COMP 302 Lecture 15 21 October 2016

Notes on A2

1) Regexes are meant to match the token that's at the beginning of the string. Not interested in what's later on in the string. When scanning, we will go through the string piece by piece, peeling off the beginning every time.
   All should match at least 1 character.
   The "anything but…" are the trickiest, but they're not so hard.
   JS regexes.
   Easy to test your regexes *see 15c302*
2) You are recognizing (and eventually) removing tokens from the start of the string.
   Your scan function -> receives a string, also receives a token set that tells it what to look for.
   The token set tells it what to look for. In Q3, you should ask yourself what token set you want to pass every time.
   If you cannot do Q1, you can still make progress on Q2. You can recognize tokens algorithmically using code, not regexes.
3) Functions correspond to rules. The functions you're building to parse things correspond to rules in the grammar for non-terminals. Need the given four, but we can add extra ones.
   Each parse function -> recognizes a rule *see 15c302-1*. If we see T1, we know it will be followed by another instance of the rule.

We know the features of a language, we know how to parse it, we started to evaluate it. A piece is missing. If we want to pass around functions, we can pass around string. A function is not just a code, it's also the environment in which it executes.

For static scope, we need to track the declared environment.
E0:

- We evaluate the name -> a string
- We evaluate the parameters -> a string
- The body -> rest of our function. I don't evaluate that, I just keep track of it.

If I encounter a call to bar right after that, I'm still in E0. Evaluate name -> bar, evaluate param -> "hello".
"bar", args["hello"]
To invoke it, I create a new environment E1.
Execute the body of bar in E1: {: foo | {{{a}}} :} {{foo}}.

We have processed the definition of foo. Now we evaluate its call.
Evaluate the body of foo in E2 ({{{a}}}). We look for the binding of "a" in E2, there is none, so we go up the linking chain, and it is found inside E1.

Unwind all the calls -> "hello".

E0:
Bar: params (a)
Body: …
Env: E0

E1
a: hello

Parent environment is E0. If I can't find things in E1, I can look in E0.

E1:
a: hello
foo: params: []
Body: …
Env: E1

E2
Parent environment is E1

E2:

We notice we used static scoping -> environment associated with our function/template definition was the environment it was declared in -> used it to construct our environment.
We could have chosen our current environment -> dynamic scoping.

Other example: *see 15c302-3*
We first get the declarations of foo and bar.

Then code to invoke bar.

In E1, evaluate the body of bar ({{foo}}).
Need to find a binding for foo. Not in E1, follow to the parent, find the binding for foo in E0.

Execute the body of foo inside E2: {{{a}}}
I don't find a binding for a, I go up to the parent, no "a" either. Just returns the string "{{{a}}}".

With dynamic scoping, I would have created E0 with foo and bar.
Created E1 that has the binding "a: hello". When I create E2, the parent of E2 *would have been* the environment I was in, therefore E1. And ultimately, we would have found the binding for "{{{a}}}".

E0: foo: param[] body env – E0 bar: param[a] body env – E0
E1: a: hello point it to E0
E2: a also points to E0

Back to functions/template passing.
We want to be able to pass functions (or templates). Pass a closure (function + environment).
Anonymous functions *see 15c302-4*
If our definitions return a closure,
eval: AST -> strings ➔ AST -> strings or closures.
A bit messy to have this mixed output. We will stick with strings only.