Language SET

In this version of our defined language, we allow for the assignment of values to variables. Languages that allow for the mutation of variables are called *side-effecting*; 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 produce high-quality software in most side-effecting programming languages.

So far, our language has treated denoted values (the things that variables are bound to) as being the same as expressed values (the values that an expression can have). This is because a variable `x`, for example, will always mean the same thing no matter where it appears in its scope.

When we add assignment, such as with

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

the meaning of `x` on the LHS is different from its meaning on the RHS of the assignment. The RHS of this "assignment" represents an expressed value, whereas the LHS represents a change in the denoted value. In order to implement variable assignment, we need to find a way to disconnect denoted values from expressed values.

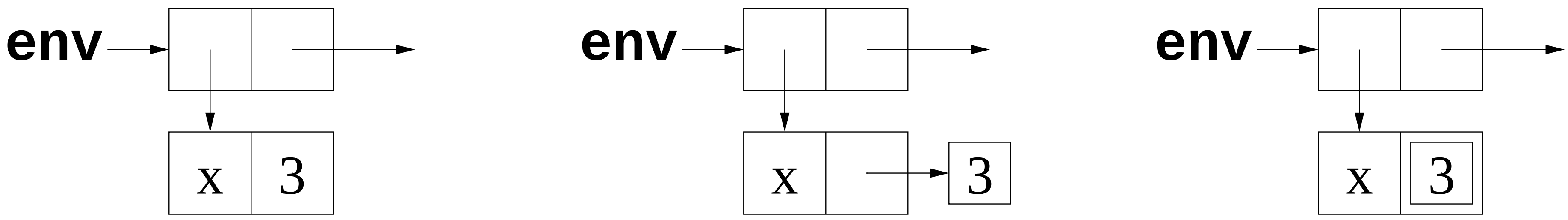
Language SET (continued)

We introduce the notion of a *reference*, something that *refers to* a mutable location in memory. Instead of binding a variable directly to an expressed value, we bind the variable to a reference, which will contain the expressed value.

Denoted value = Ref(Val)
Expressed value = Val = IntVal+ProcVal

If we want to mutate a variable, we must change the contents of the memory location the variable refers to, not the value (reference) the variable is bound to.

Denoted = Expressed   Denoted = Ref(Expressed)   (same as Ref)



The two right-hand diagrams depict the same environment. The rightmost one uses a more compact representation.

Language SET (continued)

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

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

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

We can now write the following program in our newly extended defined language:

```
let
  x = 42
in
  set x = add1(x)
```

This evaluates to 43.

Language SET (continued)

The ability to modify the value bound to a variable allows us to "capture" an environment in a procedure and use the procedure to modify its captured environment. 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 environment that defines the `proc()`. Each time we evaluate `.g()`, the procedure increments the value of `count` and returns this newly incremented value. The variable `count` persists from one invocation to the other because the `proc` captures the environment in which it is defined, namely the one with the variable `count`.

In this example, the `count` variable is unbound in the top-level environment, so an attempt to evaluate it will throw an exception:

```
count % unbound variable
```

Language SET (continued)

For our purposes, we want a reference to be a Java object whose contents can be mutated. When we bind a variable to a reference (its denoted value), this binding does not change, but the contents of the reference itself – the thing it refers to – can change.

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

```
ValRef(Val val)
```

The contents of a `ValRef` object is a `Val`, and we say that such an object is a *reference to a value*. (Recall that a `Val` object is either an `IntVal` or a `ProcVal` – 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 returning the 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 an identifier variable to a reference. (Notice that we use the terms “variable”, “identifier”, and “symbol” interchangeably.)

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

In the `Env` class, we want `applyEnv` to continue to return a `Val` object, whereas the bindings now associate identifiers with references, so we split up the responsibilities as follows:

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

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

The `applyEnvRef` method behaves exactly like the previous `applyEnv` method (but returns a `Ref` instead) and throws an exception if there is no reference bound to the given symbol. The `applyEnv` method simply gets the `Ref` object using `applyEnvRef` and dereferences it to return the corresponding value.

Language SET (continued)

In our semantics code, we need to modify all of the instances of `Binding` or `Bindings` objects so that they use references instead of values. To create a "binding" of a variable to a value, first convert the value into a reference and then bind the variable to the reference. Here's an example of how to create a binding of the variable `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`s and returns a corresponding list of `Ref`s. This is used, for example, in the code for `AppExp` objects (which need to bind formal parameter symbols to references to their actual parameter values) and for `LetExp` objects (which need to bind their LHS variable symbols to reference to their RHS expression values).

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

Language SET (continued)

```
SetExp(Token var, Exp exp)
```

So far, we have dealt only with the implementation details of environments. How do we implement the semantics of `set` expressions? Coding this is now simple:

```
SetExp
%%%
    public Val eval(Env env) {
        Val val = exp.eval(env);
        Ref ref = env.applyEnvRef(var);
        return ref.setRef(val);
    }
%%%
```

Notice that a `set` expression evaluates to the value of the RHS of the assignment. This means that multiple `set` operations can appear in one expression.

Language SET (continued)

For example, the following expression evaluates to 12:

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

Language SET (continued)

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

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

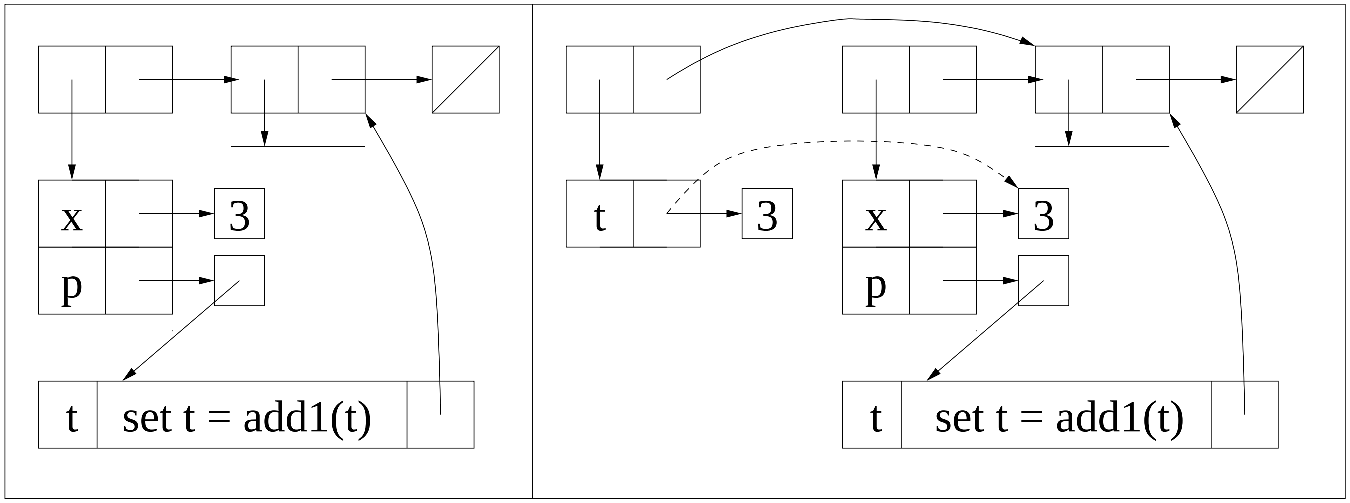
In our procedure application semantics (see the `AppExp` code), the formal parameters are bound to (references to) the *values* of the actual parameters. Since the value of the actual parameter `x` in the expression `.p(x)` is 3, this means that the variable `t` in the body of the procedure is bound to (a reference to) the value 3, and evaluating the body of the procedure modifies this binding to the value 4, but it's the variable `t`, not the variable `x`, that gets modified. Thus 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 the formal parameter `t` to a *new* reference to the value 3, not to `x`'s reference (which would be the dashed line).



Language REF

A parameter passing semantics that evaluates actual parameters and that binds the formal parameters to these actual parameter values is called *call-by-value*. This is what is used in languages `V1` to `V6`. In the language `SET`, where bindings are to references instead of values, the actual parameter values are turned into *new* references, and these references are bound to the formal parameters.

Suppose we *want* the behavior illustrated by the dashed line in the previous slide. That is, if a variable that is passed as an actual parameter to procedure, the corresponding formal parameter is bound to the *same* reference as the actual parameter variable, not a new reference with a copied value.

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

Language REF (continued)

To repeat:

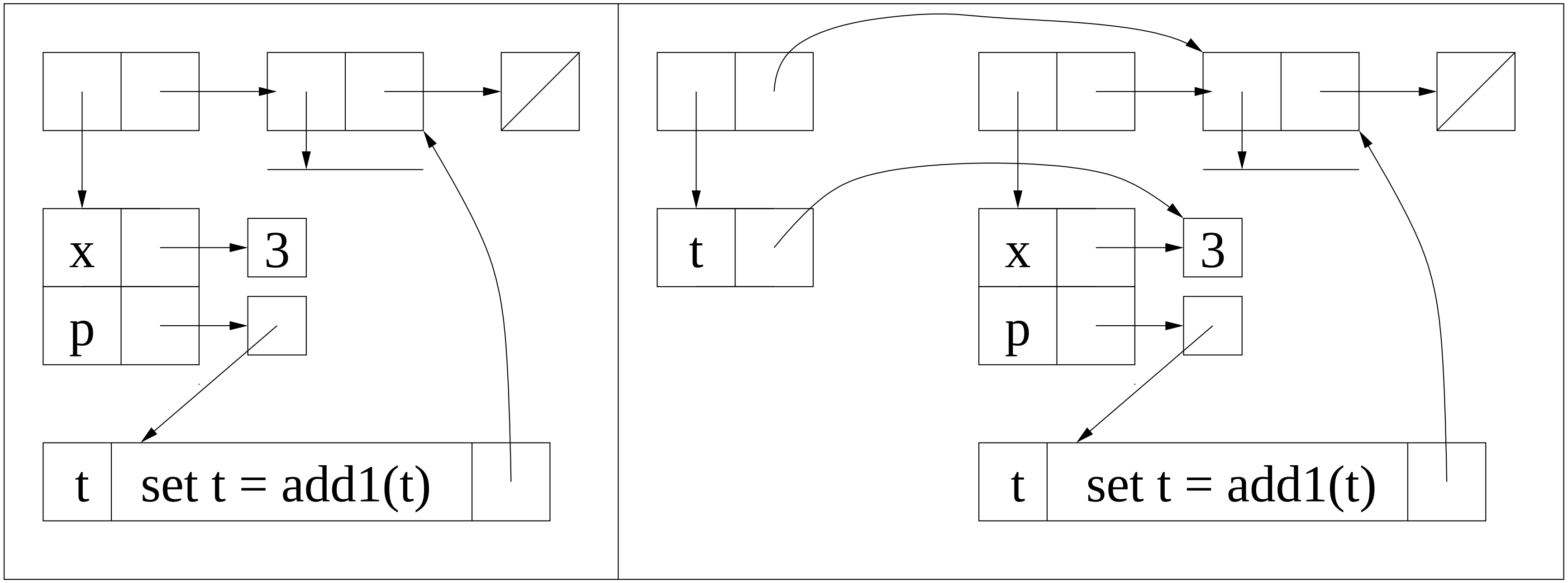
- The parameter passing semantics that we have been using up to now is called *call-by-value*. In call-by-value semantics, when an actual parameter to a procedure is a variable, the procedure's corresponding formal parameter is bound to a new reference to the value of the variable.
- In *call-by-reference* semantics, when an actual parameter to a procedure is a variable, the procedure's corresponding formal parameter denotes the *same reference* as the actual parameter, not a new reference to the actual parameter value.

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` refers to the same cell as the one `x` refers to. The environments created by the `let` declarations and then during evaluation of the expression `.p(x)` are illustrated in the following figure:



Language REF (continued)

What about actual parameters that are not themselves variables? For example, what is the value returned by the following program?

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

Clearly the *value* of `+(x,0)` is the same as that of `x`, but does it make sense to modify the value of an expression? In other words, if you replace `t` with the expression `+(x,0)` in the body of the procedure, does it make sense to perform the following?

```
set +(x,0) = add1(+(x,0))
```

Clearly the RHS of this assignment is perfectly OK – it evaluates to 4. But the LHS is not an L-value – *e.g.*, something that can occur on the LHS of a `set` operation: the only thing that can appear on the LHS of a `set` is a variable.

Language REF (continued)

We can solve this dilemma by making sure that if an actual parameter is an identifier (and therefore is bound to a reference), then the corresponding formal parameter is bound to the same reference. If an actual parameter is something other than an identifier – such as an expression like `+(x,0)`, then the corresponding formal parameter is bound to a *new* temporary reference containing the value of the expression. Thus *variables are passed by reference*, but *any other expressions are passed by value*.

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 procedure application. As the discussion on the previous slide shows, we need to handle actual parameters that are variables differently from actual paramemeters that are expressions. The idea here is to let instances of the Exp classes take care of how to translate themselves into a reference: for anything but a VarExp, we evaluate the expression and return a new reference to the value – this can be handled by a single method in the Exp class. For a VarExp, we return a reference to the same reference that the actual parameter variable referred to.

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

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

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

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

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 definition in the parent Exp class is used.

Language REF (continued)

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

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

Language REF (continued)

Notice that we are still using value semantics for `let` bindings. This means that a program such as

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

still evaluates to 3.

Our observation that any `let` can be re-written as a procedure application no longer applies when call-by-reference semantics is used for parameter passing. Specifically, if the inner `let` in the above program is re-written as a procedure application, we would get

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

which would evaluate to 4.

Language NAME

We now turn to a different parameter passing mechanism, *call-by-name*.

In call-by-name, the formal parameter is bound to the *un-evaluated actual parameter expression*. This expression is not evaluated until the corresponding formal parameter is referenced in an expression. When the formal parameter is referenced, the actual parameter expression is evaluated *in the environment where the procedure was called* – the *calling environment*, and this value becomes the expressed value of the formal parameter.

In the absence of side-effects, call-by-name and call-by-reference return the same results. Call-by-name has an advantage in that if the formal parameter is never referenced, the actual parameter expression is not evaluated. The disadvantage of call-by-name is the computational effort required to evaluate the actual parameter expression every time the formal parameter is referenced.

Language NAME (continued)

In the presence of side-effects, call-by-name has interesting properties that make it very powerful but often difficult to reason about. The language ALGOL 60 had call-by-value and call-by-name as its parameter passing mechanisms. ALGOL 60 had its greatest influence on languages such as Pascal, C/C++, and Java. Although call-by-name has been all but abandoned by modern imperative (side-effecting) programming languages – mostly because of its inefficiency, it still plays a role in functional programming. Scheme supports a form of call-by-name by means of `promise/force`. Other functional languages such as Haskell use a variant, *call-by-need*. We proceed to implement both call-by-name and call-by-need.

Language NAME (continued)

When side-effects are considered, call-by-reference and call-by-name may give different results. Consider:

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

In call-by-reference, the formal parameter `t` refers to the same cell that `x` refers to (initially containing 1), but the formal parameter `u` refers to a new cell that contains the value 6. When evaluating the expression `.f(x, +(x,5))`, modifying `t` in the body of `f` changes the value of `t` (which is bound to the same reference as variable `x`) but does not change the value of `u`. Thus this expression evaluates to 6.

Language NAME (continued)

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

In call-by-name, the formal parameter `t` still refers to the same cell that `x` refers to (initially containing 1), but the formal parameter `u` is bound to the (un-evaluated) expression `+(x,5)`.

Consider now what happens when `.f(x, +(x,5))` is invoked. The `set` operation in the body of this procedure increments the formal parameter `t`; but since `t` refers to the same cell as `x`, the value of `x` is changed, too, to two. When the formal parameter u is referenced at the end of the `proc`, the expression `+(x,5)` is evaluated *in the environment of the caller*, where `x` was originally bound to 1. Since this expression is evaluated after the `set`, and the value of `x` is now 2, the value of the expression `+(x,5)` (and thus the value returned by the procedure) is `+(2,5)` or 7. Thus this expression evaluates to 7.

Language NAME (continued)

Consider the following definition:

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

Using call-by-name, the expression

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

returns the sum

$$\sum_{x=1}^{10} x^2 = 385$$

Using call-by-reference as in the language REF, the expression never terminates because the actual parameter expression <=?(x,10) is evaluated only once, to 1 (true) when x is initially 0, and so the test? parameter is bound permanently to (a reference to) 1. Evaluating test? repeatedly always returns 1 (true), so the “loop” never terminates.

Language NAME (continued)

We proceed to implement call-by-name. We take our 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 procedure, we bind the formal parameter to (a reference to) the procedure's closure in the calling environment. If an actual parameter is an identifier, we bind the formal parameter to the same reference as the actual parameter, just as we do with call-by-reference.

If an actual parameter is any other kind of expression, we bind the formal parameter to (a reference to) an 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*.

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 closure, 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 whenever we refer to the corresponding actual parameter. A formal parameter is bound to a thunk reference only during procedure application. Thunks will otherwise not play a role in expression semantics.

Language NAME (continued)

To change from call-by-reference to call-by-name, we need to change the default `evalRef` behavior of the `Exp` objects so that `evalRef` returns a thunk for most expressions except for `LitExp`, `VarExp`, and `ProcExp`.

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

For a `LitExp` and a `ProcExp`, a thunk is not necessary, so we return an ordinary `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:

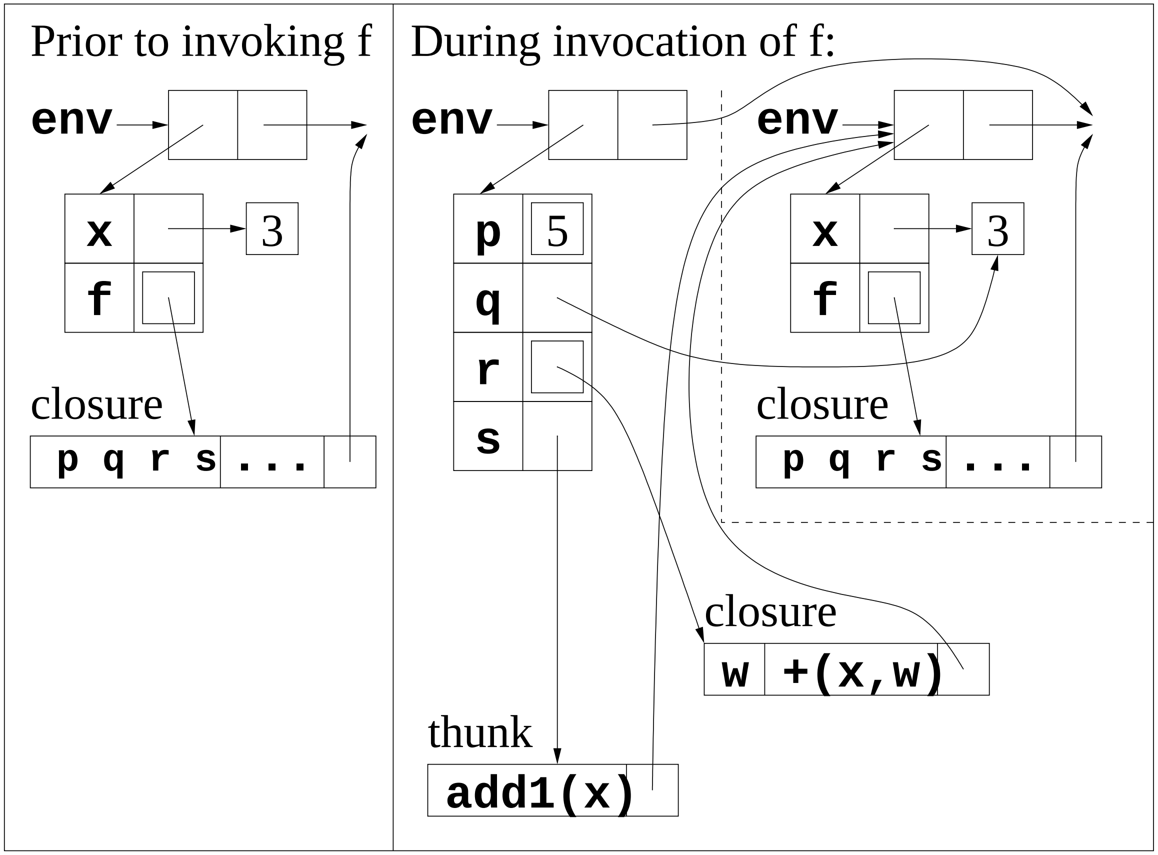
```
ThunkRef
%%%
public class ThunkRef extends Ref {

    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 RuntimeException("cannot modify a read-only expression");
    }
}
%%%
```

Observe that the setRef method throws an exception. This method is only used to evaluate set expressions, and it doesn't make sense to have an expression on the LHS of a set.

Language NAME (continued)

The following illustration may help you to understand how these bindings work. The example shows all four possible cases of actual parameter expressions: literal, 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 actual parameter `set z = add1(z)`, using call-by-need semantics. As with call-by-name, the formal parameter `x` is bound to the thunk with `set z = add1(z)` as its body, in an enclosing environment that we will assume has `z` bound to (a reference to) the value 8. When `x` is referenced in the body of calling procedure, its corresponding thunk is dereferenced, producing a result of 9 for `set z = add1(z)`. The thunk now remembers (*memoizes*) the value 9, and any further references to the formal parameter `x` in the body of the procedure will continue to evaluate to 9 without making any further changes to the variable `z`.

If call-by-name had been used in the above example, additional evaluations of `x` in the body of the procedure would result in evaluating the body of the thunk for each such evaluation, further modifying `z` and yielding values 10, 11, 12, and so forth.

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

Language NEED (continued)

Implementing call-by-need is easy, starting from the call-by-name interpreter. The principal change is to have a `Val` field named `val` in the `ThunkRef` class that is 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 checks to see if the `val` field has been memoized (*i.e.*, is non-null). If so, the `deRef` 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 that 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 modified 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;

    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 `ValRORef` is constructed by the `evalRef` methods in the `LitExp` and `ProcExp` classes. This is a slight change from the `NAME` language, where the instances were `ValRefs`. The `RO` part stands for "Read Only". The reason for doing this is that 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 saying:

```
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 *any* actual parameter expression except for a variable (where call-by-reference is the default).

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 the procedure `p` is applied, its formal parameter `t` is bound to a thunk containing the expression `set x=add1(x)`. Each time the formal parameter `t` is evaluated in the body of the procedure `p`, its thunk is dereferenced, resulting in evaluation of the expression `set x=add1(x)`. So since `t` is evaluated 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 `t` is evaluated in the body of `p`, its corresponding actual parameter expression `set x=add1(x)` is evaluated, which has the side-effect of incrementing the value of `x` to 4 and evaluates to 4. However, the thunk memoizes the expressed value of 4, so any further references 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-reference and call-by-name/need. Examine the definition of `seq`, which seems to recurse infinitely 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-reference.
%% With call-by-name or call-by-need, we get:
.first(natno)              % => 0
.first(.rest(natno))        % => 1
.first(.rest(.rest(natno))) % => 2, and so forth ...
.nth(100,natno)          % => 100
```

Order of evaluation

Let's examine the following example:

```
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 written inside curley braces for clarity) are evaluated before their values are bound to the left-hand variables. But our language does not specify the order in which the right-hand side expressions are evaluated.

In the absence of side-effects, *i.e.* in our early interpreters without `set`, the order of evaluation of the RHS expressions wouldn't matter. However, when side-effects are possible, as in our interpreters such as `SET` and `REF`, the order of evaluation does matter.

In the above example, if the second `set` is evaluated first, then `z` becomes 4 and `y` becomes 5, so the entire expression evaluates to 4 – the value of `z`. If the order of evaluation is reversed, the entire expression evaluates to 5. Our language does not specify order of evaluation; consequently, the value of this expression is ambiguous.

Order of evaluation (continued)

A similar situation exists when evaluating actual parameter expressions, as shown by this example, assuming call-by-value semantics.

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

If the actual parameters are evaluated left-to-right – which would be the "natural" evaluation order, `t` would be bound to 4 and `u` to 5, so the entire expression would evaluate to 4. If the evaluation order were right-to-left, the entire expression would evaluate to 5.

Order of evaluation (continued)

You can see that both `evalRands` and `evalRandsRef` use `for-each` loops (also called enhanced `for` loops) to traverse and evaluate the expressions in the list of actual parameters. The traversal is guaranteed by the Java API specification to be “natural”, in the sense that the elements of the list are visited in ascending item 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;
}
```

Order of evaluation (continued)

If we wished to traverse the `expList` in descending item order number, we would need to do something like this:

```
public List<Val> evalRands(Env env) {
    List<Val> valList = new LinkedList<Val>();
    int n = expList.size();
    while (n > 0) {
        n--;
        Exp e = expList.get(n);
        valList.add(0, e.eval(env));
    }
    return valList;
}
```

The reason for using `add` with the extra parameter of zero in this second version is because we want the elements in the `valList` to appear in the same order that their corresponding expressions appear in the `expList`. Also, we use a `LinkedList` here instead of an `ArrayList` because adding to the beginning of a `LinkedList` object is much more efficient than adding to the beginning of an `ArrayList` object.

Order of evaluation (continued)

Unless the language specification clearly addresses the issue of order of evaluation, the language implementor can choose any evaluation order. The C language specification, for example, explicitly states that the order in which actual parameter expressions are evaluated is undefined. *If the order of evaluation matters and is not specified, it is up to the programmer to avoid actual parameter expressions that have side-effects.* For example, none of our languages specifies the order of evaluation of expressions. To ensure that the program on slide 39 evaluates to 4, it could be re-written as follows:

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

Order of evaluation (continued)

The Java language specification is clear about order of evaluation: actual parameter expressions are guaranteed to be evaluated in left-to-right (or “natural”) order.

Order of evaluation does not matter in languages without side-effects, which makes functional languages immune to order of evaluation issues.

See `http://en.wikipedia.org/wiki/Evaluation_strategy` for more information about order of evaluation.

Order of evaluation (continued)

Another way to avoid order of evaluation problems is to require that all procedures have at most one formal parameter. In languages that use this approach coupled with 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 having multiple formal parameters using an approach called "Currying", as employed in the Haskell programming language – named after Haskell Curry. The following 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 per `proc`:

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

Aliasing

Side-effecting languages that use call-by-reference suffer from another danger. Consider, for example, the following program using the REF language semantics:

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

Aliasing (continued)

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

Using call-by-reference, when `addplus1` is applied to the actual parameters `a` and `a`, both formal parameters `x` and `y` of `addplus1` refer to the *same cell* as `a`. Therefore the `set x = add1(x)` expression is equivalent to the expression `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. Thus the value of the program is 8.

Aliasing occurs when two different formal parameters refer to the same actual parameter. As this example shows, aliasing can lead to unexpected side-effects and should be avoided. **Of course, the best way to avoid problems such as order of evaluation ambiguities and aliasing is to avoid using langauges with side-effects!**