In this lecture, we will look at SIMP, for Simple Imperative Programming Language. Under the syntax of SIMP, we will look at different semantic systems. First, we look at the syntax, which tells us what expressions/strings are valid in SIMP. We assume the existence of the system of natural numbers.

\[ \mathbb{N} = \{0, 1, 2, \ldots \} \]

We allow the following basic building blocks in our programs.

- **Arithmetic Expressions (Aexp)** - these are expressions containing natural numbers from \( \mathbb{N} \), variables (which are placeholders taken from the set \( \{x, y, z, \ldots\} \)) and valid arithmetic operation symbols between a collection of variables and natural numbers, surrounded by appropriate parentheses to avoid ambiguous operations.

- **Boolean Expressions (Bexp)** - these are expressions containing boolean variables, propositions and conjunctions/disjunctions of them.

- **Programs (P)** - composite program statements derived from Aexp, Bexp and other rules to be defined below.

As mentioned above, variables are taken from the set

\[ \mathbb{V} = \{x, y, z, \ldots\} \]

A state \( \sigma \) is defined as a function from the set of variables to the set of natural numbers.

\[ \sigma : V \rightarrow \mathbb{N} \]

We use \( \Sigma \) to denote the set of all possible states in a given context. To give a simple example of how the semantics work, see that the semantics of an Aexp with only numbers maps to a single natural number, while that of an Aexp with variables maps each state to a natural number, based on the value the variable takes in that particular state.

\[ \llbracket (3 + 4) \cdot 2 \rrbracket = 14 \in \mathbb{N} \]

\[ \llbracket (x + 4) \cdot 2 \rrbracket : \Sigma \rightarrow \mathbb{N} \]

Now we look at the rules for each expression type in detail.
Arithmetic expression - Aexp

One can compactly write the rules as a single line.

\[ e = n|x|e.e|e + e|(e) \]

More explicitly, we can have the following rules. Note that these are not unique, since we could have a better, more compact set of rules to take care of all the different constraints including parenthesis matching. Coming up with a good set of rules is a fruitful exercise.

\[
\begin{array}{c}
\text{n : Factor} \\
\hline
n \in \mathbb{N} \\
\text{x : Factor} \\
\hline
x \in \mathbb{V} \\
\text{e : Aexp} \\
\hline
(e) : \text{Factor} \\
\text{e : Aexp} \\
\hline
f : \text{prod} \\
\text{e + f : Aexp} \\
\hline
\text{e : prod} \\
\text{f : Factor} \\
\hline
\text{e.f : prod} \\
\text{e : prod} \\
\text{e : Aexp} \\
\text{e : Factor} \\
\text{e : prod}
\end{array}
\]

Boolean expression - Bexp

\[ e = true|false|e = f|e \leq f|b \land c|b \lor c|\neg b|(b) \]

The semantics of boolean expressions take values in the set \( \mathbb{B} = \{true, false\} \).

\[
\begin{array}{c}
\text{true : Bexp} \\
\hline
\text{false : Bexp} \\
\text{e : Aexp} \\
\hline
\text{f : Aexp} \\
\text{e = f : Bexp} \\
\text{e : Aexp} \\
\hline
\text{f : Aexp} \\
\text{e \leq f : Bexp} \\
\text{b : Bexp} \\
\text{\neg b : Bexp} \\
\text{b : Bexp} \\
\text{(b) : Bexp} \\
\text{b : conjunction} \\
\text{(b) : Bexp} \\
\text{b : disjunction} \\
\text{(b) : Bexp} \\
\text{b : Bexp} \\
\text{b : conjunction}
\end{array}
\]
\[
\begin{align*}
& b : \text{Bexp} \\
& b : \text{disjunction} \\
& b : \text{conjunction} \\
& c : \text{Bexp} \\
& b \land c : \text{conjunction} \\
& b : \text{disjunction} \\
& c : \text{Bexp} \\
& b \lor c : \text{disjunction}
\end{align*}
\]

Program - P

\[p = \text{skip}|x = e|p;q|\text{if } b \text{ then } p|\text{while } b \text{ do } p\]

\[
\begin{align*}
\text{skip} : & P \\
\text{e} : & \text{Aexp} \\
\text{x} = & e : P \\
p : & P \\
q : & P \\
p ; q : & P \\
b : & \text{Bexp} \\
p : & P \\
\text{if } b \text{ then } p : & P \\
b : & \text{Bexp} \\
p : & P \\
\text{while } b \text{ do } p : & P
\end{align*}
\]

Semantics

Now that we know what statements/expressions are valid in SIMP, let us turn our heads to defining the semantics for these expressions.

In general, we have to define the following functions.

- \([a] : \Sigma \rightarrow \mathbb{N} \ (a: \text{Aexp})\)
- \([b] : \Sigma \rightarrow \mathbb{B} \ (b: \text{Bexp})\)
- \([p] : \Sigma \rightarrow \Sigma \ (p:\text{P})\)

We can define semantics on different levels. The first type of semantics we look at are called Operational Semantics. These are local semantics, in the sense that they tell us how a program transforms one particular state to another. One can think of interpreter-based languages (such as Python) to understand operational semantics. The notation is as follows.

\[
< p, \sigma > \rightarrow \sigma'
\]

which is read as \textit{If program }\textit{p is executed in state }\textit{\sigma, the result is state }\textit{\sigma’}.

\[
\begin{align*}
< \text{skip}, \sigma > & \rightarrow \sigma \\
< e, \sigma > & \rightarrow n \\
< x := e, \sigma > & \rightarrow \sigma[x \mapsto n]
\end{align*}
\]
where the notation $\sigma[x \mapsto n]$ is read as the same state as $\sigma$ but with the variable $x$ mapped to $n$ instead. Aexp’s and Bexp’s derive their semantics from the respective arithmetic and boolean systems. We show a couple of examples below to make this clear.

$$\frac{\langle e, \sigma \rangle \rightarrow n_1 \quad \langle f, \sigma \rangle \rightarrow n_2}{\langle e + f, \sigma \rangle \rightarrow n} \quad n = n_1 + n_2$$

$$\frac{\langle e, \sigma \rangle \rightarrow n_1 \quad \langle f, \sigma \rangle \rightarrow n_2}{\langle e = f, \sigma \rangle \rightarrow \text{true}} \quad n_1 = n_2$$

The operational semantics for the sequential, if and while clauses are given below. The if and while semantics break into two cases based on whether the conditions are true or false.

$$\frac{\langle p, \sigma \rangle \rightarrow \sigma' \quad \langle q, \sigma' \rangle \rightarrow \sigma''}{\langle p; q, \sigma \rangle \rightarrow \sigma''}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{if } b \text{ then } p, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle p, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } p, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } p, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle p; \text{while } b \text{ do } p, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } b \text{ do } p, \sigma \rangle \rightarrow \sigma'}$$

Next we look at a complete derivation tree with a simple example.

$$\langle 0, (x, 5) \rangle \rightarrow 0 \quad \langle x = 0, (x, 5) \rightarrow (x, 0) \rangle \rightarrow 0 \quad \langle x, (x, 0) \rightarrow 0 \rangle \rightarrow 0 \quad \langle x \leq 2, (x, 0) \rightarrow 2 \rangle \rightarrow 2 \quad \langle x = x + 1, \text{while } x \leq 2 \text{ do } x = x + 1, (x, 0) \rightarrow (x, 3) \rangle \rightarrow (x, 3)$$

We continue up the while loop derivation until we reach a deadend when the loop condition becomes false.

In the next lecture we will look at axiomatic semantics which are stronger and more expressive than operational semantics. Instead of talking about how the program affects a particular state, they talk about the preconditions and postconditions that a program satisfies. Thus we can look at certain types of states and the corresponding behavior all at once, instead of specifying individual behavior for each state.