Church Numerals

Amazingly, any function that can be computed at all, such as factorial(5) or sqrt(7) or piglatin(Scratch), can be computed using only 

and 

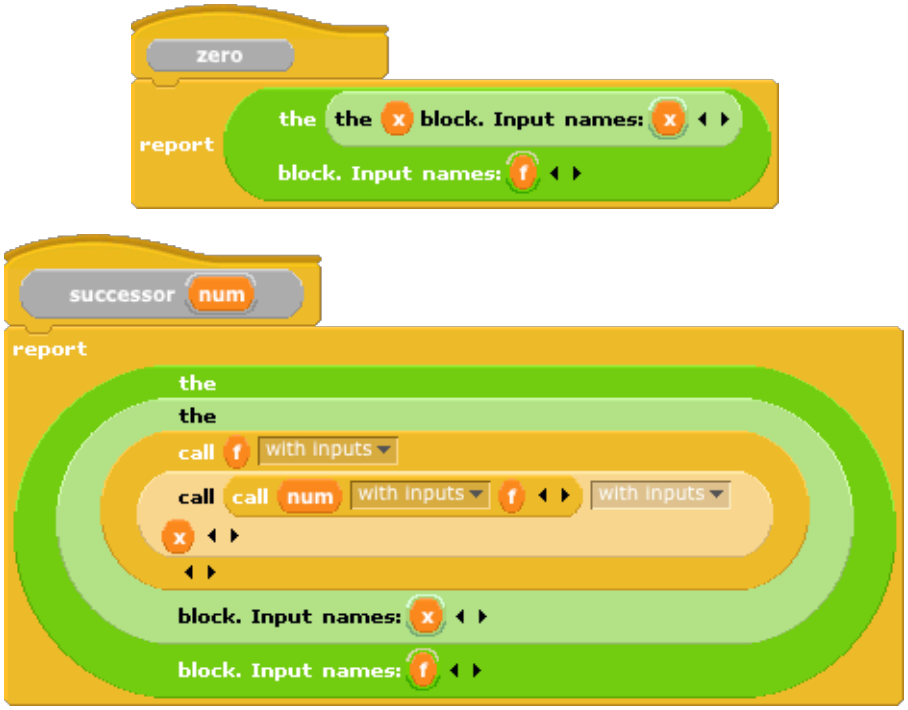
. The study of how to do this is "lambda calculus," a mathematical area invented by Alonzo Church. ("Lambda" is what the non-BYOB world calls THE BLOCK. In the notation used in lambda calculus, there's no need for a CALL block because following a function with an argument implies calling the former with the latter. Of course, since the only data type in lambda calculus is functions, the argument to a function is always itself a function!) You end up with long ugly programs, and nobody would use them in practice, but knowing that you could is useful for *reasoning about* programs. If you can prove something is true for programs made out of just lambdas, then you know it's true for real programs on real computers.

In this exercise we're going to invent arithmetic. As in the actual historical invention of numbers, we start with just the natural numbers (nonnegative integers). Once those work, you can extend arithmetic to include fractions, negative numbers, irrational numbers, and complex numbers. Although in lambda calculus there's no 

to create functions with names (if you want recursion, you have to figure out a way to let an anonymous block call itself!), for this project we're going to abbreviate, in the interest of writing readable programs, by creating named functions that aren't recursive.

Important: With one exception, noted below, you should never use the grey-border abbreviation for encapsulated blocks. Always use explicit THE BLOCK blocks when needed, and click the right arrow to give name(s) to the input(s) taken by the function you're creating.

We start by providing two named functions, ZERO and SUCCESSOR:



See what I mean about ugly programs? But the idea isn't that bad. Every number is a function that takes another

function, *f*, as input. It returns a function of *x* that computes *f(f(f(f(...f(x))))* with that number of calls to *f*. So, in particular, the number 0 returns a function of *f* that doesn't call *f* at all; what it returns is the identity function that returns its input unmodified. If we were going to cheat and write directly the number 3, it'd look like this:

This says that THREE is a function that takes a function *f* as argument, and returns the function of *x* that computes *f(f(f(x)))* — calling *f* three times.

But we don't have to build a library of named numbers, because we can make any number we want with the SUCCESSOR function, which takes a number as argument and returns the next number (NUM+1). Here's how it works: Suppose we have NUM. Then, for any function *f*, we can call NUM with argument *f*, and that gives us a function that calls *f* NUM times. The number NUM+1 should be a function of *f* that returns a function that calls *f* NUM+1 times. So we just have to call *f* one more time. Look in the definition of SUCCESSOR for the place where it calls NUM with input F, then see how it calls F with, as input, the result of calling NUM-of-F with input X.

Ex. 1: Convince yourself that 

reports a function that has the same behavior as THREE above.

Note: To help with debugging, you can use the following function, You call it with a Church numeral and it reports the ordinary number that corresponds to it. TRY isn't part of the Church numeral system, just a debugging tool:

Ex. 2: Using only ZERO and SUCCESSOR as helpers if necessary, write a SUM function that takes two numbers (that is, two Church numerals) and reports their sum, also as a Church numeral. In other words, given two numbers A and B, you want to report a function of *f* that reports the function that calls *f* A+B times on some input *x*. You can make SUM a named block, just for convenience, but you can't use recursion in defining it. (If you're thinking in terms of recursion, you're not taking advantage of what the numbers A and B mean.)

(Historical note: In traditional lambda calculus, you can't have a function of two inputs like SUM. All functions take

exactly one input. But you can write a SUM function that takes a number A as input, and reports *a function that takes B as input* and reports the sum A+B. This trick, turning a two-input function into a one-input function that returns a function of the second input, is called "Currying" after the logician Haskell Curry.)

Ex. 3: Similarly, write PRODUCT and EXPT (exponentiation) functions of two numbers.

Beyond this point we need a way to do conditional evaluation: IF-THEN-ELSE. First we need to invent Boolean (true/false) values:

Ex. 4: Invent IF-THEN-ELSE. This isn't a C-shaped (or E-shaped) block, because it's a reporter, not a command. Its first input should be one of the Booleans (TRUE or FALSE) defined above. (Presumably the actual block used in that input slot will usually not be a constant TRUE or FALSE, but rather some function call that reports TRUE or FALSE.) The other two inputs should be functions of no inputs. Your IF-THEN-ELSE should CALL only one of those functions. Note: This is the exception noted above; since the last two inputs are functions of no inputs (essentially just requests to defer the evaluation of the block they contain), you can use the grey-border shortcut notation for these inputs.

(Historical note: Lambda calculus doesn't need these no-input functions and CALLs to them, because it uses *normal order evaluation*, a system in which no expressions are evaluated until the last possible moment. BYOB uses *applicative order evaluation*, in which subexpressions are evaluated before any block is called. We are essentially faking normal order evaluation for the IF block. Don't worry if this note goes over your head. Oh, and of course in lambda calculus we'd have to Curry those two-input and three-input functions we're using here.)

Ex. 5: Invent ZERO?, our first predicate function; it takes a number (a Church numeral) as input and reports TRUE if and only if the number is ZERO.

Hint: ZERO? has to return TRUE or FALSE, that is, a function of two inputs that selects one of them. So your block will look something like this:

Ex. 6: Beyond this point we're going to need a small data aggregate, called a "pair," basically a two-item list. We need a two-input function CONS to construct a pair with the two inputs as its items, and we need selectors CAR for the first item of a pair and CDR for the second item of a pair. Here's CAR; you write the others.

Historical note: CONS, CAR, and CDR are the names for these functions in Lisp. "CONS" abbreviates "construct";

the other two names have to do with the particuar computer model on which Lisp was first implemented, in which the main internal register was divided into an Address part and a Decrement part. "CAR" abbreviates "Contents of Address [field of the] Register." These ridiculous-seeming names have survived over 50 years because they can easily be composed, so the function CDADDR is cdr(car(cdr(cdr(pair)))))!

Ex. 7: (This is the hardest exercise!) Invent PREDECESSOR, a function that takes a number NUM and reports the number NUM-1. Don't worry about what it reports if its input is ZERO.

Ex. 8: Using PREDECESSOR, invent DIFFERENCE, a function of two numbers A and B that reports A-B. Again, don't worry about what it reports if A<B.

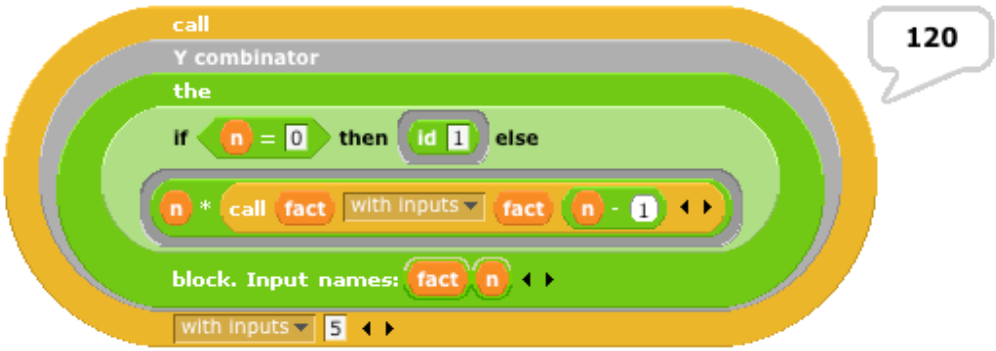
Ex. 9: Invent versions of AND, OR, and NOT that work on the Booleans given above.

Ex. 10: Using the results of the two previous exercises, invent the relational predicates LESSEQ?, GREATEREQ?, EQUAL?, LESS?, and GREATER?, each of which takes two Church numerals and reports a Boolean.

Ex. 11: (This is the other hardest exercise!) Time to invent recursion. That is, we need to invent a way for a function made with THE BLOCK, so it doesn't have a name and doesn't appear in the block palette, to call itself. Hint: If we don't have "Make a block," and we don't have "Make a variable," so there's no global naming, the only way we have to give something a name is to make it an input to the block:

(This isn't a Church-numeral example; to simplify the picture and focus only on the problem of recursion, the ordinary BYOB arithmetic operations are used here. But once we have recursion invented, you'll use it with Church numeral arithmetic.) What we need is a function that takes a two-input function like this one as its input, and reports a one-input function (corresponding to the input N above) that calls this function with *itself* as its first input.

(Historical note: The lambda-calculus solution to this problem, slightly different in that it has to take a Curried function as input, is called the "Y combinator.")



Ex. 12: Use the Y combinator to write a factorial function for Church numerals.

Ex. 13: Write a division function that returns a pair of Church numerals, one for the quotient and one for the remainder.