

Class 30: Interpreting Procedure Calls

Held: Friday, 15 April 2011

Summary: We consider a variety of strategies for relating the formal to actual parameters in a procedure call.

Related Pages:

- EBoard.
- Reading: Wikipedia on Evaluation Strategy; Parameter Passing in *Encyclopedia of Computer Science* (4th ed.).

Notes:

- We'll use about fifteen minutes at the start of class to talk about presentations.
- Readings for Monday: Graham, Paul (2002). *Revenge of the Nerds*. Graham, Paul (2002). *What Made Lisp Different*.
- EC for today's CS Table - Wolfram TED Talk, http://www.ted.com/talks/stephen_wolfram_computing_a_theory_of_everything.html [academic]
- EC for tonight's Bob's Benefit Bash [peer]

Overview:

- A Bit About Procedures.
- Parameter Passing.
- An Operational Approach.
- Jensen's Device.

A Bit About Procedures

- Most imperative languages provide a way to name and parameterize blocks of code. We traditionally refer to such blocks of code as *functions* or *procedures*.
 - Usually, functions return a value whereas procedures do not.
 - However, many use treat the two terms as being equivalent (as do I).
 - We will also treat as equivalent a host of other names, such as *subroutine* and *method*.
- *Why do we want to be able to name blocks of code?* Here are some reasons.
 - **To write shorter programs.** If you're writing the same code again and again, it is much easier to write and name it once, and then use the name afterwards.
 - **To write clearer/more readable programs.** Often, the details of how you do something are unimportant. In this sense, procedures provide a way to add new "basic operations" to the language.
 - **To write more analyzable programs.** Procedures provide one of the strongest forms of control restrictions. It is almost impossible (and probably meaningless) to enter a procedure anywhere

but at the top. We can also use preconditions and postconditions to restrict usage.

- **To support recursion.** While this was not a reason to include subroutines in early languages, it is now a clear reason to support them.
- I've been told that some programmer's used Fortran's nonrecursive nature to provide a form of error recovery: "when all else fails, call 'main'". Since there's no stack, this is, in effect, a reset.
- Note that we often consider a variety of aspects to procedures, including
 - The procedure *signature* (or *header* or *declaration* or *handler* or ...): a name and list of parameters
 - often both name and parameters are typed).
 - The procedure *definition*: the block of code corresponding to the procedure.
 - The procedure *call* or *invocation*: a request to execute the code.
 - The procedure *execution*: The status of the procedure while running.
- Languages provide a variety of structures like functions and procedures, including methods and macros.

Parameter Passing

- As we've noted, many procedures are parameterized.
- You might observe that there are two "kinds" of parameters that are used in a typical program:
 - The parameters of the function definition. We call these the *formal parameters* or *formals*.
 - The parameters of the function call. We call these the *actual parameters* or *actuals*.
- One particularly important design aspect of a language is the relationship between the formals and actuals, particularly when the actuals can be expressions rather than just variables.
- What must we consider?
 - When are the actuals evaluated?
 - What kinds of values are permitted for the actuals?
 - What is the relationship between the memory location used for the actuals and that used for the formals?
 - What are the possible effects on the actuals (when the actuals are variables or otherwise reference memory locations)?
- There are at least six basic relationships.
 - *pass-by-value* (standard parameters in Pascal and C)
 - *pass-by-reference* (var parameters in Pascal)
 - *pass-by-sharing* (an attempt to describe passing objects)
 - *pass-by-value-return* (an elegant variation of var parameters)
 - *pass-by-text* (macros in C)
 - *pass-by-name* (something like continuations)
- Evaluation
 - In pass-by-value, pass-by-reference, pass-by-sharing and pass-by-value-return, the actuals are evaluated before the call.
 - In pass-by-text and pass-by-name, the text (or a modified version of the text) of each actual is passed into the procedure and that text is substituted for the corresponding formal.
- Memory
 - In pass-by-value and pass-by-value-return, new memory is allocated to store values and the

value of the variable is copied.

- In pass-by-reference and pass-by-sharing, a reference to the variable is copied (so memory for a reference may be allocated), but the formal is, in effect, equivalent to the actual.
- This is an irrelevant issue in pass-by-`{text,name}`.
- Effect on actual
 - In all but pass-by-value, the actual may be affected by changes to the formal.
 - In all but pass-by-value-return, the effect on the actual is synchronous with the effect on the formal. In pass-by-value-return, the changes are not made until the function returns.
- Pass-by-text has some odd problems.
 - What happens if the actual is redeclared within the body of the procedure?
 - Recursion can become more difficult.
- Pass-by-name eliminates the first of these problems by associating each variable with a scope, so that the variable used is, in effect, a different variable.
 - Some find it easier to think about this as renaming of variables.

An Operational Approach

You may find it useful to think about the strategies operationally.

- In pass-by-value,
 - the actuals are evaluated before the call,
 - new memory is allocated for the formals,
 - the values of the actuals are copied into these new memory locations,
 - the body is executed.
- In pass-by-reference and pass-by-sharing
 - the actuals are evaluated before the call,
 - no new memory is allocated for the formals; rather, the formals name the memory locations of the results of the actuals (optionally, a reference is allocated, but that's equivalent);
 - the body is executed.
- In pass-by-value-return
 - the actuals are evaluated before the call,
 - new memory is allocated for the formals,
 - the values of the actuals are copied into these new memory locations,
 - the body is executed,
 - the values are copied back from the formals to the actuals.
- In pass-by-text (similar to macros),
 - the actuals are not evaluated before the call,
 - in the body of the code, the actuals are substituted for the formals,
 - the new body is executed.
- In pass-by-name,
 - the actuals are not evaluated before the call,
 - the actuals and the environment are packaged into a “thunk” that, when called,
 - in the body of the code, the actuals (with environment) are substituted for the formals,
 - the new body is executed,

- whenever a formal is used, the thunk is evaluated.

Jensen's Device

- To some, pass-by-name is very elegant.
- One instance of this elegance is Jensen's device.
- Here is the most common definition of Jensen's device.

```
function sum(exp, counter, start, finish)
begin
  tmp = 0;
  for counter = start to finish
    tmp = tmp + exp;
  end for
  return tmp
end sum
```

- What does this compute?
- Does your answer change if `exp` is `A[i]` or something similar?
 - Consider `sum(A[i], i, 1, 10)`