# 2013W1-lecture20

October 27, 2013

## Contents

## 1 Question of the Day

What does the abstract syntax tree for this program look like?

```
(let-rec ((f (lambda (n)
               (if (= n 0)
                 1
                 (f (sub1 n))))))
  (f 2))
```

**NOTE: I ACCIDENTALLY USED `let` IN THE ORIGINAL VERSION!** `let` does not work because its binding—for `f` in this case—is not available in the value expression we evaluate to get `f`'s value. `let-rec`, on the other hand, allows us to define recursive functions.

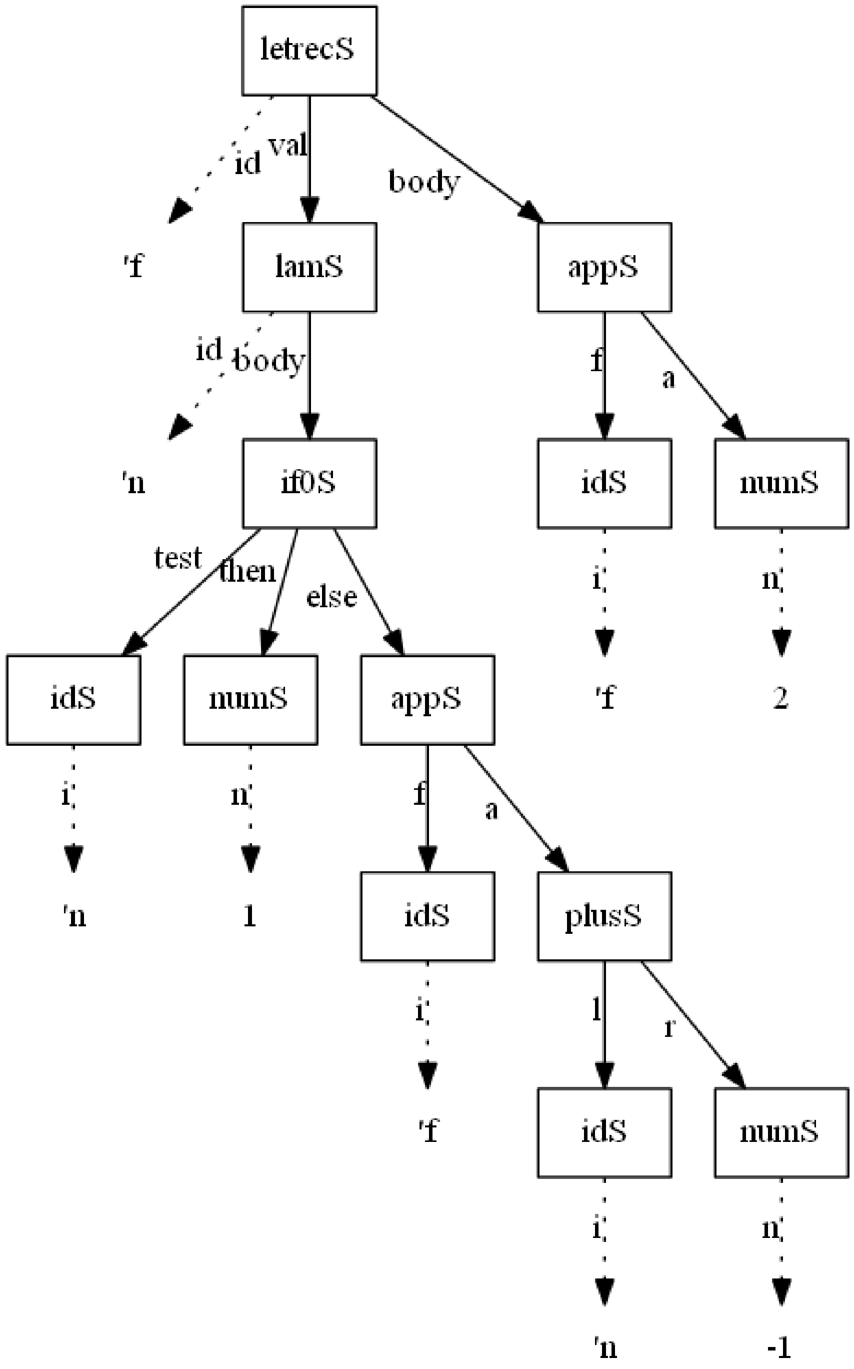
There are different options for how to do this, one of which is the way you solved `deffun` for the ParselDesugar assignments: use something like a `let` to bind `f` to a dummy value; then, inside the body (where the `f` binding is available!), assign the real value to `f` before evaluating the body of the `let-rec`.

## 1.1 Solution

You should be able to draw this yourself! I used this slightly different program (which matches the syntax I had in my interpreter):

```
(let-rec (f (lambda (n)
              (if (= n 0)
                  1
                  (f (+ n -1)))))
         (f 2))
```

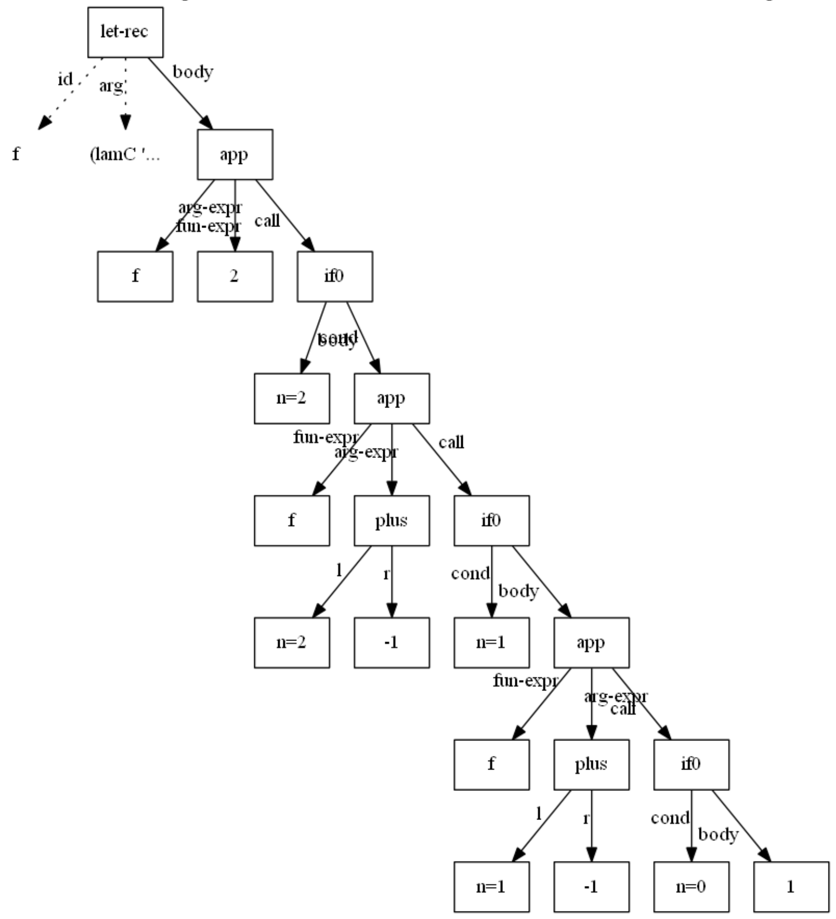
Here's what my program automatically created for the abstract syntax tree:


letrecS
id
val
body
'f
lamS
appS
id
body
f
a
'n
if0S
idS
numS
i
n
'f
2
test
then
else
idS
numS
appS
i
n
f
a
'n
1
idS
plusS
i
l
r
'f
idS
numS
i
n
'n
-1

Notice how in this tree you can check whether an identifier is bound just by traversing up the tree from the identifier reference. Also, you can tell where a binding is available by looking **down** the tree into subtrees.

(If I had used a `let` rather than a `let-rec`, only the `body` subtree would have the `let`'s binding available, not the `val` subtree.)

We can also talk about the function call tree: a tree showing the recursive function calls in our program. In a language like Java, we'd usually literally only consider actual function calls. When we're writing an interpreter, however, it's more interesting to look at each recursive call to interp (which will include a recursive call to invoke a function in the function application case). Here's what my program automatically produces for our little program:



Notice the `call` edges representing the initial and recursive calls to the function (referred to by) `f`. If we compare our "location" in the code in the function call tree to the abstract syntax tree, we see that the function call tree can "jump" from one place in the AST to another, including (on the recursive calls) from a descendant of a node in the AST back up to the node.

The store flows left-to-right and top-to-bottom in the function call tree.

## 2 Logistics

### 2.1 Programming Assignment #4

It's out; the milestone is due Sunday.

Get started, but be sure to read the chapter on state (Chapter 8) first!

### 2.2 Conceptual Assignment 4 corrections due Tue

### 2.3 Midterm exam one week from Tue

## 3 What have we learned today?

- From QotD: What do our various "flows" look like when we draw them out? Learning goals:
  - Draw the abstract syntax tree for a program
  - Draw the function call tree for a program
  - Identify how values and the store flow through the function call tree
  - Explain why it's difficult to show how values/store flow through the AST
  - Explain how the environment can be understood both in terms of the AST—in its "shape" (the identifiers bound at a given point)—and the function call tree—in its values.

## 4 Some Practice Exercises

Recording practice exercises we mention over time.

- Imagine we wanted to introduce something sort of halfway between `let` and `set!`. We'll call it `sleet`. In a language that otherwise has static scoping, `(sleet id value-expr body-expr)` introduces a new **dynamic** binding for `id`. If a static binding for `id` already exists in the current scope, that produces an error. If a dynamic binding already exists, this `sleet` shadows it. (We'll assume we **cannot** use `set!` on `sleet`-bound variables.) An `id` reference will always be to a static binding if one exists; otherwise, it will find dynamic (`sleet`) bindings.

  - Use a core language with `sleet` to desugar `(deffun name funbody body)`, and discuss why it's easier than with `let`.
  - Imagine implementing `sleet` by passing **two** environments into `interp`, `static-env` and `dynamic-env`. Implement the `appC`, `funcC`, and `idC` cases for the interpreter, being sure to manage the two separate environments correctly. (Do closures need one environment or two??)
  - Describe any concerns you have about `sleet` from a software engineering perspective.

- Implement a new syntactic form `incr!` that takes a variable and adds one to its value.
- Implement a new syntactic form `time-maximum` that evaluates to the largest value a variable has ever contained. Discuss whether we could incorporate this into a mainstream language. (Could we do it with desugaring rather than changing the underlying language?)
- Implement a new syntactic form `count-bindings`. It takes a single argument—a symbol—and evaluates to the number of bindings for that symbol presently in the environment (so, more than 1 if there's both a binding and at least one shadowed binding).
- Create multiple-argument functions and applications in a surface language by desugaring them to one-argument functions and applications in the core.
- Assume you have a language with multiple-argument function definitions and applications as well as lists (cons, first, rest, and empty?) in the core. Implement a new core form `apply` that takes a function expression and an argument list expression, checks that they evaluate to the appropriate types (function value and list), and then applies the function to the list of arguments. `apply` should give an arity (i.e., number of arguments) error if the length of the list of arguments does not match the length of the list of parameters for the function.
  - Now, desugar as much of this as you can to a core that has only single-argument function definition and application. Discuss why the arity check, in particular, is **very** hard to manage in the

desugaring. (Hint: when you apply a “two-argument function” to just one argument in this desugaring, the natural result is to get a function back waiting for the second argument. Is such a value illegal or unreasonable as a “normal value” in a language with first-class functions?)