301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-15: Recursion, tail-recursion, type inference
static int indexOf(char[] source, int sourceOffset, int sourceCount,
    char[] target, int targetOffset, int targetCount,
    int fromIndex) {

    char first = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        /* Look for first character. */
        if (source[i] != first) {
            while (++i <= max && source[i] != first);
        }

        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
            int j = i + 1;
            int end = j + targetCount - 1;
            for (int k = targetOffset + 1; j < end && source[j] ==
                target[k]; j++, k++);

            if (j == end) {
                /* Found whole string. */
                return i - sourceOffset;
            }
        }
    }
    return -1;
}
Searching a Substring: 
Exploiting Laziness

```haskell
isPrefixOf :: Eq a => [a] -> [a] -> Bool
-- returns True if first list is prefix of the second
isPrefixOf [] x = True
isPrefixOf (y:ys) [] = False
isPrefixOf (y:ys) (x:xs) = 
  if (x == y) then isPrefixOf ys xs else False

suffixes:: String -> [String]
-- All suffixes of s
suffixes[] = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs

or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or [] = False
or (b:bs) = b || or bs

isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t |
  t <- suffixes s ]
```
From loops to recursion

• In functional programming, **for** and **while** loops are replaced by using **recursion**

• **Recursion**: subroutines call themselves directly or indirectly (mutual recursion)

• **Iteration** and **recursion** are equally powerful in theoretical sense: Iteration can be expressed by recursion and vice versa

• Recursion is the natural solution when the solution of a problem is defined in terms of simpler versions of the same problem, as for **tree traversal**
On efficiency

• In general a procedure call is much more expensive than a conditional branch.

• Thus recursion is in general less efficient, but good compilers for functional languages can perform good code optimization.

• Use of higher-order functions (combinators), like `map`, `reduce` (`foldl`, `foldr`), `filter`, `foreach`,... strongly encouraged, because they are highly optimized by the compiler.
Tail-Recursive Functions

- **Tail-recursive functions** are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

  ```
  tail-recursive                not tail-recursive
  int trfun()                  int rfun()
  { ...                     { ...
  return trfun();          return 1+rfun();
  }
  ```

- A tail-recursive call could *reuse* the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed
  - Simply eliminating the push (and pop) of the next frame will do

- In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function
Tail-Recursion Optimization

• Consider the GCD function:

```c
int gcd(int a, int b)
{ if (a==b) return a;
  else if (a>b) return gcd(a-b, b);
  else return gcd(a, b-a);
}
```

• a good compiler will optimize the function into:

```c
int gcd(int a, int b)
{ start:
  if (a==b) return a;
  else if (a>b) { a = a-b; goto start; }
  else { b = b-a; goto start; }
}
```

• which is just as efficient as the iterative version:

```c
int gcd(int a, int b)
{ while (a!=b)
    if (a>b) a = a-b;
    else b = b-a;
  return a;
}
```
Tail-Call Optimization

- **Tail-call**: a function returns calling another function, not necessarily itself
- Optimization still possible, reusing the stack frame
- Note: Number/size of parameters can differ
Converting Recursive Functions to Tail-Recursive Functions

• Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call.
• For example, the non-tail-recursive function computing $\sum_{n=low}^{high} f(n)$

\[
\text{summation} = (f, low, high) \rightarrow \\
\text{if } (low == high) \text{ then } (f low) \\
\text{else } (f low) + \text{summation} (f, low + 1, high)
\]

can be rewritten into a tail-recursive function:

\[
\text{summationTR} = (f, low, high, subtotal) \rightarrow \\
\text{if } (low == high) \\
\text{then } \text{subtotal} + (f low) \\
\text{else } \text{summationTR} (f, low + 1, high, \text{subtotal} + (f low))
\]
Converting recursion into tail recursion in C: Example

- Here is the same example in C:

```c
typedef int (*int_func)(int);

int summation(int_func f, int low, int high)
{ if (low == high)
    return f(low)
  else
    return f(low) + summation(f, low+1, high);
}
```

- rewritten into the tail-recursive form:

```c
int summationTR(int_func f, int low, int high, int subtotal)
{ if (low == high) —
    return subtotal+f(low)
  else
    return summationTR(f, low+1, high, subtotal+f(low));
}
```
Converting recursion into tail recursion: Fibonacci

• The Fibonacci function implemented as a recursive function is very inefficient as it takes exponential time to compute:

```haskell
fib = \n -> if  n == 0 then  1
    else if n == 1 then 1
    else fib (n - 1) + fib (n - 2)
```

• with a tail-recursive helper function, we can run it in $O(n)$ time:

```haskell
fibTR = \n -> let fibhelper (f1, f2, i) =
          if (n == i) then f2
                        else fibhelper (f2, f1 + f2, i + 1)
    in fibhelper(0,1,0)
```
Lambdas in Java 8
Java 8: language extensions

Java 8 is the biggest change to Java since the inception of the language. Main new features:

• Lambda expressions
  – Method references
  – Default methods in interfaces
  – Improved type inference

• Stream API

A big challenge was to introduce lambdas without requiring recompilation of existing binaries
Benefits of Lambdas in Java 8

• Enabling functional programming
  – Being able to pass behaviors as well as data to functions
  – Introduction of lazy evaluation with stream processing

• Writing cleaner and more compact code

• Facilitating parallel programming

• Developing more generic, flexible and reusable APIs
Lambda expression syntax:
Print a list of integers with a lambda

List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));

• $x \rightarrow \text{System.out.println}(x)$
is a lambda expression that defines an anonymous function (method) with one parameter named $x$ of type Integer

// equivalent syntax
intSeq.forEach((Integer x) -> System.out.println(x));

intSeq.forEach(x -> {System.out.println(x);});

intSeq.forEach(System.out::println); //method reference

• Type of parameter inferred by the compiler if missing
List<Integer> intSeq = Arrays.asList(1,2,3);
// multiline: curly brackets necessary
intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
// local variable declaration
intSeq.forEach(x -> {
    int y = x + 2;
    System.out.println(y);
});
// variable capture
[final] int y = 2; // must be [effectively] final
intSeq.forEach(x -> {
    System.out.println(x + y);
});
// no new scope!!!
int x = 0;
intSeq.forEach(x -> { //error: x already defined
    System.out.println(x + 2);
});
Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function, compiling its code
- Then it generates code to call the compiled function where needed
- For example, \( x \rightarrow \text{System.out.println}(x) \) could be converted into a generated static function
  ```java
  public static void genName(Integer x) {
    System.out.println(x);
  }
  ```
- But what type should be generated for this function? How should it be called? What class should it go in?
Functional Interfaces

• Design decision: Java 8 lambdas are instances of functional interfaces.

• A functional interface is a Java interface with exactly one abstract method. E.g.,

```java
public interface Comparator<T> { //java.util
    int compare(T o1, T o2);
}

public interface Runnable { //java.lang
    void run();
}

public interface Callable<V> { //java.util.concurrent
    V call() throws Exception;
}
```
Functional interfaces and lambdas

• Functional Interfaces can be used as target type of lambda expressions, i.e.
  – As type of variable to which the lambda is assigned
  – As type of formal parameter to which the lambda is passed
• The compiler uses type inference based on target type
• Arguments and result types of the lambda must match those of the unique abstract method of the functional interface
• Lambdas can be interpreted as instances of anonymous inner classes implementing the functional interface
• The lambda is invoked by calling the only abstract method of the functional interface
public class Calculator1 {  // Pre Java 8
    interface IntegerMath {  // (inner) functional interface
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }  // parameter type is functional interface
    // inner class implementing the interface
    static class IntMath$Add implements IntegerMath{
        public int operation(int a, int b){
            return a + b;
        }
    }
    public static void main(String... args) {
        Calculator1 myApp = new Calculator1();
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, new IntMath$Add()));
        // anonymous inner class implementing the interface
        IntegerMath subtraction = new IntegerMath(){
            public int operation(int a, int b){
                return a - b;
            }
        };
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
public class Calculator {

    interface IntegerMath { // (inner) functional interface
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    } // parameter type is functional interface

    public static void main(String... args) {
        Calculator myApp = new Calculator();
        // lambda assigned to functional interface variables
        IntegerMath addition = (a, b) -> a + b;
        System.out.println("40 + 2 = " +
                myApp.operateBinary(40, 2, addition));
        // lambda passed to functional interface formal parameter
        System.out.println("20 - 10 = " +
                myApp.operateBinary(20, 10, (a, b) -> a - b));
    }
}
Other examples of lambdas: Runnable

```java
public class ThreadTest { // using functional interface Runnable
    public static void main(String[] args) {
        Runnable r1 = new Runnable() { // anonymous inner class
            @Override
            public void run() {
                System.out.println("Old Java Way");
            }
        };

        Runnable r2 = () -> { System.out.println("New Java Way"); };

        new Thread(r1).start();
        new Thread(r2).start();
    }
}
```

// constructor of class Thread

```java
public Thread(Runnable target)
```
JButton button = new JButton("Click Me!");

// pre Java 8
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Handled by anonymous class listener");
    }
});

// Java 8
button.addActionListener(
    e -> System.out.println("Handled by Lambda listener"));
New Functional Interfaces in package java.util.function

```java
public interface Consumer<T> {
    void accept(T t);
}

public interface Supplier<T> {
    T get();
}

public interface Predicate<T> {
    boolean test(T t);
}

public interface Function<T, R> {
    R apply(T t);
}
```
Other examples of lambdas

```java
List<Integer> intSeq = new ArrayList<>(Arrays.asList(1, 2, 3));

// sort list in descending order using Comparator<Integer>
intSeq.sort((x, z) -> z - x);  // lambda with two arguments
intSeq.forEach(System.out::println);

// remove odd numbers using a Predicate<Integer>
intSeq.removeIf(x -> x % 2 == 1);
intSeq.forEach(System.out::println);  // prints only '2'
```

// default method of Interface List<E>
default void sort(Comparator<? super E> c)
// default method of Interface Collection<E>
default boolean removeIf(Predicate<? super E> filter)
// default method of Interface Iterable<T>
default void forEach(Consumer<? super T> action)
```
Default Methods

Problem: Adding new abstract methods to an interface breaks existing implementations of the interface

Java 8 allows interface to include

• Abstract (instance) methods, as usual
• Static methods
• Default methods, defined in terms of other possibly abstract methods

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve backward compatibility with existing published interfaces
## Method References

- Method references can be used to pass an existing function in places where a lambda is expected.
- The signature of the referenced method needs to match the signature of the functional interface method.

<table>
<thead>
<tr>
<th>Method Reference Type</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>ClassName::StaticMethodName</td>
<td>String::valueOf</td>
</tr>
<tr>
<td>constructor</td>
<td>ClassName::new</td>
<td>ArrayList::new</td>
</tr>
<tr>
<td>specific object instance</td>
<td>objectReference::MethodName</td>
<td>x::toString</td>
</tr>
<tr>
<td>arbitrary object of a given type</td>
<td>ClassName::InstanceMethodName</td>
<td>Object::toString</td>
</tr>
</tbody>
</table>