Language Engineering Notes

by Tyler Wright

github.com/Fluxanoia fluxanoia.co.uk

These notes are not necessarily correct, consistent, representative of the course as it stands today, or rigorous. Any result of the above is not the author's fault.

These notes are marked as unsupported, they were supported up until June 2020.

These notes are incomplete and will remain so for the foreseeable future.

Contents

1 Syntax and Semantics **5**
- 1.1 What is Syntax? . . . 5
- 1.2 What are Semantics? . . . 5

2 Mathematical Notation **6**
- 2.1 Relations . . . 6
- 2.2 Total Functions . . . 6
- 2.3 Partial Functions . . . 6

3 The While Language **7**
- 3.1 Syntactic Categories . . . 7
  - 3.1.1 Extensions of the Categories . . . 7
- 3.2 State . . . 8
- 3.3 Semantic Functions . . . 8
  - 3.3.1 Numerals . . . 8
  - 3.3.2 Arithmetic Expressions . . . 9
  - 3.3.3 Boolean Expressions . . . 9
- 3.4 Free Variables . . . 10
  - 3.4.1 Arithmetic Expressions . . . 10
  - 3.4.2 Boolean Expressions . . . 10
- 3.5 Substitutions . . . 11
  - 3.5.1 Arithmetic Expressions . . . 11
  - 3.5.2 Boolean Expressions . . . 11
  - 3.5.3 States . . . 11

4 Operational Semantics **12**
- 4.1 Rules and Derivation Trees . . . 12
  - 4.1.1 Transitions . . . 12
  - 4.1.2 Rules . . . 12
  - 4.1.3 Axioms . . . 12
  - 4.1.4 Derivation Trees . . . 13
- 4.2 Natural Semantics . . . 13
  - 4.2.1 The Semantic Function . . . 14
- 4.3 Structural Operational Semantics . . . 14
  - 4.3.1 Derivation Sequences . . . 15
  - 4.3.2 Partial Execution . . . 16
  - 4.3.3 The Semantic Function . . . 16
- 4.4 Semantic Equivalence . . . 16

5 The Abstract Machine **17**
5.1 Code . . . 17
5.2 Evaluation Stack . . . 17
5.3 Storage . . . 17
5.4 Terminal States . . . 17
5.5 Operational Semantics for the Abstract Machine . . . 18
5.5.1 Computation Sequences . . . 19
5.6 The Execution Function . . . 19
5.7 Code Generation . . . 19
5.7.1 Arithmetic Expressions . . . 20
5.7.2 Boolean Expressions . . . 20
5.7.3 Statements . . . 20
5.8 The Semantic Function . . . 21
5.9 Correctness . . . 21
5.9.1 Arithmetic Expressions . . . 21
5.9.2 Boolean Expressions . . . 21
5.9.3 Statements . . . 21

6 Denotational Semantics **22**
6.1 Partial Order . . . 22
6.1.1 Weak Partial order . . . 22
6.1.2 Strong Partial order . . . 22
6.1.3 Total Partial order . . . 22
6.2 Partial Order Sets . . . 22
6.2.1 Chains . . . 22
6.2.2 Least Upper Bounds . . . 23
6.2.3 Lifted Partial-Order Sets . . . 23
6.2.4 A Partial Order on State Transformers . . . 23
6.2.5 Fixed Point Theorem . . . 23
6.2.6 The Conditional Function . . . 23
6.2.7 The Fixed Point Function . . . 24
6.2.8 A Fibonacci Definition using `FIX` . . . 24
6.3 Direct Style . . . 25
6.3.1 The Semantic Function . . . 25
6.4 Continuation Style . . . 25
6.4.1 The Semantic Function on **While** . . . 26
6.4.2 The Semantic Function on **Exc** . . . 26

7 Axiomatics Semantics **27**
7.1 Axioms and Rules 27
7.2 Partial Correctness Schemata 28
7.3 Total Correctness Schemata 28

8 Interpretation **29**
8.1 Let Expressions 29

9 Compiling **30**
9.1 Variables 30
9.2 Stack Instructions 30
9.3 Lexers 30
9.4 Parsers 30
9.4.1 Ambiguity 31
9.4.2 LR and LL Parsing 31
9.4.3 Parser State 31

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.

3.1.1 Extensions of the Categories

We define a language **Exc** identical to **While** except we extend the **Statements** to:

$$
\begin{aligned}
S := {} & x = a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S \\
& \texttt{begin } S_1 \texttt{ handle } e : S_2 \texttt{ end} \mid \texttt{raise } e,
\end{aligned}
$$

and we define the **Exception** of which e is a member of.

The `raise` instruction, when called, breaks the execution of the current statement and gives control to the handling statement.

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:

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

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:

$$n := 0 \,|\, 1 \,|\, n +\!\!+\, 0 \,|\, n +\!\!+\, 1 \,|.$$

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 in 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:

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

3.4 Free Variables

The free variables of an expression is the set of variables occuring within it.

3.4.1 Arithmetic Expressions

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

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

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:

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

3.4.2 Boolean Expressions

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

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

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

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

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

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.

4.1 Rules and Derivation Trees

4.1.1 Transitions

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

4.1.2 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.

4.1.3 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.

4.1.4 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}.

4.2 Natural Semantics

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.

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.

As there is more than one definition of these semantics we will subscript these defintions with ns where appropriate.

4.2.1 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.

As there will be more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{ns}$.

4.3 Structural Operational Semantics

In structural operation semantics, there is a particular emphasis on the individual steps of execution. The transition relation in these semantics is of the form:

$$\langle S, s\rangle \Rightarrow \gamma,$$

where γ is of the form:

$\langle S', s'\rangle$	denoting that S and s have transformed in some way and that the execution is not yet complete,
s'	denoting that the execution has completed with some terminal state s'.

We shall say that $\langle S, s\rangle$ is **stuck** if there does not exist a γ such that $\langle S, s\rangle \Rightarrow \gamma$ and **unstuck** otherwise.

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

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

$$\text{[skip]} \qquad \langle \texttt{skip}, s\rangle \Rightarrow s,$$

$$\text{[composition]} \qquad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle} \quad \text{or} \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle},$$

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

$$\text{[while]} \qquad \langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow \langle \texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle.$$

We can see by the definition of composition that with these semantics we have to fully decompose and execution S_1 before we can begin execution of S_2.

As there is more than one definition of these semantics we will subscript these defintions with *sos* where appropriate.

4.3.1 Derivation Sequences

For a statement S and state s, we can consider the sequence of configurations of S in s. The length of such a sequence is either:

- **Finite**: $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ for some k in $\mathbb{Z}_{\geq 0}$ where each i in $\{0, 1, \ldots, k-1\}$ satisfies $\gamma_0 = \langle S, s\rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ where γ_k is terminal or stuck,
- **Infinite**: $(\gamma_k)_{k\geq 0}$ where for each i in $\mathbb{Z}_{\geq 0}$ satisfies $\gamma_0 = \langle S, s\rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$.

For two configurations γ_i, γ_k, we write $\gamma_i \Rightarrow^k \gamma_j$ to denote that there are k steps in execution between the configurations or $\gamma_i \Rightarrow^* \gamma_j$ to denote that there's some finite number of steps between the two configurations.

We say for a statement S and state s, the execution of S on s:

- **Terminates**: if and only if there's a finite derivation sequence starting with $\langle S, s, \rangle$
- **Loops**: if and only if there's a infinite derivation sequence starting with $\langle S, s \rangle$.

Thus, statements can loop on some states and terminate on others. Similarly, some states always terminate and always loop.

4.3.2 Partial Execution

If for some statement S, k in $\mathbb{Z}_{\geq 0}$, and states s, s'', we have that $\langle S, s \rangle \Rightarrow^k s''$, then there exists k_1, k_2 in $\mathbb{Z}_{\geq 0}$ such that $k_1 + k_2 = k$, $\langle S, s \rangle \Rightarrow^{k_1} s'$, and $\langle S, s' \rangle \Rightarrow^{k_2} s''$ for some intermediate state s'.

4.3.3 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:

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

Thus for a given statement S, we have that:

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

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

As there is more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{sos}$.

4.4 Semantic Equivalence

For every statement S of **While**, we have that:

$$\mathcal{S}_{ns}(S) = \mathcal{S}_{sos}(S).$$

Similarly, we have that if a statement terminates to a state for one type of semantics, it will terminate in the same state for the other.

5 The Abstract Machine

We have that the configurations of the abstract machine are of the form:

$$\langle c, e, s \rangle,$$

where c is the code being executed, e is the evaluation stack, and s is the storage. We use ε to denote an empty stack or code.

5.1 Code

The code being executed is simply an instruction defined as follows:

$$\begin{aligned} c &:= \text{empty} \mid \text{inst}; c \\ \text{inst} &:= \text{PUSH} - n \mid \text{ADD} \mid \text{MULT} \mid \text{SUB} \\ &\mid \text{TRUE} \mid \text{FALSE} \mid \text{EQ} \mid \text{LE} \mid \text{AND} \mid \text{NEG} \\ &\mid \text{FETCH} - x \mid \text{STORE} - x \mid \text{NOOP} \\ &\mid \text{BRANCH}(c, c) \mid \text{LOOP}(c, c). \end{aligned}$$

5.2 Evaluation Stack

The evaluation stack is an stack of elements in $\mathbb{Z}$ or T.

5.3 Storage

For the sake of simplicity, the storage is essentially a state.

5.4 Terminal States

A configuration is terminal if the code is empty.

5.5 Operational Semantics for the Abstract Machine

The table below lays out the operational semantics for the abstract machine.

Memory Management			
$\langle$ PUSH-$n : c, e, s$ $\rangle$	$\triangleright$	$\langle c, \mathcal{N}(n) : e, s\rangle$	
$\langle$ FETCH-$x : c, e, s$ $\rangle$	$\triangleright$	$\langle c, s(x) : e, s\rangle$	
$\langle$ STORE-$x : c, z : e, s$ $\rangle$	$\triangleright$	$\langle c, e, s[x \mapsto z]\rangle$	if z is in $\mathbb{Z}$
Arithmetic Operators			
$\langle$ ADD: $c, z_1 : z_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, (z_1 + z_2) : e, s\rangle$	if z_1, z_2 are in $\mathbb{Z}$
$\langle$ MULT: $c, z_1 : z_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, (z_1 \star z_2) : e, s\rangle$	if z_1, z_2 are in $\mathbb{Z}$
$\langle$ SUB: $c, z_1 : z_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, (z_1 - z_2) : e, s\rangle$	if z_1, z_2 are in $\mathbb{Z}$
Boolean Operators			
$\langle$ TRUE: c, e, s $\rangle$	$\triangleright$	$\langle c, \text{tt} : e, s\rangle$	
$\langle$ FALSE: c, e, s $\rangle$	$\triangleright$	$\langle c, \text{ff} : e, s\rangle$	
$\langle$ EQ: $c, z_1 : z_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, (z_1 == z_2) : e, s\rangle$	if z_1, z_2 are in $\mathbb{Z}$
$\langle$ LEQ: $c, z_1 : z_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, (z_1 \leq z_2) : e, s\rangle$	if z_1, z_2 are in $\mathbb{Z}$
$\langle$ AND: $c, t_1 : t_2 : e, s$ $\rangle$	$\triangleright$	$\langle c, \text{tt} : e, s\rangle$	if $t_1 == \text{tt}$, $t_2 == \text{tt}$
	$\triangleright$	$\langle c, \text{ff} : e, s\rangle$	otherwise
$\langle$ NEG: $c, t : e, s$ $\rangle$	$\triangleright$	$\langle c, \text{tt} : e, s\rangle$	if $t == \text{ff}$
	$\triangleright$	$\langle c, \text{ff} : e, s\rangle$	if $t == \text{tt}$
Control Flow			
$\langle$ NOOP: c, e, s $\rangle$	$\triangleright$	$\langle c, e, s\rangle$	
$\langle$ BRANCH$(c_1, c_2) : c, t : e, s$ $\rangle$	$\triangleright$	$\langle c_1, e, s\rangle$	if $t == \text{tt}$
	$\triangleright$	$\langle c_2, e, s\rangle$	otherwise
$\langle$ LOOP$(c_1, c_2) : c, e, s$ $\rangle$	$\triangleright$	$\langle c_1$:BRANCH(c_2 :LOOP(c_1, c_2), NOOP): $c, e, s\rangle$	

5.5.1 Computation Sequences

Corresponding to the derivation sequences of structural operational semantics, we can also define a computation sequence of the abstract machine. For a sequence of instructions c and some storage s, the length of such a sequence is either:

- **Finite**: $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ for some k in $\mathbb{Z}_{\geq 0}$ where each i in $\{0, 1, \ldots, k-1\}$ satisfies $\gamma_0 = \langle c, \varepsilon, s \rangle$, $\gamma_i \triangleright \gamma_{i+1}$ where there is no γ such that $\gamma_k \triangleright \gamma$,
- **Infinite**: $(\gamma_k)_{k \geq 0}$ where for each i in $\mathbb{Z}_{\geq 0}$ satisfies $\gamma_0 = \langle c, \varepsilon, s \rangle$, $\gamma_i \triangleright \gamma_{i+1}$.

We have that the evaluation stack e is always empty initially. We say the computation sequence is terminating if it's finite and looping if it's infinite. Terminating sequences lead to terminal or stuck configurations.

The semantics of the abstract machine are deterministic as defined in the table above, meaning for all computation sequences $\gamma, \gamma_1, \gamma_2$:

$$\left.\begin{array}{l} \gamma_1 \triangleright \gamma \\ \gamma_2 \triangleright \gamma \end{array}\right\} \Rightarrow \gamma_1 = \gamma_2.$$

5.6 The Execution Function

Similarly to the semantic function for our types of semantics, we define a function for the execution of abstract machine code, $\mathcal{M}$:

$$\mathcal{M} : \mathbf{Code} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

$$\mathcal{M}(c)(s) = \begin{cases} s' & \text{if } \langle c, \varepsilon, s \rangle \triangleright^* \langle \varepsilon, e, s' \rangle \\ \text{undefined} & \text{otherwise.} \end{cases}$$

5.7 Code Generation

We define functions for generating code for the abstract machine from expressions and statements.

5.7.1 Arithmetic Expressions

This code generation function has the form:

$$\mathcal{CA} : \textbf{Aexp} \to \textbf{Code}.$$

The function is defined as follows:

$$\begin{aligned}
\mathcal{CA}(n) &= \text{PUSH-}n, \\
\mathcal{CA}(x) &= \text{FETCH-}x, \\
\mathcal{CA}(a_1 + a_2) &= \mathcal{CA}(a_2) : \mathcal{CA}(a_1) : \text{ADD}, \\
\mathcal{CA}(a_1 - a_2) &= \mathcal{CA}(a_2) : \mathcal{CA}(a_1) : \text{SUB}, \\
\mathcal{CA}(a_1 \star a_2) &= \mathcal{CA}(a_2) : \mathcal{CA}(a_1) : \text{MULT}.
\end{aligned}$$

5.7.2 Boolean Expressions

This code generation function has the form:

$$\mathcal{CB} : \textbf{Bexp} \to \textbf{Code}.$$

The function is defined as follows:

$$\begin{aligned}
\mathcal{CB}(\texttt{true}) &= \text{TRUE}, \\
\mathcal{CB}(\texttt{false}) &= \text{FALSE}, \\
\mathcal{CB}(a_1 == a_2) &= \mathcal{CA}(a_2) : \mathcal{CA}(a_1) : \text{EQ}, \\
\mathcal{CB}(a_1 \leq a_2) &= \mathcal{CA}(a_2) : \mathcal{CA}(a_1) : \text{LEQ}, \\
\mathcal{CB}(\neg b) &= \mathcal{CB}(b) : \text{NEG}, \\
\mathcal{CB}(b_1 \wedge b_2) &= \mathcal{CB}(b_2) : \mathcal{CB}(b_1) : \text{AND}.
\end{aligned}$$

5.7.3 Statements

This code generation function has the form:

$$\mathcal{CS} : \textbf{Stm} \to \textbf{Code}.$$

The function is defined as follows:

$$\begin{aligned}
\mathcal{CS}(x = a) &= \mathcal{CA}(a) : \text{STORE-}x, \\
\mathcal{CS}(\texttt{skip}) &= \text{NOOP}, \\
\mathcal{CS}(S_1 : S_2) &= \mathcal{CS}(S_1) : \mathcal{CS}(S_2), \\
\mathcal{CS}(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) &= \mathcal{CB}(b) : \text{BRANCH}(\mathcal{CS}(S_1), \mathcal{CS}(S_2)), \\
\mathcal{CS}(\texttt{while } b \texttt{ do } S) &= \text{LOOP}(\mathcal{CB}(b), \mathcal{CS}(s)).
\end{aligned}$$

5.8 The Semantic Function

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:

$$\mathcal{S}(S) = (\mathcal{M} \circ \mathcal{CS})(S).$$

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

As there is more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{am}$.

5.9 Correctness

5.9.1 Arithmetic Expressions

We have that for all arithmetic expressions a:

$$\langle \mathcal{CA}(a), \varepsilon, s\rangle \triangleright^* \langle \varepsilon, \mathcal{A}(a)(s), s\rangle,$$

and all intermediary states have non-empty stacks.

5.9.2 Boolean Expressions

We have that for all boolean expressions b:

$$\langle \mathcal{CB}(b), \varepsilon, s\rangle \triangleright^* \langle \varepsilon, \mathcal{B}(b)(s), s\rangle,$$

and all intermediary states have non-empty stacks.

5.9.3 Statements

We have that for all statements S:

$$S_{ns}(S) = S_{sos}(S) = S_{am}(S).$$

Also, for all states s, we have that:

$$\left.\begin{array}{c} \langle S, s\rangle \to s' \\ \text{or} \\ \langle S, s\rangle \Rightarrow^* s' \end{array}\right\} \Rightarrow \langle \mathcal{CS}(S), \varepsilon, s\rangle \triangleright^* \langle \varepsilon, \varepsilon, s'\rangle$$

6 Denotational Semantics

6.1 Partial Order

6.1.1 Weak Partial order

A weak partial order on X with a, b, c in X has the following properties:

- reflexive ($a \leq a$),
- transitive ($a \leq b$, $b \leq c \Rightarrow a \leq c$),
- antisymmetric ($a \leq b, b \leq a \Rightarrow a = b$).

6.1.2 Strong Partial order

A strong partial order on X with a, b, c in X has the following properties:

- irreflexive ($a \leq a$ for no a),
- transitive ($a \leq b$, $b \leq c \Rightarrow a \leq c$),
- antisymmetric ($a \leq b, b \leq a \Rightarrow a = b$).

6.1.3 Total Partial order

A total partial order on X with a, b, c in X has the following properties:

- connex ($a \leq b$ or $b \leq a$),
- reflexive ($a \leq a$),
- transitive ($a \leq b$, $b \leq c \Rightarrow a \leq c$),
- antisymmetric ($a \leq b, b \leq a \Rightarrow a = b$).

6.2 Partial Order Sets

For a partial order $\sqsubseteq$ on a set D, we have that $(\sqsubseteq, D)$ is a partial order set (PO-set).

6.2.1 Chains

For a PO-set $(\sqsubseteq, D)$, for $X \subseteq D$, we say that X is a chain if and only if for all x_1, x_2 in X:

$$x_1 \sqsubseteq x_2 \text{ or } x_2 \sqsubseteq x_1,$$

meaning X is totally ordered.

6.2.2 Least Upper Bounds

For a PO-set $(\sqsubseteq, D)$, the least upper bound x of $X \subseteq D$ is denoted by $x = \sqcup X$.

If such a bound exists for all subsets X that are chains, $(\sqsubseteq, D)$ is a *chain complete* PO-set.

If such a bound exists for all subsets X, $(\sqsubseteq, D)$ is a *complete lattice*.

6.2.3 Lifted Partial-Order Sets

A PO-set is 'lifted' if an artificial bottom element $\perp$ is added to the set.

6.2.4 A Partial Order on State Transformers

We have the following PO-set $(\sqsubseteq, T)$ where:

$$T = \{f \text{ where } f : \mathbf{State} \hookrightarrow \mathbf{State}\},$$

and for f, g in T, $f \sqsubseteq g$ is equivalent to saying:

$$fs = s' \Rightarrow gs = s' \text{ for all states } s.$$

This is a **weak** partial order on T, with the least element being 0 in T undefined for all inputs.

6.2.5 Fixed Point Theorem

For $f : X \to X$ a continuous function on a chain-complete partial order $(\sqsubseteq, X)$ with a least element $\perp$, we have that:

$$\texttt{FIX}\ f = \sqcup \{f^n(\perp) : n \in \mathbb{Z}_{\geq 0}\},$$

exists and is the least fixed point of f. For clarity:

$$f^0 = \texttt{id} \qquad f^n = f \circ f^{n-1},$$

where n is in $\mathbb{Z}_{\geq 0}$.

6.2.6 The Conditional Function

We define the type of `cond` as follows:

$\texttt{cond} : (\mathbf{State} \to \mathbf{T}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$,

with definition:

$$\texttt{cond}(p, f_1, f_2)s = \begin{cases} f_1(s) & \text{if } p(s) == \mathbf{tt} \\ f_2(s) & \text{otherwise.} \end{cases}$$

6.2.7 The Fixed Point Function

We define the type of `FIX` as follows:

$$\texttt{FIX} : (\textbf{State} \hookrightarrow \textbf{State}) \to (\textbf{State} \hookrightarrow \textbf{State}) \to (\textbf{State} \hookrightarrow \textbf{State}),$$

where $\texttt{FIX}(F)$ returns the least fixed 'point' (or rather function) of F (f in $\mathrm{Im}(F)$ such that $F(f) = f$). Smallest in this case is the 'most undefined' function.

6.2.8 A Fibonacci Definition using `FIX`

Consider the following definition:

```
fib :: Int -> Int
fib = fix F where
  F :: (Int -> Int) -> (Int -> Int)
  F f n
    | (n == 0) || (n == 1) = 1
    | otherwise            = (f (n - 1)) + (f (n - 2))
```

So, if we want to calculate `fib 4` we get:

```
fib 4   = fix F 4
        = F (fix F) 4
        = (fix F 3) + (fix F 2)

fix F 3 = F (fix F) 3
        = (fix F 2) + (fix F 1)

fix F 2 = F (fix F) 2
        = (fix F 1) + (fix F 0)

fix F 1 = F (fix F) 1
        = 1

fix F 0 = F (fix F) 0
        = 1
```

```
fix F 2 = F (fix F) 2
        = (fix F 1) + (fix F 0)
        = 1 + 1
        = 2

fix F 3 = F (fix F) 3
        = (fix F 2) + (fix F 1)
        = 2 + 1
        = 3

fib 4   = fix F 4
        = F (fix F) 4
        = (fix F 3) + (fix F 2)
        = 3 + 2
        = 5
```

6.3 Direct Style

6.3.1 The Semantic Function

We define the semantic function as follows:

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

We that this function is partial because some statements may never terminate. We have the following:

$$\begin{aligned}
\mathcal{S}(x = a)(s) &= s[x \mapsto \mathcal{A}(a)(s)], \\
\mathcal{S}(\texttt{skip}) &= \texttt{id}, \\
\mathcal{S}(S_1; S_2) &= \mathcal{S}(S_2) \circ \mathcal{S}(S_1), \\
\mathcal{S}(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) &= \texttt{cond}(\mathcal{B}(b), \mathcal{S}(S_1), \mathcal{S}(S_2)), \\
\mathcal{S}(\texttt{while } b \texttt{ do } S) &= \texttt{FIX } F \text{ (where } F(g) := \texttt{cond}(\mathcal{B}(b), g \circ \mathcal{S}(S), \texttt{id})).
\end{aligned}$$

As there is more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{ds}$.

6.4 Continuation Style

A continuation describes the effect of executing the remainder of the program denoted as **Cont** and typed as follows:

$$\mathbf{Cont} = \mathbf{State} \hookrightarrow \mathbf{State}.$$

For a sequence of statements $\{S_1, S_2, \ldots, S_k, \ldots, S_{n-1}, S_n\}$ we have that the continuation c from n to k can be extended to c' the continuation from n to $k-1$ illustrated here:

$$S_1, S_2, \ldots, \underbrace{S_{k-1}, \overbrace{S_k, \ldots, S_{n-1}, S_n}^{c}}_{c'}.$$

6.4.1 The Semantic Function on While

We define the semantic function as follows:

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

We that this function is partial because some statements may never terminate. We have the following:

$$\begin{aligned}
\mathcal{S}(x = a)(c)(s) &= c(s[x \mapsto \mathcal{A}(a)(s)]),\\
\mathcal{S}(\texttt{skip})(c)(s) &= c(s),\\
\mathcal{S}(S_1; S_2) &= \mathcal{S}(S_1) \circ \mathcal{S}(S_2),\\
\mathcal{S}(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2)(c) &= \texttt{cond}(\mathcal{B}(b), \mathcal{S}(S_1)(c), \mathcal{S}(S_2)(c)),\\
\mathcal{S}(\texttt{while } b \texttt{ do } S) &= \texttt{FIX } F \text{ (where } F(g)(c) := \texttt{cond}(\mathcal{B}(b), \mathcal{S}(S)(g(c)), c)).
\end{aligned}$$

As there is more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{cs}^{\mathbf{While}}$.

6.4.2 The Semantic Function on Exc

Exception environments we define exeception environments env_E of type $\mathbf{Env}_E : \mathbf{Exception} \to \mathbf{Cont}$.

We define the semantic function as follows:

$$\mathcal{S} : \mathbf{Stm} \to \mathbf{Env}_E \to (\mathbf{Cont} \hookrightarrow \mathbf{Cont}).$$

We that this function is partial because some statements may never terminate. We have the following:

$$\begin{aligned}
\mathcal{S}(x = a)(env_E)(c)(s) &= c(s[x \mapsto \mathcal{A}(a)(s)]),\\
\mathcal{S}(\texttt{skip})(env_E)(c)(s) &= c(s),\\
\mathcal{S}(S_1; S_2)(env_E) &= \mathcal{S}(S_1)(env_E) \circ \mathcal{S}(S_2)(env_E),\\
\mathcal{S}(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2)(env_E)(c) &= \texttt{cond}(\mathcal{B}(b), \mathcal{S}(S_1)(env_E)(c), \mathcal{S}(S_2)(env_E)(c)),\\
\mathcal{S}(\texttt{while } b \texttt{ do } S)(env_E) &= \texttt{FIX } F\\
&\quad (\text{where } F(g)(c) := \texttt{cond}(\mathcal{B}(b), \mathcal{S}(S)(env_E)(g(c)), c)),\\
\mathcal{S}(\texttt{begin } S_1 \texttt{ handle } e : S_2 \texttt{ end})&\\
(env_E)(c) &= \mathcal{S}(S_1)(env_E[e \mapsto \mathcal{S}(S_2)(env_E)(c)])c,\\
\mathcal{S}(\texttt{raise } e)(env_E)(c) &= env_E e.
\end{aligned}$$

As there is more than one semantic function (for our different types of semantics) we denote this function as $\mathcal{S}_{cs}^{\mathbf{Exc}}$.

7 Axiomatics Semantics

These semantics revolve around formalising properties of a program into **assertions** which are triples of the form:

$$\text{Precondition} \qquad \text{Program} \qquad \text{Postcondition.}$$

We will consider partial and total correctness semantics:

Partial Correctness The assertion (**{P} S {R}**) means that if **S** terminates with precondition **P** then **R** holds immediately afterwards.

Total Correctness The assertion (**[P] S [R]**) means that **S** always terminates with precondition **P** and **R** holds immediately afterwards.

7.1 Axioms and Rules

We define a rule which is an assertion a (the conclusion) and a set of assertions $a_1, \ldots, a_n$ (the premises) written like so:

$$\frac{a_1 \cdots a_n}{a}.$$

This means, to show a, it is sufficient to show $a_1, \ldots, a_n$. A rule with no premises is an axiom.

7.2 Partial Correctness Schemata

[skip] $\quad \mathbf{\{P\}}$ skip $\mathbf{\{P\}}$,

[assignment] $\quad \mathbf{\{P(a)\}}\ x = a\ \mathbf{\{P(x)\}}$,

[composition] $\quad \dfrac{\mathbf{\{P\}}\ S_1\ \mathbf{\{R\}} \qquad \mathbf{\{R\}}\ S_2\ \mathbf{\{Q\}}}{\mathbf{\{P\}}\ S_1; S_2\ \mathbf{\{Q\}}}$,

[conditional] $\quad \dfrac{\mathbf{\{P} \wedge b\mathbf{\}}\ S_1\ \mathbf{\{Q\}} \qquad \mathbf{\{P} \wedge \neg b\mathbf{\}}\ S_2\ \mathbf{\{Q\}}}{\mathbf{\{P\}}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \mathbf{\{Q\}}}$,

[while] $\quad \dfrac{\mathbf{\{P} \wedge b\mathbf{\}}\ S\ \mathbf{\{P\}}}{\mathbf{\{P\}}\ \text{while } b \text{ do } S\ \mathbf{\{P} \wedge \neg b\mathbf{\}}}$,

[consequence] $\quad \dfrac{\mathbf{\{P'\}}\ S\ \mathbf{\{Q'\}}}{\mathbf{\{P\}}\ S\ \mathbf{\{Q\}}}$ if $\mathbf{P}$ entails $\mathbf{P'}$ and $\mathbf{Q'}$ entails $\mathbf{Q}$.

7.3 Total Correctness Schemata

[skip] $\quad [\mathbf{P}]$ skip $[\mathbf{P}]$,

[assignment] $\quad [\mathbf{P(a)}]\ x = a\ [\mathbf{P(x)}]$,

[composition] $\quad \dfrac{[\mathbf{P}]\ S_1\ [\mathbf{R}] \qquad [\mathbf{R}]\ S_2\ [\mathbf{Q}]}{[\mathbf{P}]\ S_1; S_2\ [\mathbf{Q}]}$,

[conditional] $\quad \dfrac{[\mathbf{P} \wedge b]\ S_1\ [\mathbf{Q}] \qquad [\mathbf{P} \wedge \neg b]\ S_2\ [\mathbf{Q}]}{[\mathbf{P}]\ \text{if } b \text{ then } S_1 \text{ else } S_2\ [\mathbf{Q}]}$,

[while] $\quad \dfrac{[\mathbf{P}(z+1)]\ S\ [\mathbf{P}(z)]}{[z \in \mathbb{N}, \mathbf{P}(z)]\ \text{while } b \text{ do } S\ [\mathbf{P}(0)]}$ if $\mathbf{P}(z+1)$ entails b and $\mathbf{P}(0)$ entails $\neg b$,

[consequence] $\quad \dfrac{[\mathbf{P'}]\ S\ [\mathbf{Q'}]}{[\mathbf{P}]\ S\ [\mathbf{Q}]}$ if $\mathbf{P}$ entails $\mathbf{P'}$ and $\mathbf{Q'}$ entails $\mathbf{Q}$.

8 Interpretation

Interpretation is one-stage execution, the code is evaluated at runtime.

Most code interpretation is done as expected (refering to the earlier chapters) but there are some key concepts to consider.

8.1 Let Expressions

Let expressions have the form:

```
let x xexpr expr
```

where `x` is the variable we are setting the value of, `xexpr` is the expression of which we are setting `x` to, and `expr` is the expression in which `x` takes on the value described by `xexpr`.

Variables defined in let expressions take precedence over variables in the environment. However, there is potential (when substituting free variables) for variables to be captured by our let expressions when they shouldn't. So, when we substitute variables, we should use some method (like a counter) to make sure substituted variables are unique and thus, are not captured.

9 Compiling

Compiling is two-stage execution, the code is evaluated into machine code - then it can be evaluated.

9.1 Variables

Variable's symbolic names are lost on compilation and replaced by indices refering to the distance to the variable binding. In this case, our environment becomes a list of values which we use our variable indices to index.

9.2 Stack Instructions

We form a set of stack instructions:

Instruction	Stack before	Stack after	Description
`SCSTI` n	$\ldots$	$n, \ldots$	Pushes
`SVAR` n	$\ldots$	$s[n], \ldots$	Indexes the stack
`SADD`	$n_2, n_1, \ldots$	$n_1 + n_2, \ldots$	Adds
`SSUB`	$n_2, n_1, \ldots$	$n_1 - n_2, \ldots$	Subtracts
`SMUL`	$n_2, n_1, \ldots$	$n_1 \cdot n_2, \ldots$	Multiplies
`SDUP`	$v, \ldots$	$v, v, \ldots$	Duplicates
`SPOP`	$v, \ldots$	$\ldots$	Pops
`SSWAP`	$v_2, v_1, \ldots$	$v_1, v_2, \ldots$	Swaps

The element remaining at the top of the stack at the end of computation is the result of the computation.

9.3 Lexers

The purpose of a lexer is to break down a string of characters (a written program) into tokens. Tokens can be keywords, operators, special symbols, etc. with identifiers and number literals formed compositionally.

Lexers are generated using a specification which describes the format of tokens.

9.4 Parsers

Parsers check that grammar (syntax) is respected, arranging tokens into a syntax tree with the leaves as tokens and other nodes as operators.

Like with lexers, parsers are generated using a specification which describes well-formed streams of tokens.

9.4.1 Ambiguity

We say that a grammar is ambigious if it has multiple derivation trees. To circumvent this, we assign precedence and associativity to operators. Higher precedence operators appear closer to the leaves.

9.4.2 LR and LL Parsing

LR parsing parses from left to right, making derivations from the right-most non-terminal symbol. This corresponds to bottom-up parsing, which is difficult to hand-write but requires no grammar transformations.

LL parsing parses from left to right, making derivations from the left-most non-terminal symbol. This corresponds to top-down parsing, which is easy to hand-write but requires grammar transformations.

9.4.3 Parser State

The parser state is a stack of values consisting of parser state numbers and grammar symbols, terminal and non-terminal. The parser can perform the following actions on the state:

- Shift, read a symbol in and put it on the stack,
- Reduce, take grammar rule symbols off the stack and replace with the corresponding non-terminal, then evaluate the semantic action.

There can be ambiguity in the decision to shift or reduce, this can be solved by altering precedence and associativity.