The Y Combinator, Self-Reference and the Halting Problem

So far this semester, you have seen a lot of recursion. Perhaps too much recursion. But why is recursion important? What does a language that supports recursion have to offer over one that doesn't? Today we're going to show how recursion is a concept which defines the limits of what we can compute, and how such a powerful concept can come from relatively humble beginnings.

You have seen ugly hacks that introduce recursion into a language. In the environment model, we cheated by allocating a frame with a "junk value" before making the closure, and then fixing things up afterwards. We did the same thing with Letrec_l evaluation in problem set 5. You might be thinking that recursion is somehow ' above' the power of non-recursive languages. Without someone outside the system coming in and hacking around, is it possible for a non-recursive language to create recursion? Literally, can a language containing only finite statements express something which is infinite?

Let' s examine the function `fact`:

```
fun fact x = if x=0 then 1 else x*fact(x-1)
```

Can we construct this using only `val` declarations?
We can, but it' s a similar hack that you' ve seen before: we use ' junk values' :

```
val fact = let
  val fact':(int -> int) ref = ref (fn x => x)
  val f = fn x => if x=0 then 1 else x*(!fact'(x-1))
in
  fact' := f;
  fn x => !fact' x
end
```

Not a single `fun` declaration anywhere!
But we're still cheating, using the same kind of trick we used before. Suppose we can't step in and change things using refs…could we still construct `fact`? Let's unroll the recursion one level:

```
fun fact' x = if x=0 then 1 else x*fact'(x-1)
val fact = fn x => if x=0 then 1 else x*fact'(x-1)
```

As long as `fact'` is in scope, the second statement works without using refs or the `fun` keyword.

```
fun fact' x = if x=0 then 1 else x*fact'(x-1)
val fact'' = fn f => fn x => if x=0 then 1 else x*f(x-1)
val fact = fn x => (fact'' fact' x)
```

This is a little closer. `fact''` no longer needs the recursive function to be in scope, as long as it's passed in as an argument.

```
val fact'' = fn f => fn x => if x=0 then 1 else x*(f f)(x-1)
val fact = fact'' fact''
```

No more fun declarations! No more refs! Check to make sure you are convinced that this does the right thing.

The construct we have used can be generalized into something else, which takes in a non-recursive function and makes it recursive. This powerful construct is called the Y-Combinator <thunder, lighting>.

val Y = fn f => (fn x => f(x x)) (fn x => f(x x))

We can use this to build recursive fact simply by calling `Y fact''`.

Let's see the substitution model explanation for why this works:

```
Y fact'' =  (fn x => fact''(x x)) (fn x => fact''(x x))
         =  fact'' ((fn x => fact''(x x)) (fn x => fact''(x x)))
         =  fact'' (fact'' ((fn x => fact''(x x)) (fn x => fact''(x x)))
         =  fact'' (fact'' (fact'' ...))
         =  fact'' (Y fact'')

Y fact'' 1 = Y fact'' 1
    = fact'' (Y fact'') 1
    = if 1=0 then 1 else 1*(((Y fact'')(Y fact'')0)
    = 1*((Y fact'') (Y fact'')) 0)
    = 1*(((fn x => fact''(x x)) (Y fact'')) 0)
    = 1*(fact'' ((Y fact'')(Y fact'')) 0)
    = 1*(if 0=0 then 1 else 1 * (((Y fact'') (Y fact'')) 0-1))
    = 1*1 = 1
```

The power behind the y-combinator comes from the idea of self reference. The Y-combinator essentially makes functions 'self-aware', in that it puts them *in their own scope.* This is exactly the functionality of the `fun` keyword, and is the power we were looking for.

Does this power come at a price?

In the earlier part of the 20th century, mathematicians were looking for ways to formalize what it meant for something to be computable. Bertrand Russel and Alfred Whitehead began an enormous push to this end: *Principia Mathematica*. The idea was to create a finite set of axioms from which we could derive all true statements about the universe. To wax philosophical, mathematicians at the time believed that math was the tool God had given them by which to discover his creation. Literally, anything that was True could be Proved. Anything true could be understood by man, using the power of this formalism.

PA depended on two things: Soundness and Completeness. A proof system is sound if every statement we can prove is true. In other words, it works. If a proof system is unsound, we can prove things that are false, which doesn't help us find the truth to the universe! A system is Complete if all statements that are true can be proven. The combination of these two is necessary for us to find all true statements using the proof system.

We can use some of the ideas in this course to see why this is impossible, and where the mathematicians went wrong. The reason why these ideas fail is precisely the idea of self-reference, which the Y-combinator embodies. A mathematician by the name of Kurt Godel was able to construct a sentence using the formal language of Principia Mathematica that could not be computed by any finite set of axioms in the system: "This statement is not provable." The key to this statement is that it was written in the language of the proof system. Statements outside the system are allowed to be meaningless: they are nonsense insofar as the system is concerned. But Godel's statement was not nonsense…yet it was certainly not provable. For the mathematicians at the time, this was the equivalent of a truth that could be felt, but not known.

We can use the Y-combinator to construct such a statement. (Why not?) Suppose we have a function `not`, which simply returns the negation of its argument.

What is the value of `(Y not) (Y not)`?

```
(Y not)(Y not) = (fn x=>not(x x))(Y not)
               = not ((Y not)(Y not))
```

Suppose `(Y not)(Y not)` is `true`. The result of our expansion should be `false`. Conversely, if `(Y not)(Y not)` is `false`, the expansion will return `true`. Clearly, we have a contradiction here. We can phrase other well-known questions using the Y-combinator:

- The Halting Problem: suppose we had a function `halts`, which determined if a program halts on a given input. Now take its negation, `halts' = not halts`. `(Y halts')(Y halts')` gives us a similar contradiction.

- The Human Existential Crisis: suppose there was a way of knowing whether or not we exist. Call it a function `exists`, which takes something and tells us whether or not it exists. For example, `exists CS312` evaluates to `true`. Most of you might wish it didn't. Suppose we define `exists'` to be the negation of `exists`. `(Y exists')(Y exists')` gives us a similar quandary.

The important thing to gain from this is that almost any system is inherently incomplete.

"No sufficiently powerful deductive system is complete" – Godel

By 'sufficiently powerful', we mean that it has the ability to talk about itself. Not all systems suffer from Incompleteness. Truth tables and propositional logic are examples. However, these can only prove extremely limited things. No logical system can even prove that it is sound…which poses a problem in mathematics, since any system you make is not provable until you end up at a paradox. Soundness is something that is extremely important to a system, yet you can never prove it without going outside the system, or being comfortable with paradox.

Why would you introduce such a concept into a language? The truth is that without it, the kind of problems you can solve (in fact, the kind of questions you can ask) are extremely limited without self-reference. But even after introducing self-reference, we have seen there there are many questions for which we will still never know the answer.