Chapter 1

Semantic Introduction

You've got to mean it Harry

Voldemort

1.1 Introduction

There will be another section on operational semantics and their relation to compilers but for now, we will use operational semantics as a way to help us get introduced to a language. This will be relatively quick and just get us more used to how we can quickly convey what things mean in a new language.

1.2 The Quick Definition

If you don't want to read all of this (you really should), this is the very quick definition of what we are doing. **The goal: have a quick, concise, and easily expandable way to talk about what a line of code (or equivalent) means in a language as you learn it.** We can do so by constructing rules. There is a formal way and shorthand I use in my notes.

Suppose we have something like $1?2$. I want to say that this returns 5. I may say something like $1?2 \Rightarrow 5$. That if the line of code is "1?2", then the result is "5". Now I would want to generalize this for any 2 values so I would have a rule that says $a?b \Rightarrow c$. This rule tells you that if you start with the stuff on the left side of the $\Rightarrow$ you end up with the stuff on the right side. Now we don't know what the ? operator does, but I could tell you: it sums the square of the 2 numbers. There is a formal way to write this and it looks like:

$$\frac{A; a \Rightarrow d \qquad A; b \Rightarrow e \qquad d^2 + e^2 = c}{A; a?b \Rightarrow c}$$

This shows that we break everything down. If we start with a we end up with d. If we start with b, we end up with e. If we take the things we end up with d and e, and we perform the operation I described, we end up with some result c. c is the result of $a?b$. The A is called the environment. It acts as the lookup table for variables.

Now we may want to put a restriction on what the values of a and b could be. One type of restriction is a type restriction. Since we want to square the values of a and b, it would not make sense for us to say something like "hello"?"bye". So I may want to say something like $(a : int \ ? \ b : int) : int$. This says that a and b are an `int` type and the thing returned is also an `int` type. There is a formal way to say this:

$$\frac{G \vdash a : int \qquad G \vdash b : int}{G \vdash a?b : int}$$

Like A, G (which is called the context) is used to keep track of variables.
End of Quick Definition

1.3 Meaning

We use language as a means to convey information, or to communicate with others. A programming language is no different, we are just communicating with a computer instead of another person (though we kinda are communicating with other people since other people need to read our code. Comments are helpful about this). For spoken and written languages, we may use a dictionary to help us define words to help us convey what we want. For programming languages, we may have things like language documentation. In the programming language research space however, we have rule constructs that help us quickly and (somewhat) easily convey the information want. This is what we will use to help teach you languages quickly.

1.4 Meta and Target languages

When you define a word, you cannot use that word in the definition. You even need a good grasp of the English language to read some dictionary entries. The same is true for programming languages. We need a baseline understanding of a language to learn more about it, and we cannot really use the language itself to help define it. So we introduce the idea of a target language and a meta language.

The target language is the language we want to talk about or define. The meta language is the language we will use to talk about the target language. So if I wanted to talk about the Java programming language, I may use English as the language to describe what happens to someone who has never seen Java before. Since this class has some prerequisite classes or equivalent coursework, I will be using a mixture of English and the C programming language to help us talk about our target languages. Once we learn enough of the target language, I may use the target language itself as well (much like the English Dictionary uses English to define its own words).

I will show you what I mean if I wanted to teach you some programming language which I will name L for now. To start off, I will use English and the C programming language to tell you how Language L should work. As we learn more of language L, I will start using it to hep define more complicated parts of language L.

1.5 Rules

I mentioned we use rule constructs in the Programming Language Research space to talk about how to give meaning to things. They are called rules because they describe how something should act. We can also use these rules to make proofs about how a program should work or compute. The most basic of these rules are called axioms.

1.5.1 Axioms

An axiom rule will describe the most basic element of a language. Typically this is data: something like an integer. Axioms are rules about what things are with no justification because there is no need for it. For example, I may say something like the integer 3 does not need a justification of what 3 is. If I so choose, I could write a logical proof to convey this:

$$\frac{3}{\therefore \quad 3}$$

To be more accurate, I could remove the hypothesis, and just have the conclusion:

$$\frac{}{\therefore \quad 3}$$

Logic Rules to Semantic Rules: Changed Format

The rules we use for semantic meaning will look like logic rules but look slightly different. Let's first see what the logical rules look like before we see what the semantic rules look like.

In the case of logic rules, we are talking about derivations to get to a new conclusion. Maybe we start with p and we end up with q. From our logic rules, we know that proofs can be rewritten as a conjunction of the premises which imply the conclusion. For example:

$$\begin{array}{ll} & p \\ & p \Rightarrow q \\ \hline \therefore & q \end{array} \text{ can be rewritten as } ((p) \wedge (p \Rightarrow q)) \Rightarrow q$$

This describes what happens if we start with the premises p and $p \Rightarrow q$. For semantic rules, we want to talk about what should happen when we start with a line of code. So if we start with the line of code that is just 3, we should say we end up with just 3. Logically put $3 \Rightarrow 3$.

For our semantic rules, we will use this same idea but just change the format a bit. To describe that 3 means 3, we would use the following format:

$$\frac{}{3 \Rightarrow 3}$$

Now since the language L wants to have all integers in the language, we don't want to have to write a rule about every integer since that would take a really long time. Instead we can generalize this rule to the following:

$$\frac{}{n \Rightarrow n}$$

Assuming that n is any integer, this axiom describes that all integers just are in our language. I want to add two other axioms in our language L:

$$\frac{}{WIN \Rightarrow WIN} \qquad \frac{}{FAIL \Rightarrow FAIL}$$

So in this L language there is something that is WIN and there is something $FAIL$ that just are themselves. There is no justification, they just are.

1.5.2 Non-Axioms

If axioms are things which need no justification as to what they mean, non-axiom rules need justification. For example, suppose the following is a valid line of code in language L:

```
SUM OF 3 AN 4
```

I would need to tell you what this means by giving a rule about it. Logically, my rule would look something like:

	`SUM OF 3 AN 4`	start
	$3 \Rightarrow 3$	axiom
	$4 \Rightarrow 4$	axiom
	$3 + 4 = 7$	In C, 3 + 4 is 7
$\therefore$	7	

Now you may be looking at this and say that this is not logically valid. And you would be right. However, we are not talking about logical rules here and we are providing the rule that defines behavior. That is, we are giving the defining rule which makes something valid - so meta! So much like how we just define a Modus Ponens rule that is backed by a truth table, we are defining the "SUM OF" rule that is backed by the compiler/interpreter/machine.

To generalize this rule, and to put it in the format we will use in this class, we will start by putting the starting point on the bottom:

$3 \Rightarrow 3$	axiom
$4 \Rightarrow 4$	axiom
$3 + 4 = 7$	In C, 3 + 4 is 7
`SUM OF 3 AN 4` $\Rightarrow 7$	

We then will replace every particular value with a variable so it is generalized to any line of code we could come up with with what we know of language L so far.

$$\frac{\begin{array}{c} a \Rightarrow c \\ b \Rightarrow d \\ c + d = e \end{array}}{\texttt{SUM OF } a \text{ AN } b \Rightarrow e}$$

Note that the first 2 lines are being defined with the previously defined rules of language L, whereas the 3rd line is being defined with the meta language of the C programming language. With this generalization we can now use this rule to describe any line of code in language L that uses the "SUM OF" command. For example:

$$\frac{\begin{array}{c} 42 \Rightarrow 42 \\ 25 \Rightarrow 25 \\ 42 + 25 = 67 \end{array}}{\texttt{SUM OF } 42 \text{ AN } 25 \Rightarrow 67}$$

1.5.3 Nesting Rules

As we learn more rules for our language, we will start writing larger and larger proofs. We will have to start nesting them to have a full understanding of our program. For example:

```
SUM OF 42 AN (SUM OF 12 AN 13)
```

This line of code could be allowed by our rules. Why? Let's take a look back at the generalization:

$$\frac{\begin{array}{c} a \Rightarrow c \\ b \Rightarrow d \\ c + d = e \end{array}}{\texttt{SUM OF } a \text{ AN } b \Rightarrow e}$$

In particular, let's look at the second line. The second line says if we start with b, we end up with c. In the previous example we put an axiom there, but we haven't said there was any restriction that it had to be an axiom. There a few ways we could add restrictions which I will talk about later, but for now if we allow for non-axioms here, then our nested line of code would have a proof that looks like the following. Note: I added justifications for clarity.

$$\frac{\begin{array}{rl} 42 \Rightarrow 42 & \text{Axiom} \\ \text{SUM OF } 12 \text{ AN } 13 \Rightarrow 25 & \text{SUM OF rule} \\ 42 + 25 = 67 & \text{In C, } 42 + 25 = 67 \end{array}}{\texttt{SUM OF } 42 \text{ AN SUM OF } 12 \text{ AN } 13 \Rightarrow 67}$$

If we then expanded out the inner SUM OF rule, we would get the following:

$$\frac{\dfrac{\begin{array}{r} 42 \Rightarrow 42 \\ 12 \Rightarrow 12 \\ 13 \Rightarrow 13 \\ 12 + 13 = 25 \end{array}}{\text{SUM OF } 12 \text{ AN } 13 \Rightarrow 25} \\ 42 + 25 = 67}{\texttt{SUM OF } 42 \text{ AN SUM OF } 12 \text{ AN } 13 \Rightarrow 67}$$

Now since the subcomponets are difficult to differentiate this way, I will move sub-proofs to be horizontl rather than put them together vertically:

$$\frac{42 \Rightarrow 42 \qquad \dfrac{12 \Rightarrow 12 \qquad 13 \Rightarrow 13 \qquad 12 + 13 = 25}{\text{SUM OF } 12 \text{ AN } 13 \Rightarrow 25} \qquad 42 + 25 = 67}{\texttt{SUM OF } 42 \text{ AN SUM OF } 12 \text{ AN } 13 \Rightarrow 67}$$

This is the format we will use in the class: sub-components moved to horizontal left to right, every sub-components expanded to their target language axioms or to their Meta-Language description.

1.6 Rules of Types: An alternative Meaning

We started out and just assumed that we gave meaning through what things are in terms of values. However, semantics describes what something means. In programming languages, a lot of what something means is dependent on its type. Suppose we had the binary string "01000001". This string could mean "65" if it was an int, or it could mean "A" if a ASCII Character. Let's see what our previous defined rules would look like if we had them define meaning via type, rather than value:

$$\frac{}{n : numbr} \qquad \frac{}{WIN : troof} \qquad \frac{}{FAIL : troof} \qquad \frac{a : numbr \qquad b : numbr}{\text{SUM OF } a \text{ AN } b : numbr}$$

In this case, notice that all integers are of type *numbr*, that WIN and FAIL are troof types, and that SUM OF only works on *numbr* types (Language *L* would not allow you to SUM OF troof values). That is, something like the following is an illogical (invalid) proof with the given rules and hence, not allowed in language *L*.

$$\frac{\text{WIN} : troof \qquad 7 : numbr}{\text{SUM OF WIN AN } 7 : numbr}$$

Also note that I put a particular value in the top part of the SUM OF rule.

1.7 Variables, the Environment, and the Context

For either set of rules, if we introduce variables into the language, we would need some way to keep track what the variables are bound to. We should be aware of the something like a lookup table that keeps track of what values and variables are paired together. When we encounter a variable, we would lookup the variable in this table and whatever the table says is bound to the variable is the result. So suppose we put this process into the English metalanguage and added it to our rules.

$$\frac{\text{lookup } x \text{ in } A = v}{A; x \Rightarrow v} \qquad \frac{\text{lookup } x \text{ in } G = t}{G \vdash x : t}$$

In this class, the convention is that the table that stores variable and value pairs is called *A* or the environment, and the table that stores variable and type pairs is called *G* or the context. Conventionally, we also separate the environment from the code with a semicolon(;), and we separate the context and the code with a $\vdash$.

Aside from looking up values and types in the environment or context, we should also have a way to add things to the environment or context. Let's add a rule for this.

$$\frac{A; b \Rightarrow v \qquad \text{add } (x, v) \text{ to } A}{A; x \text{ R } b \Rightarrow void} \qquad \frac{G \vdash b : t \qquad \text{add } (x, t) \text{ to } G}{G \vdash x \text{ R } b : None}$$

In this example, we see whatever *b* is (value or type, whichever is appropriate), and then add the return, bound to the variable to the environment or context to be used later. We also see the return value and type are void or None. This is because unlike something like SUM OF, we don't really have a return value or type when we do assignment. It is possible to have one, it is just these rules that does not have one.