Parameter passing

- When passing arguments, need to support right semantics
- One issue: when is the argument expression evaluated?
  - Before call?
  - If and when needed by callee?
- Another issue: what happens if formal is assigned to in callee?
  - Is this visible to the caller? If so, when?
  - What happens with aliasing among arguments and lexically visible variables?
- Different choices lead to different representations for passed arguments and different code to access formals

Some parameter passing modes

- call-by-value
- call-by-sharing
- call-by-reference
- call-by-value-result
- call-by-name
- call-by-need
- ...

Call-by-value

- If formal is assigned, doesn’t affect caller’s value
- Implement by passing copy of argument value
  - Trivial for scalars
  - Inefficient for aggregates

Call-by-sharing

- If implicitly reference aggregate data via pointer (e.g., Java, Lisp, Smalltalk, ML, …) then call-by-sharing is call-by-value applied to implicit pointer
  - “call-by-pointer-value”
  - Efficient, even for big aggregates
  - Assignments of formal to a different aggregate don’t affect caller (e.g., f := x)
  - Updates to contents of aggregate visible to caller immediately (e.g., f[i] := x)
  - Aliasing/sharing relationships are preserved

Call-by-reference

- If formal is assigned, actual value is changed in caller
  - Change occurs immediately
  - Assumes actual is an lvalue
  - Implement by passing pointer to actual
  - Efficient for big data structures
  - References to formal must do extra dereference
Big immutable data
for example, a constant string

- Expensive to pass by-value
- Can implement as call-by-reference
  - Since you can’t assign to the data, you don’t care

Call-by-value-result

- If formal is assigned, final value is copied back to caller when callee returns
  - “copy-in, copy-out”
- Implement as call-by-value with assignment back when procedure returns
  - More efficient for

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
y := y + a;
end foo;
a := 2;
foo(a,a);
output := a;
```

Call-by-result

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
y := y + a;
end foo;
a := 2;
foo(a,a);
output := a;
```

Ada: in, out, in out

- Programmer selects intent
- Compiler decides what mechanism is more efficient
- Program’s meaning “shouldn’t” depend on which is chosen

Call-by-name, call-by-need

- Variations on lazy evaluation
  - Only evaluate argument expression if and when needed by callee
  - Supports very cool programming tricks
  - Hard to implement efficiently in traditional compilers
    - Thunks
    - Largely incompatible with side-effects
      - So more common in purely functional languages like Haskell and Miranda
      - But did appear first in Algol-60

Call-by-name

```
proc square(x);
  int x;
begin
  x := x * x
end;
square(A[i]);
```

```
proc square(x);
  int x;
begin
  x := x * x
end;
square(A[i]);
```
Jensen’s device

- How to implement the equivalent of math formulae like
- \[ \sum_{0 \leq i \leq n} A_i \]
- Try `\texttt{sum}(i, 0, n, A[i])`
- Passing by-reference or by-value does work, since we can only pass in one element of A
- So, use Jensen’s device

```c
int proc \texttt{sum}(j, lo, hi, Aj);  
int j, lo, hi, Aj, s;  
begin  
s := 0;  
for j := lo to hi do  
s := s + Aj;  
end;  
return s;  
end;
```

A classic problem:

- a procedure to swap two elements

```c
proc \texttt{swap}(int a, int b);  
int temp;  
begin  
temp := a; a := b; b := temp  
end;
```

```
\texttt{swap}(x, y);  
\texttt{swap}(i, x[i]);
```

- i = 2
- x[2] = 5

Advantages of call-by-name

- Textual substitution is a simple, clear semantic model
- There are some useful applications, like Jensen’s device
- Argument expressions are evaluated lazily

Disadvantages of call-by-name

- Repeatedly evaluating arguments can be inefficient
- Pass-by-name precludes some standard procedures from being implemented
- Pass-by-name is difficult to implement

thunks

- Call-by-name arguments are compiled to thunks, special parameter-less
- Thunks are passed into the called procedure and called to evaluate the argument whenever necessary

Parameter passing and compiling

- There is an intimate link between the semantics of a programming language and the mechanisms used for parameter passing
- Maybe more than other programming language constructs, the connection is extremely strong between implementation and language semantics in this area