Control Flow

COMP 524: Programming Languages
Srinivas Krishnan
March 1, 2011

Based in part on slides and notes by Bjoern Brandenburg, S. Olivier and A. Block.

Sequential Control Flow

Determines what is computed, and in which order.

- **Imperative PL**: order mostly explicit.
- **Declarative PL**: order mostly implicit

The Basis: Conditional Branching

Virtually all instruction sets support:

- **Unconditional branching** to a fixed address.
  - e.g., `jmp 0x123`: “jump to address 0x123”

- **Unconditional branching** to an address in a register, i.e., to an address determined at runtime.
  - e.g., `jmp (%eax)`: “jump to the address in the accumulator register.”

- **Conditional branching** to a fixed address.
  - e.g., `jne 0x123`: “jump to address 0x123 if last two values that were compared were not equal”

This is sufficient to implement a universal programming language!
The Basis: Conditional Branching

Virtually all instruction sets support:
- **Unconditional branching** to a fixed address.
  - e.g., `jmp 0x123`: “jump to address 0x123”
- **Unconditional branching** to an address in a register, i.e., to an address determined at runtime.
  - e.g., `jmp (%eax)`: “jump to the address in the accumulator register.”
- **Conditional branching** to a fixed address.
  - e.g., `jne 0x123`: “jump to address 0x123 if last two values that were compared were not equal.”

**This is sufficient to implement a universal programming language!**

Sequencing

**Sequencing:** explicit control flow.
- Abstractions that control the order of execution.
- Crucial to imperative programming.

**Levels of abstraction.**
- **Unstructured** control flow
  - hardly any abstraction over jumps
  - hard to reason about
- **Structured** control flow
  - amendable to formal proofs
  - easier to understand
  - jumps are implicit

**Sequential composition:**
- `do this first ; then this` (implied)

**Unstructured:**
- e.g., `goto`, `break`, `redo`, `last`, `continue`, `continue`, and `return` if used to “skip” over rest of subroutine

**Structured:**
- e.g. `for`, `while`, and `if` (implied)

Goto Considered Harmful

Title of a famous critique of unstructured control flow by Dijkstra, 1968.

**I could reconstruct the program’s flow or use the little ‘goto’ instead.**

**Oh, screw good practice... how bad can it be? goto main, exit()**

Source: [xkcd](https://xkcd.com) (go check it out!)
Goto Considered Harmful
Title of a famous critique of unstructured control flow by Dijkstra, 1968.

Bohm & Jacopini, 1964
every use of goto can be equivalently expressed with structured control flow constructs

Bottom line: Don't ever use goto. Try hard to avoid all other unstructured control flow constructs, too.

(Footnote: some very special settings can benefit from a goto, e.g., some kernel routines. However, this does not apply to 99% of all software, in particular business & web software.)

Loops and Conditionals

Selection: execute one choice, but not the other(s).
- if-then-else
- if-then(-elsif)*-else
- switch, case statements
  - Implementation driven: exists to facilitate generation of efficient machine code.
  - This reason is somewhat obsolete with improved compilers.

Iteration: do something a pre-determined number of times.
- for (enumeration controlled)
  - from x to y; sometimes also from y downto x.
- for each (iterator)
  - executing a loop body for each element of a "collection."
  - can be emulated with iterator pattern if not supported by language
  - hasNext() and next()

Logically-controlled loops...
Logically-controlled Loops

repeat something while a condition is satisfied

```plaintext
Do
{
...
}
while i==true;
```

```plaintext
For(;;)
{
...
if i==true break;
...
}
```

```plaintext
While (i==false)
{
...
}
```

Midtest

Post-test

Pre-test

Subroutines

subprograms, functions, procedures, methods,…

Control Flow Abstraction.

- Separate "what it does" from "how it's done."
- API: subroutine as a service provider.
- Reuse high-level code.
- DRY: write it only once.
- Maintenance: fix bugs only once.
- Reuse machine code.
- Usually, only one copy of a subroutine is included in final program.

Instead of writing a concrete sequence of instructions, a subroutine is **parametrized** sequence of instructions.
Execution Context
A subroutine is executed in the context of the (virtual) machine state (global variables, device state, …). A subroutine’s result may differ between calls with the same arguments if the global context changed.

Side Effect
A program fragment that alters the global context and thus indirectly affects the outcome of otherwise unrelated computations is said to have a \textit{side effect}.

The “main effect” is the value that is computed (i.e., the return value).
Function vs. Procedure

Pure Function.
- A pure function has **no side effects**.
- A pure function's **result only depends on its arguments**, and not on any global state, not affected by side effects.
- Always the same and "leaves no trace."

Pure Procedure.
- A pure procedure **returns no value**, and is only executed for its side effects.
- Java: any method with return type **void**.

Subroutine Parameters

```plaintext
define my_subroutine(X, Y, Z) {
    ... print X;
    ...
}

define getval() {
    return 42;
}

my_var = 4;
my_subroutine(my_var, 3 + 4, getval());
```

```
4     7      42
```
The names used in the subroutine definition are called the form
dal parameters.

The program fragments used in the subroutine call are called the actual parameters.

The values resulting from the evaluation of the actual parameters are called the arguments.
Parameter Passing

Parameter Passing: Information Flow

In Parameters
Information/data provided by the caller; (possibly) consumed by the callee.

Actual parameter remains unchanged.

Out Parameters
Receiving variable provided by caller; information stored by callee.

Callee does not use prior value (if any) of receiving variable.
Parameter Passing: Information Flow

**In Parameters**
- Information/data provided by the caller;
  - possibly) consumed by the callee.
- Actual parameter remains unchanged.

**Out Parameters**
- Receiving variable provided by caller;
  - information stored by callee.
- Callee does not use prior value (if any) of receiving variable.

**In-Out Parameters**
- Information/data provided by the caller;
  - (possibly) updated by the callee.
- Any change by callee visible to caller.

Parameter Passing: Semantics

**Pass-By-Value**
- Behaves as if arguments are copied from the caller to the callee prior to the call.

**Pass-By-Reference**
- Behaves as if formal parameter is bound to the argument.
Parameter Passing: Semantics

**Pass-By-Value**
Behaves as if arguments are copied from the caller to the callee prior to the call.

**Pass-By-Reference**
Behaves as if formal parameter is bound to the argument.

**Pass-By-Name**
Behaves as if formal parameter is replaced by actual parameter in subroutine body; evaluated whenever needed.

---

**Example: Java**
Scalar types (int, double, etc.) are in parameters and passed-by-value, whereas objects are passed-by-reference.

---

**Example: C Preprocessor**
Macro parameters are passed-by-name.
Parameter Passing: Semantics

- **Pass-By-Value**
  - Behaves as if arguments are copied from the caller to the callee prior to the call.
  - Usually implemented with actual copying, but details vary.

- **Pass-By-Reference**
  - Behaves as if formal parameter is bound to the argument.

- **Pass-By-Name**
  - Behaves as if formal parameter is replaced by actual parameter in subroutine body; evaluated whenever needed.
  - Usually implemented by copying address, but sometimes more complex (e.g., Java RMI).

Parameter Evaluation Time

- **Eager Evaluation**
  - Evaluate all arguments before call.
  - Easy to implement.
  - But can be problematic.
  - What if not needed?
  - What if error might occur?

- **Normal-order evaluation**
  - Evaluate every time when argument needed.
  - But only if needed.
  - i.e., call-by-name.
  - May be not very efficient, hard to implement.

- **Lazy evaluation**
  - Actual parameter evaluated only when the argument is used.
  - Result cached.

Monday, February 28, 2011
Parameter Evaluation Time
When to evaluate actual parameters to obtain arguments.

Eager Evaluation.
- Evaluate all arguments before call.
- Easy to implement.
- But can be problematic.
  - What if not needed?
  - What if error might occur?

Normal-order evaluation.
- Evaluate every time when argument needed.
- But only if needed.
  - i.e., call-by-name.
  - May be not very efficient; hard to implement.

Mainly used in purely-functional languages: requires that time of evaluation does not impact result.

Positional Parameters
How are actual parameters and the resulting arguments matched to formal parameters?

Matched one-to-one, based on index.
- Order of formal parameters determines the order in which actual parameters must occur.
- Simple to understand and implement.
- Sometimes too inflexible or inconvenient.
  - Infrequently used options must always be specified.
  - Rigid order required; can be tedious for many parameters.

Python Function Definition
Python Shell Output:
```python
def f(a, b, c):
    print a, b, c

f(1, 2, 3)
```

```
1 2 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 3 arguments (1 given)
```
Positional Parameters

How are actual parameters and the resulting arguments matched to formal parameters?

- Matched one-to-one, based on index.
- Order of formal parameters determines the order in which actual parameters must occur.
- Simple to understand and implement.
- Sometimes too inflexible or inconvenient.

Specifying too few or too many actual parameters results in error.

Python Function Definition

```python
def f(a, b, c):
    print a, b, c
```

Python Shell Output:

```python
>>> f(1, 2, 3)
1 2 3
```

Variable Parameters

How are actual parameters and the resulting arguments matched to formal parameters?

- Matched many-to-one.
- Zero or more actual parameter correspond to one “iterable” (list-like) formal parameter. (In Python, the formal parameter is a tuple. In Java, an array.)
- Two common uses:
  - Apply some operation to any number of objects (e.g., “delete all these files”).
  - Expect certain arguments based on “configuration argument” (e.g., printf).

Python Function Definition

```python
def f(a):
    print a, a[1]
```

Python Shell Output:

```python
>>> f(1, 2, 3)
(1, 2, 3) 2
```

How are actual parameters and the resulting arguments matched to formal parameters?

- Matched many-to-one.
- Zero or more actual parameter correspond to one “iterable” (list-like) formal parameter. (In Python, the formal parameter is a tuple. In Java, an array.)
- Two common uses:
  - Apply some operation to any number of objects (e.g., “delete all these files”).
  - Expect certain arguments based on “configuration argument” (e.g., printf).
Variable Parameters
How are actual parameters and the resulting arguments matched to formal parameters?

Matched many-to-one.
- Zero or more actual parameter correspond to one “iterable” (list-like) formal parameter. (In Python, the formal parameter is a tuple. In Java, an array.)
- Two common uses:
  • Apply some operation to any number of objects (e.g., “delete all these files”).
  • Expect certain arguments based on “configuration argument” (e.g., printf).

Python Function Definition
Python Shell Output:
```python
def f(*a):
    print a, a[1]
```
```
>>> f(1, 2, 3)
(1, 2, 3, 2)
```
In Python, all arguments are available as a tuple.

The number of required parameters not fixed.

Keyword Parameters
How are actual parameters and the resulting arguments matched to formal parameters?

Matched one-to-one, either by position or keyword.
- Parameter can occur out of order.
- If default value is provided, then parameter can be omitted, too.
- Some languages (e.g., C++) allow only default values, but not keyword parameters.
- Result: can be omitted, but not provided out of order.

```python
def f(a, b):
    print a, b
```
```
>>> f(a=1, b=2)
1 2
```
The number of required parameters not fixed.
Keyword Parameters

How are actual parameters and the resulting arguments matched to formal parameters?

Matched **one-to-one**, either by position or **keyword**.

➡ Parameter can occur out of order.
➡ If default value is provided, then parameter can be omitted, too.
➡ Some languages (e.g., C++) allow only default values, but not keyword parameters.

• Result: can be omitted, but not provided out of order.

Parameters can be **provided as needed**; by naming their **keyword**, they can occur in any order.

Function can still be called with **positional** parameters.

```python
def f(a=10, b=20, c=30):
    print a, b, c
```

```python
>>> f(c=3, b=2)
10 2 3
```

```python
>>> f(1, 2, 3)
1 2 3
```
Parameter Passing: Efficiency

Compile-time.
- Parameters with default values and keyword parameters do not necessarily incur additional runtime overheads.
- Can be automatically translated to regular positional parameters.

Run-time.
- Support for variable number of parameters ("varargs") requires construction of list-like structure and iteration.
- However, the added flexibility is usually a good tradeoff.

Example: C
On x86, most positional parameters are passed through registers (fast), but varargs must be passed via the stack (slower).

However, the added flexibility is usually a good tradeoff.

Recursion

A subroutine that calls itself.
- Either directly or indirectly.
- Requires runtime stack.
Repetition without loops.
- Natural fit for "divide-and-conquer" algorithms.
  - E.g., Quicksort.
- From a math point of view:
  - recursion is natural;
  - loops can be difficult to reason about.

\[ f_{ib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
f_{ib}(n-1) + f_{ib}(n-2) & \text{otherwise}
\end{cases} \]

Naive, recursive computation of Fibonacci numbers in Prolog.

\[
\text{fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]

Definition of the Fibonacci Sequence for \( n \geq 0 \).
Recursion

A subroutine that calls itself.

Either directly or indirectly.

Requires runtime stack.

Repetition without loops.

Natural fit for "divide-and-conquer" algorithms.

E.g., Quicksort.

From a math point of view:

- recursion is natural;
- loops can be difficult to reason about.

Definition of the Fibonacci Sequence for \( n \geq 0 \).

\[
 fib(n) = \begin{cases} 
  0 & \text{if } n = 0 \\
  1 & \text{if } n = 1 \\
  fib(n - 1) + fib(n - 2) & \text{otherwise} 
\end{cases}
\]

This causes exponential runtime complexity!

Naive, recursive computation of Fibonacci numbers in Prolog.

Exponential Call Tree for naive_fib(X, 6)

Needlessly inefficient: E.g., naive_fib for \( N=2 \) is evaluated five times!
Exponential Call Tree for naive_fib(X, 6)

Result X = 8 since the non-zero base-case occurred 8 times.

Linear Recursion

% fib/2 --- compute the Nth Fibonacci number.
% Two trivial cases first.
fib(0, 0) :- !.
fib(1, 1) :- !.
% Cases that actually require iteration.
fib(X, N) :-
    N_1 is N - 1,
    N_2 is N - 2,
    fib(A, N_1),
    fib(B, N_2),
    X is A + B.

Result X = 8 since the non-zero base-case occurred 8 times.

Linear Recursion

% fib/2 --- compute the Nth Fibonacci number.
% Two trivial cases first.
fib(0, 0) :- !.
fib(1, 1) :- !.
% Cases that actually require iteration.
fib(X, N) :-
    fib(0, 1, 2, N, X).
% fib/5 --- Fibonacci helper clause; does the actual iteration.
% Base case: have reached end of iteration.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index = Stop, !,
    Res is PrevPrev + Prev.
% Recursive case: have not yet reached the desired index.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index < Stop,
    Cur is PrevPrev + Prev,
    Next is Index + 1,
    fib(Prev, Cur, Next, Stop, Res).

% fib/5 --- Fibonacci helper clause; does the actual iteration.
% Base case: have reached end of iteration.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index = Stop, !,
    Res is PrevPrev + Prev.
% Recursive case: have not yet reached the desired index.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index < Stop,
    Cur is PrevPrev + Prev,
    Next is Index + 1,
    fib(Prev, Cur, Next, Stop, Res).
Linear Recursion

% fib/2 --- compute the Nth Fibonacci number.

fib(0, 0) :- !.
fib(1, 1) :- !.

% Cases that actually require iteration.
% Base case: have reached end of iteration.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index = Stop, !,
    Res is PrevPrev + Prev.

% Recursive case: have not yet reached the desired index.
fib(PrevPrev, Prev, Index, Stop, Res) :-
    Index < Stop,
    Cur is PrevPrev + Prev,
    Next is Index + 1,
    fib(Prev, Cur, Next, Stop, Res).

Iteration ends when desired index is reached. At this point, computing the result is simple since both previous Fibonacci numbers are known.

X = Res = 8

Stack Overflow

Subroutine call requires stack space.
Stack space is a limited resource.
Problem: max recursion depth is limited by stack space if implemented naively.
Suppose Subroutine D is recursive.

The recursion will run out of space eventually.
Stack Overflow Example

```java
public static void main(String args[]) {
    System.out.println(factorial(4));
    System.out.println(factorial(100000));
}

static long factorial(long n) {
    if (n == 0) {
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
```

Output:

Exception in thread "main" java.lang.StackOverflowError
at Factorial.factorial(Factorial.java:18)
at Factorial.factorial(Factorial.java:18)
at Factorial.factorial(Factorial.java:18)
at Factorial.factorial(Factorial.java:18)
(repeated several thousand times)

So how can we implement arbitrary loops with recursion if we have only finite memory?

If a recursive call is the last statement/ expression of a subroutine to be evaluated, then the already-allocated stack frame of the caller is reused.

Stack frame = local execution context.

- If nothing remains to be executed, then stack frame contents are no longer required.
- Conceptually, instead of allocating a new stack frame, the compiler simply generates a jump to the beginning of the subroutines code.
- A bit more complicated with indirect recursion...
- Elegant recursion compiled to efficient loop.
Recursive sub-goal is not last to be evaluated!
Tail Recursion Example

Proper tail recursion: recursive sub-goal occurs last.

Inline Expansion

Subroutine granularity.
- Using many, very short subroutines is good software engineering practice.
  - Easier to understand and debug.
- However, subroutine calls incur overhead.

Inline subroutines.
- Semantically, like a normal subroutine.
  - Type checking, etc.
- However, instead of generating a call, compiler "copy&pastes" subroutine code into caller.
  - Like macro expansion.
  - Increases code size, but call overhead is avoided.

Inline Expansion Example

```c
#include <stdio.h>

int normal_function(void) { return 1; }

inline int inline_function(void) { return 2; }

int main(int argc, char** argv) {
    printf("result = %d\n", normal_function() + inline_function());
    return 0;
}
```

C99 Example.
#include <stdio.h>

int normal_function(void)
{
    return 1;
}

inline int inline_function(void)
{
    return 2;
}

int main(int argc, char **argv)
{
    printf("result = %d
", normal_function() + inline_function());
    return 0;
}

C99 Example.

Inline keyword is a hint to the compiler to include body instead of generating a call.

Generated machine code.

```
normal_function:
    push   %ebp
    mov    %esp,%ebp
    mov    $0x1,%eax
    pop    %ebp
    ret

main:
    push   %ebp
    mov    %esp,%ebp
    and    $0xffffff,%esp
    sub    $0x10,%esp
    call   normal_function
    add    $0x2,%eax
    mov    %eax,0x4(%esp)
    movl   $0x80484a0,(%esp)
    call   printf@plt
    mov    $0x0,%eax
    leave
    ret
```

Call generated for normal function.
Inline Expansion Example

The "return 2" was inlined; no call to inline_function generated.

```c
080483a4 <normal_function>:
  80483a4:       55                      push   %ebp
  80483a5:       89 e5                   mov    %esp,%ebp
  80483a7:       b8 01 00 00 00          mov    $0x1,%eax
  80483ac:       5d                      pop    %ebp
  80483ad:       c3                      ret

080483b8 <main>:
  80483b8:       55                      push   %ebp
  80483b9:       89 e5                   mov    %esp,%ebp
  80483bb:       83 e4 f0                and    $0xfffffff0,%esp
  80483be:       83 ec 10                sub    $0x10,%esp
  80483c1:       e8 de ff ff ff           call   80483a4 <normal_function>
  80483c6:       83 c0 02                add    $0x2,%eax
  80483c9:       89 44 24 04             mov    %eax,0x4(%esp)
  80483cd:       c7 04 24 a0 84 04 08    movl   $0x80484a0,(%esp)
  80483d4:       e8 ff ff ff ff           call   80482d8 <printf@plt>
  80483d9:       b8 00 00 00 00          mov    $0x0,%eax
  80483de:       c9                      leave
  80483df:       c3                      ret
```

The "return 2" was inlined; no call to inline_function generated.

Exception Handling

How to report errors?

With error or return codes.
• Commonly done in C.
• Tedious and error-prone.
  • Hard to read, complex control flow.
  • Easy to forget.

With (unstructured) jumps.
• Also error prone.
Exceptions: structured error handling.

- Checked exceptions: anticipated failures that can occur in correct program.
  • e.g., IOException: user could have specified incorrect file.
- Unchecked exceptions: errors that indicate programmer error or catastrophic system failure.
  • e.g., IllegalArgumentException: misuse of API.
  • e.g., OutOfMemoryError: program can't do anything about it.

In many languages (e.g., C++, Python…),
all exceptions are unchecked.

(checked: compiler raises error if possible exception is not handled or propagated)
Expression Evaluation

Statement vs. Expression
- Imperative languages often differentiate between "statements" and "expressions".
- Functional languages usually focus on expressions.

Expressions
- Can be evaluated to yield a value.
- E.g., in Java, "1 + 2", "Math.sqrt(2)".

Statements
- Give imperative languages sequential nature.
- E.g., in Java, "if" is a statement; it cannot occur in expressions.

Expressions usually consist of operators, operands (literals, variables, and subexpressions), and subroutine calls.

Unary, Binary, and Ternary Operators

**Unary:** Operator has **single operand**.
Example: logical negation

**Binary:** Operator has **two operands**.
Examples: logical and, addition

**Ternary:** Operator has **three operands**.
Example: ?: (conditional expression) in C-like languages
Prefix, Infix, and Postfix Operators

**Prefix:** Operator *before* Operand
Examples: `++`, `!`

**Infix:** Operator *between* Operands
Examples: `&&`, `||`, `+=`, `==`

**Postfix:** Operator *after* Operand
Examples: `++`

---

Operators as Function Applications

Operators are *not inherently special*.

An operator is a function/subroutine with human-friendly syntax.

\[
\langle \text{operand1} \rangle \ \langle \text{op} \rangle \ \langle \text{operand2} \rangle
\]

is the same as

\[
\langle \text{op} \rangle (\langle \text{operand1} \rangle, \langle \text{operand2} \rangle)
\]

Examples:

- \(3 + 4 \equiv + (3, 4)\)
- \(3 + 4 \equiv (+ 3 4)\) (LISP)
- \(x = 4 \equiv (\text{set!} \ x 4)\) (Scheme)
- \(x = 4 \equiv (\text{set!} 4)\) (Haskell)
- \(3 + 4 \equiv (\text{+} 3 4)\) (Haskell)

This is a *purely syntactic transformation* that can be done by the parser.

The semantic analysis, optimization, and code generation phases only need to implement one concept: subroutine calls.
Operators as Function Applications

Operators are not inherently special.
An operator is a function/subroutine with human-friendly syntax.

<operand1> <op> <operand2>
is the same as
<op>({operand1}, {operand2})

Compilation of operators.
- Some operators correspond directly to machine instructions
  e.g., integer addition
- These are called built-in or primitive functions.
- Which operations are primitive is entirely machine-dependent.
  e.g., some machines require software floating point emulation.
- Avoiding a subroutine call in the case of primitive functions is a compile-time optimization similar to inlining.

However, classic imperative language design treats operators as a concept that is different from a regular subroutine abstraction. This is a serious design limitation.
- e.g., in Pascal, C, and Java, operators are unrelated to functions/procedures (even if they are implemented in software) and are syntactically different.
- e.g., in C++, the user can override select operators with custom methods, but the user cannot define new operators.

Operator Precedence

<operand1> <op1> <operand2> <op2> <operand3>

Automatically transformed by parser into subroutine calls…

Monday, February 28, 2011
Operator Precedence

Problem: how to match operators to operands?

Tie breaking rules.
- Each operator is assigned a numeric precedence value.
- Operators are evaluated in order of decreasing precedence.
- Conceptually, implicit parentheses are inserted to disambiguate expression.
- E.g., multiplication usually has higher precedence than addition.

If <\text{op1}> has higher precedence than <\text{op2}>, then:

<\text{op2}>(<\text{op1}>(<\text{operand1}>, <\text{operand2}>) , <\text{operand3}>))

Monday, February 28, 2011
Operator Associativity
Consistent placement of implicit parentheses.
- Either start on left or start on right right.
  - Called left-associative and right-associative.
- Determines result if operator is not commutative.
  - Note: addition/multiplication not necessarily commutative on a computer due to overflow/underflow/loss of precision.

What if both operators have the same precedence?

Logical and: if first operand is false.
Logical or: if first operand is true.
Short-circuit: only evaluate second operand if result is required.
  i.e., use lazy evaluation!

Uses.
- This is an optimization: put the computationally cheap tests first.
- Short-circuit operators are often used to guard potentially erroneous sub-expressions.

Java
```java
HashMap dict = null;
// possibly initialized by other code
if (dict != null && dict.containsKey("key")) {
    // do something
}
```
Short-Circuit Operators

Value of expressions does not always depend on all operands.

- Logical and: if first operand is false.
- Logical or: if first operand is true.
- Short-circuit: only evaluate second operand if result is required.
  i.e., use lazy evaluation!

Uses.
- This is an optimization: put the computationally cheap tests first.
- Short-circuit operators are often used to guard potentially erroneous sub-expressions.

Java:
```java
HashMap dict = null;
// ... possibly initialized by other code
if (dict != null && dict.containsKey("key"))
  // do something;
```

Potential null dereference (dict) guarded by short-circuit operator: equivalent to call-by-name invocation of subroutine named &&.

Nested Subroutines

Subroutines definitions within subroutines.
- Subroutine definition creates a local, nested scope.
- Orthogonality: it should be possible to define new, nested subroutines in a subroutine's local scope.
- Allows decomposing large subroutines into smaller parts without "leaking" names into surrounding namespace.

History.
- Introduced in Algol 60; adopted by many modern languages (e.g., Pascal, Python, Scheme, etc.).
- Ignored by C and most descendants.
 ‣ Originally probably for ease of implementation.
 ‣ However, gcc supports nested functions as an extension.
 ‣ In Java, there isn't really a good reason not to include it...

Nested Subroutines: Example

```python
def long_running_operation(list_of_work_items):
  def progress(i):
    print("finished %d of %d" % (i, len(list_of_work_items)))
  while not done:
    # ... complicated logic ...
    progress(current_index)
    # ... more complicated logic ...
```
def long_running_operation(list_of_work_items):
    def progress(i):
        print("finished \%d of \%d\) % (i, len(list_of_work_items))
    while not done:
        # ... complicated logic ...
        progress(current_index)
        # ... more complicated logic ...

Nested subroutine: remove UI clutter from main logic.
  (especially useful if GUI code is involved)

Nesting of scopes: bindings from enclosing scope(s) visible.

Higher-Order Functions

Subroutines as arguments and return values.

A function (i.e., subroutine) that either accepts (a reference to) another
function as an argument or yields a subroutine as its return value is called a
higher-order function.

This allows users to write very flexible functions.

Caller can "customize" implemented algorithm.

def update_elements(update, array):
    for i in range(len(array)):
        array[i] = update(array[i])

def scale_by_ten(x):
    return x * 10

a = [1, 2, 3, 4, 5]
update_elements(scale_by_ten, a)
print(a)  # prints [10, 20, 30, 40, 50]
Higher-Order Functions

Subroutines as arguments and return values.

A function (i.e., subroutine) that either accepts (a reference to) another function as an argument or yields a subroutine as its return value is called a higher-order function.

This allows users to write very flexible functions.
- Caller can “customize” implemented algorithm.

A higher-order function: caller can customize the update that is applied to each element.

```python
def update_elements(update, array):
    for i in range(len(array)):
        array[i] = update(array[i])

def scale_by_ten(x):
    return x * 10

a = [1, 2, 3, 4, 5]
update_elements(scale_by_ten, a)
print(a)  # prints [10, 20, 30, 40, 50]
```

Separation of Concerns:
The loop “knows” nothing about scaling, and the scaling operation “knows” nothing about arrays.
DRY: write the loop once and reuse it with different update functions.

Example: customized sort order.

```python
def last_char(x):
    return x[-1]

strings = ['just', 'some', 'number', 'of', 'character', 'sequences']

print('by length', sorted(strings, key=len))
print('by last', sorted(strings, key=last_char))
```

Output:
- by length ['of', 'just', 'some', 'number', 'character', 'sequences']
- by last ['of', 'just', 'some', 'number', 'character', 'sequences']

Monday, February 28, 2011
Example: customized sort order.

```python
def last_char(x):
    return x[-1]
strings = ['just', 'some', 'number', 'of', 'character', 'sequences']
print 'by length', sorted(strings, key=len)
print 'by last', sorted(strings, key=last_char)
```

# Output:
# by length ['of', 'just', 'some', 'number', 'character', 'sequences']
# by last ['some', 'of', 'number', 'character', 'sequences', 'just']

Algorithmic Customization: Python allows items in a list to be sorted based upon an arbitrary key function.

Java does not support higher-order functions: The same effect is achieved by Collections.sort() by accepting a reference to a Comparator instance. (Which is significantly less elegant and natural.)
Anonymous Functions

```python
def last_char(x):
    return x[-1]

strings = ['just', 'some', 'number', 'of', 'character', 'sequences']
print by length, sorted(strings, key=len)
print by last, sorted(strings, key=last_char)
```

### Definition of short "use once" functions.
- Defining a function and coming up with a good name for each "customization" can be tedious.
- Thus, it may be convenient to use unnamed functions.
- I.e., instead of defining a function and then referring to it, anonymous functions allow us to simply write a function literal.
- Due to their theoretical roots, these are often called lambda expressions.

Unnecessarily verbose: the definition “boilerplate code” is much longer than the actual logic, the function is only used once.

Better alternative: use an anonymous function (indicated by lambda keyword) to achieve same effect.

---

**Anonymous Functions**

```python
strings = ['just', 'some', 'number', 'of', 'character', 'sequences']
print by length, sorted(strings, key=len)
print by last, sorted(strings, key=last_char)
```

**Definition of short "use once" functions.**
- Defining a function and coming up with a good name for each "customization" can be tedious.
- Thus, it may be convenient to use unnamed functions.
- I.e., instead of defining a function and then referring to it, anonymous functions allow us to simply write a function literal.
- Due to their theoretical roots, these are often called lambda expressions.
Closures

Nested subroutines that "capture" their referencing environment.

Free variables.
- In a subroutine $F$, a variable that is neither a formal parameter of $F$ nor a local variable is called a free variable.
- What happens if $F$ is a nested subroutine and returned by the subroutine in which it was defined?
  - Or if it is otherwise passed to code that may call it after the subroutine call in which it was defined terminated.

```c
int foo(int x)
{
    int y = 0;
    return x + y + z;
}
```

Closure.
- A subroutine that is "closed over" its free variables.
- Meaning: the free variables stay bound to whatever they became bound at definition time (see lexical scoping) and remain valid.
- This requires all entities that are referenced by closures to be allocated on the heap, since they may have to "outlive" the call in which the closure was created.
- Hence, closures are usually found in garbage-collected languages.

Note: closures and anonymous functions are not the same concept!
- Closures do not have to be anonymous.
- Anonymous functions do not necessarily have free variables.
def make_stack():
mystack = []
def _push(x):
mystack.append(x)
def _pop():
val = mystack[-1]
del mystack[-1]
return val
return (_push, _pop)

(a, b) = make_stack()
(c, d) = make_stack()

(a(1); a(2); a(3))
(c(9); c(8); c(7))

print 'b:', b(), b(), b()
print 'd:', d(), d(), d()
def make_stack():
    mystack = []
    def _push(x):
        mystack.append(x)
    def _pop():
        val = mystack[-1]
        del mystack[-1]
        return val
    return (_push, _pop)

(a, b) = make_stack()
(c, d) = make_stack()
a(1); a(2); a(3)
c(7); c(8); c(9)
print 'b:', b(), b(), b()
print 'd:', d(), d(), d()

Output:
Python:
b: 3 2 1
d: 7 8 9

The name mystack is a free variable; thus _push() and _pop() are closures.

Creates hidden state that is neither global nor local nor class-based. Can be used to implement object systems.

Java does not support closures: Again, it uses (inelegant) class-based workarounds, known as "object-closures". This design choice is limiting. For example, the Swing GUI API would be a lot easier to use if Java had anonymous functions and closures; the need for "Listener" interfaces would be greatly reduced.
What happens if you supply “too few” actual parameters to a function?

- Normally, this is an error.
- Partial application allows the programmer to specialize functions by “fixing” some of the parameters.
- This is similar to a closure in that some parameters become “hidden.”

**Python:**
```python
from functools import partial
def scale_by(factor, x):
    return factor * x

scale_by_ten = partial(scale_by, 10)
scale_by_20 = partial(scale_by, 20)
print(scale_by_ten(1), scale_by_20(1))
# prints 10 20
```

**Haskell:**
```haskell
scale_by requires two parameters…

-- by partially applying one parameter, a new function is created that only requires a single parameter.
main = do
    let f = plus 10
    print f 20
    -- prints 30
```

**Explanation:**
Partial application allows the programmer to specialize functions by “fixing” some of the parameters. This is similar to a closure in that some parameters become “hidden.” It allows for the creation of specialized versions without duplicating the implemented logic (DRY, good for maintenance).
What happens if you supply “too few” actual parameters to a function?

- Normally, this is an error.
- Partial application allows the programmer to specialize functions by “fixing” some of the parameters.
- This is similar to a closure in that some parameters become “hidden.”

### Python

```python
from functools import partial
def scale_by(factor, x):
    return factor * x

scale_by_ten = partial(scale_by, 10)
scale_by_20 = partial(scale_by, 20)

print(scale_by_ten(1), scale_by_20(1))
# prints 10 20
```

### Haskell

```
Given only one parameter, Haskell automatically creates a function that maps one integer to an integer.
```

In the context of mathematics and functional programming, partial application is commonly called currying in honor of the logician Haskell Curry (1900–1982).
Continuations

Simplified: snapshot of execution state.
- Stack + registers (incl. instruction pointer).
- Execution can be resumed (continue) form snapshot at later point in time.
- Very powerful abstraction.
  - e.g., can be used to implement exception handling.

Adoption.
- Not widespread.
- Scheme is the most-prominent example.
  - Well worth studying over the summer...
  - Challenging to implement without extensive runtime system.

Co-Routines

Concurrent execution of subroutines.
- Execution of several subroutines is interleaved.
- Not by OS (e.g., processes), but by compiler / runtime system.

Uses.
- Emulate concurrency on a uniprocessor.
  - Less overhead than actual multithreading.
- Process simulation (SIMULA 67).
- Discrete-event simulation.

Adoption.
- Not supported by most main-stream programming languages.
  - Can be emulated in C with libraries (and inline assembly code).
  - Relevance likely reduced on multicore systems.