Programming Languages

Functions and Recursion
High Level Languages

Java (Object Oriented)

ACL2 (Propositional Induction)

Algorithmic Information Theory (Information Compression and Randomness) - Kolmogorov Complexity

Orc (Parallel Computing)

GpH (Parallel Computing)

RDF (Horn Clause Deduction, Semantic Web)

This Course

Jython in Java

A Snapshot of Programming Language History
Lisp Function Definition (Declaration) and Application (Invocation)

Using Scheme8 on the class calendar webpage, evaluate the following:

```
(interp (parse (quote (with ((z 7)) (with ((z 3) (a 5) (x (fun (x y) (- x (+ y z)))))
                       (with ((z 10) (a 5)) (x z a)))))
      ) (mtEnv))
```

The (parse (quote (with ((z 7)) . . . evaluates to the following:

```
(with
  (list (binding 'z (num 7)))
  (with
    (list (binding 'z (num 3)) (binding 'a (num 5)) (binding 'x (fun (list 'x 'y) (binop - (id 'x) (binop + (id 'y) (id 'z))))))
    (with (list (binding 'z (num 10)) (binding 'a (num 5)) (app (id 'x) (list (id 'z) (id 'a)))))
  )
)
```

• Where are the assignments?

• Where are the global variables?

• Are there any side affects?
Dr. Philip Cannata
Definitions

An **argument** is an expression that appears in a function call.

A **parameter** is an identifier that appears in a function declaration.

In the code on the right the call `A(a, b)` has arguments `a` and `b`.

The function declaration `A` has parameters `x` and `y`.

```c
int h, i;
void B(int w) {
    int j = 1, k = 2;
    i = 2*w;
    w = w+1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d\n", w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2;
    B(h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d\n", x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
    printf("In Main a, b, h, i: %d, %d, %d, %d\n", a, b, h, i);
}
```
Parameter Passing Mechanisms

- By value
- By reference
- By value-result
- By name
Pass by Value

Compute the value of the argument at the time of the call and assign that value to the parameter.

So passing by value doesn’t normally allow the called function to modify an argument’s value.

All arguments in C and Java are passed by value.

But references can be passed to allow argument values to be modified. E.g., void swap(int *a, int *b) { … }

int h, i;
void B(int w) {
    int j = 1, k = 2;
    i = 2*w;
    w = w+1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d\n", w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2;
    B(h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d\n", x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
    printf("In Main a, b, h, i: %d, %d, %d, %d\n", a, b, h, i);
}

$ ./a
In B - w, j, k, h, i: 6, 1, 2, 5, 10
In A - x, y, i, j, h: 3, 2, 1.100000, 2.200000, 5
In Main a, b, h, i: 3, 2, 5, 10
Compute the *address* of the argument at the time of the call and assign it to the parameter.

Since `h` is passed by reference, its value changes during the call to `B`.
int h, i;
void B(int w) {
    int j = 1, k = 2;
    i = 2*w;
    w = w+1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d,
          %d\n",  w, j, k, h, i);
}
void A(int x, int y) {
    float i = 1.1, j = 2.2;
    B(h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f,
            %d\n", x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
    printf("In Main a, b, h, i: %d, %d, %d,
            %d\n", a, b, h, i);
}

$ ./a
In B - w, j, k, h, i: 6, 1, 2, 5, 10
In A - x, y, i, j, h: 3, 2, 1.100000, 2.200000, 5
In Main a, b, h, i: 3, 2, 5, 10

Pass by Value                             Pass by Reference
int h, i;
void B(int *w) {
    int j = 1, k = 2;
    i = 2(*w);
    *w = *w + 1;
    printf("In B - w, j, k, h, i: %d, %d, %d, %d, %d
          w, j, k, h, i);
}
void A(int *x, int *y) {
    float i = 1.1, j = 2.2;
    B(&h);
    printf("In A - x, y, i, j, h: %d, %d, %f, %f, %d
            , x, y, i, j, h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(&a, &b);
    printf("In Main a, b, h, i: %d, %d, %d, %d
            a, b, h, i);
}

$ ./a
In B - w, j, k, h, i: 4206640, 1, 2, 6, 10
In A - x, y, i, j, h: 2280676, 2280672, 1.100000,
2.200000, 6
In Main a, b, h, i: 3, 2, 6, 10
Pass by Value-Results

Pass by value at the time of the call and/or copy the result back to the argument at the end of the call – also called copy-in-copy-out.
Pass by Name

Textually substitute the argument for every instance of its corresponding parameter in the function body. Originated with Algol 60 (Jensen’s device), but was dropped by Algol’s successors -- Pascal, Ada, Modula.

```plaintext
real procedure Sum(j, lo, hi, Ej);
value lo, hi; integer j, lo, hi; real Ej;
begin
  real S; S := 0;
  for j := lo step 1 until hi do
    S := S + Ej;
  Sum := S
end;
```

Exemplifies *late binding*, since evaluation of the argument is delayed until its occurrence in the function body is actually executed.

Associated with *lazy evaluation* in functional languages (e.g., Haskell).
A function that can call itself, either directly or indirectly, is a recursive function.
A stack of activation records:

• Each new call pushes an activation record, and each completing call pops the topmost one.

• So, the topmost record is the most recent call, and the stack has all active calls at any run-time moment.
A block of information associated with each function call, which includes:

- **parameters and local variables**
- **Return address**

“We call this constructed value a closure because it “closes” the function body over the substitutions that are waiting to occur. When the interpreter encounters a function application, it must ensure that the function’s pending substitutions aren’t forgotten. It must, however, ignore the substitutions pending at the location of the invocation, for that is precisely what led us to dynamic instead of static scope. It must instead use the substitutions of the invocation location to convert the function and argument into values, hope that the function expression evaluated to a closure, then proceed with evaluating the body employing the repository of deferred substitutions stored in the closure.”

[textbook. Pages 46-47]
int h, i;
void B(int w) {
    int j, k;
    i = 2*w;
    w = w+1;
}
void A(int x, int y) {
    bool i, j;
    B(h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
}

- parameters and local variables
- Return address
- Saved registers
- Temporary variables
- Return value
int factorial(int n) {
    if(n < 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number, answer;
    number = 3;
    answer = factorial(number);
    print(answer);
}

Hmm Runtime Stack for Factorial 3

Calling function: factorial
BasePtr = 3, printing runtime stack
null: null
n: 3
null: null
answer: null
number: 3

Calling function: factorial
BasePtr = 5, printing runtime stack
null: null
n: 2
null: null
n: 3
null: null
answer: null
number: 3

Calling function: factorial
BasePtr = 7, printing runtime stack
null: null
n: 1
null: null
n: 2
null: null
n: 3
null: null
answer: null
number: 3

Calling function: factorial
BasePtr = 9, printing runtime stack
null: null
n: 0
null: null
n: 1
null: null
n: 2
null: null
n: 3
null: null
answer: null
number: 3

Exiting function: factorial
BasePtr = 9, printing runtime stack
null: null
n: 0
return#factorial: 1
n: 1
null: null
n: 2
null: null
n: 3
null: null
answer: null
number: 3

Exiting function: factorial
BasePtr = 7, printing runtime stack
return#factorial: 1
n: 1
return#factorial: 1
n: 2
null: null
n: 3
null: null
answer: null
number: 3

Exiting function: factorial
BasePtr = 5, printing runtime stack
return#factorial: 1
n: 2
return#factorial: 2
n: 3
null: null
answer: null
number: 3

Exiting function: factorial
BasePtr = 3, printing runtime stack
return#factorial: 2
n: 3
return#factorial: 6
answer: null
number: 3
A Taxonomy of Functions

6.1 A Taxonomy of Functions

• **first-order** Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

• **higher-order** Functions can [be defined anywhere in a program and] return other functions as values.

• **first-class** Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures. [They can also be defined anywhere in a program.]