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 program for simulating the results of a 4-bit adder (see below).

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.

1 4-Bit Adder Program

Full-Adder circuits compute the sum and carry of two bits. They can be combined to do general addition of binary numbers. We want to carry out 4-bit addition. Given two 4-bit numbers, a 4-bit adder computes the resultant 4-bit sum and a single carry bit. Our language only supports functions that produce a single bit output, so we must tweak the 4-bit adder to account for this. The boolean arithmetic of the full-adder is given on wikipedia along with a diagram for the 4-bit adder that should help `https://en.wikipedia.org/wiki/Adder_(electronics)`.

Write your program in the *main* function style such that main takes as arguments 11 bits. The first four bits are the first input to the adder and the second 4 are the second input. The last 3 bits determine the output of the program. If they are binary 0, then output the least significant bit of the sum, if they are 1 output the second least significant, and so on. If they are 4 to 7, then output the carry bit. Here's some tests to illustrate. Assume *prog* is the string form of the program.

```
check "4bit adder":
   prog = ...

   ## sum of 15+3 in least to greatest bit order
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 0,0,0]) is 0
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 0,0,1]) is 1
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 0,1,0]) is 0
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 0,1,1]) is 0
   ## carry of 15+3
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 1,0,0]) is 1
   run-main(prog,[list: 1,1,1,1, 0,0,1,1, 1,1,0]) is 1

end
```

It's useful to note that you can do multi-line strings in Pyret by surrounding the string with ``` (the key usually next 1, not apostrophe). This makes writing your program a bit easier as you can break it across lines. Here's a Pyret test that demonstrates some key properties of multi-line strings.

```
check:
  ## the newlines don't break s-expression reading
  S.read-s-exp(
  ```
```
```

```
  (this is
  a string)
  ```
)
  is
  S.read-s-exp("(this is a string)")

  ## the newlines are added to the string
  ## the spacing around the line break is added to the string
  ## but leading and trailing white space is ignored

  ## a rare case of not indenting!
  ```
  (this is
a string)
  ```
  is
  "(this is\na string)"

  ## space at the break
   ```
  (this is
  a string)
  ```
  is
  "(this is\n a string)"

  ## leading space
   ```  (this is
a string)   ```
  is
  "(this is\na string)"
end
```

2 Extra Challenge

Allowing n-ary functions introduces a few other issues to function application semantics. These things are required, but you're strongly encouraged to incorporate them in your language. They'll probably be required in the future.

- *Language Keyword Protection*
  Programmers should not name functions or identifiers after keywords used by the language. If I name a function *xor*, then nothing good will happen. If the name of my function argument is *true* and that's also a boolean literal, then how do we parse this? We should verify that all the programmer specified names in our programs do not conflict with the operator names and other keywords of our language.

- *Argument Checking functions*
  Functions expect a specific number of arguments and when they're not given that number it's an error. Our current language allows us to check this as a static property of the program. Prior to the interpreter, during parsing or before desugaring, we can check all function applications against their definitions in order to see if they are being given the correct number of arguments. We could also check this at run-time.

- *Function Overloading*
  Overloading a function means providing more than one definition for the name with the caveat that

the signature for each definition must be distinguishable for each function. In our current language, this means allowing multiple definitions where each definition allows a different number of arguments. We can have a unary, binary, and ternary *foo* function. Clearly this capability requires a change to interpretation: we must now fetch *all* the definitions for a given name and check each against the number of arguments being passed to that function.

- *Multiple Definitions*
  Programmers shouldn't provide more than one definition of a function. If there are two binary cases of a function named *foo*, then which one did the programmer intend to call when she did *(foo 0 1)*.