

# CMSC 330: Organization of Programming Languages

## Parameter Passing

## Parameter Passing in OCaml

- Quiz: What value is bound to `z`?

```
let add x y = x + y
let z = add 3 4
```

7

```
let add x y = x + y
let z = add (add 3 1) (add 4 1)
```

9

```
let r = ref 0
let add x y = (!r) + x + y
let set_r () = r := 3; 1
let z = add (set_r ()) 2
```

6

Actuals evaluated before call

CMSC 330

2

## Call-by-Value

- In **call-by-value** (*cbv*), arguments to functions are fully evaluated before the function is invoked
  - Also in OCaml, in `let x = e1 in e2`, the expression `e1` is fully evaluated before `e2` is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;
int add(int x, int y) { return r + x + y; }
int set_r(void) {
    r = 3;
    return 1;
}
add(set_r(), 2);
```

CMSC 330

3

## Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print? 0

```
void f(int x) {
    x = 3;
}

int main() {
    int x = 0;
    f(x);
    printf("%d\n", x);
}
```

- Cbv protects function arguments against modifications

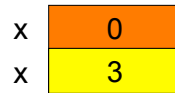
CMSC 330

4

## Call-by-Value (cont.)

- ▶ Actual parameter is copied to stack location of formal parameter

```
void f(int x) {  
    x = 3;  
}  
  
int main() {  
    int x = 0;  
    f(x);  
    printf("%d\n", x);  
}
```



## Call-by-Reference

- ▶ Alternative idea
  - Implicitly pass a **pointer** or **reference** to actual parameter
  - If the function writes to it the actual parameter is modified

```
void f(int x) {  
    x = 3;  
}  
  
int main() {  
    int x = 0;  
    f(x);  
    printf("%d\n", x);  
}
```



## Call-by-Reference (cont.)

- ▶ Advantages
  - Allows multiple return values
  - Avoid copying entire argument to called function
    - > More efficient when passing large (multi-word) arguments
    - > Can do this without explicit pointer manipulation
- ▶ Disadvantages
  - More work to pass non-variable arguments
    - > Examples: constant, function result
  - May be hard to tell if function modifies arguments
  - Introduces **aliasing**

## Aliasing

- ▶ We say that two names are **aliased** if they refer to the same object in memory
  - C examples (this is what makes optimizing C hard)

```
int x;  
int *p, *q;  
p = &x; /* *p and x are aliased */  
q = p; /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }  
struct list *p, *q;  
...  
q = p; /* *q and *p are aliased */  
/* so are p->x and q->x */  
/* and p->next->x and q->next->x... */
```

## Call-by-Reference (cont.)

- ▶ Call-by-reference is still around (e.g., C++)

```
int x = 0; // C++
void f(int& y) { y = 1; } // y = reference var
f(x); printf("%d\n", x); // prints 1
f(2); // error
```

- ▶ Seems to be less popular in newer languages
  - Older languages still use it
    - > Examples: Fortran, Ada
  - Possible efficiency gains not worth the confusion
    - > Pointer notation, if you've got it, seems easier
  - The “hardware” is basically call-by-value
    - > Although call by reference is not hard to implement and there may be some support for it

## Call-by-Value Discussion

- ▶ Cbv is standard for languages with side effects
  - When we have side effects, we need to know the order in which things are evaluated
    - > Otherwise programs have unpredictable behavior
  - Call-by-value specifies the order at function calls
  - Call-by-reference can sometimes give different results
- ▶ A note about Java
  - Language is call by value
  - But most parameters are object references anyway
    - > Still have aliasing, parameter modifications at object level

## Call-by-Name (cont.)

- ▶ In **call-by-name (cbn)**, arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y
let z = add (add 3 1) (add 4 1)
```

OCaml; cbv; arguments evaluated here

```
add (add 3 1) (add 4 1) ->
add 4 (add 4 1) ->
add 4 5 ->
4 + 5 -> 9
```

Haskell; cbn; arguments evaluated here

```
add x y = x + y
z = add (add 3 1) (add 4 1)
```

```
add (add 3 1) (add 4 1) ->
(add 3 1) + (add 4 1) ->
4 + (add 4 1) ->
4 + 5 -> 9
```

## Call-by-Name (cont.)

- ▶ What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; cbv; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; cbn; never evaluated because parameter is never used

## Two Cool Things to Do with CBN

- ▶ CBN is also called **lazy evaluation**
  - CBV is also known as **eager evaluation**
- ▶ Build control structures with functions

```
let cond p x y = if p then x else y
```

- ▶ Build “infinite” data structures

```
integers n = n::(integers (n+1))
take 10 (integers 0) (* infinite loop in cbv *)
```

## Simulating CBN with CBV

- ▶ Thunk
  - A function with no arguments
- ▶ Algorithm
  1. Replace arguments  $a_1 \dots a_k$  by thunks  $t_1 \dots t_k$ 
    - ▶ When called,  $t_i$  evaluates and returns  $a_i$
  2. Within body of the function
    - ▶ Replace formal argument with thunk invocations

```
let add1 x = x + 1 in add1 (2+3)
```



```
let add1 x = x () + 1 in add1 (fun () -> (2+3))
```

## Simulating CBN with CBV (cont.)

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

- becomes...
  - Get 1st arg
  - Return 2nd arg
  - Never invoked

```
let cond p x y = if (p ()) then (x ()) else (y ())
let rec loop n = loop n (* didn't transform... *)
let z = cond (fun () -> true)
              (fun () -> 42)
              (fun () -> loop 0) } Parameters are
                                now thunks
```

## Call by Name and References

- ▶ Call-by-name mixes poorly with updateable references
  - Everyone now agrees this is a bad idea!
- ▶  $P(x) \{x = x + x;\}$

What is:  $Y = 2;$   
 $P(Y);$   
 $write(Y)$  ← becomes  $Y = Y+Y = 4$

- ▶  $F(m) \{return m++;\}$

What is:  $int A[10];$   
 $m = 1;$   
 $P(A[F(m)])$   
 $\rightarrow A[F(m)] = A[F(m)]+A[F(m)]$

$\rightarrow A[m++] = A[m++]+A[m++]$   
 $\rightarrow A[2] = A[3]+A[4]$  (depending on eval order)

## Call by Name Anomalies

- ▶ Write a function to exchange values of X and Y
- ▶ Usual way - `swap(x,y) { t=x; x=y; y=t; }`
  - Cannot do it with call by name!
- ▶ Reason
  - Cannot handle both of following
    - > `swap(A[m], m)`
    - > `swap(m, A[m])`
  - One of these must fail
    - > `swap(A[m], m) → t = A[m]; A[m] = m; m = t;`
    - > `swap(m, A[m]) → t = m; m = A[m]; A[m] = t; // fails!`



## Three-Way Comparison

- ▶ Consider the following program under the three calling conventions
  - For each, determine `i`'s value and which `a[i]` (if any) is modified

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
           i, a[0], a[1], a[2]);
}
```

## Example: Call-by-Value

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
           i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]	f	g
1	0	1	2		
				1	1
				5	2

## Example: Call-by-Reference

```
int i = 1;

void p(int f, int g) {
    g++;
    f = 5 * i;
}

int main() {
    int a[] = {0, 1, 2};
    p(a[i], i);
    printf("%d %d %d %d\n",
           i, a[0], a[1], a[2]);
}
```

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2		10	
2		10	

## Example: Call-by-Name

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i; } i++;
                a[i] = 5*i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
         i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]
1	0	1	2
2			10
2			10

The expression `a[i]` isn't evaluated until needed, in this case after `i` has changed.

## Other Calling Mechanisms

- ▶ Call-by-result
  - Actual argument passed by reference, but not initialized
  - Written to in function body (and since passed by reference, affects actual argument)
- ▶ Call-by-value-result
  - Actual argument copied in on call (like cbv)
  - Mutated within function, but does not affect actual yet
  - At end of function body, copied back out to actual
- ▶ These calling mechanisms didn't really catch on
  - They can be confusing in cases
  - Recent languages don't use them

## CBV versus CBN

- ▶ CBN is flexible--strictly more programs terminate
  - E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate
- ▶ Order of evaluation is really hard to see in CBN
  - Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)
- ▶ Call-by-name is more expensive since
  - Functions have to be passed around
  - If you use a parameter twice in a function body, its thunk (the unevaluated argument) will be called twice
    - Haskell actually uses *call-by-need* (each formal parameter is evaluated only once, where it's first used in a function)