# COMP 325 - Interpreter 2- Functional Boolean Arithmetic

Fall 2015

**Abstract**

Your second interpreter still involves a single data type but includes Functions and Conditionals. In addition to implementing the language, you're asked to write a small, but meaningful program in your language. Languages are meant to solve problems, so we should see how our language does at real-world problems and not just contrived test cases.

**Due Tuesday, 10/6**

For this interpreter we'll be building a functional language for boolean arithmetic and then consider its usage for expressing and simulating boolean logic circuits.

Your language will provide:

- The unary *not* operator
- The *n*-ary versions of the operators *and*, *or*, *xor*, *nor*, *nand*, *xnor*
- A multi-branch conditional expression a la *if elseif ... else* or Racket/Scheme *cond*
- n-ary functions

Once you've completed the language, use it to write a program to carry out a moderately involved digital logic circuit or boolean formula. I recommend doing something like a 4-bit adder. The catch with the adder is that you'll have to write 5 circuit procedures, one for each bit and one for the carry, because our language can't operate on arrays of bits. If you're not familiar adders, try looking here: `https://en.wikipedia.org/wiki/Adder_(electronics)`. The goal is to get some feel for the pragmatics of your language and better understand it from the user's perspective. If you're not feeling the adder, you can find lots of similar circuits but looking for *digit logic circuits* on Google web and image search.

## 0.1 Implementation Notes

You have many issues to deal with and choices to make as the language implementer. We'll discuss some of them in class. Many of them are also addressed in the text. Here are a few key notes:

- Language Usage Patterns
  We should give some thought to the means by which a user interacts with our language. This is a bit forced given the bootstrapping we're doing on Pyret, but we can still capture the essence of how our language might be used. One option is very script-like. Users intend to execute a specific expression and write a series of function definitions to help abstract away elements of the expression. This is fine for fixed expressions but requires that the program be re-written if you wish to carry out different computations. Another usage paradigm is the main function. Users write only function definitions, one of which must be for a function named *main*. Executing the program then means executing main. This later option lets us think about CLI like programs that accept arguments when they're executed. You'll be given a starter file that gives you a structure in which both of these programming practices can be utilized.
- Parser Errors
  You do not have to strive for high-quality, helpful error messages in the parser, but you still need to catch all the errors yourself.

- Conditional Expression Syntax
  There are lots of standard syntax options available for multi-branch conditionals. You can use the *if*,*else if*, and *else* keywords or something like the *cond* from BSL Racket `http://docs.racket-lang.org/htdp-langs/beginner.html`. In all cases the multi-branch conditional can be desugared to the standard *if..then..else* two-way branch.

- Desugaring $n$-ary boolean operators
  An $n$-ary operator can often be desugared to nested binary operators. Doing so requires a strict binary version of the operator in the target language. We've also seen that some boolean operators can be desugared to short-circuiting conditionals. With this language you can go both ways.

- Desugaring to Universal sets of binary operators: {*xor,and*}, {*and,or,not*}, {*nand*}, etc.
  There are several subsets of the binary operators that can be used to represent all the binary operators. This presents us with a clear desugaring option similar to what we had in the last interpreter.

- Lazy vs Eager evaluation
  You can try your hand at either an Eager or Lazy language. Just document your choice and realize that it will impact your tests and some data definitions.

- Efficient Substitution via Environments
  You must do substitution using the efficient, *environment* based substitution process discussed in the text.