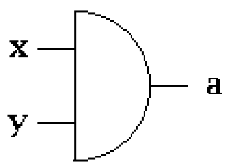
CSE 373 -- Autumn 1999 -- Programming Assignment #3

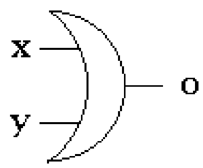
(*due Friday, November 12*)

Introduction: For this assignment, you will write a program that simulates simple circuits. This program will give you experience with Binary Trees (not Binary Search Trees) and Hash Tables, and will give you some idea of how simulators, compilers, and interpreters use these data structures to store and manipulate user-defined circuits, expressions, or code.

Simple Circuits 101: Two of the simplest building blocks for designing circuits are the "AND" and "OR" gates. Each of these gates takes two input signals. The input signals can have values 0 or 1. The gates compute the boolean "AND" and "OR" functions on their input signals (equivalent to the boolean `&&` and `||` functions in C++). To simplify large circuits, input signals are often labeled with identifiers (e.g., *i1*) with the assumption that all wires marked with the same identifier will carry the same value (0 or 1). In addition, these identifiers may be complemented by preceding them with a tilde (e.g., ~*i1*) which indicates that the signal's value should be reversed (equivalent to the C++ ! -- a 1 becomes a 0, and a 0 becomes a 1). The following diagram shows the AND and OR gates, and what their outputs are as a function of their inputs. The table on the right shows the definition of the complement notation.



the "AND" gate

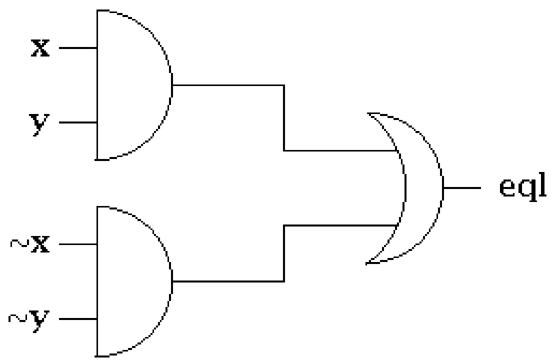


the "OR" gate

x	y	a	o
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

z	~z
0	1
1	0

These gates can be combined to form slightly more interesting functions such as the following circuit which tells whether two signals are equal (it will output 1 only when `x` and `y` are both 1 or when they're both 0).



a circuit that computes "equal"

As these circuits grow in complexity, they can be somewhat painstaking to evaluate for a given set of input values, so we'll write an interpreter that allows circuits to be specified and evaluated for particular inputs. Obviously a real-life circuit simulator would have other gates and functional units available, but we'll stick to just these three (the two gates and complement) for simplicity.

The BASIL Language: Users of your simulator will need a way to specify circuits, check them for correctness, and see what their output is. For this purpose, we will define the *Basic AND/OR-gate SImulation Language (BASIL)*. BASIL has four exciting commands:

- **s** <*output wire name*> = <*circuit definition*> -- sets a wire to have the specified circuit definition
- **p** <*wire name*> -- prints the circuit definition for the given wire
- **c** <*wire name*> -- check the value of the specified wire (0 or 1)
- **q** -- quit the simulator

A *circuit definition* is defined recursively as follows:

- It can be the constant values 0 or 1
- It can be a wire name (*e.g.*, x)
- It can be the complement of a wire name (*e.g.*, ~x)
- It can be a gate followed by two additional circuit definitions that specify its inputs (*e.g.* OR x ~y)

So for example, the BASIL circuit definitions for the diagrams above would be:

```
s a = AND x y
s o = OR x y
s eql = OR AND x y AND ~x ~y
```

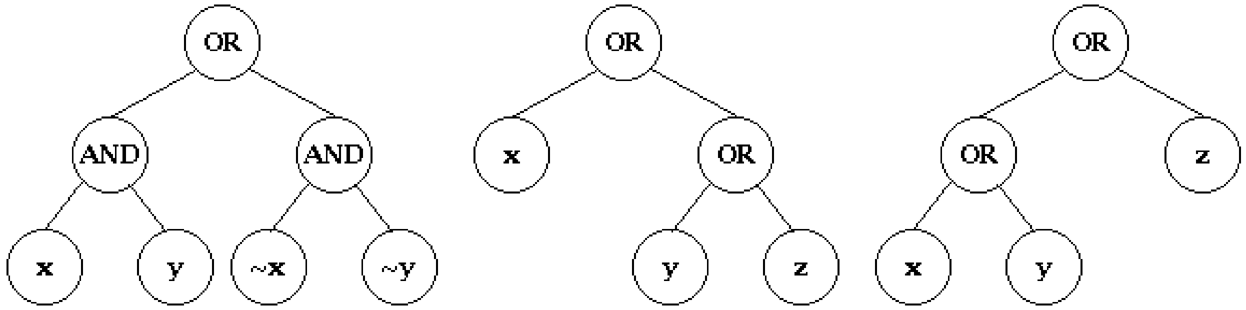
Some other circuit definitions could be:

```
s ident = AND 1 x
s xor = ~eql
s or3 = OR x OR y z
s or3v2 = OR OR x y z
s x = 1
s z = y
s notz = ~z
```

Here's an example of how a session with your interpreter might look. You can change the specific interface, but the functionality should be similar (bold and italics for emphasis only):

```
basil> eql = OR AND x y AND ~x ~y
eql defined
basil> s x = 1
x defined
basil> p eql
eql is defined as: OR AND x y AND ~x ~y
basil> c eql
ERROR: wire y undefined
basil> p y
y is currently undefined
basil> s y = 0
y defined
basil> c eql
eql has the value 0
basil> s x = 0
x redefined
basil> c eql
eql has the value 1
```

Implementation: A very reasonable way to implement such an interpreter is using a hash table and binary trees. The hash table will associate wire names (the keys) with circuit definitions (the data), so that wires may be easily looked up to print them out, check their values, etc. The recursive nature of the trees will be used to store the circuit definitions themselves. For example, the following diagram shows how the above definitions for eql, or3, and or3v2 would be stored as binary trees, respectively:



At a high level, your program should have a main loop that reads in a command, processes it, and then repeats the process for the next command. Each command is processed a bit differently:

- **s**et -- read the wire name, read the equal sign, read the circuit definition into a tree, and hash it into the table using the wire name as the key.
- **p**rint -- read the wire name, look it up in the hash table and print out its tree definition
- **c**heck -- read the wire name, look it up in the hash table and evaluate its tree. In doing so, you will generally have to recursively check wires that it uses.
- **q**uit -- exit the program

Note that as you're testing your implementation, you may get tired of typing in definitions over and over again. You ought to be able to direct a text file (containing a BASIL command per line, for example) into your program to be used in place of cin by specifying "< infile.txt" on the command line (or using *Project -> Settings -> Debug Tab -> Program Arguments Textbox* from within MSVC++). This can save you from having to type the same circuit definition over and over again.

Some of the design decisions you'll have to make include: what to store at the tree nodes, how to organize your hash table, how to deal with undefined variables and redefining variables, and what operations your binary tree will need to support.

This simulator has a fairly neat and tidy implementation: all of the tree functions are very short, elegant recursive functions. The evaluation of a wire involves mutual recursion since it requires evaluating a tree which in turn may require evaluating a wire. Again, this has a fairly short, elegant implementation. This is the type of assignment where spending a fair amount of time sketching out the recursive functions with pencil and paper can save lots of superstitious and panicked fiddling with code at the computer.

Weiss' ADTs: Weiss' ADTs will be somewhat helpful in completing this assignment, but not completely so. As always you may use other ADTs as long as they are consistent with the definitions we've discussed in class and don't solve this problem for you.

Weiss provides binary search trees, whereas we only need binary trees (we don't need any sort of searching capabilities). For this reason, it's probably wise to implement your binary tree from scratch

It'll only need to support a few operations which form the bulk of your assignment anyway, so this isn't like reinventing some well-defined ADT from scratch.

Weiss' hash tables are much more useful. He provides both separate chaining and quadratic probing implementations. His separate chaining implementation never resizes (the lists will just get longer and longer). His quadratic probing implementation rehashes when the load factor exceeds 50%. In my lecture, I described two hash table implementations: one took key and data as a single HashedObj parameter and only applied the hash function to a subset of its fields; the explicitly separated key and data. Weiss takes the first approach, so you can either use his implementation it in that way or modify it to take the other form (using it unmodified should work fine once you understand the approach).

Assumptions: You may assume that only legal BASIL commands will be entered: No need to worry about syntax checking, spacing, etc. You may also assume that no circuits will be defined such that a loop is formed (which would result in infinite recursion during evaluation). You may assume that wire names are only formed of letters and numbers if you wish.

Don't Panic/Don't Procrastinate: The description of this assignment is long in order to be complete and clear, not because the program itself is amazingly long or difficult. The interpreter should conceptually be simple after you've brained it out, though of course you'll have the normal bit of wrestling with C++ and dealing with details that are an inherent part of C++ programming. I recommend starting early and biting off small chunks at a time. If you want to brainstorm, the TAs or myself would be happy to help you flesh out the algorithms in office hours/lab hours/email. However, it'll be a more pleasant experience for both of parties if you enlist our help sometime before the last day or two before the assignment is due. Bus rides are a great time to sketch out algorithms on paper.

Competition: Just to keep those competitive spirits busy, the competition for this assignment will be to design the most interesting circuit in BASIL, where "interesting" is defined as "useful," "humorous," or "interesting." Your entry can be sent directly to Brad. You might also consider how you can extend BASIL or the interpreter to make it even more useful (though that's hard to imagine!). None of this will influence your grade, but is simply offered for those who can't get enough programming.

Turn-in: The turn-in procedure is the same as always: mail to c373@ms.washington.edu. Include a cover sheet, sample input/output, and the following annotations with your hardcopy:

1. The code that reads in a BASIL command and processes it
2. The code that reads in a circuit description
3. The code that prints out a circuit description
4. The code that evaluates a wire name
5. The code that evaluates a circuit description
6. The code that handles undefined wires
7. The code that handles redefining a wire
8. Anything else unique to your implementation
9. Any changes you made to Weiss' ADTs