CSC 530 Lecture Notes Week 7
More on Tennent-Style Denotational Semantics

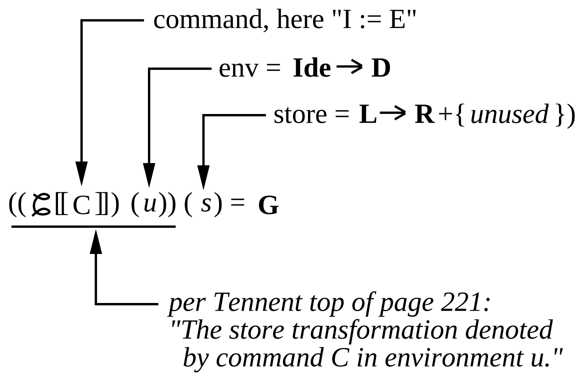
I. These notes refer to, Tennent Chapter 13 is attached in Paper number 20.

A. In these notes, we'll dissect some of the key definitions appearing in Tables 13.2 and 13.3.

B. Probably the hardest thing to "get" about these definitions is the pervasive use of functions (of functions (of functions)).

II. Assignment statements

A. The following is a dissection of the denotational definition of assignment from Table 13.3:


command, here "I := E"
env = **Ide** → **D**
store = **L** → **R** + { *unused* })
$((\mathcal{C}[\![C]\!])\ (u))\ (s) = \mathbf{G}$
per Tennent top of page 221: "The store transformation denoted by command C in environment u."

where **G** expands as follows:

$$\begin{aligned}\mathbf{G} &= s[d \mapsto e]\\ &= s[u[\![I]\!] \mapsto \mathcal{E}[\![E]\!]\ u\ s\ ]\\ &= s[(\mathbf{Ide} \to \mathbf{D})\ [\![I]\!] \mapsto\\ &\quad \mathcal{E}[\![E]\!]\ (\mathbf{Ide} \to \mathbf{D})\ (\mathbf{L} \to (\mathbf{R} + \{\mathit{unused}\}))]\end{aligned}$$

B. What this says is that the meaning of an assignment statement of the form "I := E" for some identifier "I" and expression "E" is a modified store, where the location of the modification is the memory location bound to "I" in the environment *u* and the value of the modification is the value produced by evaluating E in the environment *u* and pre-modified store *s*.

C. Now tell the truth, isn't that one the coolest things you've ever seen?

III. Procedures

A. Consider the following Tennent code fragment:

Code	**Semantics**
val n = 100	$\mathcal{D}[\![$**val** n = 100$]\!]$
new x = 0	$\mathcal{D}[\![$**new** x = 0$]\!]$
t = **true**	$\mathcal{D}[\![$**new** t = **true**$]\!]$

new Pn = (**procedure** x := x+1)	$\mathcal{D}$[[**val** Pn = (**proc** ...)]]
val Pv = (**procedure** x := x+1)	$\mathcal{D}$[[**val** Pv = (**proc** ...)]]
call(Pv)	$\mathcal{C}$[[Pv]]*u s*

B. with the resulting environment (*u*) and store (*s*):

Environment:

Ide	**Value x Tag**
n	100,**Z**
x	0xff20, **L**
t	0xff24, **L**
Pn	0xff25, **L**
Pv	$\mathcal{C}$[[x:=x+1]]u_v

Store:

Addr	**Value**
0xff20	0
1	0
2	0
3	0
4	true
5	$\mathcal{C}$[[x:=x+1]]u_n *storage for body of Pv* ...

C. Notes

1. The depiction of the environment is intended to resemble a lookup table and depiction of the store is intended to resemble a segment of computer memory;
   a. Both of these depictions are merely graphical mnemonic suggestion.
   b. Abstractly, both the environment and store are the same form of unary functions.
2. Notationally, an assoc-style lookup in the environment is denoted

   u[[x]]

   a. This means apply the environment function *u* to the identifier "x".
   b. Think of the *entire table* applied as a function to the argument "x", which application effects a lookup of the value "0xff20"
3. Note well that the procedure values bound to Pn and Pv are unevaluated lambda bodies, with an *attached environment*.
   a. This attachment is what defines this language as *statically scoped*.
   b. Note further the different static environments in the binding of Pn versus Pv.
      i. In Pn, the environment does not yet contain the binding of Pv.
      ii. This defines a declare-before-use semantics for procedures.

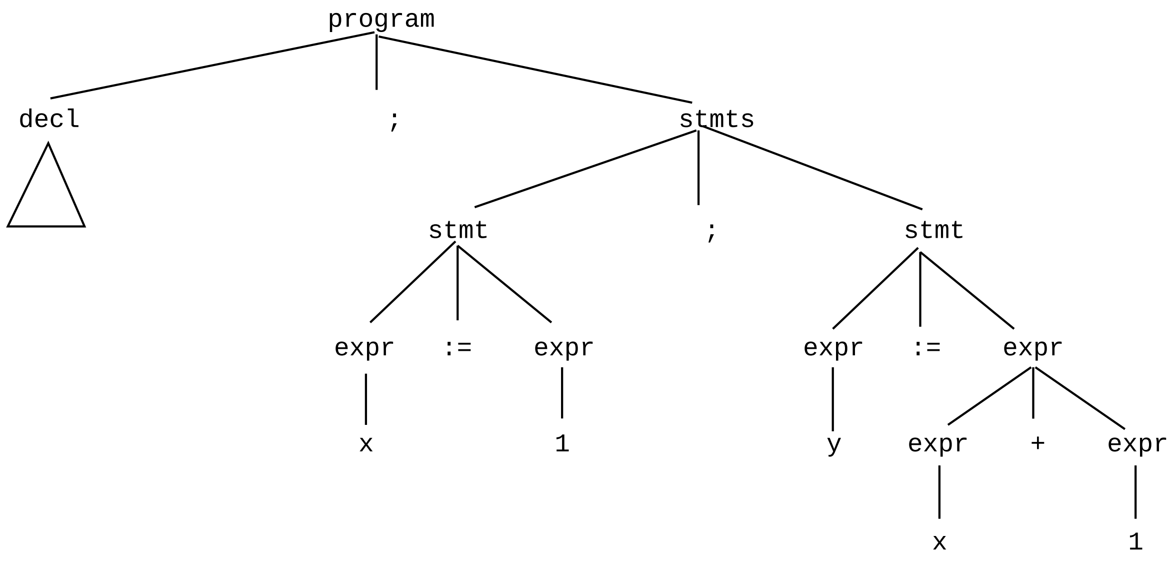
IV. Comparison of Tennent versus Knuth overall evaluation strategy

A. Consider the generic Tennent/SIL program:

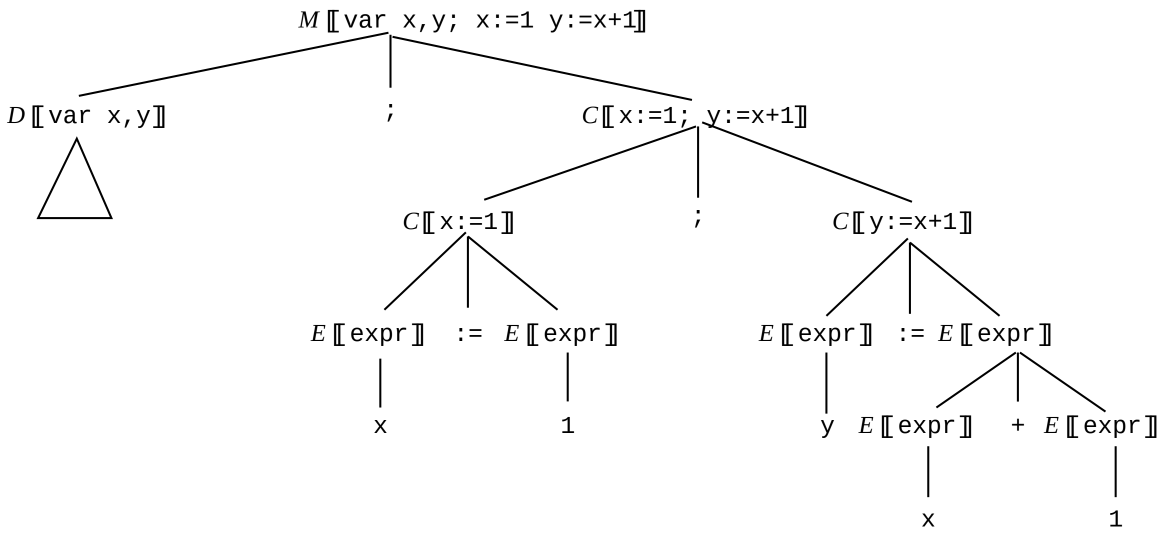
```
var x,y;
x := 1;
```

y := x+1;

B. The Knuth tree-based evaluation involves walking through the following parse tree:



C. The comparable Tennent function-based evaluation involves a series of function applications that look like this:



V. Fundamentals of infinitary program behavior, i.e., loops.

A. Thus far, we have not directly adressed the semantics of iterative loops.

1. In the lisp interpreter, we presumably relied on tail recursion to effect iterative behavior.
2. In the SIL definition, we left out any form of *while loop* construct.

B. To address the looping issue directly, we can take two approaches:

1. An "operational" mathematical approach, where loop-style iteration is defined in terms of

recursive function application directly.

2. An abstract mathematical approach, where loop iteration is defined in terms of a *recursive mathematical definition*, using a *fixpoint-as-limit* technique to deliver a meaningful semantic function.

VI. Let us consider a fixpoint definition of **while**.

A. The basic idea is

"**while** E **do** C" = **if** E **then {** C ; **while** E **do** C }

B. Using a bit of denotational notation:

$\mathcal{C}$⟦ **while** E **do** C ⟧ =
    if $\mathcal{E}$⟦E⟧ **then begin** $\mathcal{C}$⟦C⟧ **;** $\mathcal{C}$⟦**while** E **do** C⟧ **end**

where let

f = $\mathcal{C}$⟦ **while** E **do** C ⟧
F(*f*) = **if** $\mathcal{E}$⟦E⟧ **then begin** $\mathcal{C}$⟦C⟧ **;** $\mathcal{C}$⟦**while** E **do** C⟧ **end**

C. **Question**: So what's going on here? What kind of definition is this?

Ans: it's an *equation* to be solved for *f*, i.e., for *f* = F*(f)*

D. **Question**: Does this form of recursive functional equation have a solution in general? I.e., for a functional F: D→D, does there exist

Y: (D→D)→D such that Y(F) = F(Y(F))

Ans: Yes, if we make the following assumptions:

1. Function domains are properly defined (per Tennent ch 3)
2. All functions are continuous on their defined domains

E. This solution is called a *fixpoint*, since we're looking for the point *f* in the domain of F where F is fixed, i.e., where F(*f*) = *f*.

F. **Question**: What does such a fixpoint look like?

Ans:

1. For non-recursive cases it's just what you'd expect; e.g.,

fix(f(x) = 4) is 4

fix(f(x) = 8 - x) is 4

2. For recursive cases, consider this example:

let f(x) = if x=0 then 1 else if x=1 then f(3) else f(x-2)

a. A fixpoint of this is

f(x) = 1 if x is is even and x≥0
    undefined otherwise

b. But also a fixpoint is

f(x) = 1 if x is even and x≥0
    a if x is odd and x>0
    b otherwise

for any a,b

c. But the first of these is the *"least"* fixpoint.

i. In this class, we'll not develop the mathematics of this any further.

ii. It is overviewed in the Tennent CACM paper.

iii. In depth coverage is in Stoy's *Denotational Semantics*, of which Fisher has a copy. See also the references in the Tennent paper.

3. For a recursive construction such as the while loop, the fixpoint function looks like a *limit* of successive approximations.

   a. I.e.,

   (**while** E **do** C)$_0$ = "the worst loop approximation"

   (**while** E **do** C)$_{i+1}$ = **if** E **then** C; (**while** E **do** C)$_i$

   b. More specifically,

   (**while** E **do** C)$_0$ = **while true do null**;

   (**while** E **do** C)$_1$ = **if** E **then** C; **while true do null**;

   (**while** E **do** C)$_2$ =
   if E **then** C; **if** E **then** C ; **while true do null**;

   ...

   where the ";" separators are defined as *function composition*

   c. In terms of functions,

   let $f = \mathcal{C}[\![$**while** E **do** C$]\!]$

   and we want

   $$\begin{aligned} f &= \mathcal{E}[\![E]\!] \rightarrow \mathcal{C}[\![C;\ \textbf{while}\ E\ \textbf{do}\ C]\!] \\ &= \mathcal{E}[\![E]\!] \rightarrow (\mathcal{C}[\![\textbf{while}\ E\ \textbf{do}\ C]\!](\mathcal{C}[\![C]\!])) \\ &= \mathcal{E}[\![E]\!] \rightarrow f(\mathcal{C}[\![C]\!]) \end{aligned}$$

   and now, regarding the last line notationally as a function F of *f*, we want

   $f = F(f)$

   d. That is, we have arrived at the situation discussed in section 4.3 of the Tennent paper, where we're told that

   $\perp, F(\perp), F(F(\perp)), ..., F^i(\perp)$

   approximates *f*, where $\perp$ is the *bottom* value of the domain we're interested in, which here is (**while** E **do** C)$_0$.

   e. Invoking our assumptions of continuity of F and domains defined nicely, we have

   $\mathbf{Fix_D} : (D \rightarrow D) \rightarrow D$

   defined as

   $\mathbf{Fix_D} = \lim_{i\rightarrow\infty} F^i(\perp)$

   as the least solution of $f = F(f)$, i.e., it is the *least fixpoint*.