Class Information

- Project 1 grades have been posted.
- Midterm: Sample solutions will be posted by the end of the week.

  Goal: Get grades and exams to you next week (exams will be returned in recitation)

  Look at the sample solution before asking any questions.

Review: Parameter Passing Modes

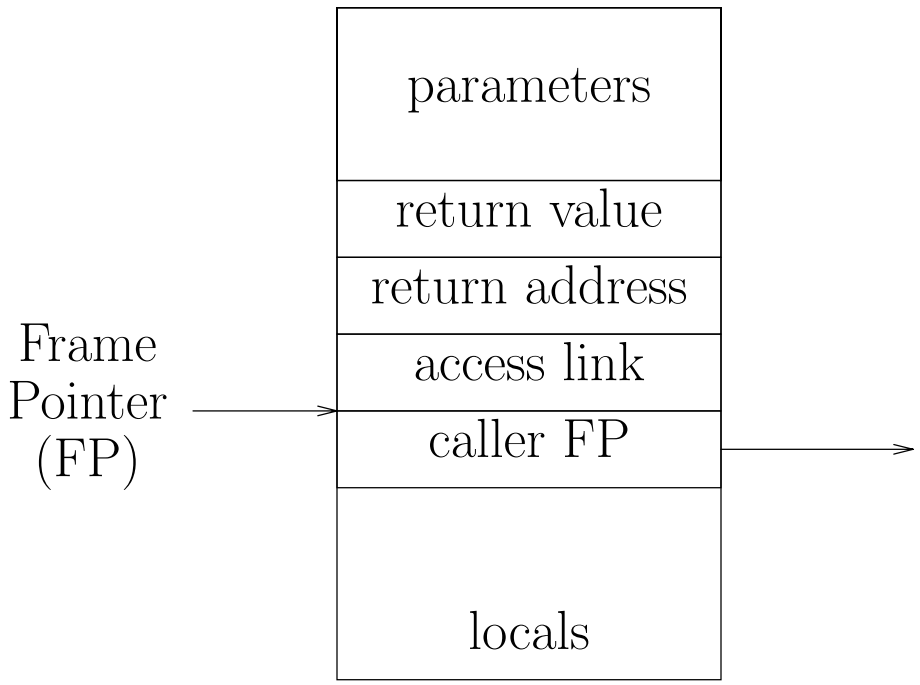
Scott: Chapter 8.3

Parameter Passing Modes

- pass-by-value: C, Pascal, Ada (`in` parameter), Scheme, Algol 68
- pass-by-result: Ada (`out` parameter)
- pass-by-value-result: Ada (`in out` parameter)
- pass-by-reference: Fortran, Pascal (`var` parameter)
- pass-by-name (not really used any more): Algol60

Review: Stack Frames

- Run-time stack contains frames for main program and each active procedure.
- Each stack frame includes:
  1. Pointer to stack frame of caller (control link)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (access link)
  4. Storage for parameters
  5. Storage for local variables
  6. Storage for final values



Pass-by-value

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

5 3

Advantage: Argument protected from changes in callee

Disadvantage: Copying of values takes execution time and space, especially for aggregate values (e.g.:arrays, structs).

Pass-by-reference

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

6 5

Advantage: more efficient than copying

Disadvantage: leads to **aliasing**: there are two or more names for the same storage location; hard to track side effects

Pass-by-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;           ==> ERROR: CANNOT USE PARAMETERS
    j := j+2;               WHICH ARE UNINITIALIZED
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output: **program doesn't compile or has runtime error**

Pass-by-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := 1;           ==> HERE IS ANOTHER PROGRAM
    j := 2;               THAT WORKS
  end r;
...
  m := 5;
  n := 3
  r(m,m);             ==> NOTE: CHANGED THE CALL
  write m,n;
end
```

Output: 1 or 2?

Problem: order of copy-back makes a difference; implementation dependent.

Pass-by-value-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  m := 5;
  n := 3
  r(m,n);
  write m,n;
end
```

Output:

6 5

Problem: order of copy-back can make a difference; implementation dependent.

Pass-by-value-result

```
begin
  c: array[1..10] of integer;
  m, n: integer;
  procedure r(k, j: integer)
  begin
    k := k+1;
    j := j+2;
  end r;
...
  /* set c[m] = m */
  m := 2;
  r(m,c[m]); ==> WHAT ELEMENT OF ‘‘c’’ IS ASSIGNED TO?
  write c[1], c[2], ... c[10];
end
```

Output:

1 4 3 4 5 ... 10 on entry

1 2 4 4 5 ... 10 on exit

Problem: When is the address computed for the copy-back operation? At procedure call (procedure entry), just before procedure exit, somewhere inbetween? (Example: ADA on entry)

Aliasing

Aliasing:

More than two ways to name the same object within a scope

Even without pointers, you can have aliasing through (global $\leftrightarrow$ formal) and (formal $\leftrightarrow$ formal) parameter passing.

```
begin
  j, k, m: integer;
  procedure q(a,b: integer);
  begin
    b := 3;
    m := m*a;
  end
  ...
  q(m,k);  ==> global/formal <m,a> ALIAS PAIR
  q(j,j);  ==> formal/formal <a,b> ALIAS PAIR
  write y;
end
```

Comparison: by-value-result vs. by-reference

Actual parameters need to evaluate to L-values (addresses).

```
begin
  y: integer;
  procedure p(x: integer);
  begin
    x := x+1;    ==> ref: x and y are ALIASED
    x := x+y;    ==> val-res: x and y are NOT ALIASED
  end
  ...
  y := 2;
  p(y);
  write y;
end
```

Output:

- pass-by-reference: 6
- pass-by-value-result: 5

Note: by-value-result: Requires copying of parameter values (expensive for aggregate values); does not have aliasing, but copy-back order dependence;

Functional Programming

Pure Functional Languages

Scott: Chapter 10

Fundamental concept: **application** of (mathematical) **functions** to **values**

1. **Referential transparency:** The value of a function application is independent of the context in which it occurs
   - value of `f(a,b,c)` depends only on the values of `f`, `a`, `b` and `c`
   - It does not depend on the global state of computation

   $\Rightarrow$ all vars in function must be local (or parameters)

Pure Functional Languages

2. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state
3. Control flow is governed by function calls and conditional expressions

   $\Rightarrow$ no iteration

   $\Rightarrow$ recursion is widely used

Pure Functional Languages

4. All storage management is implicit
    - needs garbage collection
5. Functions are *First Class Values*
    - Can be returned as the value of an expression
    - Can be passed as an argument
    - Can be put in a data structure as a value
    - (Unnamed) functions exist as values

Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```
> (define length
    (lambda (x)
      (if (null? x)
          0
        (+ 1 (length (rest x))))))

> (length '(A LIST OF 5 THINGS))
5
```

READ-EVAL-PRINT Loop

The Scheme interpreters on the ilab machines are called mzscheme, racket, and drracket. "drracket" is an interactive environment, the others are command-line based. For example: Type `mzscheme`, and you are in the READ-EVAL-PRINT loop. Use Control D to exit the interpreter.

READ: Read input from user:
a function application

EVAL: Evaluate input:
`(f arg`$_1$ `arg`$_2$ `...arg`$_n$`)`

1. evaluate `f` to obtain a function
2. evaluate each `arg`$_i$ to obtain a value
3. apply function to argument values

PRINT: Print resulting value:
the result of the function application

You can write your Scheme program in file <name>.ss and then read it into the Scheme interpreter by saying at the interpreter prompt: `(load "<name>.ss")`

Next Lecture

More on Scheme

Please see our website for an online Scheme textbook