Programming Paradigms

Lecture 7:
Control Flow (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Wake-up Exercise

What does this Java function return?

```java
boolean callMe() {
    try {
        return true;
    } finally {
        return false;
    }
}
```

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

What does this Java function return?

```java
boolean callMe() {
    try {
        return true;
    } finally {
        return false;
    }
}
```

Result: false

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

What does this Java function return?

```java
boolean callMe() {
    try {
        return true;
    } finally {
        return false;
    }
}
```

Result: false

Finally blocks are always executed, even after a return

But: Avoid returning in a finally block
Overview

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

- **Branch** that depends on a **condition**
- Different syntactic variants
  - If-else statements (sometimes with else-if)
  - Case/switch statements
If Statements

Syntactic variants across PLs

**Algol 60 and its descendants:**

```
if (A == B) then ...
else if (A == C) then ...
else ...
```

**Lisp and its descendants:**

```
(cond
  ((= A B) (...))
  ((= A C) (...))
  (T (...))
)
```

**Bash**

```
If [ $A = $B ]
then ...
elif [ $A = $C ]
then ...
else ...
fi
```
Compilation of If Statements

if ((A > B) and (C > D)) or (E ≠ F) then
  then-clause
else
  else-clause
• Short-circuited evaluation
• Fall-through to same cases

\[
\begin{align*}
  r1 &:= A \\
  r2 &:= B \\
  r1 &:= C \\
  r2 &:= D \\
  \text{if } r1 < r2 \text{ goto L4} \\
  L4 : & r1 := E \\
  & r2 := E \\
  \text{if } r1 > r2 \text{ goto L1} \\
  L1 : & \text{ then-clause} \\
  & \text{ goto L3} \\
  L2 : & \text{ else-clause} \\
  L3 : & \text{...}
\end{align*}
\]
Case/Switch Statements

Many conditions that compare the same expression to different compile-time constants

-- Ada syntax
case ... -- potentially complicated expression
if
  when 1 => clause_A
  when 2 | 7 => clause_B
  when 3..5 => clause_C
  when 10 => clause_D
  when others => clause_E
end case;
Case/Switch Statements

Many conditions that compare the **same** expression to different compile-time constants

--- Ada syntax

```ada
case ... -- potentially complicated expression
if
  when 1 => clause_A
  when 2 | 7 => clause_B
  when 3..5 => clause_C
  when 10 => clause_D
  when others => clause_E
end case;
```

Labels

Arms
Compilation of Case/ Switch Statements

r1 := ...

( calculate controlling expression )

if r1 ≠ 1 goto L1
clause_A
goto L6

L1: if r1 = 2 goto L2
    if r1 ≠ 7 goto L3

L2: clause_B
    goto L6

L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    clause_C
    goto L6

L4: if r1 ≠ 10 goto L5
    clause_D

LS: clause_E
L6:
Jump Table-based Compilation

T:
&L1 (expression = 1)
&L2
&L3
&L3
&L3
&L5
&L2
&L5
&L5
&L4

L6: r1 := ... (evaluate expression)
  if r1 < 1 goto L5
  if r1 > 10 goto L5
  r1 := r1 - 1
  r1 := T[r1]
goto * r1

advantage:
constant-time jump to right arm
Variations Across PLs

- Case/switch varies across PLs
  - What values are allowed in labels?
  - Are ranges allowed?
  - Do you need a default arm?
  - What happens if the value does not match?
Fall-Through Case/Switch

C/C++/Java

- Each expression needs its own label (no ranges)
- Control flow “falls through”, unless stopped by `break` statement

```c
switch ( /* expression */ ) {
    case 1: clause_A
        break;
    case 2:
    case 7: clause_B
        break;
    case 3:
    case 4:
    case 5: clause_C
        break;
    case 10: clause_D
        break;
    default: clause_E
        break;
}
```
Quiz: Switch/Case

What does the following C++ code print?

```cpp
int x = 3;
switch (x)
{
    case 1: { x += x; }
    case 3: { x += x; }
    case 5: { x += x; }
    default: { x += 5; }
}
std::cout << x;
```

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Switch/Case

What does the following C++ code print?

```cpp
int x = 3;
switch (x)
{
    case 1: { x += x; }
    case 3: { x += x; }
    case 5: { x += x; }
    default: { x += 5; }
}
std::cout << x;
```

Result: 17

Each of these is executed (because no `break` statement)

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

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

- Essential language construct
  - Otherwise: Amount of work done is linear to program size

- Two basic forms of loops
  - Enumeration-controlled:
    Once per value in finite set
  - Logically controlled:
    Until Boolean expression is false
Enumeration-controlled Loops

- Most simple form: Triple of
  - Initial value
  - Bound
  - Step size

Fortran 90:

```
  do i = 1, 10, 2
    ...
  enddo;
```

Modula-2:

```
  FOR i := 1 TO 10 BY 2 DO
    ...
  END
```
Enumeration-controlled Loops

- **Most simple form: Triple of**
  - Initial value
  - Bound
  - Step size

**Fortran 90:**

```fortran
do i = 1, 10, 2
  ...
enddo;
```

**Modula-2:**

```modula
FOR i := 1 TO 10 BY 2 DO
  ...
END
```

**Iterations with i = 1, 3, 5, 7, 9**
Semantic Variants

Different PLs offer different variants

- Can you leave the loop in the middle?
- Can you modify the loop variable?
- Can you modify the values used to compute the loop bounds?
- Can you read the loop variable in/after the loop?
Iterators

- Special enumeration-controlled loop: **Iterates through any kind of set/sequence of values**
  - E.g., nodes of a tree or elements of a collection
- Decouples two algorithms
  - How to enumerate the values
  - How to use the values
- Three flavors
  - “True” iterators, iterator objects, first-class functions
“True” Iterators

- Subroutine with `yield` statements
  - Each `yield` “returns” another element
- Popular, e.g., in Python, Ruby, and C#
- Used in a `for` loop
  - Example (Python):
    ```python
    # range is a built-in iterator
    for i in range(first, last, step):
        ...
    ```
class BinTree:
    def __init__(self, data):
        self.data = data
        self.lchild = self.rchild = None

    # other methods: insert, delete, lookup, ...

def preorder(self):
    if self.data is not None:
        yield self.data
    if self.lchild is not None:
        for d in self.lchild.preorder():
            yield d
    if self.rchild is not None:
        for d in self.rchild.preorder():
            yield d
Iterator Objects

■ Regular object with methods for
  □ Initialization
  □ Generation of next value
  □ Test for completion

■ Popular, e.g., in Java and C++

■ Used in for loop

for (Iterator i = c.iterator(); i.hasNext(); ) {
    ... = i.next();
}

Iterator Objects

- Regular object with **methods** for
  - Initialization
  - Generation of next value
  - Test for completion
- Popular, e.g., in Java and C++
- Used in **for** loop

```java
for (Iterator i = c.iterator(); i.hasNext(); ) {
  ... = i.next();
}
```
Iterator Objects

- Regular object with methods for
  - Initialization
  - Generation of next value
  - Test for completion
- Popular, e.g., in Java and C++
- Used in for loop

```java
for (Iterator i = c.iterator(); i.hasNext(); ) {
    ... = i.next();
}
```

```java
for (Element e : c) {
    ...
}
```

Since Java 5
Example: Binary Tree

class BinTree<T> implements Iterable<T> {
    BinTree<T> left; BinTree<T> right; T val;

    // other methods: insert, delete, lookup

    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }

    private class TreeIterator implements Iterator<T> {
        public boolean hasNext() {
            ... // check if there is another element
        }

        public T next() {
            ... // return the next element
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
Iterating with First-Class Functions

- **Two functions**
  - One function about **what to do for each element**
  - Another function that **calls** the first function for each element

- **Example (Scheme):**

```scheme
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
      (begin
        (f low)
        (uptoby (+ low step) high step f))
      '()))
```

```scheme
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
      (begin
        (f low)
        (uptoby (+ low step) high step f))
      '()))
```
Iterating with First-Class Functions

- **Two functions**
  - One function about **what to do for each element**
  - Another function that **calls** the first function **for each element**

- **Example (Scheme):**

```scheme
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
        (begin
          (f low)
          (uptoby (+ low step) high step f))
        '())))
```

```scheme
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
        (begin
          (f low)
          (uptoby (+ low step) high step f))
        '())))
```
Iterating with First-Class Functions

- Two functions
  - One function about what to do for each element
  - Another function that calls the first function for each element

- Example (Scheme):

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
      (begin
        (f low)
        (uptoby (+ low step) high step f))
      '()))
```

Defines a function with four arguments
Iterating with First-Class Functions

- Two functions
  - One function about **what to do for each element**
  - Another function that **calls** the first function for each element

- Example (Scheme):

  ```scheme
  (define uptoby
    (lambda (low high step f)
      (if (<= low high)
        (begin
          (f low)
          (uptoby (+ low step) high step f))
        '()))
  )
  ```

  Calls $f$ with the next element
Iterating with First-Class Functions

- Two functions
  - One function about what to do for each element
  - Another function that calls the first function for each element

- Example (Scheme):
  (define uptoby
    (lambda (low high step f)
      (if (<= low high)
        (begin
          (f low)
          (uptoby (+ low step) high step f))
        '())))

Recursively calls `uptoby` to handle the remaining elements.
Iterating with First-Class Functions (2)

- Originally, proposed in functional languages
- Nowadays, available in many modern PLs through libraries
  - E.g., Java
    
    ```java
    mySet.stream().filter(e -> e.someProp > 5)
    ```
  - E.g., JavaScript
    
    ```javascript
    myArray.filter(e => e.someProp > 5)
    ```
Iterating with First-Class Functions (2)

- Originally, proposed in **functional languages**
- Nowadays, **available** in many modern PLs through **libraries**
  - E.g., Java
    ```java
    mySet.stream().filter(e -> e.someProp > 5)
    ```
    Iterates through all elements and returns a filtered subset
  - E.g., JavaScript
    ```javascript
    myArray.filter(e => e.someProp > 5)
    ```
Iterating with First-Class Functions (2)

- Originally, proposed in functional languages
- Nowadays, available in many modern PLs through libraries
  - E.g., Java
    ```java
    mySet.stream().filter(e -> e.someProp > 5)
    ```
  - E.g., JavaScript
    ```javascript
    myArray.filter(e => e.someProp > 5)
    ```

Boolean function that decides which elements to keep
Logically Controlled Loops

Whether to **continue to iterate** decided through a **Boolean expression**

- **Pre-test:** \texttt{while} (\texttt{cond}) {
  
  \hspace{1cm} \ldots

}

- **Mid-test:** \texttt{for} (;;) {
  
  \hspace{1cm} \ldots

  \hspace{1cm} \texttt{if} (\texttt{cond}) \texttt{break}

}

- **Post-test:** \texttt{do} {
  
  \hspace{1cm} \ldots

  \hspace{1cm} \texttt{while} (\texttt{cond})
Quiz: Iteration

Which of the following statements is true?

- Iterators are a form of logically controlled loops.
- A “true” iterator yields one element each time it is called.
- Iterator objects have a method that yields another element each time it is called.
- Iterating with first-class functions does not require a for-loop.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Iteration

Which of the following statements is true?

- Iterators are a form of logically controlled loops.
- A “true” iterator yields one element each time it is called.
- Iterator objects have a method that yields another element each time it is called.
- Iterating with first-class functions does not require a for-loop.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

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

- Equally powerful as iteration
- Most PLs allow both recursion and iteration
  - **Iteration**: More natural in imperative PLs (because the loop body typically updates variables)
  - **Recursion**: More natural in functional PLs (because the recursive function typically doesn’t update any non-local variables)
Efficiency

Naively written or naively compiled recursive functions: **Less efficient** than equivalent iterative code

- **Reason:** New *allocation frame* for each call
- **Example:** Compute \( \sum_{\text{low} \leq i \leq \text{high}} f(i) \) in Scheme

```scheme
(define sum
  (lambda (f low high)
    (if (= low high)
        (f low)
        (+ (f low) (sum f (+ low 1) high))))
```
Efficiency

Naively written or naively compiled recursive functions: **Less efficient** than equivalent iterative code

- **Reason:** New allocation frame for each call
- **Example:** Compute \( \sum_{\text{low} \leq i \leq \text{high}} f(i) \) in Scheme

```
(define sum
  (lambda (f low high)
    (if (= low high)
      (f low)
      (+ (f low) (sum f (+ low 1) high))))
```

Then and else branches
Efficiency

Naively written or naively compiled recursive functions: **Less efficient** than equivalent iterative code

- **Reason:** New allocation frame for each call
- **Example:** Compute $\sum_{low \leq i \leq high} f(i)$ in Scheme

```
(define sum
  (lambda (f low high)
    (if (= low high)
        (f low)
        (+ (f low) (sum f (+ low 1) high))))
```

Recursive call
Tail Recursion

Recursive call is the last statement before the function returns

- Compiled code can reuse same allocation frame
- Revised example:

```scheme
(define sum
  (lambda (f low high subtotal)
    (if (= low high)
        (+ subtotal (f low))
        (sum f (+ low 1) high (+ subtotal (f low))))))
```
Example: Summation

\[(\text{sum } f \ 2 \ 4)\]

Naïve implementation

\[
\text{sum} \\
f = f \\
low = 4 \\
high = 4
\]

\[
\text{sum} \\
f = f \\
low = 3 \\
high = 4
\]

\[
\text{sum} \\
f = f \\
low = 2 \\
high = 4
\]

→ 3 allocation frames

Tail-recursive implementation

\[(\text{sum } f \ 2 \ 4 \ 0)\]

→ reusing a single allocation frame

\[
\text{sum} \\
f = f \\
low = 2 \\
high = 4 \\
s_{new}=0
\]
Overview

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