Language Engineering Notes

paraphrased by Tyler Wright

*An important note, these notes are absolutely **NOT** guaranteed to be correct, representative of the course, or rigorous. Any result of this is not the author's fault.*

1 Syntax and Semantics

1.1 What is Syntax?

Syntax is the grammatical structure of a program. For example, for the program $x = y; y = z; z = x;$, syntactic analysis of this program would conclude that we have three statements concluded with ';'. Each of said statements are variables followed by the composite symbol '=' and another variable.

1.2 What are Semantics?

The semantics of a program are what the program evaluates to or rather, the meaning of a syntactically correct program. For example, $x = y$ evaluates to setting the value of x to the value of y.

2 Mathematical Notation

2.1 Relations

For two sets X, Y we have that f is a relation from X to Y if $f \subseteq X \times Y$.

2.2 Total Functions

A total function from X to Y is a function $f : X \to Y$ that maps each value in X to a value in Y.

2.3 Partial Functions

A partial function from X to Y is a function $f : X \hookrightarrow Y$ where for some $X' \subseteq X$ we have that $f : X' \to Y$ is total.

3 The While Language

3.1 Syntactic Categories

For this language, we have five syntactic categories:

- **Numerals** (Num), denoted by n
- **Variables** (Var), denoted by x
- **Arithmetic Expressions** (Aexp), denoted by a

- **Boolean Expressions** (Bexp), denoted by b
- **Statements** (Stm), denoted by S.

We assume that the numerals and variables are defined elsewhere (for example, the numerals could be strings of digits and the variables could be strings of letters). The other structures are detailed below:

$$\begin{aligned} a &:= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\ b &:= \texttt{true} \mid \texttt{false} \mid a_1 == a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ S &:= x = a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S. \end{aligned}$$

We define representations of each structure in terms of itself as composite elements and the rest are basis elements.

3.2 State

When considering a statement, it is almost always important to consider the state of the program as the role of statements is to change the state of the program. For example, when evaluating $x + 1$, we must consider what x is. This requires us to develop the notion of state.

We define state as a set of mappings from variables to values:

$$\textbf{State} = \{f : \textbf{Var} \to \mathbb{Z}\}.$$

This is commonly listed out. For example, if x maps to 4 and y maps to 5, we have our state equal to $\{x \mapsto 4, y \mapsto 5\}$.

3.3 Semantic Functions

The use of semantic functions is to convert syntactic elements into its meaning. Each of the semantic styles will be used to define said semantic functions for variable and statements. However, for numerals, arithmetic and boolean expressions, they are defined as follows.

3.3.1 Numerals

If we assume our numerals are in base-2 (binary), we can define a numeral as follows:

$$n := 0 \mid 1 \mid n\,0 \mid n\,1 \mid.$$

In order to determine the value represented by a numeral, we define a *semantic function* $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$:

$$\begin{aligned}\mathcal{N}(0) &= 0\\ \mathcal{N}(1) &= 1\\ \mathcal{N}(n\,0) &= 2 \cdot \mathcal{N}(n)\\ \mathcal{N}(n\,1) &= 2 \cdot \mathcal{N}(n) + 1\end{aligned}$$

This definition is called *compositional* as it simply defines an output for each way of constructing a numeral.

Much like Linear Algebra, it is important to differentiate the numeral '1' and the integer 1 *and similarly for zero.*

3.3.2 Arithmetic Expressions

For a given arithmetic expression, it may be necessary to analyse the state to determine the result. Thus, the semantic expression for arithmetic expressions is a function $\mathcal{A} : \mathbf{Aexp} \to (\mathbf{State} \to \mathbb{Z})$.

This means providing $\mathcal{A}$ with an arithmetic expression gives us a function from a state to a value. So, for a state s, we can define $\mathcal{A}$ as follows:

$$\begin{aligned}\mathcal{A}(n)(s) &= \mathcal{N}(n)\\ \mathcal{A}(x)(s) &= s(x)\\ \mathcal{A}(a_1 + a_2)(s) &= \mathcal{A}(a_1)(s) + \mathcal{A}(a_2)(s)\\ \mathcal{A}(a_1 \star a_2)(s) &= \mathcal{A}(a_1)(s) \star \mathcal{A}(a_2)(s)\\ \mathcal{A}(a_1 - a_2)(s) &= \mathcal{A}(a_1)(s) - \mathcal{A}(a_2)(s).\end{aligned}$$

Similarly to the numerical semantic function, it's important to differentiate between the syntactic $+, \star, -$ in the arithmetic expressions like $a_1 + a_2$ and the usual arithmetic operations $+, \star, -$ between integers.

3.3.3 Boolean Expressions

We define the set T to be $T = \{\mathrm{tt}, \mathrm{ff}\}$ where tt represents truth and ff represents falsity.

Similarly to arithmetic expressions we can define $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathbb{Z})$ for a

state s:

$$\begin{aligned}
\mathcal{B}(\texttt{true})(s) &= \text{tt} \\
\mathcal{B}(\texttt{false})(s) &= \text{ff} \\
\mathcal{B}(a_1 == a_2)(s) &= \begin{cases} \text{tt} & \text{if } \mathcal{A}(a_1)(s) = \mathcal{A}(a_2)(s) \\ \text{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}(a_1 \leq a_2)(s) &= \begin{cases} \text{tt} & \text{if } \mathcal{A}(a_1)(s) \leq \mathcal{A}(a_2)(s) \\ \text{ff} & \text{if } \mathcal{A}(a_1)(s) > \mathcal{A}(a_2)(s) \end{cases} \\
\mathcal{B}(\neg b)(s) &= \begin{cases} \text{tt} & \text{if } \mathcal{B}(b)(s) = \text{ff} \\ \text{ff} & \text{if } \mathcal{B}(b)(s) = \text{tt} \end{cases} \\
\mathcal{B}(b_1 \wedge b_2)(s) &= \begin{cases} \text{tt} & \text{if } \mathcal{B}(b_1)(s) = \text{tt and } \mathcal{B}(b_2)(s) = \text{tt} \\ \text{ff} & \text{if } \mathcal{B}(b_1)(s) = \text{ff or } \mathcal{B}(b_2)(s) = \text{ff} \end{cases}
\end{aligned}$$

3.4 Free Variables

We will later be interested in the 'free variables' of an expression (the set of variables occuring within it).

3.4.1 Arithmetic Expressions

We define $F_V : \mathbf{Aexp} \to \{\mathbf{Var}\}$ by:

$$\begin{aligned}
F_V(n) &:= \emptyset \\
F_V(x) &:= \{x\} \\
F_V(a_1 + a_2) &:= F_V(a_1) \cup F_V(a_2) \\
F_V(a_1 \star a_2) &:= F_V(a_1) \cup F_V(a_2) \\
F_V(a_1 - a_2) &:= F_V(a_1) \cup F_V(a_2).
\end{aligned}$$

Thus, we have developed a way to decompose expressions and generate all the variables said expression depends on. This is formalised for states s_1, s_2 and an arithmetic expression a:

$$[\forall x \in F_V(a)][s_1(x) = s_2(x)] \Longrightarrow [\mathcal{A}(a)(s_1) = \mathcal{A}(a)(s_2)].$$

3.4.2 Boolean Expressions

We define $F_V : \mathbf{Bexp} \to \{\mathbf{Var}\}$ by:

$$\begin{aligned} F_V(\texttt{true}) &:= \emptyset \\ F_V(\texttt{false}) &:= \emptyset \\ F_V(a_1 == a_2) &:= F_V(a_1) \cup F_V(a_2) \\ F_V(a_1 \leq a_2) &:= F_V(a_1) \cup F_V(a_2) \\ F_V(\neg b) &:= F_V(b) \\ F_V(b_1 \wedge b_2) &:= F_V(b_1) \cup F_V(b_2). \end{aligned}$$

Similarly, for states s_1, s_2 and an boolean expression b:

$$[\forall x \in F_V(b)][s_1(x) = s_2(x)] \Longrightarrow [\mathcal{B}(b)(s_1) = \mathcal{B}(b)(s_2)].$$

3.5 Substitutions

Within expressions, we will be interested in swapping the occurances of a variable with another expression. For an expression e with x in $F_V(e)$ and another expression e_0, we write $e[x \mapsto e_0]$ for the expression e where x is substituted for e_0.

3.5.1 Arithmetic Expressions

For an arithmetic expression a_0, we can define substitution on arithmetic expressions:

$$\begin{aligned}
n[y \mapsto a_0] &:= n \\
x[y \mapsto a_0] &:= \begin{cases} a_0 & \text{if } x = y \\ x & \text{otherwise} \end{cases} \\
(a_1 + a_2)[y \mapsto a_0] &:= (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 \star a_2)[y \mapsto a_0] &:= (a_1[y \mapsto a_0]) \star (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0] &:= (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0]).
\end{aligned}$$

3.5.2 Boolean Expressions

For an arithmetic expression a_0, we can define substitution on boolean expressions:

$$\begin{aligned}
(\texttt{true})[x \mapsto a_0] &:= \texttt{true} \\
(\texttt{false})[x \mapsto a_0] &:= \texttt{false} \\
(a_1 == a_2)[x \mapsto a_0] &:= (a_1[x \mapsto a_0]) == (a_2[x \mapsto a_0]) \\
(a_1 \leq a_2)[x \mapsto a_0] &:= (a_1[x \mapsto a_0]) \leq (a_2[x \mapsto a_0]) \\
(\neg b)[x \mapsto a_0] &:= \neg(b)[x \mapsto a_0] \\
(b_1 \wedge b_2)[x \mapsto a_0] &:= (b_1)[x \mapsto a_0] \wedge (b_2)[x \mapsto a_0].
\end{aligned}$$

3.5.3 States

A similar process applies to states, for a state s and the variables x, v:

$$(s[y \mapsto v])(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{otherwise.} \end{cases}$$

4 Operational Semantics

4.1 Overview of Operational Semantics

An operational explanation of the meaning of a construct will explain how to execute said construct. For example, in C, the semicolons provide chronology and the = symbol demonstrates assignment. The semantics are not merely interested in the results of a program.

More specifically, this means we are interested in how the state of the program changes. To this end, there are two approaches we shall consider:

- **Natural Semantics**: to describe how the overall results of executions are obtained
- **Structural Operational Semantics**: to describe how the individual steps of computation take place.

During this section, we will use $\langle S, s\rangle$ representing the statement S being executed from the state s and simply s to represent a terminal state.

4.2 Rules and Derivation Trees

4.2.1 Transitions

A transition is a construct representing the transition of a state s to a state s' via some statement S, denoted by $\langle S, s\rangle \to s'$.

4.2.2 Rules

We can define 'rules' which are of the form:

$$\frac{\langle S_1, s_1\rangle \to s_2, \ldots, \langle S_n, s_n\rangle \to s'}{\langle S, s\rangle \to s'} \quad \text{if } \cdots .$$

Where the premises consisting of immediate constituents of a statement S or compositions of said immediate constituents, are written above the conclusion. Conditions may be written to the right of the rule which are necessary for when the rule is applied.

4.2.3 Axioms

For a rule where the set of premises is empty, we call such a rule an *axiom*. Axioms with meta-variables are called *axiom schema* where we can obtain an *instance* of an axiom by defining particular variables.

4.2.4 Derivation Trees

A program's execution can be modelled by a 'derivation tree' formed from rules. For example, the program $x = y; y = z; z = x;$ can be written as:

$$\dfrac{\dfrac{\langle z = x, s_{570}\rangle \to s_{575} \qquad \langle x = y, s_{575}\rangle \to s_{775}}{\langle z = x; x = y, s_{570}\rangle \to s_{775}} \qquad \langle y = z, s_{775}\rangle \to s_{755}}{\langle x = y; y = z; z = x, s_{570}\rangle \to s_{755}}.$$

Where s_{abc} represents the state where $\{x \mapsto a, y \mapsto b, z \mapsto c\}$.

This tree repesents the decomposition of the composite statement at the bottom of the tree $x = y; y = z; z = x;$. As $z = x$ transforms the state s_{570} to s_{575} and $x = y$ transforms the state s_{575} to s_{775}, we know that the composite statement $z = x; x = y$ transforms s_{570} to s_{775}.

4.3 Natural Semantics

We can define some natural semantics for the While language as follows:

[assignment] $$\langle x = a, s\rangle \to s[x \mapsto \mathcal{A}(a)(s)]$$

[skip] $$\langle \texttt{skip}, s\rangle \to s$$

[composition] $$\dfrac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

[if] $$\begin{cases} \dfrac{\langle S_1, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} & \text{if } \mathcal{B}(b)(s) = \text{tt} \\ \dfrac{\langle S_2, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} & \text{if } \mathcal{B}(b)(s) = \text{ff} \end{cases}$$

[while] $$\begin{cases} \dfrac{\langle S, s\rangle \to s' \qquad \langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s''} & \text{if } \mathcal{B}(b)(s) = \text{tt} \\ \langle \texttt{while } b \texttt{ do } S, s\rangle \to s & \text{if } \mathcal{B}(b)(s) = \text{ff} \end{cases}$$

We have that the semantics detailed above are deterministic, that is, for identical inputs, we get identical outputs.

4.4 The Semantic Function

We can now condense the meaning of statements into a partial function from **State** to **State**. We define the semantic function as follows:

$$\mathcal{S} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State}).$$

Thus for a given statement S, we have that:

$$\mathcal{S}(S) : \mathbf{State} \hookrightarrow \mathbf{State}$$
$$s \mapsto \begin{cases} s' & \text{if } \langle S, s\rangle \to s' \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We that this function is partial because some statements may never terminate. Thus giving an indeterminate answer.