Class 20 - State: Parameter Passing Modes

In this class, we will systematically explore the different language design choices for implementing parameter passing in function calls.

Parameter Passing Variants

We extend our JAKARTA SCRIPT fragment with four parameter passing modes:

- call by value
- call by name
- call by variable, and
- call by reference.

We distinguish these parameter passing modes by prefixing function parameters with a dedicated keyword that determines the mode. Before we introduce the formal semantics of these modes, we explain the differences between them using a series of examples.

Call by Value

So far, we have evaluated function calls using call by value semantics. In this mode, before the actual function call happens, the argument expression is first reduced to a value, which is then passed to the function body. In our new language, we denote call by value parameters by prefixing them with the keyword const. As an example, consider the following program, which defines a function \( f \) with a call-by-value parameter \( x \):

```javascript
const f = function(const x: number) (x + x);
const y = 3;
const r = f(y + 1);
console.log(y);
console.log(r);
```
This program will print 3 and 8. Since the call to \( f \) in this program is side-effect free, we cannot observe the specific evaluation order of the call by value semantics. To make the evaluation order explicit, consider the following modified version of the program:

```javascript
const \( f \) = function(const \( x \): number) (\( x + x \));
var \( y = 3; \)
const \( r = f(y = y + 1); \)
console.log(\( y; \)
console.log(\( r; \)
```

This program will print 4 and 8. The assignment expression \( y = y + 1 \) that is the argument to the call to \( f \) is executed once before the call. The value 4, which is the result of the assignment, is then passed into the function to compute the result of the call, which is 8.

### Call by Name

One alternative to the call by value passing mode is to change the evaluation order of function calls and call arguments so that the function call happens before the argument is evaluated. If we define the semantics in such a way that we reevaluate the argument each time the parameter is used in the function body, we speak of call by name parameter passing.

In our new extension of JAKARTA SCRIPT, we indicate call by name parameters by prefixing the parameter with the keyword `name`. As a first example, consider the following modified version of our first program above where we change the call by value parameter \( x \) in function \( f \) to a call by name parameter:

```javascript
const \( f \) = function(name \( x \): number) (\( x + x \));
const \( y = 3; \)
const \( r = f(y = y + 1); \)
console.log(\( y; \)
console.log(\( r; \)
```

This program will print 3 and 8 just like in the version with the call by value parameter. Again, we have to introduce side effects in the call to \( f \) in order to make the evaluation order explicit. To this end, consider the call by name version of the second program above:

```javascript
const \( f \) = function(name \( x \): number) (\( x + x \));
var \( y = 3; \)
const \( r = f(y = y + 1); \)
console.log(\( y; \)
console.log(\( r; \)
```

This program will now print 5 and 9 because the assignment \( y = y + 1 \) is executed two times during the evaluation of \( f \)–once for each usage of the call by name parameter \( x \) in \( f \). Depending of the control flow in the body of the called function, the argument to a call by name parameter may not be evaluated at all. For instance, consider the following variant of our program:
const f = function(const b: bool) {
    function(name x: number) (b ? x + x : 0)
};
var y = 3;
const r = f(false)(y = y + 1);
console.log(y);
console.log(r);

This program will print 3 and 0 because the call to f will not evaluate the “then” branch x x+ of the conditional expression in the body of f. Hence, the argument y = y + 1 that is passed by name to x is never used in the call and hence never evaluated.

Call by name parameter passing is a very useful programming feature. For example, suppose we have a function whose argument value is only used in the function body if certain conditions are satisfied, e.g., a logging function might only use its argument value if the program is run in debugging mode. In such cases, we would like to avoid the evaluation of the argument altogether in the cases where the value is not actually used. We can easily do this achieve this by using a call by name parameter.

Simulating Call by Name. Unfortunately, many programming languages only support call by value parameters. However, if a language supports higher-order functions and function abstraction, we can simulate call by name using call by value. The idea is to delay the execution of the argument until after the function call happened by wrapping the argument in a function abstraction. Whenever the argument is used in the body of the called function, it has to be explicitly unwrapped using an auxiliary function call that recalculates the argument value.

To see how this works, consider the following program:

const f = function(const x: () => number) (x() + x());
var y = 3;
const r = f(function () (y = y + 1));
console.log(y);
console.log(r);

This program prints 5 and 9, just like our second example for call by name parameter passing. Observe that we turned the call by name parameter x of type number into a call by value parameter of type () => number. That is, x is now a function that takes no parameters. Each call to x in f’s body will cause the wrapped argument expression to be reevaluated, which gives us the same behavior as a call by name parameter.

Call by Variable

The difference between call by value and call by name parameters is the order in which the call and the arguments to the call are evaluated. However, we
can make a more fine-grained distinction in our semantics of parameter passing, even if we fix the evaluation order to call by value semantics. Specifically, we can distinguish between parameters that are treated as constant values throughout the execution, and parameters that are essentially treated like mutable variables and can be reassigned new values. We refer to the latter type of parameters as *call by variable* parameters. We indicate such parameters by prefixing the parameter name with the keyword `var`. Essentially, the difference between a `const` parameter and a `var` parameter is that inside the function body, a `var` parameter is treated as if it was declared by a `var` declaration instead of a `const` declaration. The following example highlights this difference:

```javascript
const f = function(var x: number) { x = x + 1; x);
var y = 3;
const r = f(y);
console.log(y);
console.log(r);
```

This program will print 3 and 4. The parameter variable `x` is reassigned inside of the body of `f`. However, the effect of this assignment is not observable outside of `f`.

Note that in JavaScript (and most other imperative programming languages) function calls are implemented using call by variable semantics. Often, these languages do not make the fine-grained distinction between call by variable and call by value and both modes are simply referred to as call by value.

**Call by Reference**

Finally, we can consider a combination of call by name and call by variable. We refer to this mode as *call by reference*. This mode is indicated by the keyword `ref`. The following program highlights the difference between call by variable and call by reference:

```javascript
const f = function(ref x: number) { x = x + 1; x);
var y = 3;
const r = f(y);
console.log(y);
console.log(r);
```

This program will print 4 and 4. Unlike in the previous program where we passed the argument to `f` by variable, the assignment to `x` is now treated as an assignment to the mutable variable `y`, which is passed to `x` in the call to `f`. Note that we can only pass assignable expressions as arguments to call by reference parameters. We refer to such expressions as *location expressions*. In our current language, only mutable variables are location expressions. For example, the following program should be rejected by our type checker because `f` attempts to reassign `y` which has been declared as an immutable `const` variable:

```javascript
const f = function(ref x: number) { x = x + 1; x);
const y = 3;
```
const r = f(y); // type error because y is not assignable
console.log(y);
console.log(r);

Formalizing Parameter Passing Modes

The abstract syntax of our extended language is as follows:

\[
\begin{align*}
  n & \in \text{Num} & \text{numbers (double)} \\
  b & \in \text{Bool} := \text{true} \mid \text{false} & \text{Booleans} \\
  a & \in \text{Addr} = \mathbb{N} & \text{addresses} \\
  x & \in \text{Var} & \text{variables} \\
  \tau & \in \text{Typ} := \text{bool} \mid \text{number} \mid (\text{mode } x : \tau_1) \Rightarrow \tau_2 & \text{types} \\
  v & \in \text{Val} := n \mid b \mid a \mid \text{function } p (\text{mode } x : \tau) t e & \text{values} \\
  e & \in \text{Expr} := x \mid v \mid e_1 \text{bop } e_2 \mid uop e_1 \mid e_1 \ast e_2 : e_3 & \text{expressions} \\
  \text{mut } x & = e_4 ; e_5 \mid e_1 (e_2) & \\
  \text{bop } \in \text{Bop} := + \mid * \mid \&\& \mid || & \text{binary operators} \\
  \text{uop } \in \text{Uop} := * & \text{unary operators} \\
  p & := x \mid \epsilon & \text{function names} \\
  t & := : \tau \mid \epsilon & \text{type annotations} \\
  \text{mut } \in \text{Mut} := \text{const} \mid \text{var} & \text{mutabilities} \\
  \text{mode } \in \text{PMode} := \text{const} \mid \text{var} \mid \text{name} \mid \text{ref} & \text{passing modes}
\end{align*}
\]

The only change compared to the language of class 18 is that we now explicitly declare the parameter passing mode for each function abstraction. The parameter passing mode also becomes part of the type signature of a function.

Small-Step Semantics

Figure 1 shows the new small-step reduction rules for function calls with the different parameter passing modes. The rules for the other language constructs are as discussed in class 18. Note that the rule \text{SEARCH\_{CALL}2} only applies to the call-by-value and call-by-variable modes. For a call to a function whose parameter is passed by reference, the type system will ensure that the argument of the call is always of the form \(*a\). In this case, the argument \(*a\) should not be evaluated before the call, so that the address can be passed into the function body. Similarly, for a function whose parameter is passed by name, the argument should not be evaluated before the call.

Type Checking

The new type checking rules for the different types of function abstractions and call expressions are given in Figure 2. When the typing environment is extended
\[ \langle M, e_1 \rangle \rightarrow \langle M', e'_1 \rangle \quad \text{SearchCall}_1 \]

\[ \langle M, e_1 (e_2) \rangle \rightarrow \langle M', e'_1 (e_2) \rangle \]

\[ v_1 = \text{function } p (\text{mode } x : \tau) t e \]

\[ \langle M, e_1 (e_2) \rangle \rightarrow \langle M', e'_1 (e_2) \rangle \quad \text{mode } \notin \{ \text{ref, name} \} \quad \text{SearchCall}_2 \]

\[ \langle M, v_1 (e_2) \rangle \rightarrow \langle M, e[v_1/x] \rangle \quad \text{DoCallConst} \]

\[ v_1 = \text{function } x_1 (\text{const } x_2 : \tau_2) : \tau' e \]

\[ \langle M, v_1 (e_2) \rangle \rightarrow \langle M, e[v_1/x_1][v_2/x_2] \rangle \quad \text{DoCallConstRec} \]

\[ v_1 = \text{function } x_1 (\text{var } x_2 : \tau_2) : \tau' e \quad a \notin \text{dom}(M) \]

\[ \langle M, v_1 (e_2) \rangle \rightarrow \langle M', e[x_1/x][*a/x] \rangle \quad \text{DoCallVar} \]

\[ \langle M, v_1 (v_2) \rangle \rightarrow \langle M', e[v_1/x][*a/x] \rangle \quad \text{DoCallVarRec} \]

\[ v_1 = \text{function } x_1 (\text{name } x_2 : \tau_2) : \tau' e \]

\[ \langle M, v_1 (e_2) \rangle \rightarrow \langle M, e[x_1/x][e_2/x_2] \rangle \quad \text{DoCallName} \]

\[ v_1 = \text{function } x_1 (\text{ref } x_2 : \tau_2) : \tau' e \]

\[ \langle M, v_1 (*a) \rangle \rightarrow \langle M', e[*a/x] \rangle \quad \text{DoCallRef} \]

\[ v_1 = \text{function } x_1 (\text{ref } x_2 : \tau_2) : \tau' e \]

\[ \langle M, v_1 (*a) \rangle \rightarrow \langle M, e[v_1/x][*a/x] \rangle \quad \text{DoCallRefRec} \]

Figure 1: New inference rules that define the small-step semantics of function call expressions with the different parameter passing modes.
\[
\Gamma \vdash e_1 : (mode \ x : \tau') \Rightarrow \tau \quad \Gamma \vdash e_2 : \tau' \quad \text{mode} \notin \{\text{ref}\} \quad \text{TypeCall}
\]

\[
\Gamma \vdash e : (\text{ref} \ x : \tau) \Rightarrow \tau \quad x \in \text{dom}(\Gamma) \quad \Gamma(x) = (\text{var}, \tau') \quad \text{TypeCallRef}
\]

\[
\Gamma' = \Gamma[x \mapsto (\text{mut}(\text{mode}), \tau)] \quad \Gamma' \vdash e : \tau' \quad \text{TypeFun}
\]

\[
\Gamma' = \Gamma[x \mapsto (\text{mut}(\text{mode}), \tau)] \quad \Gamma' \vdash e : \tau' \quad \text{TypeFunAnn}
\]

\[
\Gamma' = \Gamma[x_1 \mapsto (\text{const}, \tau_1)][x_2 \mapsto (\text{mut}(\text{mode}), \tau_2)] \\
\Gamma' \vdash e : \tau' \quad \tau_1 = (\text{mode} \ x_2 : \tau_2) \Rightarrow \tau' \\
\Gamma \vdash \text{function} \ x_1 (\text{mode} \ x_2 : \tau_2) : \tau' \quad \tau_1 \quad \text{TypeFunRec}
\]

\[
\text{mut(const)} = \text{mut(name)} = \text{const} \\
\text{mut(var)} = \text{mut(ref)} = \text{var}
\]

Figure 2: New type checking rules for function abstractions and calls

with the parameter in the different TypeFun rules, the given parameter passing mode is mapped to the appropriate mutability using the function \text{mut}. Note that at the moment mutable variables are the only expressions that are allowed as arguments to functions whose parameters are passed by reference. This is because in our current language mutable variables are the only expressions that evaluate to references to memory locations.

**Custom Control Constructs with Call by Name**

One useful feature of call-by-name parameters is that it can be combined with curried higher-order functions to define custom control constructs. We discuss how this can be done in our current \text{JakartaScript} fragment as well as in \text{Scala}.

**JakartaScript while loop.** Loops are one of the most important control constructs in imperative languages. We do not have loops built into our current JakartaScript fragment. However, using call-by-name parameter passing, we can write recursive functions that can be used almost as if they were loops:

```javascript
const while = function while(name cond: bool):
  (name body: Undefined) => Undefined
|
  return function (name body: Undefined) {
```
In the above JavaScript program, we define a curried function `while` that takes a value of type `bool`, the loop condition, and a value of type `Undefined`, the loop body, to implement the semantics of a `while` loop in JavaScript. By passing the condition and body by name, they are reevaluated each time a call to the nested function of `while` is evaluated. This way, we obtain the proper semantics of a `while` loop.

When we use the `while` function, the only syntactic difference to an actual `while` loop in JavaScript is that the loop body has to be wrapped in an extra pair of parenthesis. The reason for this is that we are actually calling the function that resulted from the first call to `while`.

Another minor issue is that we have to terminate the body of the loop with an explicit `undefined` value. If we instead wrote:

```javascript
var x = 0;
while(x < 10)({
    console.log(x);
    x = x + 1;
})
```

then this program would be rejected by our type system. This is because the type of a sequence of expressions is calculated as the type of the last expression in the sequence. In the case of the loop body in the second program, this is now the assignment expression `x = x + 1`. However, the type of an assignment expression is the type of the right side of the assignment, which is `number`. This type is incompatible with the type `Undefined`, which is the expected type of the argument to the function that is returned by `while`.

Scala repeat/until loop. In Scala, a parameter can be declared as pass-by-name by putting an arrow in front of the parameter type:

```scala
def f(x: => T) e
```

Thus, a call-by-name parameter in Scala can be thought of as a function that is called with no argument and then returns a value of `T`. This syntax is reminiscent of our encoding of call-by-name parameters using call-by-value parameters with function types that take no parameters.

We can combine call-by-name parameters with Scala’s object system and its condensed method call syntax. This gives us a powerful technique for defining
custom language primitives that can be used as if they were built into the
language.

For example, some languages such as Pascal support repeat/until loops:

\[
\text{repeat body until (cond)}
\]

These loops execute body once, and then repeatedly execute it until the loop
condition cond becomes true. Although, Scala does not have repeat/until loops
built in, we can easily write a class that provides us with such a construct:

```scala
class repeat(body: => Unit) {
  def until(cond: => Boolean): Unit = {
    body
    if (!cond) until(cond)
  }
}

object repeat {
  def apply(body: => Unit) = new repeat(body)
}
```

The class `repeat` takes the loop body as a parameter and then defines a method
`until` that takes the loop condition to implement a repeat/until loop using re-
cursion. Recall that the type `Unit` is Scala’s equivalent to the type `Undefined`
in `JakartaScript`. Since both the loop body and condition are passed by
name, we obtain the expected behavior. The companion object of `repeat` de-
dines a factory method to create new `repeat` instances, saving us the explicit
calls to `new`.

We can then use this class as follows:

```scala
var x = 0
repeat {
  x = x + 1
} until (x == 10)
```

Syntactically, it now seems as if repeat/until is indeed an in-built language con-
struct. However, this code is just a syntactically more compact but semantically
equivalent version of the following nested sequence of method calls:

```scala
var x = 0
repeat.apply({
  println(x)
  x = x + 1
}).until(x == 10)
```

In particular, the first call goes to the `apply` method of the companion object
of `repeat`, the subsequent `until` call then goes to the newly created `repeat`
instance that is returned by the call to `apply`. 