

Copyright (C) 2021 Timothy Fossum

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. A copy of the license is included in the file "COPYING", entitled "GNU Free Documentation License".

Language SET

In this version of our source language, we allow for the assignment of values to variables. Languages that allow for the mutation of variables are called *imperative*. Compared to functional programming (V6 is an example), such languages are inherently more difficult to reason about, which accounts for why functional programming has received so much attention and also for why it is so difficult to write high-quality software in most side-effecting programming languages.

So far, Languages V1 through V6 have treated denoted values (the values that variables are bound to) as being the same as expressed values (the values that expressions can have). For example, a variable x in one of these languages denotes the same thing no matter where it appears in its scope.

When we add variable mutation (also called “assignment”), such as

```
set x = add1(x)
```

the meaning of x on the LHS is different from its meaning on the RHS. The expression x in the RHS of this “assignment” represents an expressed value, while the x on the LHS represents a denoted value that can be modified. In order to support variable assignment, we need to find a way to disconnect denoted values from expressed values.

Language SET (continued)

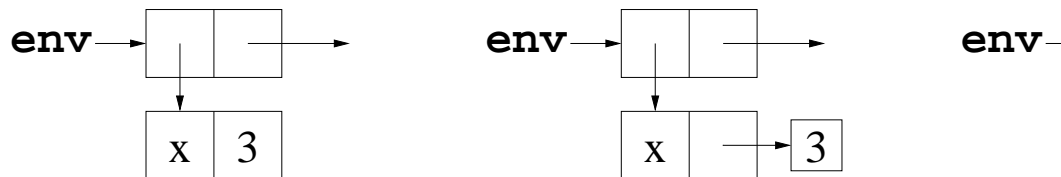
We introduce the notion of a *reference*, something that *refers to* a memory location. Instead of binding a variable directly to an expressed value, we bind a variable to a reference containing an expressed value. From a computational point of view, a reference is simply the address of a memory location. The address never changes, but the memory contents at the address can change.

Expressed value = Val = IntVal+ProcVal

Denoted value = Ref(Expressed)

To mutate a variable bound to a reference, we change the *contents* of the memory location. The variable is still bound to the same reference.

Denoted = Expressed Denoted = Ref(Expressed) (;



The two right-hand diagrams depict the same environment. The right diagram is a more compact representation.

Language SET (continued)

References will also be used to implement various parameter-passing schemes as described later in these notes.

We choose the following concrete and abstract syntax for variable mutation.

$\langle \text{exp} \rangle : \text{SetExp} ::= \text{SET } \langle \text{VAR} \rangle \text{ EQUALS } \langle \text{exp} \rangle$
 $\text{SetExp}(\text{Token } \text{var}, \text{Exp } \text{exp})$

We can now write the following program in our newly extended source language.

```
let
  x = 42
in
  { set x = add1(x) ; x }
```

This evaluates to 43.

Language SET (continued)

The ability to modify the value bound to a variable allows us to “capture the environment in a procedure and use the procedure to modify its capture”. For example, consider:

```
define g = let
    count = 0
in
    proc() set count = add1(count)

.g() % => 1
.g() % => 2
.g() % => 3
```

The value of `count` is captured in the local `let` bindings that define `g`. Each time we evaluate `.g()`, the procedure increments the value of `count` and returns this newly incremented value. The variable `count` persists from one evaluation to the other because the `proc` captures the environment in which it was created, namely the one with the variable `count`.

In this example, the `count` variable is unbound in the top-level environment. An attempt to evaluate it throws an exception:

```
count % unbound variable
```

Language SET (continued)

In our Java implementation, we want a reference to be a Java object that can be mutated. When we bind a variable to a reference object (its contents), this binding does not change, but the contents of the reference itself – what it refers to – can change.

The `Ref` abstract class embodies our notion of a reference – the thing that can be bound to. For now, the only subclass of the `Ref` class is the `ValRef`.

```
ValRef(Val val)
```

The contents of a `ValRef` object is a `Val`, and we say that such a reference is a *reference to a value*. (Recall that a `Val` object is either an `IntVal` or a `StrVal` – the only two `Val` types that we currently have.)

A `Ref` object has two methods:

```
public abstract Val deRef();
public abstract Val setRef(Val v);
```

In the `ValRef` class, The `deRef` (dereference) method simply returns the `Val` object stored in the object’s `val` field, and the `setRef` (set reference) method modifies the `val` field by changing it to the `Val` parameter `v` (and creating a new `Val` object as well).

Language SET (continued)

```
Ref
%%%
public abstract class Ref {

    public abstract Val deRef();
    public abstract Val setRef(Val v);

}
%%%
```

Language SET (continued)

```
ValRef
%%%
public class ValRef extends Ref {

    public Val val;

    public ValRef(Val val) {
        this.val = val;
    }

    public Val deRef() {
        return val;
    }

    public Val setRef(Val v) {
        return val = v;
    }

}
%%%
```

Language SET (continued)

Our denoted values (the things that variables are bound to) are now references instead of values, so we need to change our `Binding` objects to bind identifiers to a reference. (Notice that we use the terms “variable”, “identifier” interchangeably.)

```
Binding(String id, Ref ref)
```

In the `Env` class, we want `applyEnv` to continue to return the `Val` directly bound to a symbol. Since we now bind identifiers to references, we change the responsibilities as follows:

```
// returns the reference bound to sym
public abstract Ref applyEnvRef(String sym);

public Val applyEnv(String sym) {
    return applyEnvRef(sym).dereference();
}
```

The `applyEnvRef` method behaves exactly like the V6 `applyEnv` method, turning the thing (now a `Ref`) bound to the symbol and throwing an exception if there is nothing bound to the given symbol. The `applyEnv` method now returns a `Val`: it simply gets the `Ref` object using `applyEnvRef` and then returns its corresponding value.

Language SET (continued)

In our semantics code, we need to modify all of the instances of `Binding` objects so that they use references instead of values. To change the binding of a variable to a value, first wrap the value into a new reference and then change the variable to the newly created reference. Here’s an example of binding the variable named `x` to (a reference to) an integer `10`:

```
String var = "x";
Val val = new IntVal(10);
Binding b = new Binding(var, new ValRef(val));
```

The `valsToRefs` static method in the `Ref` class takes a list of `Val` objects and returns a corresponding list of `Refs`. This is used, for example, in the code for `LetExp` objects (which need to bind formal parameter symbols to references to parameter values) and for `LetExp` objects (which need to bind their symbols to references to their RHS expression values).

```
public static List<Ref> valsToRefs(List<Val> valList) {
    List<Ref> refList = new ArrayList<Ref>(valList.size());
    for (Val v : valList)
        refList.add(new ValRef(v));
    return refList;
}
```

Language SET (continued)

```
<exp>:SetExp ::= SET <VAR> EQUALS <exp>  
                SetExp(Token var, Exp exp)
```

So far, we have dealt only with the implementation details of `envir`.
do we implement the semantics of `set` expressions? Coding this is

```
SetExp  
%%%  
    public Val eval(Env env) {  
        Val val = exp.eval(env); // the RHS expression  
        Ref ref = env.applyEnvRef(var); // the LHS variable  
        return ref.setRef(val); // sets the ref and returns val  
    }  
%%%
```

Remember that an expression (`<exp>`) always evaluates to a `val`.
a `SetExp` is no exception: the value of a `SetExp` is the value of
assignment. This means that multiple `set` operations can appear in a

Language SET (continued)

Let's look the three lines in the `eval` method shown on the previous slide:

```
Val val = exp.eval(env); // the RHS expression  
Ref ref = env.applyEnvRef(var); // the LHS variable  
return ref.setRef(val); // sets the ref and returns val
```

Notice that the `exp.eval(env)` expression returns a `Val` object,
whereas the `env.applyEnvRef(var)` expression returns a `Reference`.
reference). We use the terms **value semantics** to refer to obtaining the
thing and **reference semantics** to refer to obtaining a reference to something.
code above shows that *we use value semantics for the RHS of a set expression*
reference semantics for its LHS.

Because we need to modify the value denoted by the LHS variable
(as shown above), we must use reference semantics for the LHS. `Val`
would have been useless here, since we regard “values” – instances of
– as immutable: things that cannot be modified.

In a `set` expression, the LHS variable must exist in the environment.
expression occurs, otherwise we could find no reference to modify.
to the LHS variables occurring in `let` expressions. A `let` expression
bindings to the LHS variables: we don't *modify* these variables, we

Language SET (continued)

A variable that occurs on the LHS of a set expression is not an particular, observe that the grammar rule for a set expression uses VarExp, to the left of EQUALS, and we use reference semantics, not values, for such an occurrence, because we are not treating the LHS expression.

When we evaluate a VarExp (this *is* an expression), either by itself or another expression, we use value semantics in Language SET. This is how you examine the eval semantics for a VarExp:

```
public Val eval(Env env) {  
    return env.applyEnv(var); // value semantics!  
}
```

Even though all variables are bound to references in an Env, the eval method de-references the reference bound to var to return its value.

Observe that expressed values (instances of Val) get wrapped into references (instances of Ref) when they are bound to variables in creating env. For example, when evaluating the RHS expressions in a let/letrec expression, we are creating the actual parameter expressions during a procedure application. We evaluate these expressions using value semantics in Language SET before wrapping them into ValRef objects.

Language SET (continued)

To illustrate how set expressions evaluate to something, consider the following let expression in Language SET:

```
let  
  t = 3  
  u = 42  
  v = 0  
in  
  { set v = set u = set t = add1(t) ; +(t, +(u, v)) }
```

This expression evaluates to 12. The first expression in the body (the sequence expression) gets evaluated like this:

```
set v = { set u = { set t = add1(t) } }
```

The innermost set expression sets t to 4 and evaluates to 4. Proceeding to the next set expression sets u to 4 (the value of the innermost set expression is 4). Finally, the outermost set expression sets v to 4 and evaluates to 4. The value gets discarded by the sequence expression, but the last expression in the body uses the modified values of t, u, and v.

Language SET (continued)

What happens if you try to mutate the value of an identifier that is or parameters to a procedure? For example, what value is returned by program?

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

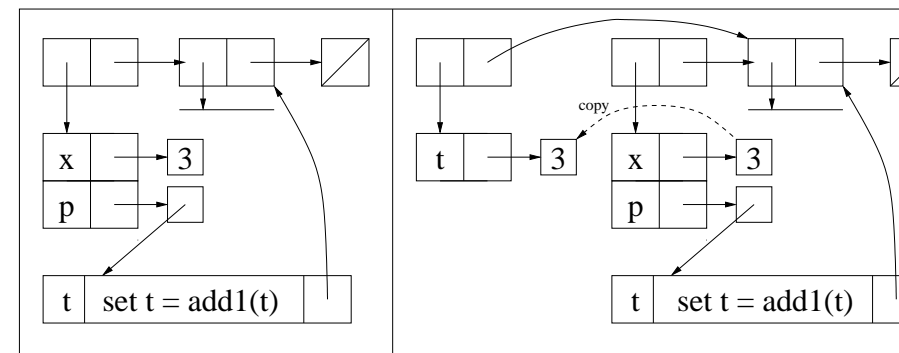
In our procedure application semantics (see the AppExp code), the identifiers are bound to (references to) the *values* of the actual parameters. Since the value of the actual parameter x in the expression $.p(x)$ is 3, this variable t in the procedure application is bound to *a new reference* and evaluating the body of the procedure modifies this reference to t , it's the reference to variable t , not the variable x , that gets modified. The value of this entire expression is 3.

Language SET (continued)

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

The following illustration shows

- the environment immediately before the procedure application $.p(x)$, in particular, the binding of x to a reference to the value 3, and
- the environment during the procedure application $.p(x)$, binding parameter t to a *new* reference containing a copy of the value 3 (indicated by the dashed line).

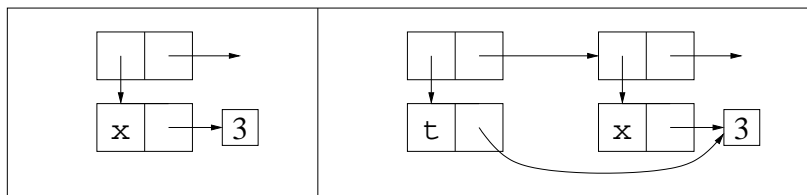


Language REF

A parameter passing approach that evaluates actual parameters using semantics and that binds the formal parameters to these actual parameter values *call-by-value*. This is what we use in languages V1 to V6.

In the language SET, where bindings are to references instead of values, the actual parameter values into *new* references, and these references are the formal parameters. Though denoted values are references in Language REF, the parameter passing approach is still call-by-value.

Considering the illustration in the previous slide, Suppose we *want* the formal parameter t to be bound to the *same* reference that is bound to x . The following diagram illustrates how the bindings in the previous diagram change when t is bound to the same reference as x :



Such a parameter passing semantics is called *call-by-reference*. We will discuss call-by-reference next, along with variants on this theme.

Language REF (continued)

To repeat:

- The parameter passing semantics that we have been using up to now is *call-by-value*. In call-by-value semantics – which we have referred to as *call-by-value semantics*, when an actual parameter expression in a procedure application is a variable, the procedure's corresponding formal parameter denotes a new reference to the expressed value of the actual parameter.
- In *call-by-reference* semantics – which we have referred to as *call-by-reference semantics*, when an actual parameter expression in a procedure application is a variable, the procedure's corresponding formal parameter denotes the same reference as the actual parameter.

The differences between call-by-value and call-by-reference semantics arise when the actual parameter expression is a variable. When the actual parameter expression is not a variable, the corresponding formal parameter denotes a new reference to the expressed value of the actual parameter, just as in Language REF.

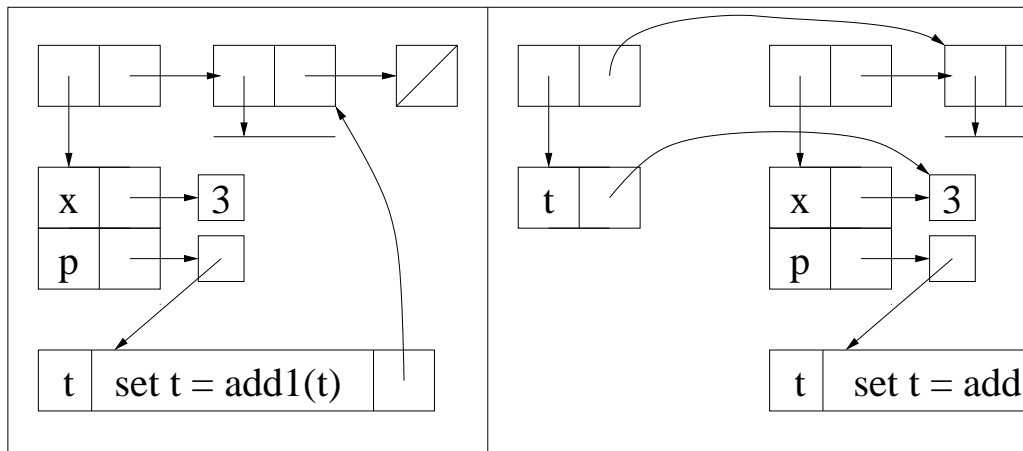
Observe that in `let` and `letrec` expressions, we always use values for the variable bindings. This means that each LHS variable in a `let` expression always denotes a new reference to the expressed value of its RHS expression.

Language REF (continued)

Using call-by-reference semantics, the program

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(x) ; x }
```

returns the value 4, since t denotes the same reference as x . The state created by the `let` and then during evaluation of the application `.p` to evaluating the procedure body) are illustrated in the following fig



Language REF (continued)

In Language REF, when actual parameter expressions are not themselves variables, we use value semantics. To illustrate this, consider the value returned by the following program:

```
let
  x = 3
  p = proc(t) set t = add1(t)
in
  { .p(+ (x, 0)) ; x }
```

Clearly the expressed value of the actual parameter `+(x, 0)` is the value of x , but the expression `+(x, 0)` is not a variable, so value semantics is used for this actual parameter. This means that when we apply the procedure to this actual parameter, the parameter t denotes a *new* reference to the value of this expression. Thus, t and x have the same expressed values, but they have different denotations. Modifying t does not affect the value of x . This expression evaluates to

Language REF (continued)

The term *L-value* refers to a semantic entity that can be considered. It's called an L-value because it is the sort of thing that can appear on the left-hand side of the '=' in a `set` expression. In Languages SET and REF, a variable is always considered as an L-value (because variables are always bound to references). An expression like `+(x, 0)` can only be considered as a value, never as an L-value.

Whether we consider a semantic entity as an L-value depends on the context. In the expression

```
set x = 3
```

the occurrence of `x` is considered as an L-value. On the other hand, in

```
set y = x
```

the occurrence of `x` is not considered as an L-value.

In Languages SET and REF, only variables can be considered as L-values. In Language SET, actual parameter expressions (even variables) *in procedure calls* are never considered as L-values. In Language REF, only actual parameters that are variables are considered as L-values.

Language REF (continued)

*To summarize: for procedure applications that appear in Language REF, an actual parameter is a variable (and therefore denotes a reference), then the corresponding formal parameter to the same reference. If an actual parameter is something other than a variable, then the corresponding formal parameter is bound to a **new** temporary reference containing the expressed value of the actual parameter.*

Language REF (continued)

Our REF language has exactly the same grammar rules as our SET language. The *only* differences are in the bindings of formal parameters during compilation. As the discussion on the previous slides show, we need to distinguish parameters that are variables differently from actual parameters (values). The idea here is to define an `evalRef` method for instance classes that takes care of how to translate themselves into a reference. For a `VarExp`, `evalRef` evaluates the expression and returns a new reference to the value. For a `FormalParamExp`, `evalRef` returns the same reference that the parameter denotes.

So in the `Exp` class, the `evalRef` method has the following *default* implementation:

```
public Ref evalRef(Env env) {  
    return new ValRef(eval(env)); // value semantic  
}
```

For the `VarExp` subclass – *and only for this class*, `evalRef` is implemented as follows:

```
public Ref evalRef(Env env) {  
    return env.applyEnvRef(var); // reference semantic  
}
```

The `evalRef` method in the `VarExp` class overrides the `evalRef` method in the `Exp` class. In all other classes that extend the `Exp` class, the default implementation of the parent `Exp` class is used.

Language REF (continued)

The other change is in the `Rands` code. In the SET language, the `eval` method was used in the implementation of `eval` for both a `LetExp` and an `AppExp` object, since both created new bindings to values. In the REF language, an `AppExp` object needs new bindings to values except for actual parameters that are variables – a situation that is described in the previous slides. To implement the correct `eval` semantics for an `AppExp` object, we need to use `evalRef` instead of `eval` values to bind them to the formal parameters. The method `evalRandsRef` in the `Rands` class does this work. The `eval` method in the `AppExp` class uses the `evalRandsRef` method to create new bindings of the formal parameters to their appropriate references. The implementation of `evalRandsRef` follows:

```
public List<Ref> evalRandsRef(Env env) {  
    List<Ref> refList = new ArrayList<Ref>(expList.size());  
    for (Exp exp : expList)  
        refList.add(exp.evalRef(env));  
    return refList;  
}
```

Language REF (continued)

Remember that we always use value semantics for `let` bindings. Take a Language REF program such as

```
let
  x = 3
in
  let
    y = x
  in
    { set y = add1(y) ; x }
```

evaluates to 3.

Our observation (see Slide 3.79) that any `let` can be re-written as a procedure application no longer applies with languages that implement reference semantics. Specifically, if we attempt to re-write the inner above Language REF program as a procedure application using the `let` on Slide 3.79, we get

```
let
  x = 3
in
  .proc(y) {set y = add1(y) ; x } (x)
```

which evaluates to 4.

Language REF (continued)

We want literals (LITs) always to have value semantics. For example, `4` appears in an expression should always evaluate to the integer 4. Consider the following code in Language REF:

```
define square = proc(x) set x=* (x,x)
define four = 4
{ .square(four) ; four }
```

By the time the variable `four` gets evaluated the second time in the sequence expression, its value has changed to 16, because Language REF uses reference semantics for the actual parameter `four`. Thus the value of the sequence expression is 16. Consider now what happens if we replace the sequence expression with the following:

```
{ .square(4) ; 4 }
```

Of course, this sequence expression evaluates to 4, because Language REF uses value semantics for everything but variables. However, languages like FORTRAN IV (in the 1970s) treated numeric literals (like ‘4’) as variables with reference semantics when passing them to procedures. The equivalent sequence expression, if written FORTRAN IV, would evaluate to 16. Further sequence statements such as

```
IF 4 = 16 THEN CALL YIKES
```

(not really a legal FORTRAN IV statement) would end up calling YIKES.

Language NAME

We now turn to a different parameter passing mechanism, *call-by-name* procedure application, we bind each procedure's formal parameter to the corresponding *un-evaluated actual parameter expression*. Each time we evaluate a formal parameter in the procedure body, we evaluate its corresponding actual parameter expression *in the environment where the procedure was called*, and this value becomes the expressed value of the formal parameter.

Call-by-name has behaviors that differ from call-by-reference: (1) in call-by-reference the formal parameter in the procedure body, we never evaluate the actual parameter expression; and (2) every time we evaluate the formal parameter in the procedure body, we re-evaluate the actual parameter expression.

In the presence of side-effects, call-by-name has interesting properties. It is very powerful but often difficult to reason about. The language *A* uses both call-by-value and call-by-name as its parameter passing mechanism. The language *A* had its greatest influence on languages such as Pascal, C/C++, and Java. Call-by-name has been all but abandoned by modern imperative programming languages – mostly because of its inefficiency, it still is used in functional programming: Scheme supports a variant, *call-by-need* (also known as *promise/force*; Haskell also supports call-by-need. We proceed with both call-by-name and call-by-need.

Language NAME (continued)

We implement Language NAME with call-by-name semantics. It is similar to Language REF.

Two actual parameter expressions that can appear in procedure applications are *constant behavior*: They are `LitExp` and `ProcExp`. Evaluating these expressions do not produce any side-effects, and their expressed values (`LitVal` and `ProcVal`, respectively) can never change. (Don't confuse evaluating these expressions with applying the procedure.) Even in call-by-name, we can evaluate these actual parameter expressions before the procedure application can bind their corresponding formal parameters to new references. In other words, we use value semantics for these actual parameter expressions in Language NAME.

Evaluating variables that appear as actual parameter expressions (`VarExp`) can produce side-effects, but such variables may have expressed values that can change, perhaps in `set` expressions. We therefore use reference semantics for these actual parameter expressions that are variables in Language NAME.

Language NAME (continued)

In the presence of side-effects, call-by-reference and call-by-name can produce different results. Consider evaluating the following expression in Language NAME.

```
let
  x = 1
  f = proc(t,u)
    { set t = add1(t) ; u }
in
  .f(x, +(x,5))
```

With call-by-reference, when we evaluate the application $.f(x, +(x,5))$, the formal parameter t in the definition of f denotes the same reference (initially containing 1), whereas the formal parameter u denotes a reference to the value 6 (using value semantics for the $+(x,5)$ expression) in the body of f changes the expressed value of x (because t and x are the same reference) but does not change the expressed value of u . Thus the entire expression evaluates to 6.

Language NAME (continued)

Now consider the same expression in Language NAME.

```
let
  x = 1
  f = proc(t,u)
    { set t = add1(t) ; u }
in
  .f(x, +(x,5))
```

Consider what happens when we evaluate $.f(x, +(x,5))$ using call-by-name. The formal parameter t still denotes the same reference that x denotes (initially containing 1), but the formal parameter u denotes the (un-evaluated) expression $+(x,5)$.

The `set` operation in the body of this procedure increments the value of t ; but since t denotes the same reference as x , the value of x changes to 2. When we then evaluate the formal parameter u at the end of the procedure, we evaluate the expression $+(x,5)$ denoted by u *in the environment* where x is 2. Since this expression gets evaluated after the `set`, and the value of x has changed, the value of the expression $+(x,5)$ (and thus the value returned by the application) is $+(2,5)$ or 7. Thus the entire expression evaluates to 7.

Language NAME (continued)

Consider the following definition in Language NAME:

```
define while = proc(test?, do, result)
  letrec loop = proc()
    if test? then {do ; .loop()} else result
  in .loop()
```

Using call-by-name, the expression

```
let x = 0 sum = 0 in
  .while(
    <=? (x, 10), % t
    { set sum=+(sum, *(x, x)) ; set x = add1(x) }, % c
    sum % i
  )
```

evaluates to the sum

$$\sum_{x=0}^{10} x^2 = 385$$

Suppose we were to consider these in Language REF. In this case, expression evaluation would never terminate. This is because the actual expression `<=? (x, 10)` is evaluated only once, to 1 (true) when `test?` so the `test?` parameter is bound permanently to (a reference to) `test?` repeatedly always returns 1 (true), so the “loop” never terminates.

Language NAME (continued)

We proceed to implement call-by-name. We take our Language REF (call-by-reference) implementation as a starting point.

If an actual parameter is a literal expression (such as 4), we bind the formal parameter to (a reference to) the literal value. If an actual parameter is a closure, we bind the formal parameter to (a reference to) the procedure’s closure object in the environment. If an actual parameter is an identifier, we bind the formal parameter to the same reference as the actual parameter, using reference semantics.

If an actual parameter is any other kind of expression, we bind the formal parameter to a `Ref` object that captures the expression in the environment in which it was called and that can be evaluated, when needed, by the called procedure. We call such an object a *thunk*.

Following our terminology for defining *value semantics* and *reference semantics* discussed earlier, we call this *name semantics*.

Language NAME (continued)

A thunk amounts to a parameterless procedure that consists of an expression and an environment in which the expression is to be evaluated. It looks just like a procedure object except that there is no formal parameter list.

```
ThunkRef(Exp exp, Env env)
```

A `ThunkRef` is a `Ref`, since we want to de-reference (`deRef`) it to refer to the corresponding actual parameter. We bind a formal parameter to a `ThunkRef` only during procedure application. Thunks will otherwise behave like procedures in expression semantics.

Language NAME (continued)

To change from call-by-reference to call-by-name, we need to change the `evalRef` behavior of the `Exp` objects so that `evalRef` returns a `ThunkRef` for all expressions *except for* `Literal`, `ProcExp`, and `VarExp`. This means we need to define `evalRef` in the `Exp` class with its default behavior as follows:

```
public Ref evalRef(Env env) {  
    return new ThunkRef(this, env);  
}
```

For a `Literal` and a `ProcExp`, a thunk is not necessary, so we return a `ValRef` as in the REF language:

```
public Ref evalRef(Env env) {  
    return new ValRef(eval(env));  
}
```

Finally, for a `VarExp`, we simply use reference semantics as in the REF language:

```
public Ref evalRef(Env env) {  
    return env.applyEnvRef(var);  
}
```

Language NAME (continued)

The ThunkRef class is straight-forward:

```
ThunkRef
%%%
public class ThunkRef extends Ref {

    public Exp exp;
    public Env env;

    public ThunkRef(Exp exp, Env env) {
        this.exp = exp;
        this.env = env;
    }

    public Val deRef() {
        return exp.eval(env);
    }

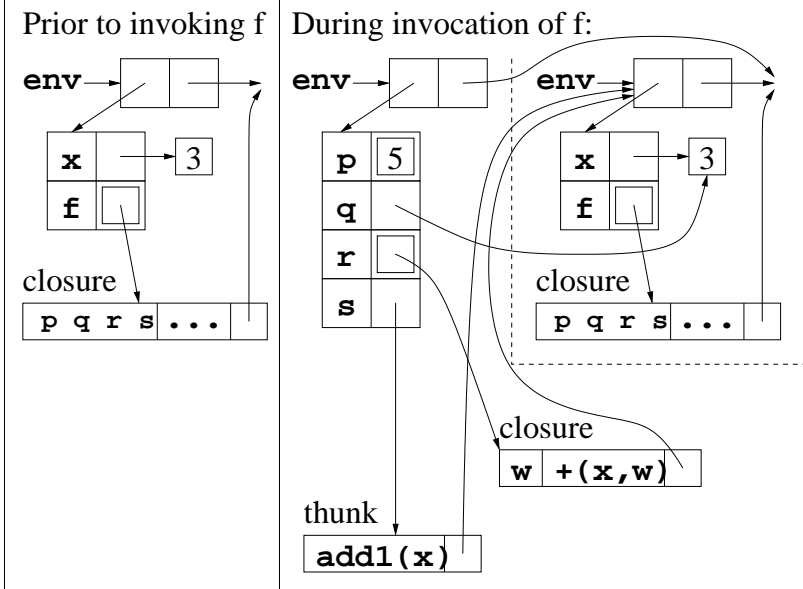
    public Val setRef(Val v) {
        throw new PLCCEException("cannot modify a read-only exp");
    }
}
%%%
```

Observe that the setRef method throws an exception. The setRef method is only used with LHS variable references during evaluation of set expressions.

Language NAME (continued)

The following illustration may help you to understand how these expressions are evaluated. The example shows all four possible cases of actual parameter expressions: variable, procedure, and other:

```
let x = 3
    f = proc(p,q,r,s) ...
in .f(5, x, proc(w) +(x,w), add1(x))
```



Language NEED

The call-by-need parameter passing mechanism is the same as call-by-name, except that a thunk is called at most once, and its value is remembered (*memoized*).

Suppose a procedure with formal parameter `x` is invoked with argument `add1(z)`. Suppose that a thunk with body `add1(z)` is created, and that the formal parameter `x` is bound to the thunk with `set x = add1(z)`. As with call-by-name, the formal parameter `x` is bound to the thunk with `set x = add1(z)` in an enclosing environment that we will assume has `z` bound to the value 8. When `x` is referenced in the body of calling procedure, the thunk is dereferenced, producing a result of 9 for `add1(z)`. The thunk now remembers (*memoizes*) the value 9, and any further reference to the formal parameter `x` in the body of the procedure will continue to evaluate to 9, making any further changes to the variable `z`.

If call-by-name had been used in the above example, additional evaluation of the body of the procedure would result in evaluating the body of the thunk again, thus further modifying `z` and yielding values 10, 11, 12, etc.

In both call-by-need and call-by-name (and unlike call-by-reference), the parameter is never referenced in the body of the procedure, the thunk is created. Compared to call-by-name, call-by-need reduces the overhead of evaluating a thunk when evaluating the corresponding formal parameter.

Language NEED (continued)

Implementing call-by-need is easy, starting from the call-by-name implementation. The principal change is to have a `Val` field named `val` in the `ThunkRef` structure, used to memoize the value of the body of the thunk. This field is initialized to `null` when the thunk is created. When the thunk's `deRef` method is invoked, it first checks to see if the `val` field has been memoized (*i.e.*, is non-null). If so, the method simply returns the memoized value. Otherwise, it evaluates the body of the thunk, saves the value in the `val` field (thereby memoizing it), and then returns the value. Subsequent `deRef` calls simply use the resulting memoized value.

In the NEED language, the `ThunkRef` constructor initializes the `val` field to `null`, indicating that the thunk has not been memoized. This field is updated when the thunk's `deRef` method is called.

Language NEED (continued)

Here are the appropriate changes to ThunkRef ...

```
ThunkRef
%%%
public class ThunkRef extends Ref {

    public Exp exp;
    public Env env;
    public Val val; // memoized value

    public ThunkRef(Exp exp, Env env) {
        this.exp = exp;
        this.env = env;
        this.val = null;
    }

    public Val deRef() {
        if (val == null)
            val = exp.eval(env);
        return val;
    }

    ...

}
```

Language NEED (continued)

You might have noticed in the `code` file that an instance of the class constructed by the `evalRef` methods in the `LitExp` and `ProcExp` is a slight change from the `NAME` language, where the instances were `ReadOnly`. The `RO` part stands for “Read Only”. We do this because it doesn’t make sense for a literal or procedure to be modified.

Consider, for example, the following code:

```
let
  f = proc(x) set x=add1(x)
in
  .f(3)
```

One’s intuition would be to think of the procedure application `.f(3)` as:

```
set 3=add1(3)
```

But of course this doesn’t make any sense.

We have already seen that the `setRef` method in the `ThunkRef` class throws an exception. What we are now doing is to have this same behavior for a procedure parameter expression except for a variable (where call-by-reference is required).

Language NEED (continued)

The following example illustrates the difference between call-by-name and call-by-need:

```
let
  x = 3
  p = proc(t) {t;t;t}
in
  .p(set x=add1(x))
```

With call-by-name, when we apply the procedure `p`, its formal parameter `t` is bound to a thunk containing the expression `set x=add1(x)`. Each time we evaluate the formal parameter `t` in the body of the procedure `p`, its thunk is evaluated, resulting in evaluation of the expression `set x=add1(x)`. So since `t` occurs three times in the body of `p`, the expression `set x=add1(x)` is evaluated three times, incrementing `x` from 3 to 6. Consequently, the entire expression evaluates to 6.

With call-by-need, the first time we evaluate `t` in the body of `p`, its actual parameter expression `set x=add1(x)` is evaluated, with the effect of incrementing the value of `x` to 4 and evaluating to 4. How does it memoize the expressed value of 4, so any further references we make to `t` evaluate to 4. Consequently, the entire expression evaluates to 4.

Language NEED (continued)

Here's another example illustrating the difference between call-by-name and call-by-need. Examine the definition of `seq`, which seems to terminate finitely but doesn't with call-by-name (why?).

```
define pair = proc(x,y)
  proc(t) if t then y else x
define first = proc(p) .p(0)
define rest  = proc(p) .p(1)
define nth   = proc(n,lst) % zero-based
  if n then .nth(sub1(n),.rest(lst)) else .first(lst)
define seq   = proc(n) .pair(n,.seq(add1(n)))
define natno = .seq(0) % all the natural numbers
%% The above never terminates with call-by-name
%% With call-by-need or call-by-need, we get
.first(natno) % => 0
.first(.rest(natno)) % => 1
.first(.rest(.rest(natno))) % => 2, and so on
.nth(100,natno) % => 100
```

Order of evaluation

Let's examine the following example in Language SET (or REF):

```
let
  x = 3
in
  let
    y = {set x = add1(x)}
    z = {set x = add1(x)}
  in
    z
```

Consider the inner `let`. We know that the right-hand side expressions (here we use braces for clarity) are evaluated before their values are bound to the left-hand variables. In Language SET and REF *specify* the order in which the right-hand side expressions are evaluated. (In Language V6, the RHS expressions in a `let` cannot produce side effects, so the order of evaluation of RHS expressions wouldn't matter. However, when side-effects are present, the order of evaluation does matter.)

Suppose we did not specify this order of evaluation. In the above example, if the expression `add1(x)` is evaluated first, then `z` becomes 4 and `y` becomes 5, so the entire expression evaluates to the value of `z`. If the order of evaluation were reversed, the entire expression evaluates to the value of `y`. In Languages SET and REF specify the order of evaluation, we have an unambiguous value for the above expression. In the absence of such a specification, the value is ambiguous. *Do you know what your favorite language does?*

Order of evaluation (continued)

A similar situation exists when evaluating actual parameter expressions in a procedure call. Consider the following example, assuming call-by-value semantics.

```
let
  x = 3
  p = proc(t,u) t
in
  .p(set x = add1(x), set x = add1(x))
```

When the actual parameters are evaluated left-to-right as specified in the example, `t` would be bound to 4 and `u` to 5, so the entire expression evaluates to 4. If the evaluation order had been right-to-left, the entire expression would evaluate to 5.

Order of evaluation (continued)

You can see that both `evalRands` and `evalRandsRef` use `for` (also called enhanced `for` loops) to traverse and evaluate the expressions of actual parameters. The traversal is guaranteed by the Java API to be “natural”, in the sense that the elements of the list are visited in number order. Here is the code for `evalRands` in the `Rands` class.

```
public List<Val> evalRands(Env env) {
    List<Val> valList = new ArrayList<Val>();
    for (Exp e : expList)
        valList.add(e.eval(env));
    return valList;
}
```

Our order of evaluation semantics depends on the behavior of Java's order of evaluation mechanism, which is guaranteed to be left-to-right. If we had chosen right-to-left semantics, we could, instead, explicitly traverse from last to first if we wished.

The point is that, to make any language semantics well-defined and it is necessary to specify the order of evaluation. Unless the language clearly addresses the issue of order of evaluation, the language implementers choose any evaluation order. *Let the buyer beware!*

Order of evaluation (continued)

Both Java and Python specify that actual parameter expressions are evaluated left-to-right. However, the C language specification explicitly states that the order in which actual parameter expressions are evaluated is *undefined* – which means that the evaluation order is implementation dependent. In the following example, the output is 3 (using the GCC compiler on the date this file was last modified), which shows that the operand expressions `foo(3)` and `foo(5)` are evaluated left-to-left. Your mileage may vary!

```
#include <stdio.h>
#include <stdlib.h>
static int xx = 0;
void foo2(int x1, int x2) {
    return x1 + x2;
}
int foo(int x) {
    xx = x;
    return x;
}
int main(int argc, char ** argv) {
    foo2(foo(3), foo(5));
    printf("xx=%d\n", xx);
    return 0;
}
```

Order of evaluation (continued)

Order of evaluation does not matter in languages without side-effects, which makes functional languages immune to order of evaluation problems. http://en.wikipedia.org/wiki/Evaluation_strategy contains more information about order of evaluation.

Another way to avoid order of evaluation problems is to require that procedures have at most one formal parameter. In languages that use this approach with call-by-need, there is never an “order of evaluation” issue because there is never more than one actual parameter to evaluate.

While you may think that a language with procedures having only one formal parameter might be limited, it's possible for such a language to behave like a language with multiple formal parameters using an approach called “Currying”, as in the Haskell programming language – named after Haskell Curry. The next slide gives an example.

Order of evaluation (continued)

Here is an example without currying:

```
let
  x = 3
  y = 5
  p = proc(t,u) + (t,u)
in
  .p(x,y) % => 8
```

Here is semantically equivalent code that has exactly one formal parameter:

```
let
  x = 3
  y = 5
  p = proc(t) proc(u) + (t,u)
in
  . .p(x)(y)
```

In the second example above, `x` must be evaluated first, so that `.p(x)` can then be applied to the value of `y`.

Aliasing

Side-effecting languages that use call-by-reference suffer from a problem called *aliasing*. Consider, for example, the following program using the REF language:

```
let
  addplus1 = proc(x,y) {set x = add1(x) ; +
in
  .addplus1(3,3)
```

It's clear that this program returns 7. But what about the following program?

```
let
  a = 3
  addplus1 = proc(x,y) {set x = add1(x) ; +
in
  .addplus1(a,a)
```

Aliasing (continued)

```
let
  a = 3
  addplus1 = proc(x,y) {set x = add1(x) ; +
in
  .addplus1(a,a)
```

Using call-by-reference, when `addplus1` is applied to the actual arguments `a` and `a`, both formal parameters `x` and `y` of `addplus1` refer to the same memory location as `a`. Therefore the `set x = add1(x)` expression is equivalent to `set a = add1(a)` which increments `a` to 4, and the next expression `+ (x,y)` is essentially equivalent to the expression `+ (a,a)` which now evaluates to 8. The value of the program is 8.

Aliasing occurs when two different formal parameters refer to the same memory location as a parameter. As this example shows, aliasing can lead to unexpected results. **Of course, the best way to avoid problems of evaluation ambiguities and aliasing is to avoid using languages with side effects!**