Section 7

Advanced abstractions

1. Preliminaries

2. Introduction

3. Values and types

4. Advanced typing

5. Storage

6. Commands

7. Advanced abstractions
   7.1 Strategies of evaluation
   7.2 From nested functions to closures
   7.3 Emulating objects with closures and vice versa
   7.4 Beyond closures
   7.5 Generators with JAVA iterators
What’s in this chapter?

- Collateral evaluation and intentionally undefined behavior
- Parameter passing modes
- Nested functions
- Closures
- Generators
- Coroutines
- Continuations.
7. Advanced abstractions

7.1. Strategies of evaluation

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage
6. Commands

7. Advanced abstractions
7.1 Strategies of evaluation
7.1.1 Motivation
7.1.2 Collateral evaluation
7.1.3 Short-circuit evaluation
7.1.4 Normal evaluation order
7.1.5 Argument passing modes
7.1.6 Call by need
7.1.7 Summary
7.2 From nested functions to closures
7.3 Emulating objects with closures and vice versa
7.4 Beyond closures
7.5 Generators with JAVA iterators
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.1. Motivation
Thinking of expressions as trees

fun min(a, b) = if a < b then a else b;

fun max(a, b) = a + b - min(a, b);

fun min3(a, b, c) = min(a, min(b, c));

fun max3(a, b, c) = max(a, max(b, c));

In the professional jargon, these trees are called \(\lambda\)-expressions.
More on the tree perspective

fun median(a, b, c) =
(a + (b + c)) -
(min3(a, b, c) + max3(a, b, c));

Leaves  Arguments, literals, references, ...

Inner nodes  Locations of evaluation
  • 1 / 2 / more children
  • functions / operators
  • builtin / defined-by-programmer
Applying a function to arguments

**Function:**

```
    -
   /  \
  +    +
 /     /
a +    min3
     /   /
b     a   b     c
     /   /
     a   b   c
```

**Application**

```
median( 0xBee, 0xCafe, 0xBed )
```

**Evaluation:**

- using each argument 3 times.
- invoking 4 more functions: `min`, `max`, `min3`, `max3`
  - some more than once.
- invoking 4 builtin operators: `if · then · else ·`, `<`, `+` and `-`
  - some more than once
- using and passing the original arguments many more times
What do we mean by “evaluation strategies”?  

Issues of evaluation, e.g.,  
- Order of operations  
- Argument passing  
- Caching  

Two approaches:  
- \(\lambda\)-calculus  
  - formal and precise  
  - but,  
    - not so intuitive  
    - heavy on notation  

  some other time...  
- Algorithmic  
  - intuitive  
  - minimal notation  
  - but, not so formal and precise  
  our approach!
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.2. Collateral evaluation
Evaluation order (revisiting expression & commands)

Is this program “correct”?

```c
#include <stdio.h>

int main() {
    return printf("Hello,\n") - printf("World!\n");
}
```

- Function `printf` returns the number of characters it prints.
  - `sizeof "Hello,\n" == 7`
  - `sizeof "World!\n" == 7`

- So, `main` returns 0, interpreted as normal termination
Expressions + side-effects = evaluation order dilemma

In evaluating, e.g., $X + Y$, the PL can decide which of $X$ and $Y$ is evaluated first, but, often PLs prefer to refrain from making a decision.

**Definition (Collateral evaluation)**

Let $X$ and $Y$ be two code fragments (expressions or commands). Then, all of the following are correct implementations of collateral execution of $X$ and $Y$

- $X$ is executed before $Y$
- $X$ is executed after $Y$
- “interleaved” execution
- simultaneous execution
What will be printed?

So, in compiling and executing

```c
#include <stdio.h>

int main() {
    return printf("Hello,\n") - printf("World!\n");
}
```

there is no telling what will be printed!
Consequence of collateral semantics

Consider an application of function:

```
- (+ (+ a b) min3 (+ b c)) max3
```

to arguments, e.g., `median( 0xBeef, 0xCafe, 0xBed )`, and using auxiliary functions:

```
if · then · else ·
< a b
```

Collateral semantics

*The PL does not place its own restrictions on the evaluation order.*
Intentionally undefined behavior in PLs

PL designers do not specify everything

- If certain patterns are “bad” programming practice...
- If many “legitimate” implementation make sense...
- If different compilers / different architectures may take a performance toll from over-specification...

Then, the PL designer will tend to consider specifying “undefined behavior”.

```c
messy() {
    int i, n, a[100];
    for (i = 0; i < n; i++)
        printf("a[%d]=%d\n", i, a[i]);
}
```

**C** Specifying that `auto` variables are zero-initialized may cause an unnecessary performance overhead.

**Java** Advances in compiler technology make it possible for the compiler to produce an error message if an uninitialized variable is used.
Side-effects $\iff$ evaluation order question

- If expressions have side-effects then there is clearly an evaluation order question.

```c
printf("Hello, \n") - printf("World! \n");
```

- Question may pop up even if there are no side effects, e.g., the evaluation of the following pure-mathematical expression over $\mathbb{R}$

$$\arcsin 2 + \frac{\sqrt{-1}}{0} \times \lg_2 0$$

whose evaluation tree is

Depending on the evaluation order, each of the red nodes may trigger a runtime error first.
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.3. Short-circuit evaluation
More on expressions’ evaluation order

Example: Rotate 13 Algorithm

Star Wars Episode V: The Empire Strikes Back (1980)

Spoiler
Qnegu Inqre vf Yhxr Fxljnyxre’f snzure.

Spoiler… Revealed!
Darth Vader is Luke Skywalker’s father.

“ROT13” encryption: add 13 to-, or subtract 13 from- all letters.

Algorithm for a “rot13” filter

1. Define a conversion table, indexed by characters.
2. Fill the conversion table for lower and upper case letters.
3. Read the input: replacing each letter with its conversion table entry.
Rotate 13 implementation

Is there a bug here?

```pascal
PROGRAM Rotate13(Input, Output);
VAR
  (* Define a translation table, indexed by characters. *)
  rot13: Array['A'..'z'] of Char;
  c: char;
  (* Fill the translation table for lower and upper case letters: *)
  Procedure fill; Begin
    For c := 'A' to 'z' do rot13[c] := chr(0);
    For c := 'A' to 'M' do rot13[c] := chr(ord(c)+13);
    For c := 'N' to 'Z' do rot13[c] := chr(ord(c)-13);
    For c := 'a' to 'm' do rot13[c] := chr(ord(c)+13);
    For c := 'n' to 'z' do rot13[c] := chr(ord(c)-13);
  end;
```

Rotate 13 implementation (cont.)

Or maybe here?

Encoding/decoding

(* Replacing letters with their translation table entry: *)

Procedure Convert(Var c: Char); Begin
    If c >= 'A'
        and
        c <= 'z'
        and
        rot13[c] <> chr(0)
    then
        c := rot13[c]
    end;
Rotate 13 implementation (main program)

Or maybe a bug in here?

```pascal
PROGRAM Rotate13(Input, Output);
...
Begin
Fill;
While not eof do begin
Read(c);
Convert(c);
Write(c)
end
end.
```
Here is the bug!

In Pascal, operator **and** evaluates all of its arguments (even if it becomes apparent that these are immaterial for the results)

```pascal
Procedure Convert(Var c: Char); Begin
  If c >= 'A'
       and
       c <= 'Z'
       and
       rot13[c] <> chr(0) (* X * )
  then
       c := rot13[c]
  end;
```

Another annoying (and typical) case...

```pascal
If p <> null and p^.next <> null then ...
```
Eager/strict/applicative evaluation

Definition (Eager (strict) evaluation order)

An **eager evaluation order also called strict evaluation order** specifies that all arguments to functions are evaluated before the procedure is applied.

- Also called **applicative order**
- Eager order does not specify in which order the arguments are computed; it can be
  - unspecified (collateral)
  - left to right
  - right to left
  - ...
- Most PLs use eager evaluation for
  - all functions,
  - the majority of operators
Eager vs. short-circuit logical operators

**Definition (Short-circuit evaluation)**

Short-circuit evaluation \( \land \) and \( \lor \) prescribes that the 2\(^{nd} \) argument is evaluated only if the 1\(^{st} \) argument does not determine the result

\( \land \) **Logical “and”** Evaluate the second argument only if the first argument is true

\( \lor \) **Logical “or”** Evaluate the second argument only if the first argument is false

<table>
<thead>
<tr>
<th>PL</th>
<th>Eager version</th>
<th>Short-circuit version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pascal</strong></td>
<td>and, or</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML</td>
<td></td>
<td>andalso, orelse</td>
</tr>
<tr>
<td><strong>Eiffel, Ada</strong></td>
<td>and, or</td>
<td>and then, or else</td>
</tr>
</tbody>
</table>
Comparing eager and short-circuit semantics

- Same result, if there are no errors
- Same computation, if there are no side-effects
Programming idioms with short-circuit semantics

Beautiful programming idiom (originated by Perl, but applicable e.g., in C)

```c
f = fopen(fileName, "r")
|| die("Cannot open file %s\n", fileName);
```

And another one:

```c
f == (FILE *)0 || ((void)fclose(f), f = (FILE *)0);
```

More tools of the trade

Java/C/C++  Operator “?”…“:” (just like if (evaluate only the branch that is required))

C/C++  Operator “,”

Similar to Prolog, except that unlike Prolog, each expression here yields only one result.
Clever short circuit evaluation in **Bash**

A Bash program to remove contents of current directory.

```bash
for f in *; do
    echo -n "\$f: \\
    [ -e \$f ] || echo "already\_removed"
    [ -d \$f ] &&
    echo -n "removing\_directory" \\
    rmdir \$f && ( \\
        [ -e \$f ] && echo "...failed"
        || echo ""
    )

    [ -e \$f ] &&
    echo -n "moving\_to\_/tmp" \\
    mv -f \$f /tmp && ( \\
        [ -e \$f ] && echo "...failed"
        || echo ""
    )

done
```

- **Bash commands may succeed** or fail:
  - Success returns `true`
    - (integer 0)
  - Failure returns `false`
    - (error code ≠ 0)

- "[]" is a command; it takes arguments; it may succeed or fail:
  - "[ -e \$f ]" 3 arguments to command "[]"; succeeds if file \$f exists
  - "[ -d \$f ]" 3 arguments to command "[]"; succeeds if directory \$f exists
Emulating short-circuit operators with conditional commands

*poor substitute, operators occur in expressions, and expressions are not commands*

Logical “And”

```pascal
If p^. <> null then
  If p^.next <> null
  (* some command *)
```

Logical “Or”

```pascal
If p^. = null then
  (* command, possibly SKIP *)
else if p^.next <> null then
  (* some command *)
```
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.4. Normal evaluation order
Normal evaluation order

**Definition (Normal evaluation order)**

In normal evaluation order arguments to functions are evaluated whenever they are used by the function.

- Is a generalization of “short circuit evaluation”
- The terms
  - “normal evaluation order” and
  - “normal order evaluation”
  are synonymous.
- Can be used to encapsulate any of the following C operators in functions:
  - `& &`
  - `| |`
  - `,`
  - `? :`
Normal vs. applicative order:

**Normal order** a “symbolic expression” (“symbolic expression” is an imprecise name for λ-expression) is passed; this expression is evaluated by the callee when it needs it.

**Applicative order** a “symbolic expression” is evaluated by the caller, the result is passed to the callee.
Capitalizing on normal order evaluation

A variant of the “die” programming idiom:

**Clever “unless” function**

```java
static <T> // exists for each type T
T unless(
    boolean b, // !(precondition)
    normal T t, // value
    normal Exception e // in case precondition fails
) {
    if (b)
        throw e; // Do not evaluate t
    return t; // Do not evaluate e
}
```

- Parameters marked **normal** are evaluated (only) when used.
- If precondition **fails**, then exception **e** is evaluated and then thrown.
- If precondition **holds**, then argument **t** is evaluated and then returned.
Function `unless` in pseudo-JAVA

Defining:

```java
static <T> T unless(boolean b,
    normal T t, normal Exception e) {
    if (b) throw e;
    /* else */
    return t;
}
```

Using:

```java
// Obtain an arbitrary integer:
final Integer a = readInteger();
// Obtain another arbitrary integer:
final Integer b = readInteger();
// Divide the first by the second (if possible):
final Integer c = unless(b == 0,
    // always evaluated
    a / b,
    // evaluated only if b != 0
    new ArithmeticException(
        // evaluated only if b == 0
        "Dividing value '' + a + "' by ' + 0!"
    )
);```
New control structures with normal order evaluation

Defining:

Return the “last” value of an expression

```pseudo-C
int lastValue(
    normal int e, // Expression to evaluate
    normal bool c // until this condition holds
) {
    // Evaluate expression once:
    const int $ = e;
    // If condition fails, recurse:
    return c ? $ : lastValue(e, c);
}
```

Using:

GCD by repeated execution of expression with side effects

```pseudo-C
int gcd(int m, int n) {
    int s;
    return lastValue(
        (s=m+n, m=min(n,m), n=s-m, n%==m, m), n==0
    );
}
```
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.5. Argument passing modes
Call by value vs. call by reference

- In applicative order, \( v \) is evaluated and passed to \( f \)
- But, what if \( v \) is a variable? (If \( v \) is an expression that has no L-Value, then the call \( f(v) \) must be “by value”)

**Call by reference**
- Pass the L-Value of \( v \).
- Follow reference semantics (??)
- Function \( f \):
  - can assign to \( v \)
  - can use the value of \( v \)

**Call by value**
- Pass the R-Value of \( v \), i.e., a copy of \( v \).
- Follow value semantics (??).
- Function \( f \):
  - can not assign to \( v \)
  - can use the value of \( v \)
Visualizing call by value

```java
void main() {
    int m = 54;
    int n = 24;
    gcd(m, n);
}
```

```java
int gcd(int m, int n) {
    if (m == 0) return n;
    if (m > n) return gcd(n, m);
    n %= m;
    return gcd(n, m);
}
```

Call chain:

```
main: →
gcd₁(54, 24) →
gcd₂(24, 54) →
gcd₃(6, 24) →
gcd₄(0, 6) = 6
```

Each activation record of `gcd` has its own copy of `m` and `n`
Visualizing call by reference

As before

```java
void main() {
    int m = 54;
    int n = 24;
    gcd(m, n);
}
```

(Almost) as before

```java
int gcd(var int m, var int n) {
    if (m == 0) return n;
    if (m > n) return gcd(n, m);
    n %= m;
    return gcd(n, m);
}
```

Call chain:

```
main: →
gcd₁(54, 24) →
gcd₂(24, 54) →
gcd₃(6, 24) →
gcd₄(0, 6) = 6
```

Each activation record of `gcd` is equipped with references to the original variables `m` and `n`
Call by reference in a recursive greatest common divisor

```java
void main() {
    int m = 54;
    int n = 24;
    gcd(m, n);
}

int gcd(var int m, var int n) {
    if (m == 0) return n;
    if (m > n) return gcd(n, m);
    n %= m;
    return gcd(n, m);
}
```

<table>
<thead>
<tr>
<th>Call</th>
<th>Action</th>
<th>Main</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>int m = 54; int n = 24;</td>
<td>54</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m_main</td>
</tr>
<tr>
<td>gcd₁(54, 24)</td>
<td></td>
<td>54</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m_main</td>
</tr>
<tr>
<td>gcd₂(24, 54)</td>
<td></td>
<td>54</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n_main</td>
</tr>
<tr>
<td>gcd₃(6, 24)</td>
<td>n %= m;</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m_main</td>
</tr>
<tr>
<td>gcd₄(0, 6)</td>
<td>= 6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n_main</td>
</tr>
</tbody>
</table>
Other argument passing modes:

Modes of applicative order:

- **Call by value** a copy of the evaluated argument is passed.
- **Call by reference** a reference to the evaluated argument is passed.
- **Call by result** no value is passed to callee, on return, value is copied back.

**Call by value-result** call by value plus call by result.

We also talk about “call by name”, which is a nothing but “normal order”

- **Call by name** the unevaluated expression is passed as argument, and is computed by the callee.
Implementation of argument passing modes

Implementation of call by ...

- **value**  before call, push value onto stack
- **reference**  before call, push address onto stack
- **result**  after call, pop value from stack
- **value-result**
  - before call, push value onto stack
  - after call, pop value from stack
- **name**  with “thunks”...
Implementation of “call by name”

Think of automatic conversion of

```java
Integer f(Integer a, b) {
    return unless(  // Recall that unless is normal-order in its 2\textsuperscript{nd} and 3\textsuperscript{rd}
        b == 0,  // Convert into a thunk:
        a / b,  // Convert into a thunk:
        new ArithmeticException("Dividing value' + a + "' by 0!"))
    ;
}
```

into

```java
Integer f(Integer a, b) {
    Integer _1() { return a / b; }  // 1\textsuperscript{st} thunk implementing 1\textsuperscript{st} argument
    Exception _2() {  // 2\textsuperscript{nd} thunk, implementing 2\textsuperscript{nd} argument
        return new ArithmeticException("Dividing value' + a + "' by 0!");
    }
    return unless(b == 0, _1, _2);
}
```
Normal order arguments can be replaced by nested functions

**Definition (Thunk)**

A thunk is a compiler generated nested function that implements a “normal” order argument

- Thunks are only passed as arguments.
- Thunks are never returned.
- Therefore, inner functions as in Pascal suffice.
- Below we shall see how nested functions are implemented.
- To implement “call by reference by name”, the thunk simulating normal order arguments, must be able to return references to variables.
**Thunks in the lastValue example**

**Before conversion to thunks**

```pseudo-C
int gcd(int m, int n) {
    int s;
    return lastValue(
        (s=m+n, m=min(n,m), n=s-m, n%=m, m), n==0
    );
}
```

**After conversion to thunks**

```pseudo-C
int gcd(int m, int n) {
    int s;
    int _1() { // 1st thunk implementing 1st argument
        s = m + n; m=min(n,m); n=s-m; n %= m;
        return m;
    }
    int _2() { // 2nd thunk implementing 2nd argument
        return n == 0;
    }
    return lastValue(_1, _2);
}
```
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.6. Call by need
Lazy evaluation order = memoization + call-by-name

Definition (Lazy evaluation order)

In lazy evaluation order arguments to functions are evaluated at most once, at the first time they are used by the function.

Definition (Call by need)

Call by need is an argument passing mode, in which the arguments are evaluated lazily.

- The term “memoization” means “caching function results”.
- Used in Haskell (the main feature which distinguishes it from ML)
- Only makes sense in PLs in which there is no “program state”
Memoization

**Definition (Memoization)**

*memoization* is

- *a compiler optimization technique*
- *part of the semantics of certain PLs*

*by which results of function applications for certain combination of the arguments are cached.*

<table>
<thead>
<tr>
<th>Call</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;Hello&quot;)</code></td>
<td>✗ side effects</td>
</tr>
<tr>
<td><code>time()</code></td>
<td>✗ access global state</td>
</tr>
<tr>
<td><code>eof()</code></td>
<td>✗ access global state</td>
</tr>
<tr>
<td><code>sin(12)</code></td>
<td>✓ No side effects, no access to global state</td>
</tr>
<tr>
<td><code>strlen(&quot;Hello, &quot;)</code></td>
<td>✓ No side effects, no access to global state</td>
</tr>
<tr>
<td><code>strcpy(s,t)</code></td>
<td>✗ Has side effects.</td>
</tr>
</tbody>
</table>
Memoization in **Haskell**

Memoization may significantly boost speed.

Find the $n^{th}$ Fibonacci number in $O(n)$ time.

```haskell
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```
7. Advanced abstractions

7.1. Strategies of evaluation

7.1.7. Summary
Concepts of evaluation order

- **Evaluation order?**
  - Short circuit
  - Non-Strict
  - Lazy/Cache
  - Normal order
  - Collateral
  - Strict/Eager
  - Ordered

- Functions with side effects...
- Runtime errors...
- Undefined behavior
Summary: argument passing modes

1. Call by value
2. Call by reference
3. Call by result
4. Call by value-result
5. Call by name
6. Call by need
Summary: other concepts

- Side effects
- Intentionally undefined behavior
- \(\lambda\)-expressions
- \(\lambda\)-calculus

**\(\lambda\)-calculus**

A mathematical formalism in which the concepts of this unit can be stated precisely.

- Memoization
- Thunks
7. Advanced abstractions

7.2. From nested functions to closures

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage
6. Commands

7. Advanced abstractions
7.1 Strategies of evaluation
7.2 From nested functions to closures
7.2.1 Functions as first class values?
7.2.2 Nested functions
7.2.3 The environment
7.2.4 Dynamic scoping
7.2.5 Lexcial scoping
7.2.6 Escaping the nest
7.2.7 Closures
7.3 Emulating objects with closures and vice versa
7.4 Beyond closures
7.5 Generators with Java iterators
7. Advanced abstractions

7.2. From nested functions to closures

7.2.1. Functions as first class values?
Functions are supposed to be values

- In discussing *type constructors* we introduced the *mapping type constructor*, e.g., in ML we have the type `int->int`.
- Values of type mapping, can be realized as
  - Arrays
  - Functions
- It is often the case that functions are not “pure” mappings.
  - In the imperative paradigm, functions may have side effects
  - Even in the functional paradigm, functions may depend on values which are *not* parameters
- Do function values behave just like other values?
Discrimination against function values

In **Pascal**

- Can define variables of almost any type, **but not of type function** (**Discrimination**).
- Functions can take function values as parameters, but **functions cannot return function values** (**Discrimination**).
Function values in C

In C, function values are realized as "function pointers"

- Can be stored in variables
- Can be stored in arrays
- Can be fields of structures
- Can be passed as parameters
- Can be returned from functions

But, C

- does not provide means for creating function values at runtime (Discrimination!)
- does not allow nested functions (Discrimination!)
Nested functions in Gnu-C

Gnu-C is a C dialect, supporting nested functions

```c
int isRightTriangle(double a, double b, double c) {
    double sqr(double x) {
        return x * x;
    }
    return sqr(a) + sqr(b) == sqr(c);
}
```

- Function `sqr` is *nested* in function `isRightTriangle`
- Currently, `sqr` is inaccessible outside `isRightTriangle`
- Function `sqr` may “escape” the boundaries of `isRightTriangle` (e.g., by saving `&sqr` in a global variable).
- Closures are all about escapes from the nest.

**Discrimination:** in Gnu-C, programmer is on his own when taking the address of nested functions!
Topics in this units

- Nested functions
- Name, entity, binding
- Scope and environment
- Lexical scoping/dynamic scoping
- Closures
7. Advanced abstractions

7.2. From nested functions to closures

7.2.2. Nested functions
The nest is accessible to nested functions

Why use nested function?

- Nested functions can access variables and definitions of nest
- Saves lots of argument passing

An implementation of the quick sort algorithm with nested functions (nesting diagram):

```c
void sort(int a[], int n) {
    void swap(int i, int j) {
        ...
    }

    void qsort(int from, int to) {
        int pivot() {
            ...
        }
        ...
    }
    ...
}
```
"Inheritance" of environment with nested functions

```c
void sort(int a[], int n) {
    void swap(int i, int j) {
        ...
    }

    void qsort(int from, int to) {
        int pivot() {
            ...
        }

        ...
    }

    ...
}
```

```c
void sort(int a[], int n) {
    void swap(int i, int j) {
        // "inherits", and uses a
        // "inherits", without using, n
        const int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

    ...
}
```
void sort(int a[], int n) {
    void qsort(int from, int to) {
        int pivot() {⋯}
        if (from == to)
            return;
        const int p = pivot();
        qsort(from, p); // 1st recursive call
        qsort(p, to); // 2nd recursive call
    }
    qsort(0, n);
}

At run time, there might be several instances of the recursive function qsort;

All these instances “inherit” array a and integer n of function sort

"Inheritance" of environment from function sort
Environment “inherited” by function **pivot**

```c
int pivot() {
    // “inherits”
    // from sort: a,
    // from qsort: from, to
    int last = to - 1;
    int pivot = a[first];
    int split = from;
    swap(split, last);
    for (int i = first; i < last; i++)
        if (a[i] < pivot)
            swap(split++, i);
    swap(split, last);
    return split;
}
```

- When **pivot** executes, there is
  - a single instance of function **sort**
  - one or more instances of the recursive function **qsort**
- Function **pivot** inherits
  - integers **from** and **to** from the correct instance of **qsort**
  - integer array **a** from the single instance of **sort**
7. Advanced abstractions

7.2. From nested functions to closures

7.2.3. The environment
Reminder: CPU & memory in the classical model
Understanding the machine stack

```c
int gcd(int m, int n) {
    if (m == n)
        return m;
    int m1 = m;
    int n1 = n;
    if (m > n)
        m1 = m % n;
    else
        n1 = n % m;
    return gcd(m1, n1);
}
```

How does this function “exist” at runtime?

- **Common** sequence of bytes, representing machine code
- **Per activation** the “environment”:
  - Parameters
  - Local variables
A stack frame of function \texttt{gcd}

\begin{verbatim}
int gcd(int m, int n) {
    if (m == n)
        return m;
    int m1 = m;
    int n1 = n;
    if (m > n)
        m1 = m % n;
    else
        n1 = n % m;
    return gcd(m1, n1);
}
\end{verbatim}
Structure of an activation record

<table>
<thead>
<tr>
<th>Region</th>
<th>Managed by</th>
<th>Content</th>
<th>Fixed Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Caller</td>
<td>?</td>
<td>✓</td>
</tr>
<tr>
<td>Parameters</td>
<td>Caller</td>
<td>Return variable, actual</td>
<td>✓</td>
</tr>
<tr>
<td>Saved state</td>
<td>Architecture</td>
<td>Saved registers, saved PC, saved SP</td>
<td>✓</td>
</tr>
<tr>
<td>Local state</td>
<td>Callee</td>
<td>Automatic variables, intermediate results</td>
<td>X</td>
</tr>
</tbody>
</table>

- Each call of `gcd`
  - **Caller**: push arguments
  - **Architecture**: push registers and jump to `gcd` (in response to a hardware `call` instruction)
  - **Callee**: Allocate local variables and use stack for intermediate results.

- Each return from a call
  - **Callee**: pop the frame out of the stack
  - **Architecture**: pop registers and jump saved location (in response to a hardware `ret` instruction)
  - **Caller**: pop arguments

---

1. To be discussed below
2. Variable size for variadic functions, i.e., functions such as `printf` whose number of parameters is not fixed
Activation record as a contract

A function call has two parties

- The caller
- The callee

An activation record represents the contract of interface between the two:

- **Saved state** everything that is required to reconstruct the caller’s state at the end of the call; typically, saved registers.
- **Local state** local variables of the callee, intermediate results, etc.
- **Arguments** values that the caller sent to the callee
- **Environment** variables, functions, definitions, etc. defined by the caller and used by the callee.

An activation record is often realized by a stack frame.
What’s the “environment”?

**Definition (Environment (first approximation))**

Variables, functions, constants, etc., of the caller, which can be used by callee.

- Does not exist in C/C++
- In **Pascal**, definitions made in a function, are available to any nested function.
- Limited forms available in Gnu-C and **Java**
Acquired bindings in nested functions

What’s the environment of \texttt{pivot}?

```c
void sort(int a[], int n) {
    void swap(int i, int j) { ...}
    void qsort(int from, int to) {
        int pivot() { ???}
    }
    ...
}
```

- **Function** \texttt{sort}
- **Function** \texttt{swap}
- **Function** \texttt{qsort}
- **Function** \texttt{pivot}

- **Arguments to function** \texttt{sort}
  - Array \texttt{a}
  - Integer \texttt{n}
- **Arguments to function** \texttt{qsort}
  - Integer \texttt{from}
  - Integer \texttt{to}
Name vs. entity

- But what’s really in the phrase
  
  "variables, functions, constants, etc."

- We distinguish between
  - Name
  - Entity

Note that:

- Entities may have no name (e.g., anonymous functions, classes, allocated variables, etc.)
- Entities may have more than one name (e.g., type aliases)
- In some cases, a name might have a name, (e.g., tetragrammaton, shemhamphorasch, but also found in reflective programming)
Binding

**Definition (Binding)**

*Binding is the tie between a name and an entity*

The phrase “variables, functions, constants, etc.”, means

the set of bindings in the caller available to the callee
The environment: a more precise definition

**Definition (Environment (second approximation))**

The environment of a function, is the set of set of bindings of the caller which are available to the callee.

In **Pascal**, the environment includes

- The **CONST** section of the caller (binding of names to values)
- The **VAR** section of the caller (binding of names to variables)
- The **TYPE** section of the caller (binding of names to types)
- The **LABEL** section of the caller (binding of names (integers) to program locations)
- Functions and procedures defined within the caller (binding of names to functions and procedure values)

In **C/C++**, there is no “caller environment”
Environment of a nested function

What’s the environment of function **pivot**?

```c
void sort(int a[], int n) {
    void swap(int i, int j) { ... }
    void qsort(int from, int to) {
        int pivot() { ??? }
    }
}
```

Function names in the environment:

- The binding of the names: **sort**, **swap**, **qsort** and **pivot** to the “functions”
- Binding here is of names to the pointers to functions.

Function arguments in the environment:

- Binding of names of arguments to function **sort**
- Binding of names of arguments to function **qsort**
  - These bindings are distinct in each recursive calls.
  - The semantics of Gnu-C are such that **pivot** acquires the “correct” bindings.
Environment vs. scope

In simple words...

- **Q:** What’s the environment?
- **A:** All variables which are in “scope”
- **Q:** What’s scope?
- **A:** The extent of locations in the program in which a binding is recognized.

Two scoping (binding) rules are found in PLs

**Lexical scoping** Bindings not made in function, are determined by where the function is defined.

**Dynamic scoping** Bindings not made in function, are determined by when the function is active.

Environment could be

- Defined statically (lexical scoping)
- Defined dynamically (dynamic scoping)
7. Advanced abstractions

7.2. From nested functions to closures

7.2.4. Dynamic scoping
What’s dynamic scoping

Inheritance of bindings?

When does $f_2$ “inherit” bindings made in $f_1$?

Answer I/II: (dynamic scoping) If $f_2$ is called by $f_1$.

(sometimes called dynamic binding)

- Inheritance is inductive: $f_2$ inherits bindings of $f_0$ if $f_1$ is called by $f_0$.
- Environment defined by program dynamics
- Conversely, the scope of bindings made in $f_1$ is:
  - all functions called, directly, or indirectly by $f_1$
  - determined dynamically
  - hence the term *dynamic scoping*
Dynamic scoping

Definition (Dynamic scoping)

The environment is determined by calls; if $f_1$ calls $f_2$, then $f_2$ “inherits” bindings made by $f_1$.

Dynamic scoping cares not whether there is any relationship between $f_1$ and $f_2$.

- Found in **LISP** (but not in **SCHEME**)
- Found in **TEX** (text processing PL used to produce this material)
- Found in **PERL**!
- Found in **BASH**!
- Found in **HTML/CSS**!

The de-facto semantics of the **C** preprocessor.
Semantics of dynamic scoping

```c
void (*exchange)(int, int);

void sort(int a[], int n) {
    void swap(int i, int j) {
        const int t = a[j];
        a[i] = a[j];
        a[j] = t;
    }
    ...  
    exchange = swap;
    ...
}
```

```c
...  
#define SIZEOF(z) (sizeof(z) / sizeof(z[0]))

int main(int argc, char **argv) {
    int a[500];
    int b[500];
    ...
    sort(b, SIZEOF(b));
    ...
    (*exchange)(10, 12);
    ...
}
```

What would work?

- **sort calling qsort?** Yes.
- **qsort calling qsort?** Yes.
- **qsort calling pivot?** Yes.
- **main calling swap?** Yes.
The runtime bindings dictionary

The “binding dictionary”:

- Local state is represented by a dictionary
- Dictionary supports mapping between names and entities
- Typical entities are variables, constants, and functions.
- Typical implementations of dictionary are linked list and hash table.

The environment:

- The environment is represented as a “back pointer” (BP) to the most recent stack frame.
- Thus, we have a “stack of dictionaries”.
- Search for name is carried out by searching in the stack of dictionaries.
- There are means to make this search quite efficient.
7. Advanced abstractions

7.2. From nested functions to closures

7.2.5. Lexcial scoping
Lexical scoping

Inheritance of bindings?
When does \( f_2 \) “inherit” bindings made in \( f_1 \)?

**Answer II/II:** (lexical scoping) If \( f_2 \) is defined inside \( f_1 \)
(sometimes called static scoping, or static scoping)

**Program code**

```
Function \( f_0 \) {
    Function \( f_1 \) {
        Function \( f_2 \) {
            
        }
    }
}
```

- By induction, \( f_2 \) inherits bindings of \( f_0 \) if \( f_1 \) is defined inside \( f_0 \).
- Environment defined by static code structure.
Intricacies in lexical scoping

Definition (Lexical scoping)

The environment is determined by scope; if \( f_1 \) is defined in \( f_2 \), then \( f_2 \) “inherits” bindings made by \( f_1 \).

- There might be more than one activation record of \( f_1 \) on the stack.
- The caller of \( f_2 \) might not be \( f_1 \).
- Lexical scoping means the most recent version of \( f_1 \).

Lexical is static?

Lexical scoping is not entirely static. The correct edition of the nest must be determined dynamically.
Lexical nesting does not imply call order

```c
void sort(int a[], int n) {
    void swap(int i, int j) { … }
    void qsort(int from, int to) {
        int pivot() { … }
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>pivot</th>
<th>qsort</th>
<th>sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>called by</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>nested in</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>called by</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- “nested in” does not imply “called by”.
- “called by” does not imply “nested in”!
Convoluted calls

Who can call function \textit{f}_2()?

The caller of \textit{f}_2 is not necessarily \textit{f}_1; it could be

- function \textit{f}_2 itself
- function \textit{f}_3 defined in \textit{f}_2
- function \textit{f}_4 defined in \textit{f}_3
- ...
- another function \textit{g}_1 defined in \textit{f}_1
- function \textit{g}_2 defined in \textit{g}_1
- function \textit{g}_3 defined in \textit{g}_2
- ...

But not function \textit{f}_0 within which \textit{f}_1 is defined!
Back pointers in the quick sort example
Consider the following chain of calls:

```
main() → sort() → qsort() → qsort() → qsort() → pivot() → swap()
```

**Frames on the hardware stack**
*(grows from high memory to low memory, depicted right to left)*

Legend:
- **Caller's saved state**
- **Back pointer**
- **Arguments**
- **Local variables**
Managing lexical scoping with back pointers

- The environment is represented as a “back pointer” (BP)
- The BP is part of the stack frame
- BP points at the stack frame of the most recent version of nest

Using the back pointer:

- To access variables defined in $f_1$ function $f_2$ traverses the BP, to find the local state of $f_1$
- To access variables defined in $f_0$ function $f_2$ hop twice through the BP list, to find the local state of $f_0$
- To access variables defined in $f_{-1}$ (the function within which $f_0$ is defined), make three hops along the BP list.
- ...

...
Maintaining the back pointer

- Let

\[ f \in \{ f_1, f_2, f_3, \ldots, g_1, g_2, g_3, \ldots \} \]

be a function ready to call \( f_2 \).

- \( f \) must traverse the back pointers list to find the "most recent" stack frame of \( f_1 \).

- Copy this address to the BP of the stack frame of the newly created stack frame for \( f_2 \).

Number of hops?

- If \( f = f_1 \), then no hops.
- If \( f = f_2 \), then one hop.
- If \( f = f_3 \), then two hops.
- If \( f = g_1 \), then one hops.
- If \( f = g_2 \), then two hops.
- ...
A simplified implementation of lexical scoping

- Suppose that all bindings in the environment are read-only.
- Then, instead of juggling the BP, one can simply copy the entire set of bindings as arguments.
7. Advanced abstractions

7.2. From nested functions to closures

7.2.6. Escaping the nest
Function variables & dangling references

Similarly, suppose that Pascal had function variables...

- The environment “dies” as a stack frame is popped.
- C forbids nested functions for this reason.
- Pascal forbids function variable for this reason
- To make first class “function values”, the activation record cannot be allocated on the hardware stack.

```pseudo
Program Main;
VAR savedF: Integer -> Char; (* X *)
  (* (no function variables) *)
Procedure P;
  VAR m: Integer;
Function f(n: Integer): Char;
  Begin (* uses inherited m *)
    if (m - n > 0) f := 'p' else f := 'n'
  end; (* function f*)
Begin (* Procedure p *)
  readln(m);
  savedF := f; (* X! *)
End (* Procedure p *)
Begin (* Program main *)
P;
...
savedF(2) (* X! *)
...
end.
```
Nested functions escaping the nest (Gnu-C)

Function add nested in makeAdder

```c
#include <stdio.h>

typedef int (*F)(int);

// The nest:
F makeAdder(int a) {
    // The nested:
    int add(int x) {
        // The use of environment (variable a)
        const int $ = a + x;
        printf("f. add(): return %d\n", $);
        return $;
    }
    // The escape:
    return add;
}

// Refuge for escaped values:
F add5, add7;
```

- **F** is an alias for the type of a function that takes an integer and returns an integer.
- Function **add** is nested in function **makeAdder**, the nest.
- The nested uses the environment (e.g., variables) of the nest.
- The nested escapes the nest.
- The escaped value will be stored in global variables **add5** and **add7**, the refuge.
Forcing access to lost environment

Lost environment

```c
// Provide refuge to the escaping:
void initialize() {
    add5 = makeAdder(5);
    add7 = makeAdder(7);
}

int dozen() {
    // Use the nested that escaped:
    return add7(add5(0)); // X
}

// Force program crash:
int main() {
    initialize();
    printf("dozen=\%d\n", dozen());
    return 0;
}
```

- Store in the refuge two versions of function `add`, the nested that escaped.
- Both use the environment of the nest, function `makeAdder`.
- But function `makeAdder` is not active anymore.
- So, we are doomed for trouble…

---

`f. add(): return 7
Segmentation fault (core dumped)`
7. Advanced abstractions

7.2. From nested functions to closures

7.2.7. Closures
What are closures?

- Closures can be thought of as
  - First class nested functions.
  - Functions that close over their environment.
  - Functions whose activation record is heap managed (rather than stack managed).

- Many modern PLs support closures, where a function can be created in a way that it “keeps with it” the variables that exist in its scope.

- Often used in functions-that-return-functions

- You can’t do that in Pascal, C, C++

- Very common in functional languages, ML, Python, JavaScript and Java (since version 8.0; uses a hack).
**Closures in JavaScript**

A closures factory

```javascript
function makeAdder(i) {
    var delta = i;
    function adder(n) {
        return n + delta;
    }
    return adder;
}
```

Usage

```javascript
var add3 = makeAdder(3);
document.write(add3(7));
document.write(add3(12));
var add8 = makeAdder(8);
document.write(add8(12));
```

Output:

```
10
15
20
```
Closures and lifetime

- **Q:** what’s the lifetime of a variable enclosed in a closure?
- **A:** the lifetime of the enclosing value.
  - In this example, the lifetime of “delta” for each function returned by `makeAdder` is the lifetime of the return value of `makeAdder`.
  - The same as lifetime of fields in a record allocated on the heap: live as long as the record is still allocated.

In general, you cannot have closures without using GC, rather than the stack for implementation of activation records. With closures, activation records are allocated from the heap.
Closures in ML

Just as in JavaScript, in ML, all functions are closures.

- Standard programming idiom of the language
- Also supports anonymous functions
- Function values are first class values (including the environment)
7. **Advanced abstractions**

7.3. Emulating objects with closures and vice versa
7. Advanced abstractions

7.3. Emulating objects with closures and vice versa

7.3.1. Objects with closures
Motivation: objects in ML using closures & references

Accumulator: *adds, counts, and computes the mean*

```ml
fun makeAccumulator() = let
  val n = ref 0
  val sum = ref 0.0
  in {
    count = fn() => !n,
    total = fn() => !sum,
    add = fn r:real =>
      (n := !n + 1; sum := !sum + r),
    mean = fn() =>(!sum / real(!n))
  } end;

val makeAccumulator = fn
  : unit -> {
    add:real -> unit,
    count:unit -> int,
    mean:unit -> real,
    total:unit -> real
  }
```
Computing your average grade

create a new accumulator

```ml
val grades = makeAccumulator();
```

```ml
val grades = {
    add=fn, count=fn, mean=fn, total=fn
}: {
    add:real -> unit, count:unit -> int, mean:unit -> real, total:unit -> real
}
```

record grade in CS101

```ml
(#add grades)(82.0);
val it = () : unit
```

record grade in calculus

```ml
#add grades (37.0);
```

```ml
val it = () : unit
```

how many grades so far?

```ml
#count grades ();
val it = 3 : int
```

what’s their total?

```ml
#total grades ();
val it = 219.0 : real
```

what’s my GPA?

```ml
#mean grades ();
val it = 73.0 : real
```
7. Advanced abstractions

7.3. Emulating objects with closures and vice versa

7.3.2. Function objects of C++
Emulation closures with C++ function objects

Class of “adding something” function objects:

```cpp
class FunctionObjectForAdding {
public:
    FunctionObjectForAdding(int _b): b(_b) {}
    int operator()(int a) { return a + b; }
private:
    int b; // Saved environment
};
```

- A function object stores a copy of relevant pieces of the environment as data members of the class.
- Environment copy is managed by the function object, which, just like any other C++ variable, can be
  - present on the stack
  - allocated from the heap
  - global
- Memory management is the programmer’s responsibility.
Using C++’s function objects

Usage:

```cpp
#include <iostream>

int main() {
    FunctionObjectForAdding add3(3);
    std::cout << add3(7);
    std::cout << add3(12);
    FunctionObjectForAdding add8(8);
    std::cout << add8(12);
    return 0;
}
```

Output:

```
10
15
20
```
7. Advanced abstractions

7.3. Emulating objects with closures and vice versa

7.3.3. Function objects in Java

6. Commands

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage

7. Advanced abstractions
7.1 Strategies of evaluation
7.2 From nested functions to closures
7.3 Emulating objects with closures and vice versa
7.3.1 Objects with closures
7.3.2 Function objects of C++
7.3.3 Function objects in Java
7.4 Beyond closures
7.5 Generators with Java iterators
Closures in **JAVA**?

- In **JAVA** the only first class values are objects.
- Hence, in **JAVA** functions are second class citizens.
- **JAVA** does not have real closures.
- Still, you can imitate closures with objects.

**Imitation is just like C++'s function objects**
Function objects with Java

Class of function object

class Log {
    final double logBase; // Captured environment
    Log(final double base) {
        logBase = Math.log(base);
    }
    public double apply(double v) {
        return Math.log(v) / logBase;
    }
}

Using the function object:

public class L {
    public static void main(String[] args) {
        final Log log2 = new Log(2);
        System.out.println("Log base 2 of 1024 is " + log2.apply(1024));
    }
}

Log base 2 of 1024 is 10.0
7. Advanced abstractions

7.3. Emulating objects with closures and vice versa / 7.3.3. Function objects in Java

An interface for function objects

Abstract function type

```java
interface Function<T, R> { R apply(T t); }
```

```java
class LOG implements Function<Double, Double>
{
    // captured environment:
    final Double logBase;
    LOG(final Double base) {
        logBase = Math.log(base);
    }
    public Double apply(Double t) {
        return Math.log(t) / logBase;
    }
}
```

Why **Double**? Why not **double**?

- **Java** primitive types follow **value** semantics
- all other types follow **reference** semantics
- **Java** generics require **reference** semantics
- hence, type **double** is not first class
- use reference type **Double** instead:
7. Advanced abstractions

7.3. Emulating objects with closures and vice versa

7.3.3. Function objects in Java

Function objects in Java

class LOG implements Function<Double, Double> {
   // captured environment:
   final Double logBase;
   LOG(final Double base) {
      logBase = Math.log(base);
   }
   public Double apply(Double t) {
      return Math.log(t) / logBase;
   }
}

Using the function object:

public class M {
   public static void main(String[] args) {
      Function<Double, Double> log2 = new LOG(2.0);
      System.out.println("Log base 2 of 1024 is "+ log2.apply(1024));
   }
}

Log base 2 of 1024 is 10.0
Another use of the interface for function objects

- **interface** `Function` is handy in other situations
- It is part of the standard library of **Java 8**

```java
class IntMap implements Function<Integer, String> {
    private final String[] map;
    public IntMap(String[] map) {
        this.map = map;
    }
    public String apply(Integer v) {
        return v >= 0 && v < map.length
            ? map[v]
            : "" + v;
    }
}
```

Note minor problem:

- Generics do **not** take as parameters atomic types such as `int` and `double`
- Use instead equivalent reference types: `Integer`, `Double`, ...
Using objects of type Function<T,R>

Storing function objects in variables:

```java
Function<Integer,String> inRoman =
    new IntMap(
        new String[] {
            "i", "ii", "iii", "iv", "v"
        }
    );
Function<Integer,String> inEnglish =
    new IntMap(
        new String[] {
            "one", "two", "three"
        }
    );
```

Applying function objects:

```java
for (int i = 0; i < 10; ++i)
    System.out.println(i + "\t" + inRoman.apply(i) + "\t" + inEnglish.apply(i));
```

```
0 i one
1 ii two
2 iii three
3 iv 3
4 v 4
5 5 5
6 6 6
7 7 7
8 8 8
9 9 9
```
Functions returning function objects

Instead of calling the constructor, one may use factory functions:

```java
class IntMap implements Function<Integer, String> {
    ...
    // Constructor is private!
    private IntMap(String[] map) { ... }
    // Factory function:
    public static IntMap mapping(String[] map) {
        return new IntMap(map);
    }
}
```

Now, write `mapping(...)` instead of `new IntMap(...):

- Slightly shorter syntax
- Conceal the fact that a new object is created
Using factory functions

Using *factory function* `mapping` to create two function objects:

```java
IntMap inLatin = mapping(
    new String[] { "i", "ii", "iii", "iv", "v" }
);
IntMap inEnglish = mapping(
    new String[] { "one", "two", "three", }
    );
```

Using the returned function objects:

```java
for (int i = 0; i < 10; ++i)
    System.out.println(i + "\t" + inRoman.apply(i) + "\t" + inEnglish.apply(i));
```

0 i one
1 ii two
2 iii three
3 iv 3
4 v 4
5 5 5
6 6 6
7 7 7
8 8 8
9 9 9
Smarter function objects with inner classes

In the above, the programmer must manually store the “environment”:

```java
class Log {
    final double logBase;
    Log(final double base) {
        logBase = Math.log(base);
    }
    ...
}
```

```java
class LOG implements Function<Double, Double> {
    final Double logBase;
    LOG(final Double base) {
        logBase = Math.log(base);
    }
    ...
}
```

**Idea:**

- In (all but most ancient versions of) Java you can define local classes: classes inside the scope of a method.
- Bindings made in a class \( C \) are available to:
  - are available to a function \( f \) defined in \( C \),
  - are available to a class \( C' \) defined in \( f \),
  - are available to a function \( f'' \) defined in \( C' \),
  - are available to a class \( C'' \) defined in \( f'' \),
  - ...

Function objects with inner classes

The trick: Use inner classes; inner classes do close on the environment:

```java
public class C {
    static Object foo(final int a) {
        class Inner {
            public int hashCode() {
                return (a-1)*a*(a+1)*(2*a + 3);
            }
        }
        return new Inner();
    }
    public static void main(String[] args) {
        Object o = foo(6);
        System.out.println(o.hashCode());
    }
}
```

Output is 42!

Q: How does Java save the value of `a`?
A1: Activation records are allocated by the GC.
A2: All accessible variables in the environment are copied to the activation record.

- No BP!
- Simplified implementation of environment.
- Variables of the environment must be final
Inner class of factory function

More power is drawn from the combination of:
- factory functions
- inner classes

```java
// Should be public to be useful:
public

// Factory methods tend to be static
// (why require an existing object before creating
// a new object?)
static

// Abstract function type:
Function<Integer,String>
    mapping(String[] map) {
        // Local class of function mapping:
        class IntMap
            implements Function<Integer, String> {
                public String apply(Integer v) {
                    ... // as before
                }
            }

        // Function mapping returns a newly
        // created instance of its local class:
        return new IntMap();
    }
```

If the factory function returns an instance of an inner class, function objects become even simpler:
7. Advanced abstractions

7.4. Beyond closures
7. Advanced abstractions

7.4. Beyond closures

7.4.1. Generators
A simple **Python** generator

A generator is defined just like a function, except that it uses `yield` instead of `return`

```python
def countdown(n):
    while n >= 0:
        yield n
    n -= 1
```

Now,

**on December 31\(^{st}\), 23:59:50:**

```python
for i in countdown(10):
    print i, "...",
print "Happy new year!", "X!", "X!", "X!"
```

```
10 ... 9 ... 8 ... 7 ... 6 ... 5 ... 4 ... 3 ... 2 ... 1 ... 0 ...
Happy new year! X! X! X!
```
A generator built on top of another

`range` is a builtin generator; `range(n)` yields `0, 1, ..., n − 1`.

**A generator yielding an arithmetic progression**

```python
def arithmeticProgression(a1, d, n):
    for i in range(n):
        yield a1 + i*d
```

Using the “compound” generator

```python
for ai in arithmeticProgression(3, 4, 5):
    print ai,
3 7 11 15 19
```
Not much distinction between functions and generators:

```python
def fu(n):
    return n

def countdown(n):
    while n >= 0:
        yield n
    n -= 1
```

Checking out values of `fu` and `countdown`:

```python
>>> fu
<function fu at 0x7fc0e82f47d0>

>>> countdown
<function countdown at 0x7fc0e82f4848>
```
Functions vs. generators in **Python**

A generator looks like a function, but behaves like an iterator:

<table>
<thead>
<tr>
<th>Function <code>fu</code></th>
<th>Generator <code>gen</code></th>
</tr>
</thead>
</table>
| **Definition:** | Defined as:
| `def fu(x):` | `def gen(x):` |
| `return x` | `yield x` |
| **Invoking as function:** | `fu(3)` |
| `3` | `<generator object gen at 0x7f65af3eae60>` |
| **Invoking as generator:** |
| `for i in fu(3):` | `for i in gen(3):` |
| `print i` | `print i` |
| `Traceback (most recent call last):` | |
| `File "<stdin>"`, line 1, in ` | |
| `<module>` | |
| `TypeError: 'int' object is not iterable` | `3` |
Generators vs. iterators

Generator is a function...

Definition (Generator)
A generator is a function that produces a sequence of results, instead of returning a single result.

The sequence may be empty, or infinite.

Iterator is an object...

Definition (Iterator)
An iterator is an object which can be used to traverse a datastructure such as a list.
Generators vs. functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td># results:</td>
<td>1</td>
</tr>
<tr>
<td>Terminology:</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>call / invoke</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A function must return one value, even if it has nothing to offer.
- Generators have two kinds of “invocations”:
  - **initiate** start the production process
  - **retrieve** obtain the next element from the sequence
Generators in **ICON**

**In ICON, all expressions are generators:**

- **Singleton generators** \(3\) is a generator of the sequence \(\{3\}\)

- **Or generator** \(3|4|5\) is a generator of the sequence \(\{3, 4, 5\}\)

- **Depth first generation** \((30|40|50)+(1|2)\) is a generator of the sequence \(\{31, 32, 41, 42, 51, 52\}\)

- **“Product” of sequences** \((1|2|3)*f() + (1|2) * g()\) is a generator of a sequence with \(6 \times n \times m\) values, where
  - \(n\) is the number of values that \(f\) yields
  - \(m\) is the number of values that \(g\) yields
Comparison generators

Comparison generators realize arithmetical conditions

- $i == j$ yields
  \[
  \{j\}
  \]
  if $i = j$; otherwise, the empty sequence
  \[
  \{}
  \]

- $i == (3|4|5)$ succeeds if $i \in \{3, 4, 5\}$; it yields either one of
  \[
  \{3\}, \{4\}, \{5\}, \{}
  \]
  depending on the value of $i$. 
Neat mathematical syntax with comparison generators

A familiar mathematical notation:

\[ 0 \leq i < j \leq n. \] (4.1)

**Java:** cumbersome \( 0 \leq i \land i < j \land j \leq n \)

**Icon:** elegant \( 0 \leq i < j \leq n \) yields:

- \( \{n\} \) if the condition holds
- \( \{\} \) if the condition fails

Same \( i \leq i \land i < j \land j \leq n \) with added parenthesis

\[ (((0 \leq i) < j) \leq n) \]

Each conditional expression yields its second argument as input to the next conditional

- \( 0 \leq i \) yields either \( \{i\} \) or \( \{\} \)
- \( 0 \leq i < j \) yields either \( \{j\} \) or \( \{\} \)
- \( 0 \leq i < j \leq n \) yields either \( \{n\} \) or \( \{\} \)
Control with generators

Iteration and conditional are both generating contexts:

- **if** statement:
  - Execute body if a value is yielded
  - Triggered at most once, by the *first* yielded value

- **while** statement:
  - Execute body for each value yielded
  - Triggered by all yielded values

A simple **ICON** loop

```icon
while line := read() do write(line)
```

A loop with empty body

```icon
while write(read())
```
Generators in C#  

In statically typed PLs, generators call for sophisticated typing

```csharp
using System;
using System.Collections.Generic;

public class Program {
    static void Main() {
        foreach (int p in Powers(2, 30)) {
            Console.Write(p); Console.Write("");
        }
    }

    public static IEnumerable<int> Powers(int base, int count) {
        int result = 1;
        for (int i = 0; i < count; ++i) yield return result *= base;
        yield break; // optional
    }
}
```

Type of function **Powers** is  

```
int*int->IEnumerable<int>
```
Summary of generators

- Like closures, generators close over their environment.
- Activation records of generators, include also the Saved context of the callee makes it possible for the callee (the generator) to resume its state.
- For convenience, we shall call the term “generating record” for this augmented activation record.
- A generating record supports three conceptual operations:
  - `resume()` resume execution from the last saved context.
  - `resumable()` determine whether resumption is possible.
  - `retrieve()` obtain the value of the generator.
- A “generation contexts” tries to exhaust the generator using these three operations.

In many ways, generators are just like predicates of Prolog.
7. Advanced abstractions

7.4. Beyond closures

7.4.2. Coroutines
What are coroutines?

**Definition (Coroutine)**

A *coroutine* is a function, which can

- suspend anywhere in its execution by executing a **yield** command.
- resume a suspended execution, proceeding to the command that follows the suspending **yield**

Coroutines vs. generators:

- Generators are also called semicoroutines
- Generator can only **yield** to its caller
- Coroutine typically **yield** to another coroutine
Overloading of “yield”: two unrelated meanings

Two overloaded meanings of the word “yield” in English:

- **to produce**
- **to let other take control**

In theory of PL:

- **Generators** *to produce*
- **Coroutines** *to let other take control*
Unconvincing example of coroutines

A sow and its two little piglets may all die in the pigsty

```pseudo
void sow() { for (;;) {
    defecate();
    if (hungry) feed();
    if (thirsty) drink();
    yield piglet1;
}}
```

```pseudo
void piglet1() { for (;;) {
    defecate();
    if (hungry || thirsty) suck();
    yield piglet2;
}}
```

```pseudo
void piglet2() { for (;;) {
    defecate();
    if (hungry || thirsty) {
        suck();
        yield sow;
    }
}}
```

If `piglet2` is not hungry neither thirsty.

- `sow` may die
- `piglet1` will then die
- `piglet2` himself will eventually die
Issues with the pigsty

It is not clear at all

- how the coroutines are initiated?
- how each coroutine gains access to the other?
- how can one create multiple instances of the same coroutine? (create several active invocations of (say) piglet2())
- why coroutines are better than plain threads?

So, why the heck do we need these?
Coroutines vs. threads

Both offer multitasking:

**Threads** preemptive; execution can be interrupted at any point

**Coroutine** cooperative; control passed when the

Zillions of weird scheduling schemes?

**Threads** Yes!

**Coroutine** only with careless programming

Race conditions?

**Threads** Yes!

**Coroutine** No!

Generally speaking, coroutines grant the programmer fine control of scheduling, but it may take skill to effectively use this control.
The sad story of coroutines

- invented in the early 60’s (if not earlier)
- mostly forgotten; often discarded as “cumbersome multitasking”
- have (partial) implementation in mainstream PLs such as C, C++, Java, Python, Perl, Object-Pascal, Smalltalk and more
- implementations did not really catch

Hence, the poorly designed syntax in the pigsty.
Killer application of coroutines

Event loops:

**Events dispatcher**

```pseudo-c#
void dispatcher() {
    // Yields to appropriate coroutine
    for (;;) {
        Message m = getMessage();
        push(m); // save message for later
        Area a = findScreenArea(m);
        Coroutines cs = findWindows(a);
        for (Coroutine c: cs) {
            yield(c);
            if (consumed(m))
                break;
            // otherwise, try another c
        }
    }
}
```

**A co-routine window function**

```pseudo-c#
void myWindow(Rectangle r) {
    // Initialize window here, and, then, ...
    yield dispatcher;
    for (;;) {
        switch (Message m = getMessage()) {
            case M1: ...
            case M2: ...
            ...
            default: // Message is not my responsibility:
                push(m);
                yield dispatcher;
        }
    }
}
```

Useful in browser applications such as **Gmail**
7. **Advanced abstractions**

7.4. Beyond closures

7.4.3. **Continuations**

---

1. Preliminaries
2. Introduction
3. Values and types
4. Advanced typing
5. Storage
6. Commands
7. **Advanced abstractions**
   7.1 Strategies of evaluation
   7.2 From nested functions to closures
   7.3 Emulating objects with closures and vice versa
   7.4 Beyond closures
      7.4.1 Generators
      7.4.2 Coroutines
      7.4.3 Continuations
      7.4.4 Summary
   7.5 Generators with Java iterators
Continuations

Quite an obscure and abstract definition:

**Definition (Continuation)**

A *continuation* is an abstract representation of the control state of a computer program

More plainly, a continuation is making the activation record (more generally, the generating record) into a first class citizen.

- Invented many years ago
- Has full implementation in many dead and obscure PLs
- Has partial/non-official implementation in many mainstream PLs
- Did not catch (yet!)
Data stored in a continuation

**Code**  Where the function is stored in memory, its name, and other meta information, e.g., for debugging.

**Environment**  Activation record, including back-pointers as necessary.

**CPU state**  The saved registers, most importantly, the PC register

**Result**  Last result *returned/yielded* by the function

**Termination status**  can the continuation be continued
### Operations on continuations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \leftarrow \text{initiate}(f(...))$</td>
<td>Create a new continuation for the call $f(...)$</td>
</tr>
<tr>
<td>$c' \leftarrow \text{clone}(c)$</td>
<td>Save execution state for later</td>
</tr>
<tr>
<td>$\text{resumable}(c)$</td>
<td>Determine whether resumption is possible</td>
</tr>
<tr>
<td>$c \leftarrow \text{resume}(c)$</td>
<td>Resume execution of continuation $c$</td>
</tr>
<tr>
<td>$\text{retrievable}(c)$</td>
<td>Determine whether computation produced a result</td>
</tr>
<tr>
<td>$x \leftarrow \text{retrieve}(c)$</td>
<td>Obtain the produced result of $c$</td>
</tr>
<tr>
<td></td>
<td>generator a generated value</td>
</tr>
<tr>
<td></td>
<td>continuation a continuation record</td>
</tr>
</tbody>
</table>

Continuations are slightly more general than “generation records” in that they may **yield** continuations, rather than just plain values.
Creating continuations by the dispatcher in the event loop example

When user hits a “new window” button:

1. Create a rectangle:
   
   ```java
   Rectangle r = new Rectangle(...);
   ```

2. Initiate the continuation record (without running it):
   
   ```java
   Continuation c = initiate(myWindow(r));
   ```

3. Let function `myWindow` “run” to initialize itself a bit:
   
   ```java
   c = resume(c);
   ```
   The function inside `c` will `yield` to the dispatcher, so we may continue working.

4. Associate rectangle with window:
   
   ```java
   dictionary.add(r,c);
   ```

Now, the dispatcher may continue waiting for events, and dispatch to the right continuation.
Dispatcher main loop with continuations

1. Obtain a keyboard event, a mouse event, or some other message:
   ```
   Message m = getMessage();
   ```

2. Save this message so it can be consumed by one of our windows coroutines:
   ```
   push(m);
   ```

3. Find the screen area into which this message belongs:
   ```
   Area a = findScreenArea(m);
   ```

4. Find all window coroutines associated with this message
   ```
   Continuations cs = findWindows(a);
   ```

5. Iterate over all of them:
   ```
   for (Continuation c: cs)
     c = resume(c);
   ```
   ```
   if (!resumable(c)) dictionary.remove(c);
   ```
   ```
   if (consumed(m)) break;
   ```
What can you can with continuations?

- Good old function calls
- Closures
- Generators
- Exceptions
- Coroutines

With continuations, you can conceal the fact that Web server should be stateless; (you just provide the client with the server’s continuation record)
7. Advanced abstractions

7.4. Beyond closures

7.4.4. Summary
Main concepts of this unit

- Generators: **PYTHON**, **C#**, and **ICON**
- Coroutines vs. threads,
- Continuations
# Implementation of advanced abstractions

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinary function</td>
<td>Stack frame</td>
</tr>
<tr>
<td>Nested function</td>
<td>Stack frame with back pointers</td>
</tr>
<tr>
<td>Call by name</td>
<td>Thunks</td>
</tr>
<tr>
<td>Thunks</td>
<td>Nested functions</td>
</tr>
<tr>
<td>Closures</td>
<td>Activation record on heap</td>
</tr>
<tr>
<td>Generators</td>
<td>Generating record</td>
</tr>
<tr>
<td>Coroutines</td>
<td>Continuations</td>
</tr>
</tbody>
</table>
7. Advanced abstractions

7.5. Generators with Java iterators

7.5.1. Understanding generators
The “magic” behind a generating context

The generating context

```csharp
foreach (int p in Powers(2, 30)) {
    Console.Write(p);
    Console.Write("\n");
}
```

is implemented like so:

```csharp
// This variable is inaccessible to programmer
GeneratingRecord _;

for (_ = Powers(2, 30);
     _.resumable();
    int p = _.retrieve())
{
    Console.Write(p);
    Console.Write("\n");
}
```
Nested generation

Each generating context has its own generating record; so, the following inner loop

```csharp
foreach (int p in Powers(2, 30)) {
    foreach (int q in Powers(2, 30)) {
        Console.Write(p + q);
        Console.Write(" ");
    }
    Console.Write("\n");
}
```

Will print twice each integer $n$, if $n$ can be written as

$$n = 2^p + 2^q$$

where

$$1 \leq p \leq 30$$

and

$$1 \leq q \leq 30$$
Distinct generating record for each generation context

Implementation of nested generation with two generation records:

```csharp
// This variable is inaccessible to programmer:
GeneratingRecord _1;
for (_1 = Powers(2, 30);
    _1.resumable();
    int p = _1.retrieve())
{
    // This variable is inaccessible as well:
    GeneratingRecord _2;
    for (_2 = Powers(2, 30);
        _2.resumable();
        int q = _2.retrieve())
    {
        Console.Write(p + q);
        Console.Write(" ");
    }
}
```

Original: nested generation

```csharp
foreach (int p in Powers(2, 30)) {
    foreach (int q in Powers(2, 30)) {
        Console.Write(p + q);
        Console.Write(" ");
    }
}
```
7. Advanced abstractions

7.5. Generators with Java iterators

7.5.2. Iterators in Java
Emulating generators with Java iterators

In Java (up to version 7):
- no closures
- no generators

Still, it is possible to emulate generators in Java, especially given that Java has GC:
- A clever programmer can manually “invert control” to emulate a generator
- A clever programmer can save the environment in an object
- A clever programmer can save the local state in an object

So, all you need is to be clever enough. (In Java 8, this exact “cleverness” was packaged into the language.)
Java iterators: poor man’s generators

We shall have two classes

- Main class which creates and then uses the iterator.
  - Will create the “environment” for the our poor-man’s generator
  - Environment includes the values 2 and 30. in the C# call `Powers(2,30)`
- class `Powers` the iterator object itself.
  - Will save the “environment” that function `main` passes on to it.
  - Will save the local variables of the generator as class variables.
Invoking the iterator

### Main class in JAVA

```java
import java.util.Iterator;

public class E {
    public static void main(String[] args) {
        Integer p;
        Iterator<Integer> it;
        for (it = new Powers(2,30); it.hasNext(); ) {
            p = it.next();
            System.out.println(p);
        }
    }
}
```

### C# original

```csharp
using System;
using System.Collections.Generic;

public class Program {
    static void Main() {
        foreach (int p in Powers(2, 30)) {
            Console.Write(p);
            Console.Write(" ");
        }
    }
}
```

In JAVA, the iterator
- is **visible** to the programmer.
- is **managed** by the programmer.
The iterator class: data members

Saved environment and local state

class Powers implements Iterator<Integer> {
    // Saved environment:
    private final Integer base, count;
    // Saved local state:
    private Integer result = 1, i = 0;
    public PowersIterator(
        // Environment is passed with arguments
        Integer base, Integer count
    ) {
        // Save the environment in class
        this.base = base; this.count = count;
    }
}

Generator of C#

Powers(int base, int count) {
    int result = 1;
    for (int i = 0; i < count; ++i)
        yield return result *= base;
    yield break;  // optional
}

All four variables used by generator Powers of C# must be saved in an instance of class Powers in JAVA.
The iterator class: member functions

Inverted control

class Powers implements Iterator<Integer> {

    // At each iteration:
    public Integer next() {
        result *= base; // Result we shall yield
        ++i; // Continue iteration
        return result; // Yield result
    }

    // Termination test:
    public boolean hasNext() {
        return i < count;
    }

    // Historic baggage, courtesy of
    // a library design error
    public void remove() { /* empty */ }
}

Generator of C#

Powers(int base, int count) {
    int result = 1;
    for (int i = 0; i < count; ++i)
        yield return result *= base;
    yield break; // optional
}

To invert the control, one must understand when functions `next` and `hasNext` are called.
The two interfaces offered by Java for iteration

Java’s standard library offers two related notions:

**Definition (Iterable in Java)**
- something on which “iteration” is possible.
- e.g., a list, a set,
- e.g., an arithmetical progression,
- factory of Iterators

**Definition (Iterator in Java)**
- provides a service of “iteration” on an Iterable
- at each step of the “iteration”:
  - if there is a “next” item
  - yields the “next” item

Both notions are subject to parametric polymorphism (??), i.e.,

For every non-atomic type $\tau$, we have,
- type `Iterable<\tau>`
- type `Iterator<\tau>`
The complex **Iterable vs. Iterator contract**

1. **The only thing an** *Iterable* **does:**
   
   *generates* *Iterators*

2. **An** *Iterator* **is the ultimate** *disposable* **construct:**
   - provides iteration.
     - single run
     - forward only
   - associated with an *Iterable*:
     - from *birth*,
     - with *one*,
     - and only *one*

3. **An** *Iterable* **can be associated with**
   - one *Iterator*,
   - many *Iterators*,
   - or none at all.

4. **An** *Iterator* **may also remove** an item from a *Iterable*
   
   we will try to ignore this historical accident
7. Advanced abstractions

7.5. Generators with Java iterators

7.5.3. Advanced Java iterators
Java special syntax for iterating over an "Iterable"

The following

```java
Foreach syntax
for (Thing thing: things)
doSomethingWith(thing);
```

where

- **Thing** is some class
- **thing** is a newly defined variable of type **Thing**
- **things** is a "collection" of **things**, i.e., an instance of class that implements the interface **Iterable<Thing>**

is syntactic sugar for

```java
Foreach semantics
Iterator<Thing> it = things.iterator();
while (it.hasNext()) {
    Thing thing = it.next();
doSomethingWith(thing);
}
```
Definition (Pseudo generator)

A pseudo generator is a JAVA function that returns an Iterable;

When the returned Iterable is used in JAVA’s extended for, the pseudo generator looks like a generator, e.g.,

```java
// pseudo generators are almost always public static:
public static
    // a pseudo generator must return an Iterable:
    Iterable<Integer>
powers(final Integer Base, final Integer count) {
    return new POWERS(base, count);
}
```

It looks just like C# generator when used in an extended for loop:

```java
for (Integer p: powers(2,30))
    System.out.println(p);
```
Using JAVA’s syntactic sugar for foreach

To use this sugar:

Foreach syntax

```java
for (Thing thing: things)
    doSomethingWith(thing);
```

Foreach semantics

```java
Iterator<Thing> it = things.iterator();
while (it.hasNext()) {
    Thing thing = it.next();
    doSomethingWith(thing);
}
```

We must define a new class `POWERS`:

```java
class POWERS implements Iterable<Integer> {
    ...
}
```

and then write

```java
for (Integer p: new POWERS(2,30))
    System.out.println(p);
```
Less parameter juggling with inner classes

An inner class in Java “closes” over its environment; let’s use this to simplify the code

class POWERS implements Iterable<Integer> {
    final Integer base, count; // environment
    // Class constructor saves the environment
    public Powers(Integer base, Integer count) {
        this.base = base; this.count = count;
    }
    // Has access to data members base and count of enclosing class:
    class Powers implements Iterator<Integer> {
        // Saved local state:
        private Integer result = 1, i = 0;
        public Integer next() {
            result *= base;
            ++i;
            return result;
        }
        public boolean hasNext() {
            return i < count;
        }
        public void remove() { /* empty */ }
    }
    public Iterator<Integer> iterator() {
        return new Powers(base, count);
    }
}

Even less clutter with anonymous classes

An anonymous class in **Java** is

- an inner class which does not have a name
- defined where it is used

Let’s use this to simplify the code

```java
An anonymous Iterator<Integer>

class POWERS implements Iterable<Integer> {
    final Integer base, count; // Saved environment:
    public Powers(Integer base, Integer count) {
        this.base = base; this.count = count;
    }
    public Iterator<Integer> iterator() {
        return new Iterable<Integer>() {
            // Same as before; nothing new here
            public void remove() {
                // Nothing to remove
            }
        };
    }
}
```
An even closer imitation of generators

A Java function returning an instance of an anonymous class implementing `Iterable<Integer>`:

- is called pseudo generator.
- gets rid of class `POWERS`
- achieves a closer imitation of generators.

```java
public static Iterable<Integer> powers(
    // Environment variables passed as arguments
    final Integer Base,
    final Integer count
) {
    return new Iterable<Integer>() {
        public Iterator<Integer> iterator() {
            return new Iterable<Integer>() {
                // Same as before; nothing new here
            };
        }
    };
}
```

Using a pseudo generator in Java

```java
for (final Integer p: powers(2, 30))
for (final Integer q: powers(2, 30))
    System.out.println(p + q);
```
## Summary: Generators vs. iterators

<table>
<thead>
<tr>
<th></th>
<th>Generators</th>
<th>Iterators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PL</strong></td>
<td>C#, Python, Icon,...</td>
<td>Java</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>sequence of values</td>
<td>sequence of values</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>language construct</td>
<td>by clever programmer</td>
</tr>
<tr>
<td><strong>Underlying concept</strong></td>
<td>function</td>
<td>object</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>builtin</td>
<td>must be managed manually by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>programmer</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>forward</td>
<td>inverted</td>
</tr>
<tr>
<td><strong>Conceptual complexity</strong></td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td><strong>Pedagogical value</strong></td>
<td>low</td>
<td>high!</td>
</tr>
</tbody>
</table>
7. Advanced abstractions

7.5. Generators with JAVA iterators

7.5.4. Boxing primitive types
Reference types boxing type \texttt{int}

```java
class Int {
    private int inner;
    public int value() {
        return inner;
    }
    Int(int inner) {
        this.inner = inner;
    }
    public boolean lt(int than) {
        return inner < than;
    }
    public int inc() {
        return inner++;
    }
}
```
The actual interfaces
From the JAVA runtime library:

```java
public interface Iterable<T> {
    /** Returns an iterator over elements of type T. */
    Iterator<T> iterator();
}
```

```java
public interface Iterator<T> {
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```
Using **interface** **Iterator**

To use

```java
class X implements Iterator<Int> {
    ...
    ...
    public Iterator<Int> iterator(){
        ...
        ...
        return new ???();
    }
}
```

... have to create another **class**; one that **implements**...

```java
public interface Iterator<T> {
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```
Using the **Iterator interface**

```java
class Between implements Iterable<Int> {
    Between(int from, int to) {
        this.from = from;
        this.to = to;
    }

    Iterator<Int> iterator() {
        return new Iterator<Int>() {
            final Int current = new Int(from);
            public boolean hasNext() {
                return current.lt(to);
            }

            public Int next() {
                return current.inc();
            }
        };
    }
}
```

```java
private final int from, to;
```

```java
Between(int from, int to) {
    this.from = from;
    this.to = to;
}
```

```java
Iterator<Int> iterator() {
    return new Iterator<Int>() {
        final Int current = new Int(from);
        public boolean hasNext() {
            return current.lt(to);
        }

        public Int next() {
            return current.inc();
        }
    };
}
```
The entire code together

class Between implements Iterable<Int> {
    Between(int from, int to) {
        this.from = from;
        this.to = to;
    }
    private final int from, to;
    Iterator<Int> iterator() {
        return new Iterator<Int>() {
            final Int current = new Int(from);
            public boolean hasNext() {
                return current.lt(to);
            }
            public Int next() {
                return current.inc();
            }
        };
    }
}

The entire code together
7. Advanced abstractions

7.5. Generators with Java iterators

7.5.5. Unbounded generators
A generator for the $3n + 1$ sequence

The famous $3n + 1$ sequence

\[ a_{i+1} = \begin{cases} 
\frac{a_i}{2} & a_i \text{ is even} \\
3a_i + 1 & a_i \text{ is odd} 
\end{cases} \]

It is conjectured that there is no integer starting value $a_1$ for which the sequence diverges to infinity.

```java
class ThreeNPlus1 implements Iterable<Integer> {
    public static void main(String[] args) {
        for (Integer i : new ThreeNPlus1())
            System.out.println(i);
    }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            int i = 0;
            public boolean hasNext() {
                return true;
            }
            public Integer next() {
                i = (i % 2 == 0) ? i / 2 : 3 * i + 1;
                if (i <= 0) i = new Random().nextInt();
                return i;
            }
            public void remove() { /*empty*/ }
        };
    }
}
```
Refute the Goldbach conjecture with \texttt{JAVA} iterators

**Conjecture (Christian Goldbach)**

For every even integer \( n > 2 \) there exists primes \( p \) and \( q \) such that

\[ n = p + q. \]

All subsequent definitions will be made within our main class:

```java
import java.util.Iterator;
// Library type representing immutable arbitrary−precision integers:
import java.math.BigInteger;

public class NaiveNumberTheory {
    ...
}
```
The Goldbach conjecture and the halting problem

- Our `main` function terminates if and only if the Goldbach conjecture is false.
- Hence, the figuring out the Goldbach conjecture is easier than the halting problem.
- We may now understand a little better why the halting problem is so tough.

```java
public class NaiveNumberTheory {
    public static void main(String[] args) {
        System.out.println("Goldbach conjecture refuted by " + refutationOfGoldbach());
    }
}
```
Iterating over all even numbers

Our main loop:

```java
public static BigInteger refutationOfGoldbach() {
    // Iterate over the infinite sequence e = 4, 6, 8, ...
    for (BigInteger e = big(4);; e = succ2(e))
        if (refutesGoldbach(e))
            return e;
}
```

Three auxiliary functions:

```java
public static BigInteger big(long n) {
    return BigInteger.valueOf(n);
}
public static BigInteger succ(BigInteger p) {
    return p.add(BigInteger.ONE);
}
public static BigInteger succ2(BigInteger p) {
    return succ(succ(p));
}
```
Check whether an integer refutes the conjecture

```java
// Determine whether integer n can
// be written as sum of two primes:
private static boolean refutesGoldbach(BigInteger n) {
    for (BigInteger p : primes()) {
        if (p.compareTo(n) > 0)
            break;
        final BigInteger q = n.subtract(p);
        if (!isPrime(q)) continue;
        System.out.println(n + "=" + p + "+" + q);
        return true; // p and q refute Goldbach for n
    }
    return false; // No refutation found for n
}
```

primes() a pseudo generator of the infinite sequence of all primes.

compareTo defined in class BigInteger so that

\[
p . compareTo(n) = \begin{cases} 
  1 & \text{if } p > n \\
  0 & \text{if } p = n \\
  -1 & \text{if } p < n 
\end{cases}
\]  

(5.1)
Determine whether an integer is prime

- Use pseudo generator `primes()` for efficiency; no point in checking non-prime divisors
- Iterate until square root of tested number

```java
// Determine whether n is prime
private static boolean isPrime(BigInteger n) {
    // Potential divisors must be prime
    for (final BigInteger p : primes()) {
        // Stop iterating if p^2 > n
        if (p.multiply(p).compareTo(n) > 0)
            break;
        if (n.mod(p).compareTo(big(0)) == 0)
            return false; // Divisor found
    }
    // No divisor found
    return true;
}
```
Maintaining a list of previously found primes for efficiency

The cache list

```java
static class Cache {
    static final Node first = new Node(2);
    static Node last = first.append(3).append(5);
    static void extend() {
        last = last.append(nextPrime(last.prime));
    }
}
```

Cache list item

```java
static class Node {
    final BigInteger prime;
    Node next = null;
    Node(int p) { this(big(p)); }
    Node(BigInteger prime) {
        this.prime = prime;
    }
    Node append(BigInteger p) {
        return next = new Node(p);
    }
    Node append(int p) {
        return append(big(p));
    }
}
```
Extending the cache list

Function nextPrime

```java
public static BigInteger nextPrime(BigInteger n) {
    for (BigInteger $ = succ(n);; $ = succ($))
        if (isPrime($))
            return $;
}
```

Function extend

```java
static class Cache {
    ...
    static void extend() {
        last = last.append(
            nextPrime(last.prime));
    }
}
```
The **primes()** pseudo generator

Let's begin with the trivial, technical parts

```java
public static Iterable<BigInteger> primes() {
    return new Iterable<BigInteger>() {
        public Iterator<BigInteger> iterator() {
            return new Iterator<BigInteger>() {
                // There are infinitely many primes:
                public boolean hasNext() {
                    return true;
                }
                // Cannot remove primes
                public void remove() {
                    throw new UnsupportedOperationException();
                }
            };
        }
    };
}
```

The challenge is in the iterator function `next()`
Function \texttt{next()} in pseudo generator \texttt{primes()}

\textit{Iterator function next()} \textit{must}:  
- Retrieve the next prime from the cache  
- If the cache is exhausted, extend it

\textbf{Tricky recursion:}  
- To extend the cache, \texttt{nextPrime()} is called  
- \texttt{nextPrime()} uses \texttt{isPrime()}  
- \texttt{isPrime()} iterates over \texttt{primes()}  
- But to yield a prime, \texttt{nextPrime()} may need to extend the cache

\textbf{Luckily}  
- To yield a prime in the order of \(n\), you need primes until about \(\sqrt{n}\).  
- Cache was initialized with sufficiently primes to get this clockwork going

\begin{verbatim}
...  
return new Iterator<BigInteger>() {  
  ...  
  // Data member of the inner class  
  Node next = Cache.first;  
  // Yield the next prime from the cache  
  public BigInteger next() {  
    // Save value to be returned  
    final BigInteger $ = next.prime;  
    // Was cache exhausted?  
    if (next == Cache.last)  
      Cache.extend();  
    // Advance to next node of the cache  
    next = next.next;  
    return $;  
  }  
};
\end{verbatim}