




Church Numerals

Amazingly, any function that can be computed at all, such as `factorial(5)` or `sqrt(7)` or `piglatin(snap)`, can be computed using only  and . The study of how to do this is "lambda calculus," a branch of mathematics invented by Alonzo Church. ("Lambda" is what the non-Snap! world calls gray rings. 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 you'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. 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! There are no global variables, either; if you want to use the value of an expression twice, you have to put the expression in the program twice. For this project, though, you're going to abbreviate, in the interest of writing readable programs, by using global variables to hold the functions. But you can't cheat by writing a call to itself in the definition of a function.

Snap! functions are named specially, separate from the use of variables to name other things. Once a function is defined, the block that represents it visually implies *calling* the function, with slots ready to accept inputs.



In this project, though, you'll make all function *creation* (lambda expressions) and function *calls* explicit:



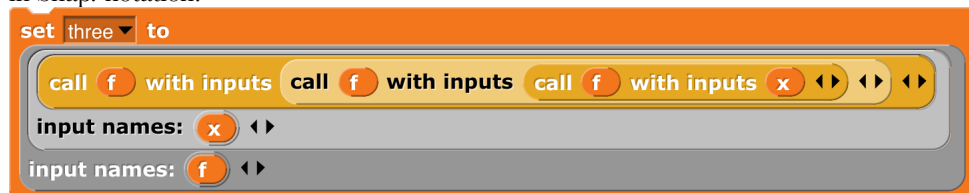
When you see  in the [starter project](#), you're supposed to replace it with code you write!

Important: Almost all the time, when someone drags a ringed expression into a ringed input slot, the person wants one ring, and doesn't understand that the ring already in the slot will stay there, around the input expression. So Snap! tries to help out by removing the outer (already provided) ring and keeping the inner (user-made) one. It doesn't matter which ring is deleted except in the case of nested rings with explicit input names. Look at the picture of *successor* below. Do this experiment: Try to recreate it by first making an outer ring with input name *number*. Then, separately, make another ring with input name *f* and drag it into the *number* ring. Poof! Snap! did you a favor and got rid of the *number* ring.

The secret is to make nests of rings from the inside out. First make a ring with input name *x*. Then right-click on that ring and choose "ringify" from the menu. That wraps a new ring outside your *x* ring; give it the input name *f*. Then ringify that one, and give the outer ring an input named *number*. Now you have all the input names you need to build whatever function is going to be inside the three rings.

But actually, you'll equally often have to use the "unringify" option when using the result of a *call* as the function input to another *call*.

If the only data type you're given is functions, how do you use functions to represent numbers? The answer is that the number three means to call some function repeatedly three times: $f(f(f(x)))$. Here's how we could say that in Snap! notation:



This says that the function *three* takes some function, which we'll call *f*, as input; *three* returns a function of *x* that calls *f* three times, starting with the value *x*.

But we don't want to have to write explicit definitions for one, two, three, four... and so on forever. Instead, we start by giving you two 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 $f(f(f(\dots f(x))))$ with that number of calls to f . So, in particular, the number *zero* returns a function of f that doesn't call f at all; what *zero* returns is the identity function that returns its input unmodified.

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 input and returns the next number ($number + 1$). Here's how it works: Suppose we have *number*. Then, for any function f , we can call *number* with argument f , and that gives us a function that calls f *number* times. The number $number + 1$ should be a function of f that returns a function that calls f $number + 1$ times. So we just have to call f one more time. Look in the definition of *successor* for the place where it calls *number* with input f , then see how it calls f with, as input, the result of calling *number*-of- f with input x .

Note: We are always using the name *number* for inputs that we know will be Church numerals. We always use f for an arbitrary function, and x for something we're thinking of simply as data. But there's nothing magic about the names! Just because you use the name *number* for an input doesn't guarantee that your code treats it like a number. You could keep the default names #1, #2, and so on, and the meaning of your program wouldn't change.

A Church numeral is always two nested functions:



Ex. 1: Convince yourself that



reports a function that has the same behavior as *three* above.

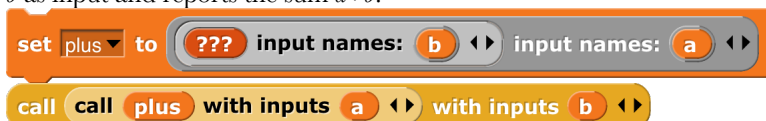
Note: To help with debugging, you can use the following function, **TRY**. You call it with a Church numeral and it reports the ordinary number that corresponds to it. **TRY** isn't part of the Church numeral system, just a debugging tool. That's why its name is in capital letters, to remind you that you can't make it part of the solution to a problem.



Make sure you understand why TRY has an ordinary Snap! + block instead of calling *successor*.

Ex. 2: Using only *zero* and *successor* as helpers if necessary, write a **plus** 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 *plus* the value of a global variable, 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 *plus*. All functions take exactly one input. But you can write a *plus* function that takes a number *a* as input, and *reports a function* that takes *b* as input and reports the sum $a+b$:

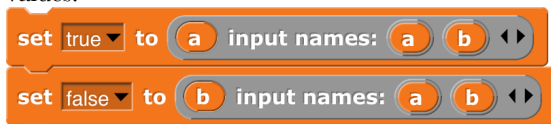


This trick, turning a two-input function into a one-input function that returns a function of the second input, is called "Currying" after the logician Haskell Curry. But you may abbreviate this way:



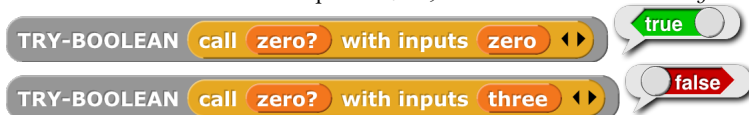
Ex. 3: Similarly, write *times* and *expt* (exponentiation).

Beyond this point you need a way to do conditional evaluation: *if-then-else*. First you need Boolean (true/false) values:

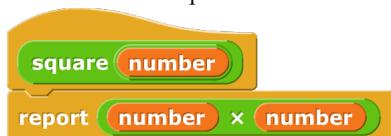


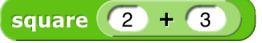
Ex. 4: Invent *zero?*, our first predicate function; it takes a number (a Church numeral) as input and reports *true* if and only if the number is *zero*.

Hint: *zero?* has to return *true* or *false*, that is, a function of two inputs that selects one of them. So your block will include a call to the *number* input to *zero?*, and it'll involve *true* and *false* somehow. We provide a debugging aid:



Ex. 5: Invent *if-then-else*. This function has to be defined as a standard Snap! custom block, rather than by assigning a lambda expression to a variable. The reason won't be obvious until you generate a mechanism for recursion, quite a while from now, so we're going to take a detour to talk about how Snap! evaluates procedure calls. Take a simple function like this:



What are the steps, in detail, that Snap! takes when you call it with, for example, this:  ? First, Snap! computes $2+3$, getting 5. Second, it calls *square*, with 5 as its input. During this call to *square*, *number* has the value 5. Third, it evaluates the body of *square*, which says to multiply 5 by 5. The multiplication function knows only that it has to multiply 5 by 5. It doesn't know or care that both of those fives were computed by the expression $2+3$. First compute the input values, then call the function. This sequence of steps is called *applicative order evaluation*, and it's how most programming languages work.

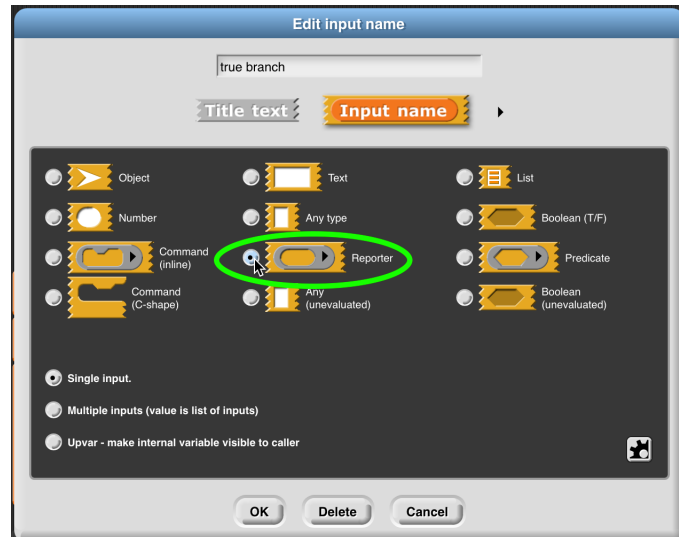
But applicative order doesn't work for Snap!'s *if-then-else* block. Depending on the value of its first input, *if-then-else* has to evaluate *either* its second input *or* its third input, *but not both*. The block has to be called *before* its input expressions are evaluated, not after. Why is this important? Don't you get the same answer either way? Well, yes, if you get an answer at all, it'll be the same answer either way. But we'll see in a moment that you might never get the answer, using applicative order evaluation. (The alternative, in which the function is called with its input *expressions* rather than its input *values*, is called *normal order evaluation*.) As a first step in the argument, imagine that one of the two alternative expressions takes a very long time to compute—the millionth prime number, let's say. Then sometimes, depending on the value of the first input, the entire computation of the *if* expression will take much longer in applicative order (in which we have to compute both alternatives before we get to decide which we want) than in normal order (in which if we're lucky we don't have to do that long

computation at all). A similar but more extreme case would be that one of the branches has an infinite loop; with applicative order evaluation the computation would *always* take forever, regardless of the first input, but normal order lets one of the alternatives, the one without the infinite loop, produce a result.

The inputs to a gray-ring lambda expression in Snap! are always computed in applicative order. That's why you can't define *if-then-else* the way we're defining everything else, with

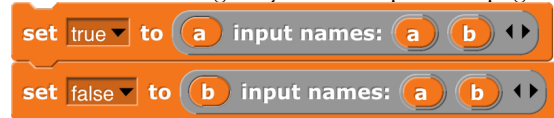


But Snap! does provide a mechanism to achieve the effect of normal order evaluation in its function definitions. You declare the inputs to be of a Procedure type:



To make this fully equivalent to normal order evaluation, the first input should be of Procedure type also, but we don't have to do that because the first input is *always* evaluated, first thing in the *if-then-else* algorithm. Also, you might think that first input should be of type Boolean, but it's not going to be a *Snap!* Boolean, but rather a lambda-calculus Boolean. All three inputs to *if-then-else* will be functions, because that's the only data type we have!

Okay, now you're ready to finish the definition of *if-then-else*. It should expect its first input to be one of the Boolean values we gave you on the previous page:



It'll use that input to choose one of its other inputs, which you'll evaluate with a call block: **call true branch** (but probably not that simple...). The **call** sort of undoes the wrapping of a gray ring around the input.

Remember that if the expression you drag into one of those rings is a *variable*, Snap! will do you a favor and unringify it. You'll have to re-ringify it.

After this we're back to the usual rule, inventing new functions as explicit lambda expressions (gray rings).

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 particular 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 verbally, so the function *cdaddr* ("kuh DA duh der") is *cdr(car(cdr(cdr(pair))))*!

Ex. 7: (This is the hardest exercise!) Invent ***predecessor***, a function that takes a number *number* and reports the number *number*-1. Don't worry about what it reports if its input is *zero*. Hint: Our solution uses pairs.

Better hint: It involves these expressions:

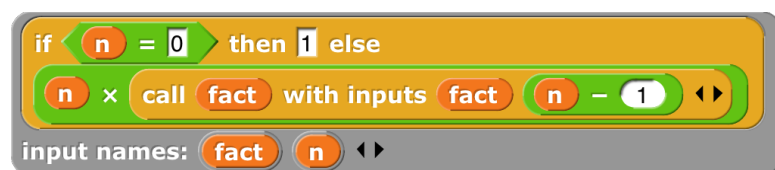


Ex. 8: Using *predecessor*, invent ***difference***, a function of two numbers *a* and *b* that reports *a-b*. Again, don't worry about what it reports if *a < b*.

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.

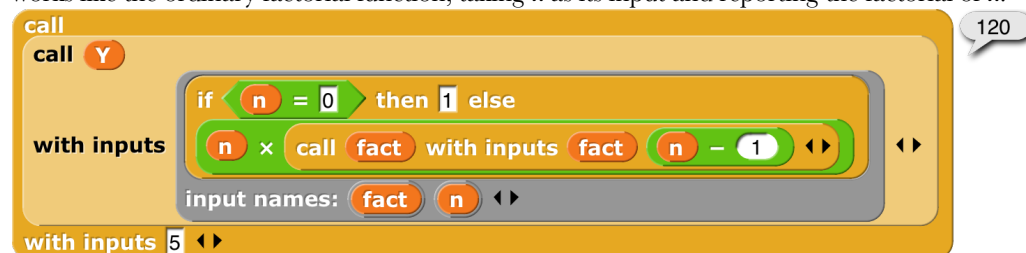
Ex. 11: (This is the other hardest exercise!) Time to invent recursion. That is, we need to invent a way for a function made with a gray ring, so it doesn't have a name and doesn't appear in the block palette, to call itself. This is the point at which you have to remember that putting the functions you create in global variables is just an abbreviation to make your code not as hideous as it would be in strict lambda calculus, which has no global variables. In inventing recursion, it would be cheating to give a function a name, by putting it in a global variable, and then using that name to call it. So, no ***set*** blocks in this exercise. 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. Here's the factorial function written that way:



(This isn't a Church-numeral example; to simplify the picture and focus only on the problem of recursion, the ordinary Snap! arithmetic operations and ***if-then-else*** 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 (whose input corresponds to the input *n* above) that calls this function with *itself* as its first input.

(Historical note: The lambda-calculus solution to this problem is called the "***Y*** combinator." If you've heard of the venture capital company called that, this is where it gets its name from.)

The function you're going to write is much simpler than the official ***Y*** combinator, because [reasons](#), but its job is to take a two-input function such as the picture above, and report a one-input function that (in this example) works like the ordinary factorial function, taking *n* as its input and reporting the factorial of *n*:



Ex. 12: Use **Y** to write a factorial function for Church numerals. That is, use your *if-then-else*, *equal?*, *times*, and so on instead of the ordinary Snap/ functions. For ease of use, put your function in a **factorial** variable, but don't use *factorial* inside its definition.



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

To debug your function, you'll want an easy way to make biggish Church numerals:



But you'll find that dividing 87 by something takes a long time; you'll probably prefer an example such as 26/3.

The end! You've made the basics of nonnegative-integer arithmetic, starting with only **call** and **call**, as advertised. For example, you should be able to write **prime?**, although it'd be painful.

Another direction you could pursue is to create lambda-calculus rational numbers, represented as a pair of two Church numerals, the numerator and the denominator. The relational operators are a bit tricky because the result of adding two rational numbers often isn't a fraction in lowest terms. So either you have to reduce all arithmetic results to lowest terms or you have to have relational operators that understand that $3/6 = 1/2$.

And then the next step is negative numbers, represented as a pair whose car is a Boolean and whose cdr is a rational number as defined above. The number is negative if the Boolean is *true*.

Extending this project to real numbers would be quite a lot harder than all of the above. Irrational numbers are problematic even in normal arithmetic, because you can't represent them exactly: $\sqrt{2} = 1.41421356237309 \dots$ The problem is with those three dots. If you can't represent $\sqrt{2}$ exactly, how can *equal?* know whether one string of digits is really equal to another? The value $\sqrt{2} + 10^{-50}$ will look a lot like $\sqrt{2}$ in the first 40 digits. In fact, it's theoretically impossible to have a representation that works for *all possible* real numbers; there are numbers that are literally *not describable*. So don't try!