COMP317 Notes - Week 1

February 6, 2016

Contents

$$F = ma \tag{1}$$

Figure 1: Force = Mass x Acceleration

1 Lecture 1 - Hello COMP317

1.1 Semantics and Syntax

This module is based on Semantics, so it is important to know what semantics is and why you would want to use/understand it.

Semantics is 'The meaning of things.' Semantics are most commonly used within languages. As a mild example, "The man ran" and "The man runs" are semantically correct, but "The man run" is not - it is the modification of the verb 'run' which provides the correct semantics.

The written form of semantics is known as 'Syntax' (in this module at least). If you have ever compiled your own code before and come across 'Syntax errors,' this means that you have written code that the compiler doesn't understand as the offending code is not a part of the language's syntax.

1.2 A mild warning

This module has a fairly high amount of formality in an attempt to define the semantics of languages. I'll attempt to make it as easily readable as possible but it's no guarantee that it won't give you a headache afterwards.

1.3 Formal Semantics

Formal Semantics are often a representation of Mathematical Models. By formally defining these models, it becomes easier to understand both the model and semantics in general. As an example:

You can make this equation sound simpler by saying "If you kick something, it moves. If you kick it harder, it moves further."

With formal semantics, you can build models (or 'worlds') and then query and reason with them in order to understand a particular problem, or syntax. Semantics comes in three main variants:

- Operational : How something is evaluated. Example: The method in which the Turing Machine works
- Denotational: What the program is and it's functions (input/output rules). Example: Expressions = Numbers
- Axiomatic : The logical inferences of the semantics. Example:

$$\{(y+1) < 0\}x = y + 1$$

Operational Semantics also outline the limits of what computers can do. As an example, it is difficult to test for bugs in recursion. There currently is no way for a machine to verify if there are bugs by outputting "yes" or "no"

1.4 Computers (in a sense)

For the purposes of this module, a computer is something which can perform algorithms on data structures in the form of programs. Broken down:

- Algorithms : A sequence of steps (which can modify)...
- Data Structures: Things which can store information - numbers, text etc. (in the form of)...
- Programs : Algorithms which a computer can understand.

These days we rarely, if ever, write in machine code. There are multiple layers of abstraction which make what is written easier to understand for humans. Writing in assembly can be considered difficult for those who do not easily understand the syntax, whereas writing in Java or C is considered an easier task.

The main changes are only differing syntax and the levels of abstraction once compiled/interpreted it makes no difference to a computer as it is still modifying memory addresses. A high voltage could be considered the value '1' in binary for a machine, low voltage being '0.'

Taking into account the 'Laws of Thought'[5], the main thing that differs between two programming languages are the semantics/syntax.

1.5 A Quick History Lesson

A big turn for computing in general was the Turing Machine. This machine formalised what computers are capable of.

We can compare the Turing Machine to the Java Virtual Machine (JVM). Both are simple machines - JVM was designed this way so that it is easy to port to other Operating Systems. The compiled Java Bytecode could be considered the tape that was fed into the Turing Machine. Java itself is a complicated language due to the amount of abstraction, but is broken down so that the JVM can understand the programs due to the Bytecode. This is how Java can adopt the slogan "Write once, run anywhere."

The 'Church-Turing Thesis[3]' worked on and defined recursive functions and something known as the 'Lambda Calculus[4]'. Lambda Calculus played a large part in the development of Functional Programming Languages, such as Haskell[1] or LISP, as well as Imperative Languages, but was a smaller influence.

1.6 "Hello Maude"

Maude is the programming language that will be used within the module to define the syntax and semantics for the use on computers. This makes the theory work executable, so long as you have correctly written it. Maude is a logic-based, Functional programming language that is interpreted, instead of compiled.

Denotational semantics is written in Maude to then allow Operational semantics (term-rewriting) to be given to the interpreter.

1.7 This is great and everything, but why do all this?

One main reason is the intellectual satisfaction of solving a problem or successfully modelling an algorithm. By understanding how to model a problem in a different language/paradigm, you are able to use this knowledge to better understand problems later in your life. Semantics blends Mathematics with Computer Science.

Another reason would be Language Design. There are many languages currently available and they all need semantics to both define themselves as well as make each language separate from others.

Formal methods is based on semantics, and helps verify the algorithms and theories that you are attempting to test. The US Army, for example, requires proof that the algorithm you're offering to them works for any and all applicable cases without error - formal methods and semantics can help with this. Things to look at would be AVL Trees and Scanning techniques for metamorphic viruses

2 Lecture 2 - Starting Semantics (SS)

2.1 There are many forms like it, but this one is Backus

John Backus and Peter Naur developed a way in which to help describe syntax in a context-free manner. This is known as the 'Backus-Normal-Form' or the 'Backus-Naur-Form' (BNF)[2]. For easing into semantics without the need of a computer, BNF will be used to help describe the semantics of a language. As this is theoretical, we will be defining our own simple language to demonstrate the capabilities of knowing semantics and syntax.

2.2 Language crash course

Language can be described as a set of strings where each string defines the format in which you can call a command. This would be why you may get syntax errors when you compile incorrect code as you are using strings which do not fit within the language. By defining the language, you are defining the set of strings that describe it's capabilities. This is known as the 'word problem.'

2.3 101 (Read: 5) Dalmations

2.3.1 The Problem and it's definition

As a way to describe the importance of semantics, we will define the binary system through BNF. Don't worry - it's simple.

Binary is a base-2 number system which uses two possible states to indicate particular numerical values - 0 and 1. These can be strung together to make larger numbers by having each 'column' represent a particular base-10 number. It goes up in multiples of two, so:

1101 \ 8 + 4 + 1 = 13 / 8421

where '1' means that that column is 'true' so should be counted. That is why the above binary's value is 13 and not 15. We can therefore see that the language of binary numbers is this:

$\text{Lang}^{\text{BinNum}} = \{0, 1, 10, 11, 100, \dots\}$

By splitting this up, we can create another language which defines the two possible values that the binary numerals can contain:

$\text{Lang}^{\text{Digit}} = \{0, 1\}$

2.3.2 Sidetrack: Variables and Expressions

If you have spent time coding in a language such as Java or C, you should be able to state which of the following two statements are incorrect:

1. i = i + 1;
2. i + 1 = i;

You would be correct if you yelled out loud that the second statement is wrong. You may not know why though - only that it is considered a syntax error. This is because in languages such as Java and C, you cannot have an 'expression' on the left side of a statement, only declarations and variables (I'm sure that there are others but for this example they are the only two worth noting). As 'i + 1' is an expression, it would be syntactically wrong if it appears on the left side of the assignment character (=) and therefore it is not a part of the language that you are using.

2.3.3 formally declaring the Languages

Moving into BNF, we can define Digits with the following format:

<Syntactic Category> ::= (value) [can be] which becomes this:

<Digit> ::= 0 <Digit> ::= 1 Nothing else is a <Digit>.

So now we have successfully defined the Language of digits, stated in the 'Problem and it's definition' section. This can also be written in one line using the OR symbol in BNF (|):

<Digit> ::= 0 | 1

As we are defining binary, only 0s and 1s appear, so we can define a new syntactic category for Binary Numerals using Digit:

<BinaryNumeral> ::= <Digit>

This is fine for the first two values of binary, but we cannot currently get values larger than 1 (10, 11 etc.). With the Binary Numeral language being a set which is infinite, we cannot exhaustively define the category like what was done with the Digit category. Therefore we have to use recursion in order to help define Binary Numerals. This can be done like so:

<BinaryNumeral> ::= <BinaryNumeral>0 <BinaryNumeral> ::= <BinaryNumeral>1

This finishes the declaration of Binary Numerals in BNF - the recursive declarations allow the value '1101' with the following deductions:

1 –> <Digit> 1 –> <BinaryNumeral> 11 –> <BinaryNumeral>1 110 –> <BinaryNumeral>0 1101 –> <BinaryNumeral>1

These declarations aren't quite as clean as we're able to currently make them though - as we are able to place the string '0' and '1' to the end of a Syntactic Category, we are also able to attach multiple categories on the right side of a declaration in BNF. Using the OR operator, we are able to slim down the entirety of binary numerals in just two declarations:

<Digit> ::= 0 | 1 <BinaryNumeral> ::= <Digit> | <BinaryNumeral><Digit>

The entire binary system in just two declarations. That's not bad.

2.3.4 Functionality would be nice though

So we have defined binary numerals but with these declarations by themselves, we cannot show the link between '1101' and '13.' This is where functions come in. Functions take some form of input, and generate output. A more formal definition would be that Functions define the relationship between an input and an output. It's important to note that syntax doesn't equal output - just in case there's any confusion.

We can state explicitly that 1101 stands for 13 with the following Function:

⟦1101⟧= 13 BinaryNumeral

This states that the binary numeral '1101' stands for the base-10 number '13.' We can use exhaustive definitions for the functions involving <Digit>s in the same manner we did with the Syntactic Category declarations:

⟦1⟧= 1 Digit

⟦0⟧= 0 Digit

Now that the basic definitions have been done for the basic digits, we can now work towards figuring out the functions for the Binary Numerals. To

help do this, we can use variable names within the double square brackets. First we define a Digit variable known as 'D' and a Binary Numeral variable known as 'BN'. Now we can write the functions:

⟦D⟧= ⟦D⟧BinaryNumeral Digit

⟦BN D⟧= ⟦BN⟧× 2 + ⟦D⟧BinaryNumeral BinaryNumeral Digit

The first statement is easy enough to understand - The function transforms a digit in a <BinaryNumeral> format into a Digit in the <Digit> format. The above functions can then turn these <Digit> inputs into a base-10 number. The second function however is a little more difficult to understand.

Taking the input of a BinaryNumeral and a Digit (such as '1101'), the value takes the Digit to one side and doubles the value of the Binary Numeral. I'm finding it difficult to explain so here is the break down:

⟦1101⟧= ⟦110⟧× 2 + ⟦1⟧BinaryNumeral BinaryNumeral Digit (6 × 2 + 1) –> 13

⟦110⟧= ⟦11⟧× 2 + ⟦0⟧BinaryNumeral BinaryNumeral Digit (3 × 2) –> 6

⟦11⟧= ⟦1⟧× 2 + ⟦1⟧BinaryNumeral BinaryNumeral Digit (1 × 2 + 1) –> 3

⟦1⟧= 1 Digit

We have now been able to define a language (Binary Numerals) through multiple Syntactic Categories (BinaryNumeral and Digit), and we have created Functions to transform Binary Numerals into base-10 numbers. Being able to do this is the basis of the entire module, so hopefully this has been written well enough for you to understand the basic concepts.

3 Lecture 3 - Can you call the operator?

3.1 Quick re-cap

There's a basic pattern to Languages. You go from semantics, use Denotational Functions which then create a Semantic Domain. This is how we went from Binary Numerals into the numbers we know and love.

3.2 Triangle games

Formal definitions play a part in this too. For the module, we will obtain syntax using BNF and our denotation will be created with Inductive definitions.

Now take this little block of code:

sum = 0; count = 0; while(count < 100) { sum = sum + count; count = count + 1 ; }

Hopefully this makes sense - the 'sum' and 'count' variables start off with the value of 0, then a while loop is called which will run a block of code until count is no longer lower than 100. The block of code adds count to the current value of sum, then count is incremented. If you run the first couple of values in your head, you should notice that the code is the triangle numbers. Now we're going to implement the basics of how to implement that code in BNF.

For the variable declarations, we can do this by defining:

<Variable> ::= <Expression> <Expression> ::= <Number> | <Variable>

This solves the first two lines of the sample code, providing we have properly defined <Number> and <Variable> (Which is a part of the problem sheets).

3.3 Sidetrack: Basic Maude

Variables can be defined in Maude by prefixing with the apostrophe symbol. If we were modelling the triangle numbers in Maude, the variables would be known as 'sum and 'count.

This doesn't really mean much at the moment, but keep it in mind for now.

3.4 Operators

Adding operators to the category is fairly easy - the following makes a valid expression have addition capabilities in a semantic sense:

<Expression> ::= <Expression> + <Expression>

This statement allows an expression to be made up of two expressions with an add symbol between them to be a valid expression. The same thing can be done using subtraction, multiplication, division... all operators. So long as you define Functions which perform what is declared, it wouldn't matter.

We can also have negative values in BNF, so we can inductively create a subtraction by adding a negative by using "-<Expression>", looking like so:

<Expression> ::= <Expression> + -<Expression>

If you found it easier, you could use an <Operator> Syntactic Category to store +, - etc.

Now we need to define the while section of the code, and for this we should create a Category called <BooleanExp>. A common boolean expressions would be equality, OR, NOT and greater than. For ease of use, we will use the formatting used in many C-based languages:

<BooleanExp> ::= <Expression> `= <Expression> <BooleanExp> ::` <Expression> || <Expression> <BooleanExp> ::= <Expression> > <Expression> <BooleanExp> ::= !<Expression>

Other concepts, such as AND and Less than, can be generated through manipulating the above statements.

3.5 Programming a program

In order to link the statements together, we need to define a program. An important thing in programs is to do nothing. If you think of threads waiting for a resource, they would have to do nothing in order to wait. For defining 'nothing,' this theoretical language will use the term 'skip':

<Program> ::= skip

We also need to define the syntax for assigning variables. We can use '=' but as a nod to Algol, I'll be using ':=' which would look like this:

<Program> ::= <Variable> := <Expression>

We also need to be able to link the statements together, but because each statement is a part of the <Program> Category, we can write:

<Program> ::= <Program><Program>

So now the statements can be linked together and it would be valid syntax. Whitespace wouldn't matter in this language as we have not explicitly defined any use for it, so it will be simply ignored.

3.6 Controlling Flow

We've successfully defined linking statements and assigning variables, but now we need to define the while statement. You may have noticed that for some declarations uses characters not wrapped in Angled Brackets. An example being in Lecture 2 with the function which turns BinaryNumerals into Numbers. So for writing the syntax of some conditional statements in BNF, it would be written like this:

<Program> ::= if(<BooleanExp>) { <Program> } else { <Program> } <Program> ::= while(<BooleanExp>) { <Program> }

Now we have defined all the types of statements found in the triangle code at the beginning of this chapter. For the moment, we are not defining the Functions but instead go into the theory of Functions, once we've brought all the BNF we've written for this chapter together.

3.7 Conclusive section

<Variable> ::= <Expression>

<Expression> ::= <Number> | <Variable> | <Expression> + <Expression> <Expression> ::= <Expression> + -<Expression>

<BooleanExp> ::= <Expression> `= <Expression> | <Expression> || <Expression> <BooleanExp> ::` <Expression> > <Expression> | !<Expression>

<Program> ::= skip | <Variable> := <Expression> | <Program><Program> <Program> ::= if(<BooleanExp>) { <Program> } else { <Program> } <Program> ::= while(<BooleanExp>) { <Program> }

And if you don't like the look of some of the concepts, such as denoting equality with '==', then you can write up something which follows BNF and is more appealling to you. So long as you follow BNF and have your Functions use the same Categories and syntax, it doesn't matter.

3.8 Sidetrack: Functional Theory

For programming languages, the names in which we give variables don't really mean anything. We could easily name 'count something different or confusing, such as 'x. Imperative languages often use a table which contains variable names and their corresponding values. These tables are known as 'State Tables.' If we have an example table below, we can look at how functions interact with the input/output:

State Table 'S': 'count |'sum ——+—— 3 | 24

If we have the Function $[\![$'count + 1$]\!]$(S), then the state table would change the Expression value of 'count to be 'count+1, which for this State, would be 4.

So now we've seen how to convert a simple program into Syntactic Categories within BNF, and how Imperative Languages use State Tables to store the values of variables, making the 'useful' naming of variables arbitrary.