COP-5555 PROGRAMMING LANGUAGE PRINCIPLES
NOTES ON DENOTATIONAL SEMANTICS.

1. Introduction.

Here we show, by way of example, the use of denotational semantics for the specification of semantics of programming languages.

In general, a denotational semantic description consists of three parts: a set of syntactic domains, a set of semantic domains, and a set of semantic functions. The semantic functions map syntactic domains into semantic domains. Typically the syntactic domains are AST's; a semantic function might map an AST into a value (in a semantic domain), which "denotes" the meaning of the construct.

Example:

Suppose we were to give the denotational semantic description of a two-operand machine language, for a 16-bit machine with 8 registers. The syntactic domain might be the set of AST's of the form < Opcode Operand Operand >. The semantic domains might as follows:

Register:	{0,1,2,3,4,5,6,7}
Value:	(16-bit binary number)
Address:	(16-bit binary number)
Memory:	Address $\rightarrow$ Value
RegValues:	Register $\rightarrow$ Value

A semantic function might be EE (for EEvaluate):

$$\text{EE: AST} \rightarrow (\text{RegValues} \times \text{Memory}) \rightarrow (\text{RegValues} \times \text{Memory})$$

EE takes an AST, then the current configuration of the machine, i.e. a (registers,memory) pair, and produces a new (registers,memory) pair that shows the effect of the operation. For example, consider a "register-to-memory" add operation:

$$\text{EE}[+\ r\ m] = \lambda(R,M).\ \text{let Result} = R\ r + M\ m\ \text{in}\ (R,(\lambda a.a\ \text{eq}\ m \rightarrow \text{Result} \mid M\ a))$$

The meaning of an operation "+ r m", where r is a register and m is a memory address, is a function that:

1.- Takes a RegValues function R, and a Memory function M;

2.- Applies R to r (obtaining the contents of register r), applies M to m (obtaining the contents of address m), adds the two values, and calls the result "Result";

3.- Returns a pair consisting of:

   a) The same RegValues fucntion R, and

   b) A new memory function that, when applied to m, returns Result, and behaves like M otherwise.

The "meaning" of the operation is evident in this description: register-to-memory operation "+" adds the contents of the given register and the given memory location. It also updates the given memory location with the result of the addition, and leaves the registers unchanged.

We now give the denotational semantic description of the CSE machine, for which there was (up to now) no formal, rigorous specification.

2. Denotational Semantics of the CSE Machine.

2.1. Syntactic Domains.

The only syntactic domain is ST, the set of standardized trees for RPAL programs. We should note that the standardization is partial: the "$\rightarrow$" is not standardized. Control structures are also unnecessary.

2.2. Semantic Domains.

State:	Env $\times$ Output
Env:	Id $\rightarrow$ Value
Output:	Value* (tuples of values)
Value:	Boolean+Integer+String+Tuple+Function+ST+dummy+{error}
Tuple:	Value*
Function:	Value $\rightarrow$ Value
Result:	Value $\times$ Output

2.3. Semantic Functions.

First, some notation. "x = > f" denotes the expression "x eq error $\rightarrow$ error | f x". This is the "pipeline" operator. An expression of the form a= >b= >c means: send a into b, and send the result into c. If anywhere along the way there is an error, propagate it. It is important to note that "= >" is LEFT associative.

The principal semantic function is EE: ST $\rightarrow$ (State $\rightarrow$ Result). EE is a function that takes a standardized tree, and returns a State to Result function. The State is an (environment,output) pair, i.e the current configuration of the machine. The Result is a (value,output) pair, i.e. the value of the program and its output.

We now describe how EE operates, on a case-by-case basis.

EE[name] = λ(e,o).(e name,o)

EE[<unop E>] = λ(e,o).(e,o) = > EE[E] = > (λ(v,o).PE unop v (e,o))

Examples:

a) PE Print = λv.λ(e,o).(dummy, o aug v)

b) PE neg = λv.λ(e,o).(-v,o)

c) PE not = λv.λ(e,o).(not v, o)

```
EE[<binop E$_1$ E$_2$>] = $\lambda$(e,o).(e,o) = > EE[E$_2$] = >
                    ($\lambda$(v$_2$,o).(e,o) = > EE[E$_1$] = >
                        ($\lambda$(v$_1$,o).(PE binop) v$_1$ v$_2$ (e,o))
                    )
```

Examples:

a) PE(+) = $\lambda v_1.\lambda v_2.\lambda(e,o).$

$(v_1 \in \text{Integer+Real})$ & $(v_2 \in \text{Integer+Real}) \rightarrow (v_1 + v_2,o)$ | error

b) PE(aug) = $\lambda v_1.\lambda v_2.\lambda(e,o).$

$v_1 \in \text{Tuple} \rightarrow (v_1 \text{aug} v_2,o)$ | error

EE[<$\rightarrow$ B T F>] = λ(e,o).(e,o) = > BB[B] = > (Cond e EE[T] EE[F]) , where

BB: ST $\rightarrow$ State $\rightarrow$ Result

BB = λast.λ(e,o).(e,o) = > EE[ast] = > (λ(v,o).v$\in$Boolean$\rightarrow$(v,o)|error)

Cond: Env$\rightarrow$(State$\rightarrow$Result)$\rightarrow$(State$\rightarrow$Result)$\rightarrow$Result$\rightarrow$Result

λe.λT.λF.λ(v,o).v$\rightarrow$T(e,o)|F(e,o)

EE[<λ v E>] = λ(e,o).((λ,v,E,e),o)

EE[<γ F A>] = λ(e,o).(e,o) = > EE[A] = >

```
    (λ(v,o).(e,o) = > EE[F] = >
        (λ(f,o).IsClosure f → EE[f 3] (Addenv v (f 2) (f 4),o)
            | Iseta f → EE[<γ<γ<λ <f 2> <f 3> > f > v > ] (e,o)
            | (f eq Y) & (IsClosure v) → ((η,v 2,v 3,v 4),o)
            | (f v,o)
        )
    )
```

Addenv: Value $\rightarrow$ Identifier $\rightarrow$ Env $\rightarrow$ Env

Addenv = λv.λi.λe.(λN.N eq i $\rightarrow$ v | e N)

IsClosure: Value $\rightarrow$ Boolean

IsClosure = λf.not(Istuple f) $\rightarrow$ false | ((Order f)eq 4) & (f 1 eq 'λ')

Iseta: Value $\rightarrow$ Boolean

Iseta = λf.not(Istuple f) $\rightarrow$ false | ((Order f)eq 4) & (f 1 eq 'η')

So we have defined EE in its entirety, as a function that takes a standardized tree, and then an (environment,output) pair, and returns the appropriate (value,output) pair. All that is left is to define the starting points: the initial environment (the primitive environment) and the initial output (empty):

The meaning of an RPAL program 'ST' is EE[ST](PE,nil)

Example:

Consider the following RPAL program:

let x=1 in Print(x+2).

The standardized tree is

<γ<λ x <Print<+x2>>>1>

The meaning of the program is, then:

```
EE[<γ<λ x <Print<+x2>>>1>] (PE,nil) =
(λ(e,o).(e,o) = > EE[1] = >
            (λ(v,o).(e,o) = > EE[<λ x <Print<+x2>>>] = >
                  (λ(f,o).IsClosure f → EE[f 3] (Addenv v (f 2) (f 4),o)
                        | Iseta f → EE[<γ<γ<λ <f 2> <f 3> > f > v > ] (e,o)
                        | (f eq Y) & (IsClosure v) → ((η,v 2,v 3,v 4),o)
                        | (f v,o)
                  )
            )
) (PE,nil)  =

(PE,nil) = > EE[1] = >
            (λ(v,o).(PE,o) = > EE[<λ x <Print<+x2>>>] = >
                  (λ(f,o).IsClosure f → EE[f 3] (Addenv v (f 2) (f 4),o)
                        | Iseta f → EE[<γ<γ<λ <f 2> <f 3> > f > v > ] (PE,o)
                        | (f eq Y) & (IsClosure v) → ((η,v 2,v 3,v 4),o)
                        | (f v,o)
                  )
            ) =

(PE,nil) = > EE[<λ x <Print<+x2>>>] = >
      (λ(f,o).
             IsClosure f → EE[f 3] (Addenv 1 (f 2) (f 4),o)
            | Iseta f → EE[<γ<γ<λ <f 2> <f 3> > f > 1 > ] (PE,o)
            | (f eq Y) & (IsClosure 1) → ((η,1 2,1 3,1 4),o)
            | (f 1,o)
      ) =

((λ,x,<Print<+x2>>,PE),nil) = >
      (λ(f,o).IsClosure f → EE[f 3] (Addenv 1 (f 2) (f 4),o)
            | Iseta f → EE[<γ<γ<λ <f 2> <f 3> > f > 1 > ] (PE,o)
            | (f eq Y) & (IsClosure 1) → ((η,1 2,1 3,1 4),o)
            | (f 1,o)
      ) =

EE[<Print<+x2>>] (Addenv 1 x PE,nil) =
EE[<Print<+x2>>] ((λN.N eq x → 1 | PE N),nil) =
((λN.N eq x→1|PE N),nil) = > EE[<+x2>] = > (λ(v,o).PE Print v ((λN.N eq x→1|PE N),o)) =
```

$EE[2]\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),nil) =>$

$\quad (\lambda(v_2,o).EE[x]\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o) =>$

$\quad\quad (\lambda(v_1,o).PE + v_1\ v_2\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o))$

$\quad) => (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$(2,nil) =>$

$\quad (\lambda(v_2,o).EE[x]\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o) =>$

$\quad\quad (\lambda(v_1,o).PE + v_1\ v_2\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o))$

$\quad) => (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$EE[x]\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),nil) => (\lambda(v_1,o).PE + v_1\ 2\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =>$

$\quad (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$((\lambda N.N\ eq\ x \rightarrow 1|PE\ N)\ x,nil) => (\lambda(v_1,o).PE + v_1\ 2\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =>$

$\quad (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$(1,nil) => (\lambda(v_1,o).PE + v_1\ 2\ ((\lambda N.N\ eqx \rightarrow 1|PE\ N),o)) =>$

$\quad (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$PE + 1\ 2\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),nil) => (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$(1 \in Integer+Real)\&(2 \in Integer+Real) \rightarrow (1+2),nil)|error =>$

$\quad (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$(3,nil) => (\lambda(v,o).PE\ Print\ v\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),o)) =$

$PE\ Print\ 3\ ((\lambda N.N\ eq\ x \rightarrow 1|PE\ N),nil) =$

(dummy, nil aug 3)   WHEW !!!