Wake-up Exercise

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);
    }
}

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

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

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

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 \texttt{i} and then increments it

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

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
Wake-up Exercise

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
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
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:

```plaintext
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

■ Postfix
  □ a b op

Example: Java

(1 + 3) * 2
Expressions: Notation

Three popular notations

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

- **Infix**
  - \( a \text{ op } b \)

- **Postfix**
  - \( a \ b \text{ op} \)

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:
  \[\ast p++\]
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>

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:
  - *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:
  - \( a \&\& b \mid\mid c \) means \( (a \&\& b) \mid\mid c \)

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

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
    - $2 \times^2 3 \times^2 2$ 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 = 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;
   ```

https://ilias3.uni-stuttgart.de/vote/0ZT9
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 \texttt{&&} has higher precedence than \texttt{||} ? \\
(b) when \texttt{||} has higher precedence than \texttt{&&}? \\
\begin{verbatim}
bool x = false, y = false, z = true;
bool z = x || y && y || z;
\end{verbatim}
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:

\[ a - f(b) - c * d \]

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 * 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:
    ```
    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:**

```c
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:**

```java
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
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;
}
```
Overview

- Expression Evaluation
- Structured and Unstructured Control Flow
- Selection
- Iteration
- Recursion
Control Flow with \texttt{gotos}

- Most assembly languages: Control flow via conditional and unconditional jumps
- Early PLs: \texttt{goto} statements
  - Jump to a statement label
  - Target label can be anywhere in the code
// 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);
```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

What does this code print?

```c
// C code
int result = 0;
int number = 3;
one : for (int i = 0; i < number; ++i)
{
three:
    result += i;
    goto two;
}
goto one;
two : if (result < 2)
{
goto three;
}
printf("%d\n", result);
```
Quiz: Goto Hell

// C code
int result = 0;
int number = 3;
one : for (int i = 0; i < number; ++i)
{
    three:
        result += i;
        goto two;
}
goto one;
two : if (result < 2)
{
    goto three;
}
printf("%d\n", result);

What does this code print?

Nothing! It never terminates.
Beyond *goto*s

- *Go To Statement Considered Harmful* article by Edsger Dijkstra (CACM, 1968)

- Instead: **Structured control flow**

- Express algorithms with
  - Sequencing
  - Selection
  - Iteration
Avoiding \texttt{goto}s

Use case of \texttt{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 $\text{goto}$s
- 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
# 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
# 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
```

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

# Ruby code
```ruby
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
```
d is a reference to the continuation
# 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
# 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

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

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

Code prints:
start 1; start 2; start 3; got 3
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

```ruby
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
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, when gets stored in C
Another Example

```ruby
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

```ruby
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
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
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
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

Code prints:
start 1; start 2; start 3; end 3; end 2; end 1; end 3; end 2; end 1;

n = n - 1
c = bar(1)

if n > 0 then c.call(c) end

puts "done"
Overview

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