Define

The `define` *program* adds a binding to a top-level environment. New bindings are added to the front of the top-level `Bindings` object so that new bindings shadow old bindings, effectively allowing one to "redefine" a symbol.

define is not an expression!!! It is a program. This means that you cannot nest it inside other expressions (e.g., ***can't do*** `let ... in { define ... }`).

Syntax

Tokens

```
DEFINE 'define'
EQUALS '='
```

BNF

```
<program>:Define ::= DEFINE <VAR> EQUALS <exp>
<program>:Eval ::= <exp>
```

Semantics

```
Program
%%%
    public static Env env = Env.initEnv(); // the initial environment
%%%

Define
%%%
    // notice that calling $run() triggers a modification
    // of the initial environment
    public void $run() {
        Env env = Program.env;     // the top-level environment
        String s = var.toString(); // the LHS of the define
        Val val = exp.eval(env);   // the RHS value
        Binding b = env.lookup(s); // only look at local bindings
        if (b != null)
            b.val = val;           // replace the binding
        else
            env.add(new Binding(s, val));
        System.out.println(s);
    }
%%%

Eval
%%%
    public void $run() {
        Val val = exp.eval(env);
        System.out.println(val);
    }
%%%
```

Examples

```
define i = 1
define ii = add1(i)
define iii = add1(ii)
define v = 5
define x = 10
define f = proc(x) if zero?(x) then 1 else *(x,.f(.g(x)))
.f(v) % ERROR: g is unbound
define g = proc(x) sub1(x)
.f(v) % => 120 -- g is now bound
.f(iii) % => 6
```

Since RHS of a define are evaluated in the top-level environment, it supports recursion without the use of `letrec`. Here are some examples.

```
define even? = proc(x)
if zero?(x) then 1 else .odd?(sub1(x))
.even?(11) % => Error: unbound procedure odd?
define odd? = proc(x)
if zero?(x) then 0 else .even?(sub1(x))
.even?(11) % => 0
.odd?(11) % => 1
```

We can *redefine* a symbol. Effectively the original binding remains, but the lookup algorithm stops when it finds the first occurrence of the symbol and returns its value.

Here is an example.

```
define x = 2
define f = proc() x % x is the top-level x
.f()                % evaluates to 2
define x = 3        % redefine top-level x
.f()                % now evaluates to 3
```

Let's look at a slightly different example.

```
define x = 2
define f =
    let
        x = x       % the RHS is the current value (2),
                    % and the LHS is a local copy
    in
```

```
        proc() x    % the proc captures the local copy
.f()                % evaluates to 2
define x = 3        % redefine top-level x
.f()                % local copy still evaluates to 2
```