Programming Paradigms

Control Flow (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2022

Control Flow

Control flow: Ordering of instructions

- Fundamental to most models of computation
- Common language mechanisms
  - Sequencing, selection, iteration, recursion, concurrency, exceptions
- Each PL defines its rules
  - Think in terms of concepts, not specific syntax

Quiz: Argument Evaluation

What does the following Java code print?

```
class ArgEval {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }

    public static void main(String[] args) {
        int i = 7;
        f(i++, --i);
    }
}
```

Quiz: Argument Evaluation

What does the following Java code print?

```
class ArgEval {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }

    public static void main(String[] args) {
        int i = 7;
        f(i++, --i);
    }
}
```

Result: 7, 7

Quiz: Argument Evaluation

What does the following Java code print?

```
class ArgEval {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }

    public static void main(String[] args) {
        int i = 7;
        f(i++, --i);
    }
}
```

Post-increment: Returns `i` and then increments it

Result: 7, 7

Quiz: Argument Evaluation

What does the following Java code print?

```
class ArgEval {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }

    public static void main(String[] args) {
        int i = 7;
        f(i++, --i);
    }
}
```

Pre-decrement: Decrements `i` and then returns it

Result: 7, 7

Quiz: Argument Evaluation

What does the following Java code print?

```
class ArgEval {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }

    public static void main(String[] args) {
        int i = 7;
        f(i++, --i);
    }
}
```

Evaluation order:
Left-to-right

Result: 7, 7

Overview

- Expression Evaluation ←
- Structured and Unstructured Control Flow
- Selection
- Iteration
- Recursion

Expressions

Operator vs. operand

- Operator: Built-in function with a simple syntax
- Operand: Arguments of operator
- Examples:

```
i++

foo() + 23

(a * b) / c
```

Expressions: Notation

Three popular notations

- Prefix
  - □ `op a b` or `op(a, b)` or `(op a b)`
- Infix
  - □ `a op b`
- Postfix
  - □ `a b op`

Expressions: Notation

Three popular notations

- Prefix
  - `op a b` or `op(a, b)` or `(op a b)`
- Infix
  - `a op b`
- Postfix
  - `a b op`

Example: Lisp

```
(* (+ 1 3) 2)
```

Expressions: Notation

Three popular notations

- Prefix
  - `op a b` or `op(a, b)` or `(op a b)`
- Infix
  - `a op b` —— **Example: Java**

    `(1 + 3) * 2`
- Postfix
  - `a b op`

Expressions: Notation

Three popular notations

- Prefix
  - `op a b` or `op(a, b)` or `(op a b)`
- Infix
  - `a op b`
- Postfix
  - `a b op`

Example: C

`a++`

Multiplicity

Number of arguments expected by an operator

- Unary
  - `a++` or `!cond`
- Binary
  - `a + b` or `x instanceof MyClass`
- Ternary
  - `cond ?  a :  b`
- (More are possible, but uncommon in practice)

Order of Evaluating Expressions

Given a complex expression, in what order to evaluate it?

Examples:

- Multiple arithmetic operations in Python:

  ```
  2 + 3 * 4
  ```

- Mix of boolean and other expressions in Java:

  ```
  !x && a == false
  ```

- Dereference and increment a pointer in C:

  ```
  *p++
  ```

Precedence and Associativity

Choice among evaluation orders:

Specified by precedence and associativity rules of the PL

- Precedence: Specify which operators group “more tighly” than others
- Associativity: For operators of equal precedence, specify whether to group to the left or right

Precedence Levels in C

Operator	Meaning
++, --	Post-increment, post-decrement
++, -- *	Pre-increment, pre-decrement Pointer dereference
<, >	Inequality test
==, !=	Equality test
&&	Logical and
||	Logical or
=, +=	Assignment

Higher means higher precedence

This list is incomplete.

Precedence Levels in C

Operator	Meaning
++, --	Post-increment, post-decrement
++, --	Pre-increment, pre-decrement
*	Pointer dereference
<, >	Inequality test
==, !=	Equality test
&&	Logical and
||	Logical or
=, +=	Assignment

Same precedence level

This list is incomplete.

Examples

- **Dereference and increment a pointer:**
  - □ `*p++`
- **Mix of logical operators:**
  - □ `a && b || c`
- **Mix of inequality and equality tests:**
  - □ `x < y == foo`

Examples

- **Dereference and increment a pointer:**
  - □ `*p++` means `*(p++)`
- **Mix of logical operators:**
  - □ `a && b || c` means `(a && b) || c`
- **Mix of inequality and equality tests:**
  - □ `x < y == foo` means `(x < y) == foo`

Examples

- **Dereference and increment a pointer:**
  - □ `*p++` means `*(p++)`
- **Mix of logical operators:**
  - □ `a && b || c` means `(a && b) || c`
- **Mix of inequality and equality tests:**
  - □ `x < y == foo` means `(x < y) == foo`

General rule:

When in doubt, use parentheses

Associativity Rules

- Decide about same-level operators
- Arithmetic operators:

  Mostly left-to-right a.k.a. left-associative

  - `12 - 3 - 2` yields `7` in most languages
  - Exception: Exponentiation is mostly right-associative
    - `2 ** 3 ** 2` yields `512` in most languages
    - But: `2 ^^ 3 ^^ 2` yields `64` in Excel
- Assignments: Mostly right-associative
  - `a = b = a + c` assigns `a + c` into `b` and then `a`

Quiz: Precedence and Associativity

1) What are the values of `foo` and `bar`

(a) when assignments are left-associative?

(b) when assignments are right-associative?

```
int foo = 2, bar = 3;
foo = bar = foo + bar;
```

2) What is the value of `z`

(a) when `&&` has higher prededence than `||`?

(b) when `||` has higher prededence than `&&`?

```
bool x = false, y = false, z = true;
bool z = x || y && y || z;
```

Quiz: Precedence and Associativity

1) What are the values of `foo` and `bar`

(a) when **assignments are left-associative**?

foo=3, bar=6

(b) when **assignments are right-associative**?

foo=5, bar=5

```
int foo = 2, bar = 3;
foo = bar = foo + bar;
```

2) What is the value of `z`

(a) when `&&` has **higher prededence** than `||`?

true

(b) when `||` has **higher prededence** than `&&`?

false

```
bool x = false, y = false, z = true;
bool z = x || y && y || z;
```

Ordering within Expressions

- **Discussed so far:**
  Order of performing operations
- **But: In what order are the operands evaluated?**
- **Example:**
  `a - f(b) - c * d`

Ordering within Expressions

- **Discussed so far:**

  Order of performing operations

- **But: In what order are the operands evaluated?**

- **Example:**

```
a - f(b) - c * d
```

Has precedence over subtraction

Ordering within Expressions

- **Discussed so far:**
  Order of performing operations

- **But: In what order are the operands evaluated?**

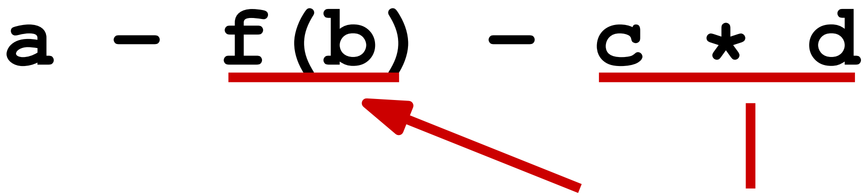
- **Example:**

$$\boxed{\texttt{a - f(b)}}\ \texttt{- c * d}$$

Subtraction is left-associative:
This is computed first

Ordering within Expressions

- Discussed so far:
  Order of performing operations
- But: In what order are the operands evaluated?
- Example:



```
a - f(b) - c * d
```

But: Which of these two operands is evaluated first?

Why Does It Matter?

- **Reason 1: Side effects**
  - Evaluating `f(b)` may modify `c` or `d`
- **Reason 2: Compiler optimizations**
  - Influences register allocation and instruction scheduling

Example:

```
a - f(b) - c * d
```

Ordering: Language-specific

Different PLs: Different ordering within expressions

- Java and C#: Left-to-right
- C and many other languages: Undefined
  - Compiler can choose best order
  - Earlier example again:

    ```
    int i = 7;
    f(i++, --i);
    ```

Ordering: Language-specific

Different PLs: Different ordering within expressions

- Java and C#: Left-to-right
- C and many other languages: Undefined
  - Compiler can choose best order
  - Earlier example again:

```
int i = 7;
f(i++, --i);
```

May pass 7, 7 (left-to-right) or 6, 6 (right-to-left) to `f`

Short-circuit Evaluation

- **Saving time when evaluating boolean expressions**
- **Example:**

```
if (very_unlikely && very_expensive())
{
    ...
}
```

Short-circuit Evaluation

- Saving time when evaluating boolean expressions
- Example:

```
if (very_unlikely && very_expensive())
{
    ...
}
```

If first operand is false, no need to evaluate the second

Short-circuit Evaluation

- **Saving time** when evaluating **boolean expressions**
- **Example:**

```
if (very_unlikely && very_expensive())
{
      ...
}
```

But: Side effects of second operand may or may not happen

Short-circuit Evaluation (2)

- **Most PLs implement short-circuit evaluation**
  - Boolean and: Ignore second operand if first is false
  - Boolean or: Ignore second operand if first is true
- **One (relatively) popular exception: Pascal**

Short-circuit Evaluation (3)

- Beware that side effects in some boolean expressions may not happen
- Use it to your advantage:

```
// C code
p = my_list;
while (p && p->key != val) {
     ...
     p = p ->next;
}
```

Overview

- Expression Evaluation
- Structured and Unstructured Control Flow
- Selection
- Iteration
- Recursion

Control Flow with `gotos`

- **Most assembly languages: Control flow via conditional and unconditional jumps**
- **Early PLs: `goto` statements**
  - □ Jump to a statement label
  - □ Target label can be anywhere in the code

Example

```
// C code
int a = 10;
my_label: do {
    if(a == 12) {
        a = a + 1;
        goto my_label;
    }
    printf("%d\n", a);
    a++;
} while(a < 15);
```

Example

```c
// C code
int a = 10;
my_label: do {
  if(a == 12) {
    a = a + 1;
    goto my_label;
  }
  printf("%d\n", a);
  a++;
} while(a < 15);
```

Output:

```
10
11
13
14
```

Quiz: Goto Hell

```c
// C code
int result = 0;
int bound = 3;
here : for (int i = 0; i < bound; ++i)
{
there:
    result += i;
    goto elsewhere;
}
goto here;
elsewhere : if (result < 2)
{
    goto there;
}
printf("%d\n", result);
```

What does this code print?

Quiz: Goto Hell

```
// C code
int result = 0;
int bound = 3;
here : for (int i = 0; i < bound; ++i)
{
there:
    result += i;
    goto elsewhere;
}
goto here;
elsewhere : if (result < 2)
{
    goto there;
}
printf("%d\n", result);
```

What does this code print?

Nothing! It never terminates.

Beyond `gotos`

- ***Go To Statement Considered Harmful*** **article by Edsger Dijkstra (CACM, 1968)**
- **Instead: Structured control flow**
- **Express algorithms with**
  - Sequencing
  - Selection
  - Iteration

Avoiding `goto`s

Use case of `goto`

- Jump to end of subroutine
- Escape from middle of loop
- Propagate to surrounding context

Structured control flow alternative

- `return` statement
- `break` and `continue` statements
- Exceptions

Continuations

- **Generalization of `gotos`**
- **Powerful language feature:**
  Allows programmer to define new control flow constructs
  - □ Exceptions
  - □ Iterators
  - □ Coroutines
  - □ etc.

Continuations (2)

- **High-level definition: Context in which to continue execution**
- **Low-level definition: Three parts**
  - □ Code address (where to continue)
  - □ Referencing environment (for resolving names)
  - □ Another continuation (to use when code returns)

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end


v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
```

Creates a continuation, i.e., execution will continue here

```
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
```

`d` is a reference to the continuation

```
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end


v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

`foo` gets called and calls itself two more times

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end

v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Jumps into context captured by `c` and makes `callcc` appear to return `i`

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
```

Code prints:

start 1; start 2; start 3; got 3

```
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

`bar` gets called and calls itself two more times

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

Creates a continuation, which gets stored in `c`

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

`n` is 2, therefore execution jumps to the continuation

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

We are here again!

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

← **We are here again!**

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

`n` is 1, therefore execution jumps to the continuation

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

n is 0. We are finally done

Another Example

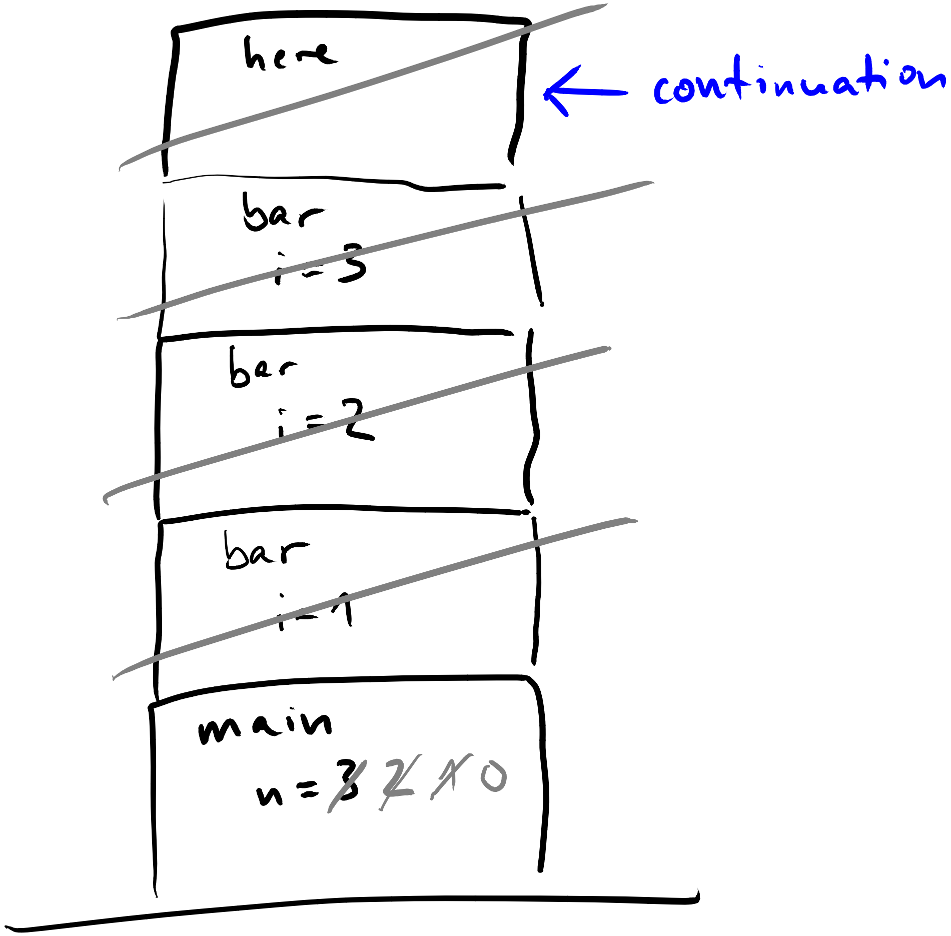
```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

Code prints:

```
start 1; start 2; start 3; end 3; end 2; end 1;
end 3; end 2; end 1;
end 3; end 2; end 1;
done
```



```
start 1
start 2
start 3
end 3
end 2
end 1
end 3
end 2
end 1
end 3
end 2
end 1
done
```