Programming Paradigms

Control Flow (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart
Summer 2020
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 Warmup {
    static void f(int a, int b) {
        System.out.println(a +", " + b);
    }

    public static void main(String[] args) {
        int i = 5;
        f(i++, --i);
    }
}
Quiz: Argument Evaluation

What does the following Java code print?

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

    public static void main(String[] args) {
        int i = 5;
        f(i++, --i);
    }
}

Result: 5, 5

Please vote via Ilias
Quiz: Argument Evaluation

What does the following Java code print?

class Warmup {
    static void f(int a, int b) {
        System.out.println(a + ", " + b);
    }
    public static void main(String[] args) {
        int i = 5;
        f(i++, --i);
    }
}

Result: 5, 5

Post-increment: Returns i and then increments it

Please vote via Ilias
Quiz: Argument Evaluation

What does the following Java code print?

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

    public static void main(String[] args) {
        int i = 5;
        f(i++, --i);
    }
}

Result: 5, 5

Please vote via Ilias
Quiz: Argument Evaluation

What does the following Java code print?

class Warmup {
    static void f(int a, int b) {
        System.out.println(a + " , " + b);
    }
    
    public static void main(String[] args) {
        int i = 5;
        f(i++, --i); // Evaluation order: Left-to-right
    }
}

Result: 5, 5
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

□ \( \text{op a b} \) or \( \text{op(a, b)} \) or \( (\text{op a b}) \)

■ Infix

□ \( a \ \text{op} \ b \)

■ Postfix

□ \( a \ b \ \text{op} \)
Expressions: Notation

Three popular notations

- **Prefix**
  - $\text{op } a \ b$  or  $\text{op}(a, \ b)$  or  $(\text{op } a \ b)$

- **Infix**
  - $a \ \text{op} \ b$

- **Postfix**
  - $a \ b \ \text{op}$

Example: Lisp

$(\ast \ (+ \ 1 \ 3) \ 2)$
Expressions: Notation

Three popular notations

- **Prefix**
  - \[ \text{op} \ a \ b \text{ or } \text{op}(a, b) \text{ or } (\text{op} \ a \ b) \]

- **Infix**
  - \[ a \ \text{ op } \ b \] **Example: Java**

- **Postfix**
  - \[ a \ b \ \text{ op} \]

Example: Java

\((1 + 3) * 2\)
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:
  
  \[2 + 3 \times 4\]

- 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 tightly” than others

- **Associativity**: For operators of equal precedence, specify whether to group to the left or right
# Precedence Levels in C

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>Post-increment, post-decrement</td>
</tr>
<tr>
<td>++, --</td>
<td>Pre-increment, pre-decrement</td>
</tr>
<tr>
<td>*</td>
<td>Pointer dereference</td>
</tr>
<tr>
<td>&lt;, &gt;</td>
<td>Inequality test</td>
</tr>
<tr>
<td>==, !=</td>
<td>Equality test</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=</td>
<td>Assignment</td>
</tr>
</tbody>
</table>

This list is incomplete.

Higher means higher precedence.
## Precedence Levels in C

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>Post-increment, post-decrement</td>
</tr>
<tr>
<td>++, --</td>
<td>Pre-increment, pre-decrement</td>
</tr>
<tr>
<td>*</td>
<td>Pointer dereference</td>
</tr>
<tr>
<td>&lt;, &gt;</td>
<td>Inequality test</td>
</tr>
<tr>
<td>==, !=</td>
<td>Equality test</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=</td>
<td>Assignment</td>
</tr>
</tbody>
</table>

This list is incomplete.
Examples

- Dereference and increment a pointer:
  - \( \ast p++ \)

- Mix of logical operators:
  - \( a \&\& b \mid\mid c \)

- Mix of inequality and equality tests:
  - \( x < y == \text{foo} \)
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:
  \[ *p++ \text{ means } *(p++), \]

- Mix of logical operators:
  \[ a && b || c \text{ means } (a && b) || c, \]

- Mix of inequality and equality tests:
  \[ x < y == foo \text{ means } (x < y) == foo, \]

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
      - $2 \text{ } ^\wedge \wedge \text{ } 3 \text{ } ^\wedge \wedge \text{ } 2$ yields $512$ in most languages
      - But: $2 \text{ } ^\wedge \wedge \text{ } 3 \text{ } ^\wedge \wedge \text{ } 2$ 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 \texttt{foo} and \texttt{bar}
   (a) when assignments are left-associative?
   (b) when assignments are right-associative?

   \begin{verbatim}
   int foo = 1, bar = 2;
   foo = bar = foo + bar;
   \end{verbatim}

2) What is the value of \texttt{z}
   (a) when \&\& has higher precedence than || ?
   (b) when || has higher precedence than \&\& ?

   \begin{verbatim}
   bool x = false, y = false, z = true;
   bool z = x || y && y || z;
   \end{verbatim}
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 = 1, bar = 2;
   foo = bar = foo + bar;
   ```

2) What is the value of z
   (a) when && has higher precedence than ||?
   (b) when || has higher precedence than &&?

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

true
false
foo=2, bar=4
foo=3, bar=3
foo=2, bar=4
foo=3, bar=3
Please vote in Ilias
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:
  \[ a - f(b) - c \times d \]

Has precedence over subtraction
Ordering within Expressions

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

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:
  \[ a - f(b) - c \times d \]
  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:

$$a - f(b) - c \times d$$
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 = 5;
    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:
    ```java
    int i = 5;
    f(i++, --i);
    ```
    May pass 5, 5 (left-to-right) or 4, 4 (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:**

```c
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:**

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

But: Side effects of second operand may or may not happen
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
// C code
p = my_list;
while (p && p->key != val) {
    ...
    p = p ->next;
}
```