Scoping, Functions, and Parameter Passing

Hans Hüttel

Programming Language Design 2022

February 2022

Incorporating material by Andrzej Filinski, Fritz Henglein and Torben Æ. Mogensen

Learning goals for scope rules

- To understand the definitions of dynamic and static scope rules
- To be able to apply the definitions of scope rules to reason about the behaviour of a program, given specific scope rules
- To be able to reason about the strengths and weaknesses of dynamic and static scope rules.
- To be able to describe how function calls are implemented for dynamic and static scope rules and how memory is allocated (deep/shallow binding and activation records).

Learning goals for parameter passing mechanisms

- To be able to explain the differences between different parameter passing mechanisms.
- To be able to apply the definition of a parameter passing mechanism to reason about the behaviour of a program.
- To be able to reason about the strengths and weaknesses of parameter passing mechanisms.

Part I

Scopes – What are they?

What are scope rules, and why do we need them?

A programming language often has several different *kinds* of identifiers/names. We often have names for

- Constants
- Variables
- Functions
- Types
- Classes, modules, data constructors, fields, labels, exceptions

  . . .

What are scope rules, and why do we need them?

Three very natural questions that a programming language design must answer are

- Which names do we know?
- Where in the program do we know them?
- At what times during program execution do we know them?

The answers to these questions give us the scope rules for our language.

Binding occurrences and bound occurrences

A binding occurrence of an identifier is a declaration. An applied occurrence is one that either refers statically to a binding occurrence of the same identifier (a bound occurrence) or does not have a binding occurrence to refer to yet (a free occurrence).

The *same* occurrence may be free within one code fragment, but bound in a larger one!

Binding occurrences and bound occurrences

```
void f(int y) {
                int y = z;
                int z = 4;
                y = y + z;  }
```

The two bound occurrences of y refer to the second binding occurrence of y.

The first binding occurrence of y has no bound occurrence! z occurs free in the definition of y, but there is another z with a bound occurrence in the last line.

Declarations and definitions

A declaration introduces an identifier, often with some intrinsic properties (such as its type). In C an example of a declaration that is not a definition is

```
int x;
```

A definition supplies the actual value/meaning of the identifier. In Haskell the following is both a declaration and a definition.

```
z = 17
```

In imperative languages, the meaning of a variable is really its *address* that contains the value of the variable.

Definitions and declarations may obey different scoping disciplines!

Declarations meet definitions

Declarations and definitions may be matched up:

- Immediately (in the same syntactic construct)
- At compile time (in the same compilation unit)
- At linking time (for modular programs)
- At runtime (e.g., function parameters)

Global scopes

Global scopes are *flat* – an identifier with global scope is unique throughout the program.

This is the case for e.g., module names and `extern` vars/functions in C.

Early programming languages – notably FORTRAN and COBOL – only had flat scopes of this kind. When these languages appeared, this was the only approach known; it is easy to implement.

Partitioned scopes

A programming language allows for partitioned scopes, if identifiers are unique within a context (such as a module) but in that context, scopes are flat.

C static vars/functions and HASKELL types are examples of constructs that have partitioned scopes in this way.

Nested scopes

Contemporary programming languages allow for nested scopes in which inner declarations can temporarily override outer ones.

Examples (and there are many!) are functions in HASKELL and types in SML.

Sometimes it may look as if we can have multiple declarations in a single scope, but this is just a shorthand for implicit inner scopes:

```
let x = 5
    x = True
in
    x
```

```
let x = 5
  in
    let x =
       True
    in
      x
```

Does the order of definitions matter?

When several definitions given in same scope, does order matter? (*cumulative* vs. *independent* definitions)

```
let x = 2
in
    let y = x + 3
        x = 7
    in y
```

```
int x = 2;
{ int y = x + 3;
  int x = 7;
  return y; }
```

Are definitions recursive?

Does an occurrence of identifier in a definition refer to a previous definition, or the current one?

```
let f(x) = x + 1
in let f(x) = 2*f(x)
   in f(5)       -- 12 or infinite recursion?
```

Significant variation between languages

- And often also within a language, for different kinds of identifiers, e.g., variables vs. functions in C.

Environments

A central notion here **and in the rest of the entire session on scopes and parameters** is that of environment.

An environment tells us what identifiers we know and what values they are bound to. So an environment tells us what bindings we know.

In formal semantics this notion is made completely precise. Here, all we need to is that an environment is a partial function

$$\mathit{env} : \mathbf{Identifiers} \hookrightarrow \mathbf{Values}$$

We write $\mathit{env}\ x = v$ if the value of x is v in environment env.

Part II

Static and dynamic scoping

What is the final value of y?

```
begin
      var x:= 0;
      var y:= 42

      proc p is x:= x+3;
      proc q is call p;

      begin
          var x:= 9;
          proc p is x:= x+1;

          call q;
          y:=x
      end
end
```

It depends on our choice of scope rules

In many programming languages, a function (procedure, method . . .) is allowed to have free variables inside its body. What do these free variables refer to?

That is, what is their binding occurrence?

Scope rules answer that question.

Static vs. dynamic scoping

Static scoping The free identifiers in a function body are the ones that were known when the function was declared. This means that free identifiers refer to *textually* enclosing declaration(s).

Dynamic scoping The free identifiers in a function body are the ones that are known when the function gets called. This means that free variable(s) refer to variable binding(s) at the *call site*.

What is the final value of y? (Dynamic scope rules)

```
begin
      var x:= 0;
      var y:= 42

      proc p is x:= x+3;
      proc q is call p;

      begin
          var x:= 9;
          proc p is x:= x+1;

          call q;
          y:=x
      end
end
```

What is the final value of y? (Static scope rules)

```
begin
    var x:= 0;
    var y:= 42

    proc p is x:= x+3;
    proc q is call p;

    begin
        var x:= 9;
        proc p is x:= x+1;

        call q;
        y:=x
    end
end
```

Pros and cons of dynamic scoping

Because dynamic scoping rules refer to the variable bindings that exist at call time, bindings

- May refer to different definitions in different calls.
- May fail to find matching definition at run time.
- May refer to declaration with wrong type at run time.

To understand if our bindings make sense, we have to simulate or run the program.

On the other hand, dynamic scoping is easy to implement!

Pros and cons of static scoping

Because static scoping rules refer to the variable bindings that exist at the time the function was declared, it is easy to check what the bindings are and if they make sense. We can simply read the program.

On the other hand, static scoping requires more of an effort to implement.

Part III

Implementing dynamic scoping

Dynamic scoping: Deep binding

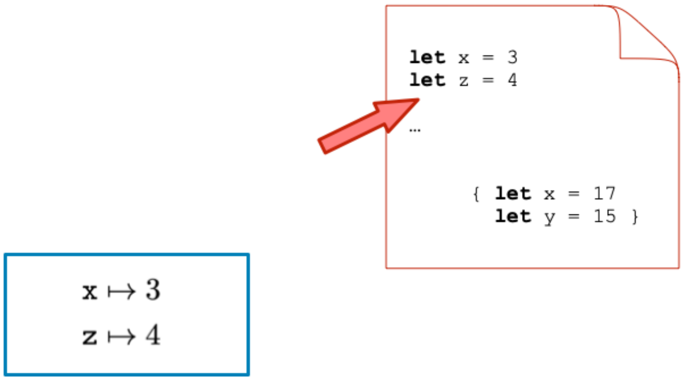
One way of implementing dynamic scoping is to use a stack to represent the environment. This is deep binding.

We use a stack of local environments, one for each scope entered. We

- Push a local environment to the stack when we enter a new scope.
- Pop the stack when we leave a scope.
- The environment given by a stack is the function *env* given by $env\ x = v$ if the topmost binding of x in the stack is to value v.

Stack

Code

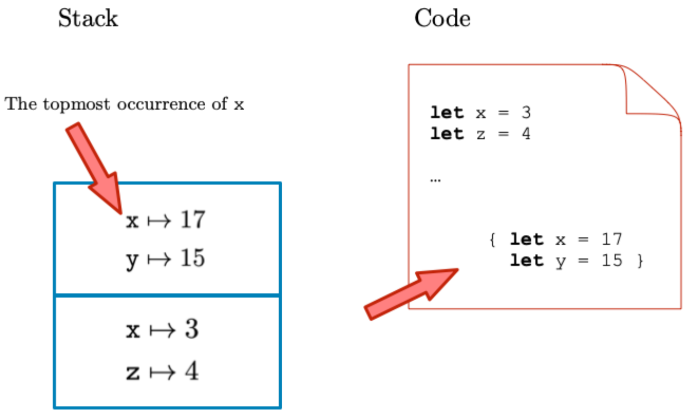


```
let x = 3
let z = 4

...

      { let x = 17
        let y = 15 }
```

$$\boxed{\begin{array}{c} \mathbf{x} \mapsto 3 \\ \mathbf{z} \mapsto 4 \end{array}}$$



```
let x = 3
let z = 4

...

        { let x = 17
          let y = 15 }
```

In the inner block, that we are now in, we have that

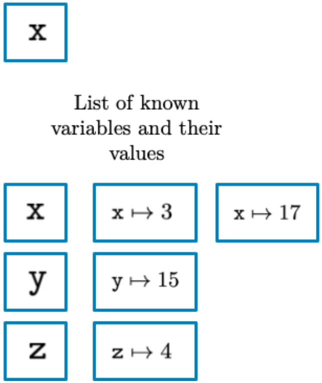
$$env = [x \mapsto 17, y \mapsto 15, z \mapsto 4]$$

Dynamic scoping: Shallow binding

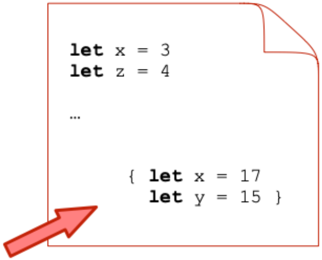
Instead of using a stack, we can use a current environment for active bindings together with a stack of hidden bindings. This is called shallow binding

- When we enter a new scope, for each new binding we
  - push current the binding for same variable to the stack of hidden bindings and
  - update the current environment with our new binding.
- When we leave the scope, we restore the previous bindings in the current environment by popping them from hidden bindings.

Hidden bindings



Code



```
let x = 3
let z = 4

...


        { let x = 17
          let y = 15 }
```

Pros and cons: Deep vs. shallow

Deep	• Fast entering and leaving of scopes. Looking up the value of a non-local variables is slow (we need to dig into the stack). • This is a simple model: A stack of bindings. To look up a value: simply find the topmost binding for identifier.
Shallow	• Fast lookup for all variables. Entering and leaving a scope that has many definitions is slow.

Part IV

Implementing static scoping

Static scoping: Nonnested functions

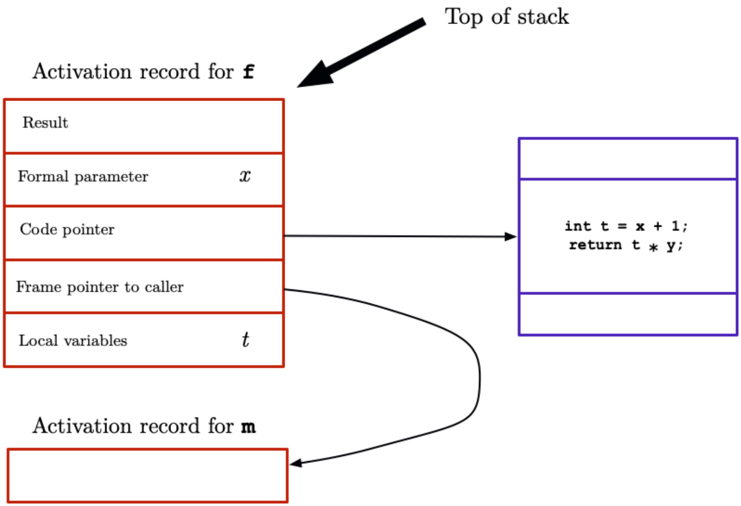
Here is a C fragment.

```
int y;
int f(int x) { int t = x + 1;  return t * y; }
int m(int z) { y = 10; return f(2*y) + z; }
```

When we call a function, we allocate a new *activation record* it in memory. It has space for the result, the parameters, local variables and pointers to the code of the function and to the activation record of the function that called it.

Activation records are allocated on a *stack*. There is a *stack pointer* pointing to the first available memory cell.

Activation records on the stack



Function calls using the stack

In the case of static scope rules, every time we call a function *f*, the following must happen:

1. We place a copy of the activation record for *f* on the stack with all the appropriate content filled in.
2. We follow the function pointer to the code of the body of *f* and execute it.
3. When the function body finishes, we pop the activation record from the stack.

Nested (local) functions

In C, we cannot have local function declarations inside other functions. But many other languages allow that – gcc, Haskell, Python, Java, etc. etc.

In GCC, we can write

```
int m(int z) {
  int y = 10;
  int f(int x) {int t = x+1; return t * y;}
  return f(2*y) + z;
}
```

Here, the body of f mentions y from the outer scope. So it needs access to the value of *stack*-allocated y. But how?

Nested (local) functions

There are two approaches to this:

- We kan make y an extra, implicit parameter of f (this is essentially the notion of λ-lifting that we will meet in a later session).
  If f can modify y, it be passed by reference (the next part tells us what this is) In general, we must do same for all variables of m that are mentioned in the body of f.
- Or we can store all variables of m in a single struct and pass its address to f. This is also known as a static link.
  We must then use a chain of links for deeper scope nestings.

Part V

Parameter passing: Eager approaches

Parameter passing mechanisms

Most programming language have notions of abstraction that use parameters.

We know functions from F#, C and many other languages – but there are also other abstractions: Procedures, methods . . .

Parameter passing mechanisms: Eager approaches

Parameters that occur in definitions of abstractions are called formal parameters.

Parameters that occur when an abstraction is invoked are called actual parameters (or sometimes arguments).

```
fun f x = x + 2

fun g y = f(17) + y

fun h n = if n == 0 then 1 else n * h (n-1)
```

What are the formal and actual parameters in the above example?

What is a parameter passing mechanism?

A parameter passing mechanism defines the way the formal parameter is associated with the actual parameter when an abstraction is invoked.

Different designs

- No parameters! We use global variables instead. COBOL and BASIC do this in their procedure calls.
- Use one single parameter passing mechanism for everything. HASKELL and F# do this (but disagree on what to use).
- Allow different parameter passing mechanisms and leave the choice to the programmer. The ALGOL languages do this.

A fundamental distinction is whether actual parameters are evaluated by the *caller* (eager approaches) or *callee* (lazy approaches).

Eager approach: Call by value

If we use call-by-value, then the formal parameter is bound to the value of the actual parameter.

In this approach, actual parameter can be general expressions of appropriate types.

Inside the body of the abstraction that we invoke, the formal parameter behaves as a local variable.

Eager approach: Call by value

Assume that we have defined a function `f` that has formal parameters $x_1, \dots, x_n$.

Here is a call of `f` with the actual parameters being a list of expressions $\vec{e} = e_1, \dots, e_n$.

$$\texttt{call}\ f(e)$$

When we use call-by-value, we proceed as follows:

1. Evaluate $\vec{e} = e_1, \dots, e_n$ to values $v_1, \dots, v_n$.
2. Bind x_1 to v_1, . . . , x_n to v_n.
3. Execute the body of `f`.

Pros and cons of call-by-value

- In a language with side effects, the order of the actual parameters can matter.
- Depends on language whether evaluation order specified.
- A function cannot pass results back from the call by means of the parameter.
- Even if an actual parameter is a mutable variable, assignments to formal parameter in body will not modify the value of the actual parameter.
- But if the *address* of mutable object/variable (i.e., pointer) is passed by call-by-value, the callee can still modify the *content* of passed memory.
  We must be careful about deep vs. shallow copy For instance, C passes structs by value, but arrays by reference.

Another eager approach: Call by value–result

This is also known as *copy-in/copy-out* and *call-by-value-return*.

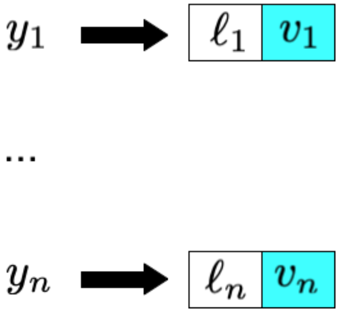
Here, the actual parameters must be variables or other expressions that have a location (such as entries in an array). The formal parameters are bound to the values of the actual parameters and behave as local variables, whose values are written to the actual parameters at the end of the call.

Call by value–result in more detail

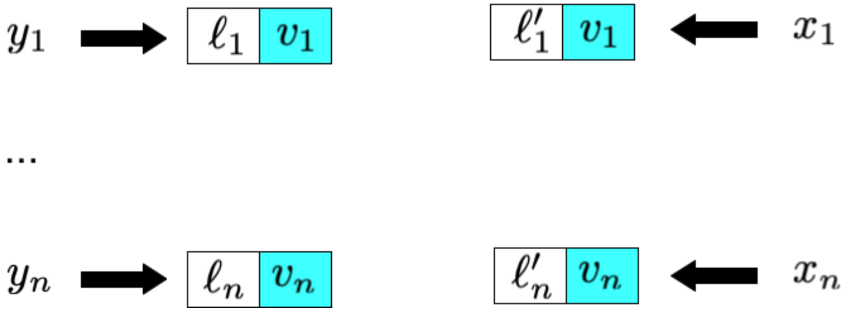
Assume that we have defined a function `f` that has formal parameters $x_1, \ldots, x_n$.

Here is a call of `f` with the actual parameters $\vec{y}$ being a list of variables (or expressions associated with locations) $\vec{y} = y_1, \ldots, y_n$.

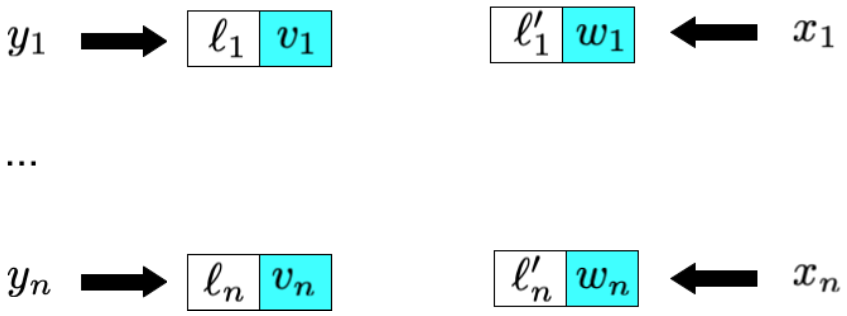
$$\texttt{call } f(\vec{y})$$



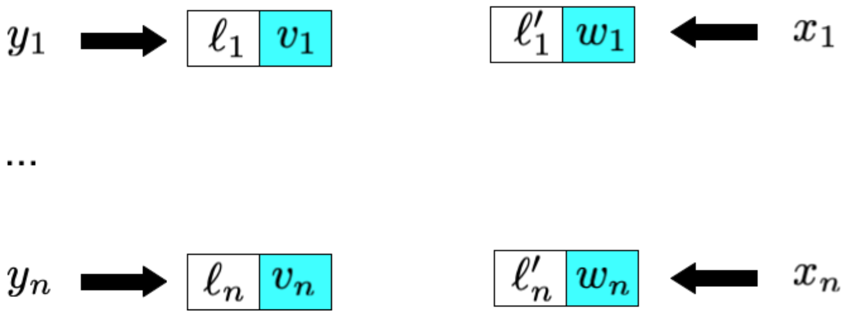
Find the locations of $\vec{y} = y_1, \ldots, y_n$.



Allocate new locations for the formal parameters $x_1, \ldots, x_n$.
Initialize them with $v_1, \ldots, v\ :_n$.



The call is about to finish; the values in the new locations are $w_1, \ldots, w_n$.



When the call finishes, we write back the values $w_1, \dots, w_n$ to the locations $y_1, \dots, y_n$.

Why is this an eager parameter passing mechanism?

The locations of the actual parameters are determined by the caller.

If we call a procedure with an array as actual parameter as in `p(x[i+1])`, the index expression `i+1` is evaluated *before* the call, so already at this time we know where to look in the array.

Pros and cons: Call by value–result

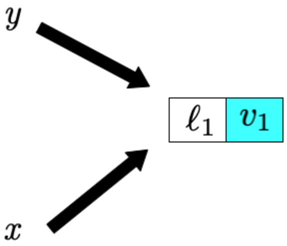
- In general, the order of writebacks may matter.
- The language may require that the actual parameters are *distinct* variables.
- But this is hard to enforce statically, e.g., `p(x[i], x[j])` – for how do we know that the indexes are different? We will only find out at run-time.

Call by reference

Here the formal variable is a reference to the actual parameter.

This means that the actual parameter must be a variable (or another expression that has a location).

Call by reference



The actual parameter y and formal parameter x share the location ℓ_1 that contains the value v_1. If we change the value in ℓ_1, both y and v will have the same new value.

So call-by-reference makes deliberate use of side effects.

Pros and cons: Call-by-reference

- We can avoiding copying large values; a reference is enough.
- We can pass data to and from a function data transfer. This is an alternative to returning values from a function.
- But it is not always evident direction of data transfer is intended!
- Side effects can be tricky. If the same actual parameter variable is used for two distinct formal parameters, (or is globally accessible) things can get complicated!

Call-by-reference in Fortran

FORTRAN has call-by-reference only but allows the actual parameter to be a general expression, e.g., `X+3`.

The caller creates an *anonymous* local variable for each actual parameter and initializes it to the value of the expression.

So any assignments to parameter in callee have no net effect.

The value of 4 is . . .

A natural "optimization": if the argument is a constant (literal), is to pass the address of the constant in program code.

This is often preferable to allocating space for an extra variable. With roughly 2^{16}-bytes address spaces, every word counts!

And if the same constant is used multiple times in the same program, we only need to keep a single copy ("literal pool").

On the other hand . . .

. . . this allows us to change the value of 4.

The value of 4 is . . .

A natural “optimization”: if the argument is a constant (literal), is to pass the address of the constant in program code.

This is often preferable to allocating space for an extra variable. With roughly 2^{16}-bytes address spaces, every word counts!

And if the same constant is used multiple times in the same program, we only need to keep a single copy (“literal pool”).

On the other hand . . .

. . . this allows us to change the value of 4.

Part VI

Parameter passing mechanisms: Lazy approaches

A lazy approach: Call-by-text

Here the text of the actual parameter is bound to the formal parameter. No evaluation or interpretation happens.

TEX and LATEX use this. The `fexpr` functions in LISP use this. So do many scripting languages – BASH, TCL ...

Pros and cons: Call-by-text

- This requires interpretation (or even string parsing) at run time!
- This forces us to have dynamic scope rules.
- Poorly suited for compilation, or even high-performance interpretation.
- The semantics is unpredictable, error prone and extremely insecure, since we can inject any text into a function body. This allows evil users to carry out injection attacks and cross-site scripting attacks become possible. We can pass arguments that arbitrarily update local variables!

Call-by-name

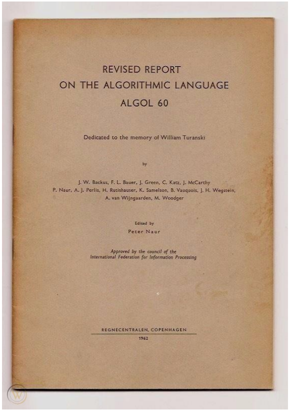
Call-by-name is a sanitized version of call-by-text.

The actual parameter is substituted for all occurrences of the formal parameter in such a way that name clashes are avoided.

This notion of *capture-avoiding substitution* deals with the many problems that call-by-text has – and allows us to have static scope rules.

Algol 60

Call-by-name was the parameter passing mechanism in ALGOL60.



REVISED REPORT
ON THE ALGORITHMIC LANGUAGE
ALGOL 60

Dedicated to the memory of William Turanski

by

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy
P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein,
A. van Wijngaarden, M. Woodger

Edited by
Peter Naur

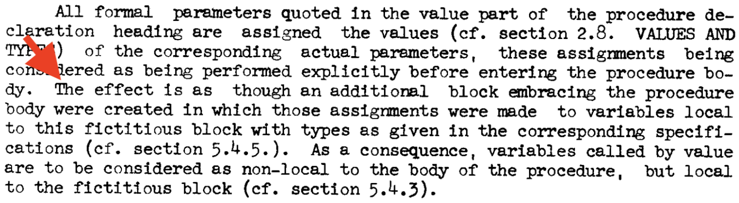
Approved by the council of the
International Federation for Information Processing

REGNECENTRALEN, COPENHAGEN

1962

Algol 60

Call-by-name was defined in terms of declaring an extra local variable, initialized to value of parameter passed by name.

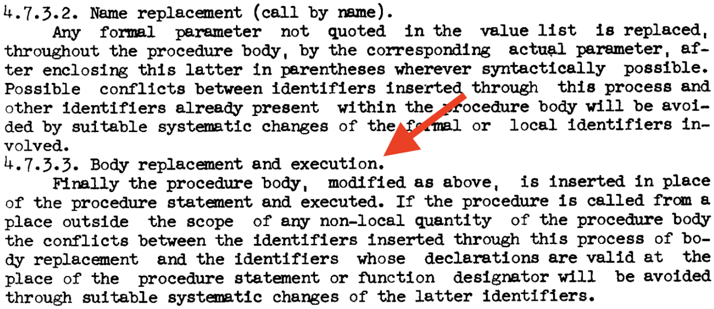


All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. VALUES AND TYPES) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which those assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5.). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

(In practice, ALGOL60 had also call-by-value.)

Algol 60

Here is the passage in the report about capture-avoiding substitutions.



4.7.3.2. Name replacement (call by name).

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution.

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

Why capture-avoiding substitution is needed

Here is an Algol60 procedure.

```
procedure p(x, r); integer x, r;
  begin integer y; y := x+2; r := x*y end;
integer y;
```

Consider a procedure call p(y−3, y) (where y is some global variable in our program). With a simple call-by-text approach the procedure call would consist in executing the body

```
      begin
          integer y; y:= (y−3)+2; y := (y−3)*y
      end
```

But then the global y will not have its value updated!

Why capture-avoiding substitution is needed

We have to re-name the local variable y inside the procedure body to make sure that the correct variable gets updated.

```
begin
     integer y1; y1:= (y−3)+2; y := (y−3)
        ∗y1
end
```

Note that we re-named the local variable y inside the procedure body to make sure that the correct variable was updated.

Implementing call-by-name

The copy rule is easy to explain and is used for inlining in optimizing compilers at compile time.

But how do we implement parameter passing?

We pass the actual parameter expression as a thunk. This is a compiled parameterless function that is invoked by the callee when the value of the parameter is needed.

We can assign to the formal parameter, so we also need a “setter” procedure that can do this. And if the actual parameter is not a variable, the “setter” signals a runtime error.

Thunks

If we call the procedure

```
procedure p(x, r); integer x, r;
  begin integer y; y := x+2; r := x*y end;
integer y;
```

with p(y−3, y) the callee executes the code

```
begin
    integer y1;
    set(y1,thunk1()+2);
    set(thunk2(),thunk1() *y1)
end
```

Pros and cons: Call-by-name

- If we use local variables for parameters, the FORTRAN behaviour can re-appear when constants are used as parameters. There is not enough information to detect this problem at compile time.
- Call by name with mutable variables allows for correct but nasty code. A famous example is "Jensen's Device".

Another lazy day: Call-by-need

- In a pure functional language (that is, with no side effects), evaluating an expression twice always gives the same result.
- So if we use call-by-name, we can *cache* any successful result of first evaluation, and simply return that every time we need it.
- We normally implement the use of a thunk by using a mutable cell to store the value. But now the contents of the cell will never change! The observable behaviour is still like call by name, only faster!

General lazy evaluation

We can generalize from parameter passing to definitions in general, also for data structures.

```
let xs = [1+2, 3+4, 5+6]
    x = head (tail xs)
in  x*x
```

This returns 49, but only evaluates 3+4, and only once.

Part VII

Comparing parameter passing mechanisms

Being lazy is better!

Call-by-name can succeed in situations in which we would get a run-time error if we used call-by-value.

A simple example is

```
f x = 3.1415926
```

Notice that the body of f does not mention the formal parameter. The call f(1/0.0) will fail, if we used call-by-value but would succeed if we used call-by-name.

Properties of parameter passing mechanisms

Let f be function with body $E = \ldots x \ldots y \ldots$ containing free occurrences of x, y: $f(x, y) = E[x, y]$

Write $E[u/x, v/y]$ for E where all free occurrences of x and y are replaced by u and v, respectively. Substitution must be *capture avoiding*: No free variable in u or v must become bound.

Properties of parameter passing methods

Call by reference: $f(u, v) \cong E[u/x, v/y]$

Call by value: $f(e_1, e_2) \cong x := e_1, y := e_2; E[x, y]$

Call by value-return: $f(u, v) \cong x := u, y := v; r := E[x, y]; u := x, v := y; r$

Call by name: $f(e_1, e_2) \cong E[e_1/x, e_2/y]$

Call by need: $f(e_1, e_2) \cong E[e_1/x, e_2/y]$
Operationally, replace e_i by its value v_i if and when the value of e_i is *needed the first time*.

Parameter passing methods: Summary

- Call by value: Evaluate argument expression and bind or assign its value to formal parameter in function.
- Call by name: Substitute (capture-avoiding) argument expression for formal parameter in function.
  - Call by need: Optimized version of call by name; if value of expression is always the same (no side effects).
  - Not applicable to Algol: Has side effects (Jensen's device).
- Call by reference: Alias (l-value of) argument expression with formal parameter in function; they are the same variable.
  - Cannot rely on distinct identifiers in function to be distinct variables.
- Call by value-result: Evaluate (l-value of) argument expression and assign it to formal parameter in function; and value of formal parameter back to argument expression at end.
  - Similar to call by reference, but with no aliasing of formal parameters during evaluation of function body.