the end of today, you should be able to

- Describe the 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.
- ***Note – Now that you're more used to the difference between syntactic and semantic objects, I won't be as strict about using italics to indicate syntactic items.*** For example, writing $\sigma(x+3) = \sigma(x)+3 = 6+3 = 9$ instead of $\underline{\sigma(x+3)} = \underline{\sigma(x) + 3} = \underline{6+3} = \underline{9}$ (assuming $\underline{\sigma(x)} = \underline{6}$).

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've omitted declarations because we don't need them when studying the semantics of our programs' execution.
- ***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 $b[e_1][\dots][e_n] := e$
  - For an n–dimensional array, all 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 [1]
  - ***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***.
  - Since we can simulate an ***if–then*** statement (without an ***else*** clause) with ***if … else skip***, we don't need to define a separate semantics for an ***if–then*** statement, which reduces the amount of analysis work we need to do.
- ***Iterative*** statement
  - ***Syntax***: ***while*** B ***do*** S_1 ***od***
  - Similarly to how we defined an ***if–then*** statement to be a particular kind of ***if–else*** statement, we can omit ***for*** loop and ***do–while*** loop statements, because we can simulate them using ***while*** loops. For example,

    for $x := e_1$ ***to*** e_2 ***do*** S

    turns into

    $x := e_1$; ***while*** $x \le e_2$ ***do*** S ; $x := x + 1$ ***od***.
- ***Program***: A program is simply a statement, typically a sequence statement.

Example 1: A Sample Program

- The program below calculates powers of 2. We run it with a value of n and it returns $y = 2^n$, unless $n < 0$, in which case it returns $y = 1$. Formally, on termination, the program establishes $(n \ge 0 \rightarrow y = 2^n) \wedge (n < 0 \rightarrow y = 1)$. (This is only one way to write the program, of course.)

```
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. So the statement, its sub–

[1] (Meant to say write this down with ***if…fi*** for conditional expressions.) The ***fi*** and ***od*** keywords are from Algol 68, which created them to solve the dangling else problem. (in C terms, take `if (B1) S1 else if (B2) S2;` — is `S2` the else clause for `if B1` or `if B2`? Nowadays people often use ***end if*** and ***end do*** instead.

statements, its sub–sub–statements, etc.) The program in Example 1 contains 10 embedded sub–statements. In no particular order:

1. $y := 1$	*(the first one; it sets the result if $n < 0$)*
2. $x := 0$	*(part of loop initialization)*
3. $y := 1$	*(part of loop initialization)*
4. $x := 0 ; y := 1$	*(loop initialization)*
5. $x := x + 1$	*(part of loop the body)*
6. $y := y + y$	*(part of loop the body)*
7. $x := x + 1 ;\ y := y + y$	*(the loop body; let's call this statement S)*
8. ***while*** $x < n$ ***do*** S ***od***	
9. $x := 0 ;\ y := 1 ;$ ***while*** $x < n$ ***do*** S ***od***	*(let's call this statement W)*
10. ***if*** $n < 0$ ***then*** $y := 1$ ***else*** W ***fi***	

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, the value of `++z` is the value of `z` *after* doing `z=z+1`.
  - As an expression, the value of `z++` is the value of `z` *before* doing `z=z+1`. I.e., first do `temp=z; z=z+1;` and then yield the value of `temp` as the value of the `z++` expression.

Example 2

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

Example 3

- The C statement: `x=a*z++;` is equivalent to $temp := z ;\ z := z + 1 ;\ x := a * temp$. Another equivalent piece of code is : $x := a * z ;\ z := z + 1$

Example 4

[2023-01-25]

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

  $x := x - 1 ;$ ***while*** $x \geq n$ ***do*** $z := z * x ; x := x - 1$ ***od***
- The decrement of x before the loop is for the first execution of $--x$ `>=` n (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 >= n`, where we jump to the top of the loop, decrement x, and test vs. n.

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 last `x-- >0` test, which evaluates to false.

Example 6

- The C program below calculates `p=n!`, if `n≥0`.
  - `p=1; for(x=2; x<n; ++x) p=p*x;`
- 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 the program is equivalent to
  - `p=1; x=2; while(x<n) {p=p*x; ++x;}`
- In our language, an equivalent program is
  - $p := 1\,;\ x := 2\,;$ ***while*** $x < n$ ***do*** $p := p * x\,;\ x := x + 1$ ***od***
- (There are certainly other ways to translate the C `for` loop.)

Notes on Translations

- There can be more than one possible translation, especially for complicated programs. The important point is that translation is not difficult and doesn't result in grossly different code.
- 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, we'll start with 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.
  - We use angle brackets instead of parentheses just to make it clear that we're talking about a configuration, not an arbitrary pair.
- ***Definition***: The ***operational semantics*** of programs is given by a relation on configurations: $\langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle$ means that ***executing*** *S* in state σ ***for one step*** yields $\langle S_1, \sigma_1 \rangle$. S_1 is the ***continuation*** of *S*. The formal definition of $\rightarrow$ will be given in the next section, but we can present some intuitive examples.

- ***Example 1***: $\langle x := x+1 ; y := x, \{x=5\}\rangle \rightarrow \langle y := x, \{x=6\}\rangle$ because execution of the first assignment changes the value of x and leaves us with one more assignment to execute. (The formal definition of $\langle x := e, \sigma\rangle \rightarrow \ldots$ will be given in the next section.)
- ***Example 2***:
  - $\langle x := x+1 ;\ y := x ;\ z := y+3, \{x=5\}\rangle \rightarrow \langle y := x ; z := y+3, \{x=6\}\rangle$,
  - $\langle y := x ; z := y+3, \{x=6\}\rangle \rightarrow \langle z := y+3, \{x=6, y=6\}\rangle$, and
  - $\langle z := y+3, \{x=6, y=6\}\rangle \rightarrow \langle E, \{x=6, y=6, z=9\}\rangle$.
- ***Notation***: In the line above, the symbol E stands for the empty program. We use it to indicate a program that's finished execution because $\langle E, \tau\rangle$ is more readable than $\langle\ , \tau\rangle$.
- ***Notation***: We can compress Example 2 by writing it as a chain of $\rightarrow$ steps:

$$\begin{aligned}&\langle x := x+1 ;\ y := x ;\ z := y+3, \{x=5\}\rangle\\ &\quad\rightarrow \langle y := x ; z := y+3, \{x=6\}\rangle\\ &\quad\rightarrow \langle z := y+3, \{x=6, y=6\}\rangle\\ &\quad\rightarrow \langle E, \{x=6, y=6, z=9\}\rangle\end{aligned}$$

- ***Definition***: ***Execution of*** S ***starting in state*** σ ***converges to state*** τ if there is a sequence of executions steps $\langle S, \sigma\rangle \rightarrow \ldots \rightarrow \langle E, \tau\rangle$. If we're not interested in τ, we can abbreviate this to say that execution of S ***converges***. Equivalent phrasings are "σ ***terminates in state*** τ" and "σ ***terminates***".
- ***Definition***: The opposite of convergence is ***divergence***. For us, that's infinite loops or an infinitely long sequence of calculations. E.g., executions of $\langle$ ***while*** $x=0$ ***do*** $x := 0$ ***od***, $\{x=0\}\rangle$ and $\langle$ ***while*** $x \geq 0$ ***do*** $x := x+1$ ***od***, $\{x=0\}\rangle$ diverge. More generally in languages one can also have infinite recursion.

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.
  - $\langle$ ***skip***, $\sigma\rangle \rightarrow \langle E, \sigma\rangle$

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.
  - $\langle v := e, \sigma\rangle \rightarrow \langle E, \sigma[v \mapsto \sigma(e)]\rangle$.
- ***Example 3:*** $\langle x := x+1, \sigma\rangle$ [2023-01-25] $\rightarrow \langle E, \sigma[x \mapsto \sigma(x+1)]\rangle = \langle E, \sigma[x \mapsto \sigma(x)+1]\rangle$.

- ***Example 4***: $\langle x := 2*x*x+5*x+6, \sigma\rangle$ [2023-01-25] $\rightarrow \langle E, \sigma[x \mapsto \alpha]\rangle$ where $\alpha = \sigma(2*x*x+5*x+6) = 2\beta^2+5\beta+6$ where $\beta = \sigma(x)$. For complicated expressions, it can be helpful to introduce new symbols, like β, but they aren't required: we can also write $2\sigma(x)^2+5\sigma(x)+6$ here.

Array Element Assignment Statements

- To execute $b[e_1] := e$ 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.
  - $\langle b[e_1] := e, \sigma\rangle \rightarrow \langle E, \sigma[b[\alpha] \mapsto \beta]\rangle$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$. [2023-01-25]
- ***Example 5***: If $\sigma(x) = 8$, then

  $\langle b[x+1] := x*5, \sigma\rangle$

  [2023-01-25] $\rightarrow \langle E, \sigma[b[\beta] \mapsto \delta]\rangle$ where $\beta = \sigma(x+1) = \sigma(x)+1 = 8+1 = 9$

  $= \langle E, \sigma[b[9] \mapsto 40]\rangle$ and $\delta = \sigma(x*5) = \sigma(x)*5 = 8*5 = 40$
- 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 $\sigma(B) = T$ then $\langle$***if*** B ***then*** S_1 ***else*** S_2 ***fi***, $\sigma\rangle \rightarrow \langle S_1, \sigma\rangle$
  - If $\sigma(B) = F$ then $\langle$***if*** B ***then*** S_1 ***else*** S_2 ***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 $\langle$***skip***, $\sigma\rangle$ is pretty trivial.)
  - If $\sigma(B) = T$ then $\langle$***if*** B ***then*** S_1 ***fi***, $\sigma\rangle \rightarrow \langle S_1, \sigma\rangle$
  - If $\sigma(B) = F$ then $\langle$***if*** B ***then*** S_1 ***fi***, $\sigma\rangle = \langle$***if*** B ***then*** S_1 ***else skip fi***, $\sigma\rangle \rightarrow \langle$***skip***, $\sigma\rangle$
- ***Example 6***: Let $S \equiv$ ***if*** $x > 0$ ***then*** $y := 0$ ***fi***; we'll evaluate in a state where x is 5.
  - Then $\langle S, \sigma[x \mapsto 5]\rangle \rightarrow \langle y := 0, \sigma[x \mapsto 5]\rangle \rightarrow \langle E, \sigma[x \mapsto 5][y \mapsto 0]\rangle$.
    - (The first arrow is for the ***if–then***; the second is for the assignment to y.)
  - Similarly, $\langle S, \sigma[x \mapsto -1]\rangle \rightarrow \langle$***skip***, $\sigma[x \mapsto -1]\rangle \rightarrow \langle E, \sigma[x \mapsto -1]\rangle$.
    - (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 $W \equiv$ ***while*** B ***do*** S ***od***; then
  - If $\sigma(B) = F$, then $\langle W, \sigma\rangle \rightarrow \langle E, \sigma\rangle$
  - If $\sigma(B) = T$, then $\langle W, \sigma\rangle \rightarrow \langle S; W, \sigma\rangle$
  - 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 $S_1;\ S_2$, 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 U_1, then we continue by executing $U_1 ; S_2$.
  - If $\langle S_1, \sigma\rangle \rightarrow \langle E, \sigma_1\rangle$ then $\langle S_1 ; S_2, \sigma\rangle \rightarrow \langle S_2, \sigma_1\rangle$
  - If $\langle S_1, \sigma\rangle \rightarrow \langle U_1, \sigma_1\rangle$ then $\langle S_1 ; S_2, \sigma\rangle \rightarrow \langle U_1 ; S_2, \sigma_1\rangle$
- The rule for executing sequences is somewhat different from the other execution rules because it is a ***rule of inference***: We need a recursive call to the definition of "$\rightarrow$". Here, we need to know how $\langle S_1, \sigma\rangle$ executes before we can say how $\langle S_1 ; S_2, \sigma\rangle$ executes.
  - All the other execution rules are ***axioms***; they do not need a recursive use of the "$\rightarrow$" relation.
- ***Example 7***: Since $\langle x := y, \sigma\rangle \rightarrow \langle E, \sigma[x \mapsto \sigma(y)]\rangle$, we know
  - $\langle x := y;\ y := 2, \sigma\rangle \rightarrow \langle y := 2, \sigma[x \mapsto \sigma(y)]\rangle$ and
  - $\langle y := 2, \sigma[x \mapsto \sigma(y)]\rangle \rightarrow \langle E, \sigma[x \mapsto \sigma(y)][y \mapsto 2]\rangle$
  - Say the two statements are the true branch of an ***if*** $x > y$… ***fi*** where $\sigma(x > y) = T$, then
    - $\langle \textbf{if } x > y \textbf{ then } x := y ; y := 2 \textbf{ else } z := 3 \textbf{ fi},\ \sigma\rangle \rightarrow \langle x := y ; y := 2, \sigma\rangle$
      $\rightarrow \langle y := 2, \sigma[x \mapsto \sigma(y)]\rangle \rightarrow \langle E, \sigma[x \mapsto \sigma(y)][y \mapsto 2]\rangle$
  - On the other hand, if $\sigma(x > y) = F$, then we get [2023-01-25]
    - $\langle \textbf{if } x > y \textbf{ then } x := y;\ y := 2 \textbf{ else } z := 3 \textbf{ fi}, \sigma\rangle \rightarrow \langle z := 3, \sigma\rangle \rightarrow \langle E, \sigma[z \mapsto 3]\rangle$.

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 can involve writing an awfully large amount of formal manipulation using the notation. We'll do something about that in the following section.
- ***Example 8***: Let $W \equiv \textbf{while } x \geq 0 \textbf{ do } x := x - 1 \textbf{ od}$, then (in maximal detail)

  $\langle W, \sigma[x \mapsto 1]\rangle$
  $\rightarrow \langle x := x - 1 ; W, \sigma[x \mapsto 1]\rangle$ // *Since* $\sigma[x \mapsto 1](x \geq 0) = T$
  $\rightarrow \langle W, \sigma[x \mapsto 0]\rangle$ // *Since* $\sigma[x \mapsto 1][x \mapsto 0] = \sigma[x \mapsto 0]$
  $\rightarrow \langle x := x - 1 ; W, \sigma[x \mapsto 0]\rangle$ // *Since* $\sigma[x \mapsto 0](x \geq 0) = T$
  $\rightarrow \langle W, \sigma[x \mapsto -1]\rangle$ // *Evaluate assignment of* x
  $\rightarrow \langle E, \sigma[x \mapsto -1]\rangle$ // *Since* $\sigma[x \mapsto -1](x \geq 0) = F$

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

  $\langle S, \sigma\rangle = \langle s := 0 ; k := 0 ; W, \sigma\rangle$
  $\rightarrow \langle k := 0 ; W, \sigma[s \mapsto 0]\rangle$ //*Loop initialization*
  $\rightarrow \langle W, \sigma_0\rangle$ // *Where* $\sigma_0 = \sigma[s \mapsto 0][k \mapsto 0]$

$\rightarrow \langle S_1 ; W, \sigma_0 \rangle$ // *Since* $\sigma_0(k < n) = T$
$= \langle s := s + k + 1;\ k := k + 1 ; W, \sigma_0 \rangle$ // *By defn of* S_1*; note this step is "=", not "→"*
$\rightarrow \langle k := k + 1;\ W, \sigma_0[s \mapsto 1] \rangle$ // *Update* s *to* $\sigma_0(s{+}k{+}1) = \sigma_0(s) + \sigma_0(k)$
$\rightarrow \langle W, \sigma_1 \rangle$ // *Let* $\sigma_1 = \sigma_0[s \mapsto 1][k \mapsto 1]$ *and evaluate asgt of* k
$\rightarrow \langle S_1 ; W, \sigma_1 \rangle$ // *Since* $\sigma_1(k < n) = T$
$= \langle s := s + k + 1;\ k := k + 1 ; W, \sigma_1 \rangle$ // *By defn of* S_1*; note this step is "=", not "→"*
$\rightarrow \langle k := k + 1 ; W, \sigma_1[s \mapsto 3] \rangle$ // *Update* s *to* $\sigma_1(s{+}k{+}1) = \sigma_1(s) + \sigma_1(k)$
$\rightarrow \langle W, \sigma_2 \rangle$ // *Where* $\sigma_2 = \sigma_1[s \mapsto 3][k \mapsto 2]$
$\rightarrow \langle E, \sigma_2 \rangle$ // *Since* $\sigma_2(k < n) = F$

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 $\langle S_0, \sigma_0 \rangle \rightarrow^n \langle S_n, \sigma_n \rangle$ if there are $n + 1$ configurations $\langle S_0, \sigma_0 \rangle, \ldots, \langle S_{n-1}, \sigma_{n-1} \rangle$ such that $\langle S_0, \sigma_0 \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \ldots \langle S_{n-1}, \sigma_{n-1} \rangle \rightarrow \langle S_n, \sigma_n \rangle$. For $n = 0$ we define $\langle S_0, \sigma_0 \rangle \rightarrow^0 \langle S_0, \sigma_0 \rangle$.
- ***Definition***: We say $\langle S_0, \sigma_0 \rangle$ ***evaluates to*** $\langle U, \tau \rangle$ and write $\langle S_0, \sigma_0 \rangle \rightarrow^* \langle U, \tau \rangle$ if $\langle S_0, \sigma_0 \rangle \rightarrow^n \langle U, \tau \rangle$ for some n.
  - An equivalent way to say all of this is that $\rightarrow^n$ is the n–fold composition of $\rightarrow$ and $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.
- ***Definition***: Execution of S starting in σ ***terminates in*** (= ***converges to***) τ if $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$. Execution of S starting in σ ***terminates*** (= ***converges***) if it terminates in some τ.
  - Recall: "terminate" and "converge" are synonyms, "diverge" is the antonym.
- ***Example 10***: Let S be $s := 0 ; k := n + 1 ; W$, where $W \equiv$ ***while*** $k > 0$ ***do*** S_1 ***od*** and $S_1 \equiv k := k - 1;$ $s := s + k$. 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 $\tau[k \mapsto \alpha][s \mapsto \beta]$. We find

  $\langle S_1, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle = \langle k := k - 1;\ s := s + k, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle$
  $\rightarrow \langle s := s + k, \tau[s \mapsto \beta][k \mapsto \alpha - 1] \rangle$
  $\rightarrow \langle E, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1]) \rangle$
- Combining the two steps gives us $\langle S_1, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle \rightarrow^2 \langle E, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1] \rangle$. Adding the test at the top of the loop (for when $\alpha > 0$) lets us give behavior of an arbitrary loop iteration, namely $\langle W, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle \rightarrow^3 \langle W, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1] \rangle$.
- Now let's evaluate S when n is 2; if $\sigma(n) = 2$, then

  $\langle S, \sigma \rangle = \langle s := 0 ; k := n + 1 ; W, \sigma \rangle$
  $\rightarrow^2 \langle W, \sigma[s \mapsto 0][k \mapsto 3] \rangle$ // *After loop initialization*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 2][s \mapsto 2] \rangle$ // *After one iteration of the loop*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 1][s \mapsto 3] \rangle$ // *After two iterations*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 0][s \mapsto 3] \rangle$ // *After three iterations*
  $\rightarrow \langle E, \sigma[k \mapsto 0][s \mapsto 3] \rangle$ // *And now we stop, since* $k > 0$ *is false*