Note 5

Control Structures

Yunheung Paek
Associate Professor
Software Optimizations and Restructuring Lab.
Seoul National University

Topics

- Evaluation of expressions
  - precedence
  - associativity
- Evaluation of statements
  - sequential
  - selective
  - iterative
- exceptions
- recursion

Control structures

- Control structures control the order of execution of operations in a program.
- **expression-level** control structures

  → precedence/associativity rules, parentheses, function calls
- **statement-level** control structures
  1. sequential structures: *$stmt_1$; $stmt_2$; …; $stmt_n$;*

     → a sequence of compound statements
  2. selective structures: *if-then-else*, *case*/*switch*
  3. iterative structures: *for*, *while*, *do*, *repeat*
  4. escape/exception/branch: *exit*, *break*, *goto*, *continue*
  5. recursive structures: by recursive function calls

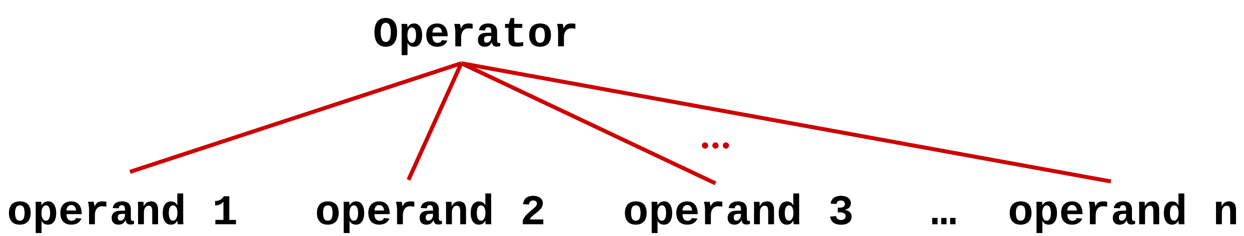
An expression is …

- a means of specifying computations in a program.
- composed of one or more operations.
- An operation = an operator + zero or more operands
  - operators: arithmetic, logical, relational, assignment, procedure call, reference/dereference, comma, id, constant, …
  - operands: sub expressions

C++

```
p = &z;
x = y + *p / 0.4;
a[i] = (z > 0 ? foo(x, y, p) : -1);
```

- Syntax tree: abstract representation of expressions

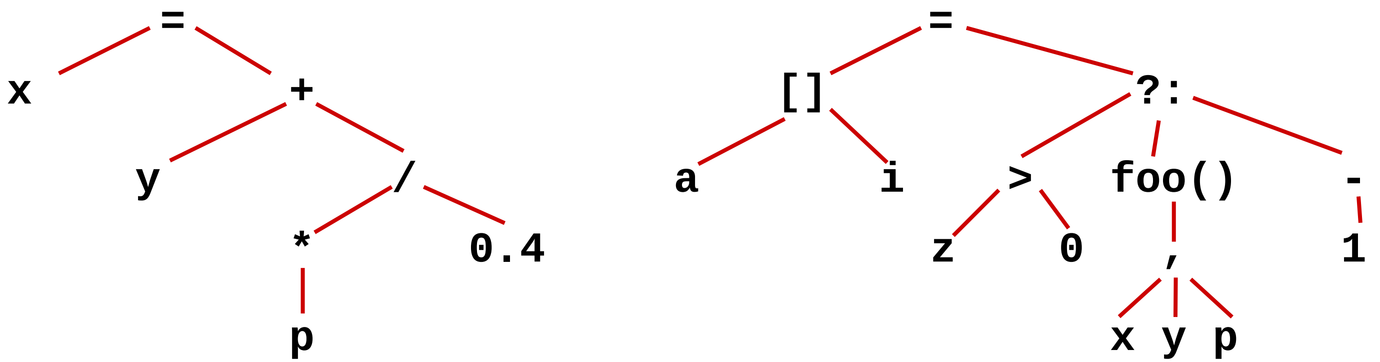


→ a node = an operator, children of a node = operands

Evaluation of expressions

- Executing a program is actually a sequence of evaluation of expressions in the program.
- How does the compiler/machine determine the evaluation order of an expression?

  → *use a syntax tree*



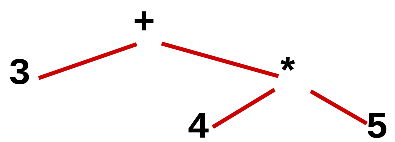
- The expression evaluation order in a language (in other words, the way to build a syntax tree) is defined by the language semantics.

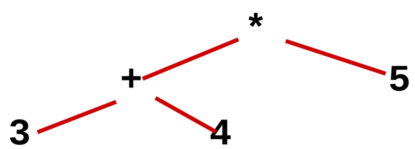
Rules specifying evaluation orders

- **Precedence rule**: the relative priority of operators when more than one kinds of operator are present

  Ex: “* has higher precedence than +”

  → thus, 3+4*5 is equal to 3+(4*5), not (3+4)*5.

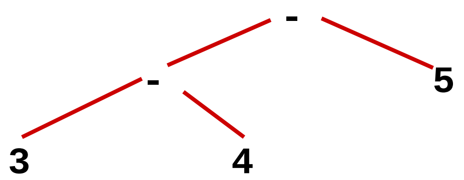


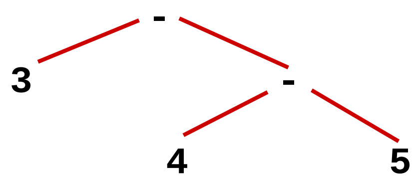


- **Associativity rule**: the relative priority of operators when two adjacent operators with the same precedence occur in a expression

  Ex: “- is left-associative”

  → thus, 3-4-5 is equal to (3-4)-5, not 3-(4-5).





Operator precedence/associativity in C

Precedence	Operators	Associativity
15	`-> . [] ()`	Left
14	`++ - ~ ! unary+ unary- * &`	Right
13	`* / %`	Left
12	`+ -`	Left
11	`<< >>`	Left
10	`< > <= =>`	Left
9	`== !=`	Left
8	`&`	Left
7	`^`	Left
6	|	Left
5	`&&`	Left
4	||	Left
3	`?:`	Left
2	`=`	Right
1	`,`	Left

Evaluation order of function arguments

- Given a function invocation **`func(arg`$_1$`,arg`$_2$`,…,arg`$_n$`)`**, all the arguments are usually evaluated before **`func`** is called. (consider the syntax tree)

  $\mathbf{arg_i}$ represents an expression for the i-th argument for the invocation

  → Then, what is the order of evaluation of them?

  1. no order imposed (Fortran)
  2. left-to-right order (C++, Dr Scheme)
  3. right-to-left order (MIT Scheme)

- The order is important due to ____________of expressions.

  Ex: Dr Scheme

```
> (define x 3)
> (define inc-x (lambda ()
        (begin (set! x (+ 1 x)) x)))
> (define foo (lambda (m n) (+ (* 100 m) n)))
> (foo (inc-x) (inc-x))
405
> (+ (* 100 (inc-x)) (inc-x))
607
```

evaluated in the DFS order

1. What are the values of **`m`** and **`n`** in the procedure call for **`foo`**?
2. What if MIT Scheme is used?

Sequential structures

- When is the order of a sequence of compound statements important?

C

```
{
      statement1;
      statement2;
          …
      statementn;
}
```

Scheme

```
(begin
      (function1 …)
      (function2 …)
          …
      (functionn …))
```

➔ It is when there is *data dependence* between the statements.
That is, when the same location is modified by different statements.

- Find data dependences in the following statements.

```
x = 3.4
y = x – 2.1;
```

```
x = z;
y = 2.3 + z;
w = z / 0.6;
print z;
```

```
read y;
x = 5.0;
x = 7.1;
y = x * 1.1;
```

```
a[2] = a[1] + 1
x = a[3] * x
a[4] = a[3] / y
```

```
x = 8.8;
y = 3.9 * 1.2;
z = 4.1;
```

```
b[i] = j + 0.1;
b[i+1] = b[i-1];
I = I + 1;
m = b[i] + b[i-1];
```

Selective, iterative structures

- Selective structures:
  - choose control flow depending on conditional test
  - Most languages support → if/then/else, switch/case
- Iterative structures
  - looping construct
  - C → while, for, do-while
- *goto* statements
  - efficient, general purpose, easy to use and translate to machine codes
  - flattens hierarchical program structures into a linear collection of statements -> difficult to read/understand
  - difficult to optimize or verify programs

Exceptions

- Diverse types of error may occur in program execution
  - overflow, type error, segment faults, divide by zero
  - **Exceptions** are such errors detected at run time.
- What would happen if your program ignores exceptions?

```
$ a.out
Abort (core dump)
```

  - Errors will eventually cause low-level message (from O/S or hardware) to be printed and to terminate the program execution.
  - What is the problem with low-level messages? → *They do not provide sufficient information about the error that caused your program to end.*
- Alternative solution: *use test code defined by languages or users*

```
test result = foo(a,b,c);
if (test result is error) raise exception;
```

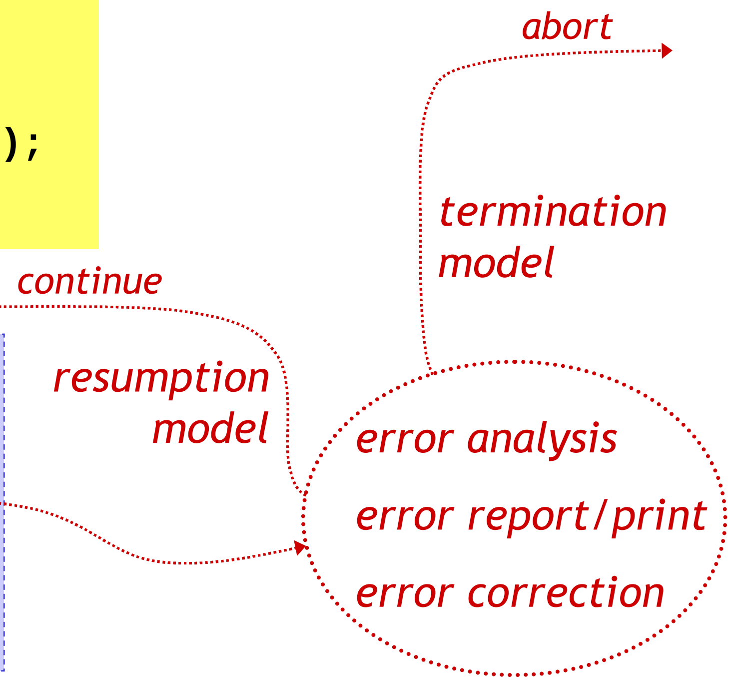
→ When an exception is raised, the normal program control is interrupted and the control is transferred to an **exception handler**, a special routine that handles the exception.

Exception handlings

- control flow for exception handling

```
foo (int i, char c) {
        float a[10];
        …
        if (error occurs)
                raise exception(error-type);
                …
}
```

```
exception_handler {
        switch (error-type) {
                case 1: …
                case 2: …
                …
                case n: …
        }
}
```



- Exception handling makes programs robust & reliable. But, it may be tedious because it needs to test possible errors. This might be inefficient if errors occurs infrequently.

Recursive structures

- A function *f* is **recursive** if it contains an application of *f* in its definition.

C++

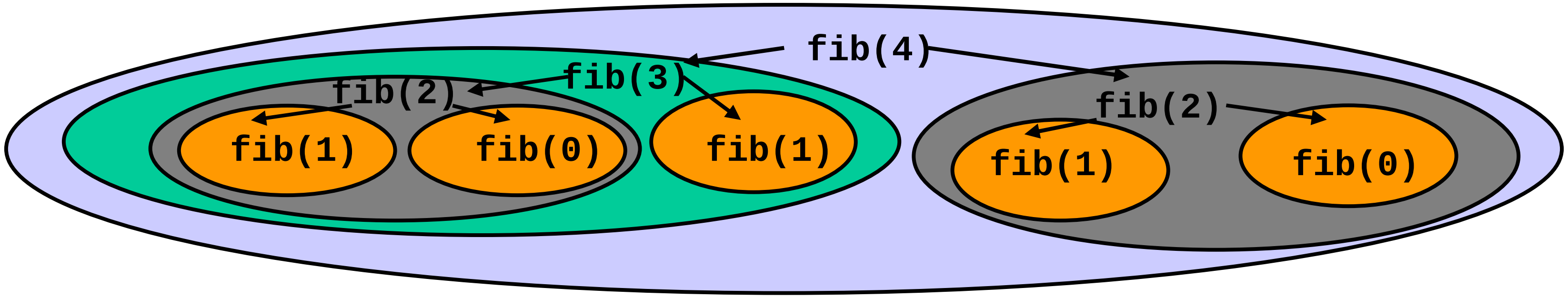
```
int fib(int n) {
   return ((n==0||n==1) ?
     1 : fib(n-1)+fib(n-2)));
}
```

Scheme

```
(define fib (lambda (n)
    (if (or (= n 0) (= n 1))
      1
      (+ (fib (- n 1))
         (fib (- n 2))))))
```

→ *How does Scheme implement recursion?*

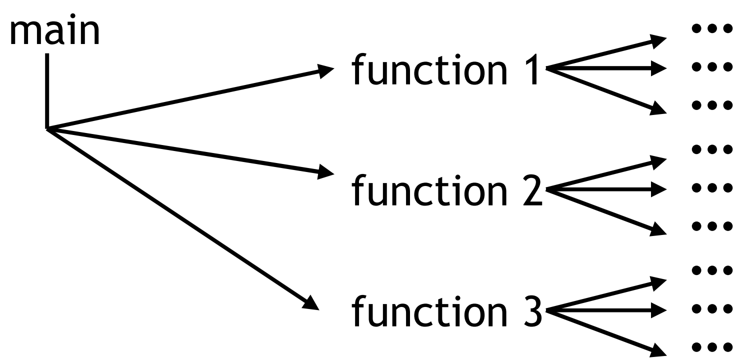
- Recursion simplifies programming by exploiting the *divide-and-conquer* method → *"divide a large problem into smaller ones"*



→ Can you rewrite the finobacci function without using recursion, and find how many lines you need for your version?

More facts about recursion

- Recursion allows users to implement their algorithms in the applicative style rather than the imperative style.



Applicative/Functional Programming

```
main {
        do this;
        do that;
        …
        do this;
        do that;
}
```

Procedural/Imperative Programming

- Recursion can be expensive if not carefully used.

→ *Compare these two functions that compute the factorial*

compute the factorial with recursion

```
int fac(int n) {
   return (n==0 ? 1 : n*fac(n-1));
}
```

compute the factorial with iteration

```
int fac2(int n) {
  int p = 1;
   for (; n > 0; n--) p = n * p;
     return p;
}
```

Comparison of `fac` and `fac2`

Computation of `fac(4)`

```
fac(4)
4 * fac(3)
4 * (3 * fac(2))
4 * (3 * (2 * fac(1)))
4 * (3 * (2 * (1 * fac(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2))
4 * 6
24
```

function call

return

five function calls

four words to store the temporal data

Computation of `fac2(4)`

```
fac2(4)
p = 1  < -------  n = 4
p = 4  < -------  n = 3
p = 12 < -------  n = 2
p = 24 < -------  n = 1
                  n = 0
```

return p

one function call

one word to store the temporal data

The main problem with the recursive version is that **`fac`** needs more memory space and function calls as the problem size **`n`** increases. In contrast, **`fac2`** always needs only 1 function call and 1 word regardless of the value of n. → *Suppose* n *is 1000!*

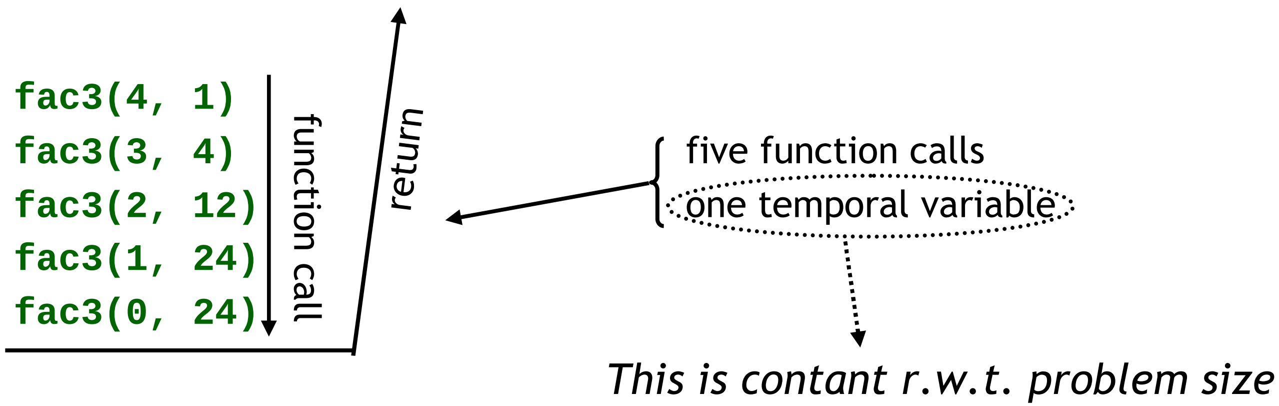
Tail recursion

- A function f is **tail-recursive** if it is a recursive function that returns either *a value without needing recursion* or *the result of a recursive activation*.

Ex:
```
void fac3(int n, int& p) { if (n > 0) { p*=n; fac3(n-1,p);} }
```
no **`int`**!

→ cf: Neither **`fib`** nor **`fac`** is tail-recursive.

- What are tail-recursive functions so great about?
  - *While still taking advantage of recursion, they can execute programs in constant space.*



Application of tail recursion

- Write a tail-recursive version of fib.

```
void fib2(int n, int& l, int& r) {
    if (n > 0) { l+=r; r=l-r; fib2(n-1,l,r); }
}
```