CS4450 Lecture 5 Notes

Wednesday, August 29, 2018 3:04 PM

First HW assignment will be given on Friday!

We will be talking about boolean logic today.

<u>Separating Syntax and Semantics</u>

Syntax: How do we define precisely what the well-formed sentences of a language are?

Semantics: Given a well-formed sentence, what does it mean?

<u>Solving linear equation: 5x + 4 = 9</u>

Assume
    5x + 4 = 9 (i)
Subtract 4 from each side of Equation (i)
    5x = 5
Divide 5 from both sides.
    x = 1

Two rules:

1. "Subtract from both sides": kx + l = m -> kx = m - l
2. "Divide both sides" kx = l -> x = 1/k (k cannot equal 0)

Consider:
    5x + 4 = 9 -> 5x = 9 - 4 -> x = (9-4)/5

Question: Can we solve 3x + 5 + 6x = 0 with these rules?
    No because the syntax does not match either rule #1 or rule #2

<u>Derivation System (High Level)</u>
    Has some notion of a "sentence" or "formula"

-(E + -E) 2B || !2B

Rules for producing new formulae from existing ones

E -> -E (You don't think about what the E means. A chunk of syntax becomes another chunk of syntax)

Notion of "proof" or "derivation". Sequence of sentences:

S1,…,Sn

Where Si result of applying a rule to (members of) {S1,…,Si-1}

Simple expression language (also represented as a derivation system):

Defining syntax

E -> i (where i is some integer)
E -> -E (what this says is that if you have E, you can get -E)

(Note: -E is "not E", the - represents "not" in boolean logic)

Derivation: E -> -E -> --E -> --9 (really simple example of a context-free grammar)

If you look at E as the dumbest programming language, --9 is a valid "E" program.

While this is really simplistic, every programming language is defined by this. Literally every one of them is defined by a context-free grammar of this form.

Writing parsers (which is what these are) can be a challenge, because it's easy to write a context-free grammar that seems correct but isn't.

There are other types of derivation systems that we encounter in programming languages, for example type systems.

In Haskell, the :: is pronounced "has type".

Example: If e :: int, then -e :: int. We can derive that if 9 :: int, -9 :: int.

Even very expressive, complex type systems like the one used in Haskell is expressed like this with this notion of derivation (the notion is purely syntactic).

Why is it important to have the notion be purely syntactic? Because it gives you machine checkability, which means it's computable (you can write a program that checks the syntax to determine if the program is syntactically correct or well-typed).

Defining meaning:

Definiton of length:

```
-- [a] is a list of items
length :: [a] -> Int

-- Here are the rules we would use to calculate length:
length [] = 0
length (i:is) = 1 + length is
        -- [x,y,z] (x is i and y,z is is)
```

In Digital Logic, you'll have logic gate diagrams that show logic flow.

An Equivalent Boolean Expression to the logic gate diagram on the slides:
(A and B) or ((B or C) and (C and B))

Propositional Logic:

Proposition: A statement that is either true or false (e.g. "It is raining", "Socrates was Greek", etc.)
Propositional Sentences: e.g., Let p and q stand for "it is raining" and "the street is wet", respectively, then 'p implies q' is a propositional sentence.

The propositional calculus is the simplest form of mathematical logic.

A propositional formula has one of the following forms:
A propositional variable
A negation

We can write the syntax using a context-free grammar now.