==
Parameter passing

Three basic implementations:
    value, value/result (copying)
    reference (aliasing)
    closure

Closures used not only for formal subroutines, but also ***name*** (lazy) parameters and label parameters (Algol 60, 68)

Some languages (e.g., Pascal) have provided `val` and `ref` directly
problem:
    pass big thing by `val` or `ref`?

Solution? (Modula-2):
    '`const`' mode that is read-only but passed by reference
but then `val` and `const` for small things are ~semantically redundant

Ada went for semantics: who can do what

`in`	formal initialized; actual not modified
`out`	formal not initialized; actual modified
`in out`	formal initialized; actual modified

Ada `in out` is always implemented as value/result for scalars, and either value/result or reference for structured objects. The language manual says your program is "erroneous" if it can tell the difference.

Call by reference is the only option in Fortran. If you pass a constant, the compiler creates a temporary location to hold it. If you modify the temporary, who cares?

In a language with a reference model of variables (Lisp, ML, etc.), the obvious approach is to make the formal parameter refer to the same thing as the actual parameter. I like the name ***call by sharing*** for this, because it isn't clear whether to think of it as value (formal parameter is a copy of the actual) or reference (formal parameter is a reference, and can change from caller's perspective). Note that with call by sharing you can change the value of the referred-to thing (assuming it isn't immutable), but you can't change which thing is referred to.

Call-by-name is an old Algol technique. Think of it as call by textual substitution (a procedure with all name parameters works like a macro). What you pass are hidden procedures called ***thunks***.

Jensen's device example:

```
function sum (expr, index : name real; low, high : const integer)
    returns answer : real;
begin
    answer := 0;
    for i in low..high loop
        index := i;
        answer +:= expr;
    end loop;
end sum;

S := sum (A[2*i-1], i, 1, 10);
```

(Curiously, it doesn't seem to be possible to write a general-purpose swap routine with name parameters.)

Call-by-name is a naive implementation of normal-order evaluation. ***Call-by-need*** does memoization. It's used in Haskell, which is purely functional, and in R, which is not. Both call-by-name and call-by-need are considered "lazy evaluation" by the functional programming community (there's some inconsistency of nomenclature here -- compiler people sometimes use "lazy" only for call-by-need).

In a pure functional language call by name and call by need are semantically indistinguishable; with side effects they aren't.

Note that passing dynamic arrays by value is tricky.
The actual parameter list on the stack contains a fixed-size dope vector and a pointer, but where does the data go?
One option is below the arguments. Another is to let the callee copy the data into the new stack frame immediately after the prologue.

Summary:

Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
`in out`	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write†	yes†	yes†

* Behavior is undefined if the program attempts to use an r-value argument after the call.
† Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Other parameter issues

- conformant arrays -- variable-size array parameters in a language that otherwise doesn't support variable-size arrays
- default (optional) parameters -- don't avoid cost
- named parameters -- great for long parameter lists
- variable number of parameters -- typesafe?

Function returns

Pascal and Fortran return values from functions by assigning to the function identifier.

User cannot re-use the name of a function inside. Later languages tend to have an explicit 'return' statement (as in C or Ada), or a named return value (as in Algol 68, above, or Go).

Another advantage of named returns is that you can use the name inside expressions:

```
-- Ada
type int_array is array (integer range <>) of integer;
    -- array of integers with unspecified integer bounds
function A_max (A : int_array) return integer is
rtn : integer;
begin
    rtn := integer'first;
    for i in A'first .. A'last loop
        if A(i) > rtn then rtn := A(i); end if;
    end loop
    return rtn;
end A_max;


-- Go
func A_max(A []int) (rtn int) {
    rtn = A[0]
    for i := 1; i < len(A); i++ {
        if A[i] > rtn { rtn = A[i] }
    }
    return
}
```