6 Imperative programming

6.1 Commands (70 frs.) ........................................... 2
   6.1.1 Commands vs. expressions (16 frs.) ................ 2
   6.1.2 Recursive definitions (4 frs.) ....................... 4
   6.1.3 Expressions’ evaluation order (6 frs.) .......... 6
   6.1.4 Atomic commands (12 frs.) ......................... 8
   6.1.5 Block commands (16 frs.) ......................... 9
   6.1.6 Conditional commands ............................... 10
   6.1.7 Iterative commands ................................. 12
   6.1.8 Exercises ............................................. 14

6.2 Structured programming (11 frs.) ......................... 14

6.3 Sequencers (1 fr.) ........................................... 16

6.4 Exceptions (35 frs.) ........................................ 17
   6.4.1 Robustness (3 frs.) .................................. 17
   6.4.2 Policy I: resumption (5 frs.) ...................... 18
   6.4.3 Policy II: error rolling (6 frs.) ................. 19
   6.4.4 Policy III: setjmp/longjmp of C (1 fr.) ....... 20
   6.4.5 Policy IV: exceptions (4 frs.) .................... 21

6.5 Functional abstractions (102 frs.) ....................... 23
   6.5.1 Closures (37 frs.) .................................... 23
   6.5.2 Function objects (12 frs.) .......................... 30
   6.5.3 Generators (11 frs.) .................................. 33
   6.5.4 Iterators (17 frs.) .................................... 35
   6.5.5 Iterator examples (15 frs.) ......................... 38
   6.5.6 Coroutines (10 frs.) .................................. 42
   6.5.7 Exercises ............................................. 43

Figure 6.1: Imperative programming: a visual mindmap

6.1 Commands

6.1.1 Commands vs. expressions (10 frs.) ........... 2
6.1.2 Recursive definitions (11 frs.) .................. 4
6.1.3 Expressions’ evaluation order (16 frs.) ...... 6
6.1.4 Atomic commands (4 frs.) ........................ 8
6.1.5 Block commands (6 frs.) .......................... 9
6.1.6 Conditional commands (12 frs.) ............... 10
6.1.7 Iterative commands (11 frs.) .................. 12
6.1.8 Exercises ............................................. 14

1 No commands in purely functional languages
5. Expressions changing the program’s state?

Nasty CS101 Exam Question You are given a seemingly innocent Pascal code, and asked…

```
Procedure Hamlet;
VAR
happy: Boolean;
Function toBe: Boolean;
Begin
  happy := not happy;
toBe := happy
End;
Begin
  happy := false;
  If toBe and not toBe
  WriteLn("The Answer!");
End;
```

Could “The Answer” ever be written?

- Suppose that toBe is a function nested in procedure Hamlet,
  - which may have access to a global variable,
  - whose initial value is false,
  - In fact, function toBe returns the value of this global variable,
  - just after flipping it!
  - So, the answer is,...

6. Expressions without side-effects?

What happened here?

- Expressions do not make sense without function calls
- Functions may invoke commands
- Commands, by definition, alter the program state!
- Worse, in some PLs, certain operators have side-effects

Would it be possible to prevent side-effects at the PL design level?

- Representation of state?
- How would you do I/O?
- In general, tough, but awkward

Obvious example, pure-ML

7. “Statement-expressions” in GNU-C

An excerpt from Section 6.1 Statements and Declarations in Expressions of Chapter 6 Extensions to the C Language Family of the Gnu-C manual:

```
(( int y = foo (); int z;
  if (y > 0) z = y;
  else z = -y;
  z ; ))
```

is a valid (though slightly more complex than necessary) expression for the absolute value of foo().

Note

Gnu-C uses the misnomer “statement” instead of command
6.1.2 Recursive definitions

Frames: Expressions are recursively defined \(\Box\) Function call expression constructor \(\Box\) Commands are also recursively defined! \(\Box\) Three atomic commands in PASCAL \(\Box\) More on PASCAL’s atomic commands \(\Box\) The advent of “expression oriented languages” \(\Box\) Two kinds of atomic commands in C++ \(\Box\) Command expressions in C \(\Box\) More on atomic expressions in C \(\Box\) Two kinds of atomic commands in JAVA\(\widetilde{\Box}\) Not all JAVA expressions make commands

---

12. Expressions are recursively defined

Naturally, each PL is different, but the general scheme is:

Atomic expressions • literals
  • variable inspection

Expression constructors • Operators such as
  • “\(\Rightarrow\), …
  • Function call:

The set of atomic expressions and the constructors’ set are PL dependent, but the variety is huge.

1. Three atomic commands in Pascal

Empty Can you figure out where it hides?

Assignment As in the above,

Procedure call As in the above,

15. Three atomic commands in Pascal

(Ignoring goto, the only sequencer of the language)

Empty no change to state; no computation; no textual representation; existence determined solely by context.

Assignment

Definition 6.4 (Assignment atomic command). Let \(v\) be a variable of type \(\tau\), and let \(E\) be an expression of type \(\tau\), or of compatible type \(\tau’, \tau \leq \tau\). Then,

\[
\begin{align*}
\tau \rightarrow \tau' & \longleftrightarrow E
\end{align*}
\]

is an atomic command.

Procedure call

Definition 6.5 (Procedure call atomic Command). If \(p\) is a procedure taking arguments of types \(\tau_1, \ldots, \tau_n\), where \(n \geq 0\) and \(E_1 \in \tau_1, \ldots, E_n \in \tau_n\) are expressions, then the procedure call

\[
\begin{align*}
\tau \rightarrow \tau' & \longleftrightarrow p(E_1, \ldots, E_n)
\end{align*}
\]

is an atomic command.

16. More on Pascal’s atomic commands

Pascal sharp distinction between expressions and commands

• distinction between Function and Procedure
• distinction between expression and command

C, Java, Go,... blurred distinction:

• a procedure is a function returning Unit
• an expression is a command, more or less, and subject to PLs variety.

---

2Ignoring sequencers
3Each PL is different
4WTF? sequencers will be discussed later
5Each PL is different
18. Two kinds of atomic commands in C++

The empty Command does not change the program state; does not perform any computation; textual representation is the semicolon, i.e., “;”

Expression marked as Command An atomic command is also “an expression followed by a semicolon”, e.g.,

\[
\text{while } (*s++ = *t++) \\
\]

Empty command; no need for loop “body”; all work is carried out by the side-effects of the expression used in the loop condition

19. Command expressions in C

Definition 6.6 (Command expressions in C). If \( E \) is an expression, then \( E; \) is a command.

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>evaluates to a value</td>
<td>( f() ) + ( a + b ) : ( a - b )</td>
</tr>
<tr>
<td>Command</td>
<td>change program state (even vacuously)</td>
<td>( f() ) + ( a + b ) : ( a - b )</td>
</tr>
<tr>
<td>Variable definition</td>
<td>creates a variable and binds a name to it</td>
<td>( i = 0; )</td>
</tr>
<tr>
<td>Variable declaration</td>
<td>makes a binding; variable must be created elsewhere</td>
<td>( \text{extern int i; } )</td>
</tr>
<tr>
<td>Definition + initializer</td>
<td>creates a variable, binds a name to it, and initializes it</td>
<td>( \text{int } i = 3; )</td>
</tr>
<tr>
<td>Declaration + initializer</td>
<td></td>
<td>( X )</td>
</tr>
</tbody>
</table>

Table 6.1: C program elements

20. More on atomic expressions in C

All C’s atomic commands (including sequencers) are semicolon terminated
- Not every command includes a semicolon
- Not every semicolon is part of a command

Can you locate the atomic command(s) in this code?

```c
struct Complex {
    double x, y;
};

main() {
    struct Complex c;
    c.x = 1.0;
    c.y = 2.0;
    return 0;
}
```

21. Two kinds of atomic commands in Java

Just as in C,

; the empty command is a lonely semicolon;

Expression; provided that the first step in the recursive decomposition of expression is “something” that has (might have) side-effects:
- Function call
- Operator with side effects: Assignment e.g., \( =, +=, <<= \)...
- Increment/decrement ++ and --; either prefix or postfix.
- Object creation e.g., \( \text{new Object}() \)
- Nothing else!

22. Not all Java expressions make commands

- \( ; \)
- \( i++ \)
- \( ++i \)
- \( ++i \times \)
- \( i++, j++ \times \)
- \( ; ; \) (two commands)
- \( i = f() \)
- \( f() \)
- \( \text{new String}() \)
- \( \text{new String}() \times \)
- \( j <<= g() \)
- \( 0; \)
- \( \text{f} \) \( ; \) \text{f}() \)
- \( i++, j++ \times \)
- \( f() + 0 \) \( \times \)
- \( a ? f() : g() \times \)
- \( 1 << f() \) \( ; \)

no comma operator in JAVA
6.1.3 Expressions’ evaluation order

Frames: Evaluation order (revisiting expression & commands)
Expressions + side-effects = evaluation order dilemma
What will be printed? “Undefined behavior” in PLs
Side-effects ⇔ evaluation order question
Mere on expressions’ evaluation order
Rotate 13 implementation
Here is the bug!
Eager evaluation
Eager vs. short-circuit logical operators
Eager vs. short-circuit evaluation
Using short circuit evaluation in Bash
Emulating short-circuit operators with conditional commands
Normal evaluation order
Expressive order of normal order evaluation
Lazy evaluation order

23. 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, which means normal termination

24. 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,
most PLs prefer to refrain from making a decision.

Definition 6.7 (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`:

1. `X` is executed before `Y`
2. `X` is executed after `Y`
   1. “interleaved” execution
   2. simultaneous execution

25. 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!

26. “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.

27. Side-effects ⇔ evaluation order question

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

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

- But, even if expressions have no side-effects, consider, e.g., the evaluation of the following pure-mathematical expression over \(\mathbb{R}\):

\[
\arcsin 2 + \sqrt{-1} \times \log 0
\]

whose evaluation tree is

Figure 6.2: Evaluation tree for an expression with many faulty subexpressions

Depending on the evaluation order, each of the red nodes may trigger a runtime error first.
28. More on expressions’ evaluation order

Example: Rotate 13 Algorithm

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

Spoiler

Qnegu Inqre vf Yhrx Fxljnyxre’f sngure.

“ROT13” algorithm: add 13 or subtract 13 from all letters.

Spoiler... Revealed!

Darth Vader is Luke Skywalker’s father.

29. Rotate 13 implementation

Where’s the bug?

Pascal

PROGRAM Rotate13(Input, Output);
VAR
  rot13: Array[‘A’..’z’] of Char;
  c: char;

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;

Procedure Convert(Var c: Char);
Begin
  If c >= ‘A’
    and c <= ‘Z’
    and rot13[c] <> chr(0)
    then
    c := rot13[c]
end;

Pascal

If p <> null and p^.next <> null then ...

30. Here is the bug!

Array bounds violation!

Another annoying (and typical) case...

31. Eager evaluation

Definition 6.8 (Eager evaluation order). An eager 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, and the majority of operators

32. Eager vs. short-circuit logical operators

Definition 6.9 (Short-circuit evaluation). Short-circuit evaluation of logical operators \( \land \) and \( \lor \) prescribes that the second argument is evaluated only if the first argument does not suffice to 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>C</td>
<td>and, or</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>ML</td>
<td></td>
<td>andalso, orelse</td>
</tr>
<tr>
<td>EIFFEL, ADA</td>
<td>and, or</td>
<td>and then, or else</td>
</tr>
</tbody>
</table>

Table 6.2: Eager vs. short-circuit logical operators

33. Eager vs. short-circuit evaluation

Comparing eager and short-circuit operators:

- Same result, if there are no errors
- Same computation, if there are no side-effects

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

```c
if = fopen(fileName, "r")
  || die("Cannot_open_file\%s\n", fileName);
```
### 34. Using 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 "removing directory"
    && rmdir $f && ( [ -e $f ] && echo "...failed" || echo ""
        && mv -f $f /tmp && ( [ -e $f ] && echo "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

### 35. Emulating short-circuit operators with conditional commands

**Logical “And”**

```pseudo-JAVA
If p^ <> null then
  If p^\next <> null (* some command *)
```

**Logical “Or”**

```pseudo-JAVA
If p^ = null then (* command, possibly SKIP *)
  else if p^\next <> null then (* some command *)
```

### 36. Normal evaluation order

**Definition 6.10** (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:
  - &&
  - ||
  - ,
  - ?:  

### 37. Expressive order of normal order evaluation

Useful (but missing) for time efficient generalization of the “die” programming idiom:

**Definition of function “unless”**

```pseudo-JAVA
static <T> // exists for each type T
T unless(boolean b, T t, Exception e) {
  if (b)
    throw e;
  return t;
}
```

which can then be used to write

**Using function “unless”**

```java
final Integer a = readInteger();
final Integer b = readInteger();
final Integer c = unless(
  b == 0, // always evaluated
  a / b, // evaluated only if b != 0
  new ArithmeticException("Dividing " + a + " by 0!"))
```

### 38. Lazy evaluation order

**Definition 6.11** (Lazy evaluation order). In lazy evaluation order arguments to functions are evaluated at the first time they are used by the function.

- Used in HASKELL (the main feature which distinguishes it from ML)
- Only makes sense in PLs in which there is no “program state”
- Makes it possible to cache results, e.g., the following requires \( O(n) \) time to compute the \( n^{th} \) Fibonacci number

```haskell
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

### 39. Vanilla assignment command

**v ← e**

- Expression \( e \) is evaluated
- Its value is assigned to variable \( v \)
40. Two variation of vanilla assignment

Multiple

\[ v_1, v_2, \ldots, v_n \leftarrow e \]  
(6.4)

- Expression \( e \) is evaluated
- Its value is assigned to variables \( v_1, \ldots, v_n \)

Update

\[ v \leftarrow \varphi(v_1, e_1, e_2, \ldots, e_n) \]  
(6.5)

- syntactic sugar for
  \[ v \leftarrow \varphi(v, e_1, e_2, \ldots, e_n) \]
- as in COBOL’s
  Add 1 to a
- as in C/JAVA
  \( i++ \)
- as in C/JAVA
  \( i \leftarrow 3 \)

41. Two more varieties of the assignment command

Collateral

\[ v_1, v_2 \leftarrow e_1, e_2 \]  
(6.6)

- \( e_1 \) is evaluated and assigned to \( v_1 \)
- \( e_2 \) is evaluated and assigned to \( v_2 \)
- the two actions take place \textit{collaterally}
- cannot be used for swapping contents of variables
- theoretically possible, but not very useful

Simultaneous

\[ \langle v_1, v_2 \rangle \leftarrow \langle e_1, e_2 \rangle \]  
(6.7)

- \( e_1 \) is evaluated and then assigned to \( v_1 \) (as in collateral assignment)
- \( e_2 \) is evaluated and then assigned to \( v_2 \) (as in collateral assignment)
- the two actions take place \textit{simultaneously}
- can be used for swapping
- we had tuples of values; \( \langle v_1, v_2 \rangle \) can be thought of as a tuple of variables; simultaneous assignment can be thought of as “tuple assignment”

42. And, what about the “forgotten”
atomic commands?

The SKIP command aka \texttt{NOP}, aka \texttt{\textbackslash relax}, aka “;”, aka ...

- is not really interesting
- syntactically necessary on occasions

Procedure call command

- is not really interesting
- occurs only when procedures are distinct from functions; in most PLs, a procedure is just a function that returns \texttt{void} aka the \texttt{Unit} type.

6.1.5 Block commands

Frames:
- Sequential block constructor \( \sqcup \)
- Collateral block constructor \( \sqcap \)
- Programmatical identical vs. semantically equivalent \( \sqcap \)
- Concurrent block constructor \( \sqcap \)
- Concurrent collateral vs. concurrent collateral \( \sqcap \)
- Concurrent execution in Occam

43. Sequential block constructor

Definition 6.12 (Sequential block constructor).
If \( C_1, \ldots, C_n \) are commands, \( n \geq 0 \), then

\[ \{C_1; C_2; \ldots; C_n\} \]  
(6.8)

is a composite command, whose semantics is sequential:\( C_{i+1} \) is executed after \( C_i \) terminates.

- Most common constructor
- Makes it possible to group several commands, and use them as one, e.g., inside a conditional
- If your language has no skip command, you can use the empty sequence, \{\}. 

Separatist Approach: semicolon separates commands; used in PASCAL; mathematically clean; error-prone.

Terminist Approach: semicolon terminates commands (at least atomic commands); used in C/C++/JAVA/C# and many other PLs; does not match the above definition.

44. Collateral block constructor

Definition 6.13 (Collateral block constructor).
If \( C_1, \ldots, C_n \) are commands, \( n \geq 0 \), then

\[ \{C_1 \sim C_2 \cdots \sim C_n\} \]  
(6.9)

is a composite command, whose semantics is that \( C_1, \ldots, C_n \) are executed \textit{collaterally}.

- Very rare, yet (as we shall see) important
- Order of execution is \textit{non-deterministic}
- An optimizing compiler (or even the runtime system) can choose “best” order
- Good use of this constructor, requires the programmer to design \( C_1, \ldots, C_n \) such that, no matter what, the result is
  - programmatically identical, or
  - at least, semantically equivalent
45. Programmatically identical vs. semantically equivalent

Programmatically Identical

Now these are the generations of the sons of Noah, Shem, Ham, and Japheth: and unto them were sons born after the flood.

1. The sons of Japheth; Gomer, and Magog, and Madai, and Javan, and Tubal, and Meshech, and Tiras...
2. And the sons of Ham; Cush, and Mizraim, and Phut, and Canaan
3. The children of Shem; Elam, and Asshur, and Arphaxad, and Lud, and Aram

Definition 6.14 (Concurrent block constructor). If \( C_1, \ldots, C_n \) are commands, then
\[
\{C_1|C_2|\cdots|C_n\} \tag{6.10}
\]
is a composite command, whose semantics is that \( C_1, \ldots, C_n \) are executed concurrently.

Common in concurrent PLs, e.g., OCCAM

- Just like “collateral”...
- Commands can be executed in any order; Order of execution is non-deterministic: An optimizing compiler (or even the runtime system) can choose “best” order; Good use of this constructor, requires the programmer to design \( C_1, \ldots, C_n \); such that, no matter what, the result is, programmatically identical, or semantically equivalent

46. Concurrent execution in Occam

The cow

\[
\text{PROC \ coW(CHAN \ INT \ udder!)}
\]
\[
\text{INT \ milk: -- definitions are ':' terminated}
\]
\[
\text{SEQ}
\]
\[
\text{milk := 0}
\]
\[
\text{WHILE TRUE}
\]
\[
\text{SEQ}
\]
\[
\text{udder ! milk}
\]
\[
\text{milk := milk + 1}
\]
\[
: -- \text{end of PROC coW}
\]

The calf

\[
\text{PROC \ caLF(CHAN \ INT \ nipple?)}
\]
\[
\text{WHILE TRUE}
\]
\[
\text{INT \ milk:}
\]
\[
\text{SEQ}
\]
\[
\text{nipple ? milk}
\]
\[
: -- \text{end of PROC caLF}
\]

The cowshed

\[
\text{PROC \ cowShed()}
\]
\[
\text{CHAN \ INT \ mammaryGland:}
\]
\[
\text{PAR}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{coW(mammaryGland!)}
\]
\[
: -- \text{end of PROC cowShed}
\]

47. Collateral vs. concurrent collateral

Collateral really means “not guaranteed to be sequential”, or “undefined”; PL chooses the extent of defining this “undefined”, e.g.,

"the order of evaluation of \( a \) and \( b \) in \( a + b \) is unspecified. Also, the runtime behavior is undefined in the case \( a \) and \( b \) access the same memory".

Concurrent may be executed in parallel, which is an extent of definition of a collateral execution.

"the evaluation of \( a + b \) by executing \( a \) and \( b \) concurrently; as usual, this concurrent execution is fair and synchronous, which means that..."

48. Concurrent execution in Occam

The cow

\[
\text{PROC \ coW(CHAN \ INT \ udder!)}
\]
\[
\text{INT \ milk: -- definitions are ':' terminated}
\]
\[
\text{SEQ}
\]
\[
\text{milk := 0}
\]
\[
\text{WHILE TRUE}
\]
\[
\text{SEQ}
\]
\[
\text{udder ! milk}
\]
\[
\text{milk := milk + 1}
\]
\[
: -- \text{end of PROC coW}
\]

The calf

\[
\text{PROC \ caLF(CHAN \ INT \ nipple?)}
\]
\[
\text{WHILE TRUE}
\]
\[
\text{INT \ milk:}
\]
\[
\text{SEQ}
\]
\[
\text{nipple ? milk}
\]
\[
: -- \text{end of PROC caLF}
\]

The cowshed

\[
\text{PROC \ cowShed()}
\]
\[
\text{CHAN \ INT \ mammaryGland:}
\]
\[
\text{PAR}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{caLF(mammaryGland?)}
\]
\[
\text{coW(mammaryGland!)}
\]
\[
: -- \text{end of PROC cowShed}
\]

49. Conditional commands

Definition 6.15 (Conditional command constructor). If \( E_1, \ldots, E_n \) are boolean expressions, then
\[
\{E_1?E_1; E_2?E_2; \cdots; E_n?E_n\} \tag{6.11}
\]
is a conditional command, whose semantics can be

Sequential: Evaluate \( E_1 \), if true, then execute \( C_1 \), otherwise, recursively execute the rest, i.e., \( \{E_2?E_2; \cdots; E_n?E_n\} \).

Collateral: Evaluate \( E_1, E_2, \ldots, E_n \) collaterally. If there exists \( i \) for which \( E_i \) evaluates to true, then execute \( C_i \). If there exists more than one such \( i \), arbitrarily choose one of them.

Concurrent: Same as collateral, except that if certain \( E_i \) are slow to execute, or blocked, the particular concurrency regime, prescribes running the others.

Example of a concurrency regime

\begin{itemize}
  \item \textbf{Strong fairness:}
  \end{itemize}

\text{In any infinite run, there is no process which does not execute infinitely many times.}
50. CSP: Communicating sequential processes

Occam features a concurrent conditional command:

```
Jacob and his four wives
INT kisses:
ALT -- a list of guarded commands
rachel ? kisses
  out ! kisses
leah ? kisses
  out ! kisses
bilibah ? kisses
  out ! kisses
zilpah ? kisses
  out ! kisses
```

If none of the “guards” is ready, then the ALT commands waits, and waits, and waits.

- **Deep theory of “communicating sequential processes”**
- **ALT** is a only a small part of it
- **but we must proceed in our course...**

51. The “else” variants

**Definition 6.16** (Conditional command constructor with else clause). If $C_1, \ldots, C_n, C_{n+1}$ are commands, $n \geq 1$, and $E_1, \ldots, E_n$ are boolean expressions, then

$$
\{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n : C_{n+1}\}
$$

(6.12)

is a conditional command, whose semantics is the precisely the same as the familiar

$$
\{E_1 ? C_1 : E_2 ? C_2 : \cdots : E_n ? C_n\},
$$

where we define

$$
E_n = \neg E_1 \land \neg E_2 \land \cdots \land \neg E_{n-1}
$$

(6.13)

The “else” clause is sometimes denoted by:

- **default**
- **otherwise**

52. Variant #1 / many: the “else” clause

Almost all languages use “else”

```
If thouWiltTakeTheLeftHand
then
  iWillGoToTheRight
else
  iWillGoToTheLeft
```

PASCAL uses “Otherwise”

```
switch(c) {       case 0:      case 1: return 0;
case 2:      case 3: return 1;
default:      return isPrime(c);}
```

53. Variant #2 / #3 / many: if-then-else & cases

- Special construct for the case $n = 1$ in the form of

```
if Condition then Statement    [ else Statement ]
```

your syntax may vary

- Special construct for the case that
  - each of $E_i$ is in the form $e = c_i$
  - $e$ is an expression (usually integral), common to all $i = 1, 2, \ldots$
  - $c_i$ is a distinct constant expression for all $i = 1, 2, \ldots$

```
case Expression of { constantExpression   Statement }+ [ otherwise Statement ]
```

your syntax may vary

54. Cases with range in Pascal

```
ROT13 Filter in Pascal
Program Rot13(Input, Output);
VAR
  c:Char;
Begin
  While not eof do begin
    Read(c);
    Case c of
      'a'..'m', 'A'..'M': Write(chr(ord(c)+13));
      'n'..'z', 'N'..'Z': Write(chr(ord(c)-13));
      otherwise Write(c);
    end
  end
end.
```

A selector of PASCAL’s case statement may contain

- Multiple entries
- Range entries

55. Why special switch/case statement?

Because the PL designer thought...

- it would be used often
- it has efficient implementation on “wide-spread” machines
  - Dedicated hardware instruction in some architecture
  - Jump-table implementation
  - Binary search implementation

The above two reasons, with different weights, probably explain many features of PL.

This is probably the reason for the particular specification of conditional in the form of if-then-else for the cases $n = 1$
Efficient implementation + usability considerations = wrong conclusion?

Early versions of FORTRAN relied on a very peculiar conditional statement, namely arithmetic if

\[
\text{IF (Expression)} \ell_1, \ell_2, \ell_3
\]

where

- \( \ell_1 \) is the label to go to in case Expression is negative
- \( \ell_2 \) is the label to go to in case Expression is zero
- \( \ell_3 \) is the label to go to in case Expression is positive

could be efficient, but not very usable in modern standards

Another weird (& obsolete) conditional statement

Early versions of FORTRAN had a “computed goto” instruction

\[
\text{GO TO (} \ell_1, \ell_2, ..., \ell_n) \text{ Expression}
\]

where

- \( \ell_1 \) is the label to go to in case Expression evaluates to 1
- \( \ell_2 \) is the label to go to in case Expression evaluates to 2
- \( \ell_n \) is the label to go to in case Expression evaluates to \( n \)

likely to have efficient implementation, but not very usable in modern standards

58. Cases variants?

- Range of consecutive integer values (in PASCAL)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of JAVA after overwhelming programmers’ demand
- Regular expressions in selectors
  - Exists in BASH
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in ML and other functional PLs
  - In the spirit of the PL type system
- No cases statement
  - In EIFFEL a pure OO language
  - Language designer thought it encourages non OO mindset

59. Vanilla multi-way conditional?

- Exists in many languages, in the form of a special keyword
- elseif, or elsif or ELIF,
- e.g., in PHP you can write

```
“elseif” in PHP

if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
```

60. else if? elseif? what’s the big difference?

- There is no big difference!
- else if many levels of nesting
- elseif one nesting level
- this might have an effect on automatic indentation, but modern code formatters are typically smarter than that!
- another small difference occurs if the PL requires the else part to be wrapped within “{“ and “}”.

6.1.7 Iterative commands

Definition 6.17 (Iterative command constructor). If \( S \) is a “program state generator” and \( C \) is a command, then

\[
\text{forall } S \text{ do } C
\]

is an iterative composite command whose semantics is the (sequential / collateral / concurrent) execution of \( C \) in all program states that \( S \) generates.

Note that with “sequencers” such as break and continue, iterative commands can be even richer!

62. State generator? answer #1/5

Range of integer (ordinal) values, e.g.,

```
For i := gcd(a,b) to lcm(a,b) do
    If isPrime(i) then
        Writeln(i);
```

63. State generator? answer #2/5

The state generator \( S \) may be... Any arithmetical progression, e.g., in FORTRAN

```
Comment WHAT IS BEING COMPUTED???

INTEGER SQUARE11
SQUARE11=0
DO 1000 I = 1, 22, 2
SQUARE11 = SQUARE11 + I
1000 CONTINUE
```
The state generator \( S \) may be... Expression, typically boolean:
- expression is re-evaluated in face of the state changes made by the command \( C \);
- iteration continues until expression becomes true, or,
- until expression becomes false,

The state generator \( S \) may be... Generator, e.g., in Java

```java
List<Thing> things = new ArrayList<Thing>();
for (Thing t : things)
    System.out.println(t);
```

The state generator \( S \) may be... Cells in an array, e.g., in Java

```java
public static void main(String[] args) {
    int i = 0;
    for (String arg: args)
        System.out.println(
            "Argument " + "+i + "+i" + arg
        );
}
```

### 68. The iteration variable
Several iteration constructs, e.g., ranges and arithmetical progressions, introduce an “iteration variable” to the iteration body, e.g.,

```gawk
#!/usr/bin/gawk
BEGIN {
    antonym["big"] = "small"
    antonym["far"] = "near"
    ... for (w in antonym)
        print w, antonym[w]
}
```

```java
int[] primes = new int[100];
for (int p = 1, i = 0;
    i < primes.length; i++)
    primes[i] = p = nextPrime(p);
for (int p: primes)
    System.out.println(p);
```

### 69. Subtleties of the iteration variable
Can you make an educated guess as to what should happen in the following cases
1. the value of the expression(s) defining the range/arithmetical progression change during iteration?
2. the loop’s body tries to change this variable?
3. the value of the iteration variable is examined after the loop?

### 70. Definite vs. Indefinite Iteration
To make an educated guess, Let’s educate ourselves:

**Definite Loop** Number of iterations is known before the loop starts

**Indefinite Loop** A loop which is not definite

It is easier to optimize definite loops.
- Many PL try to provide specialized syntax for definite loops, because they perceived as more efficient and of high usability.
- Only definite loops may have collateral or concurrent semantics
- Even if a PL does not specify that loops are definite, a clever optimizing compiler may deduce that certain loops are definite, e.g.,

```c
for (int i = 0; i < 100; i++)
    ...
```

### 71. So, let’s make our guesses...

1. the value of the expression(s) defining the range/arithmetical progression change during iteration...

The iteration range, as well as the step value are computed only at the beginning of the loop. (Check the FORTRAN/Pascal manual if you are not convinced)
2. the loop’s body tries to change this variable…
   The loop body should not change the iteration variable; The PL
could either issue a compile-time error message (PASCAL), run-
time error message (JAVA), or just state that program behavior
is undefined.

3. What’s the value of the iteration variable after the
   loop?
   The iteration variable may not even exist after the loop (JAVA);
or, its value may be undefined (PASCAL).
   - the PL designer thought that programmers should not use the
     iteration variable after the loop ends
   - if the value is defined, then collateral implementation is more
difficult
   - many architectures provide a specialized CPU instructions for
     iterations;
   - the final value of the iteration variable with these instructions
     is not always the same.

6.1.8 Exercises

1. Revisit the example of using references in ML. Is \( r \) an
   L-value? An R-value? Both? None of these? Explain.

2. AWK designers chose the iteration variable to range
   over the indices of an array, instead of the values. Why
   was this the only sensible decision?

3. What’s the iteration variable of a while loop?

4. What does the acronym “CSP” stand for?

5. Write the most feature-rich class in JAVA, without
   using the semicolon character, “;”, even once.

6. How come C does not offer any rules regarding the
   iteration variable?

7. How come JAVA, despite being similar to C, recognize
   the notion of and iteration variable.

8. Explain why it is impossible to use the PERL die(...) 
   programming idiom in JAVA.

9. JAVA designers chose the iteration variable to range
   over the array cells, instead of the indices. Why was
   this the only sensible decision?

10. Revisit the example of using references in ML. Is \( ! r \) an
    L-value? An R-value? Both? None of these? Explain.

11. Could there be an iteration variables in non-definite
    loops? Explain.

12. Write a C function with at least three commands in it,
    without using the semicolon character, “;”, even once.

13. What are the circumstances in which

   prints true; concretely, add some PASCAL code around
   to make this happen.

6.2 Structured programming

---

Figure 6.3: An intriguing finite automaton computing (in a
non-sensible manner) a function that does make sense

<table>
<thead>
<tr>
<th>72. Flowcharts: another method for command composition</th>
</tr>
</thead>
</table>
| - Nodes:
  - I/O: *
    - read
    - print
    - display
    - ...
  - Controls:
    - start
    - stop
  - Empty: skip
  - assignment
  - Decision point
  - ...

  - Edges: goto

nodes are in fact atomic commands

73. Flowchart for “getting things done (GTD)”

How to process items in your incoming mailbox?
Figure 6.4: Flowchart for “getting things done (GTD)”

74. Pros & cons of flowcharts

Pros

• Very visual

• Very colorful

• Can be aesthetically pleasing

• Can be understood by almost anyone

Cons

• Do not scale

• Many competing standards

• Not necessarily planar

• Spaghetti code

• No one can really understand them

75. Challenge of understanding spaghetti code

• The program on the right does something useful!

• Many intersecting spaghetti edges

• No obvious meaningful partitioning of the chart

• Only a few nodes with one entry and one exit

• All decision nodes have two (or more) outgoing edges

• Some nodes with two incoming edges, even three!

76. Structured programming

...is a programming paradigm, characterized by

• “Three Controls”: precisely three ways for marshaling control:

  1. Sequence, e.g., \( \text{begin } C_1; C_2; \ldots; C_n \text{ end for } n \geq 0 \)

  2. Selection, e.g., \( \text{if } \ldots \text{ then } \ldots \text{ elseif } \ldots \text{ else } \ldots \text{ endif} \)

  3. Iteration, e.g., \( \text{while } \ldots \text{ do } \ldots \text{ done} \)

• Structured Control:
  – all control commands are in fact, command constructors.
  – control is marshalled through the program structure.

Theorem 6.18 (The structured programming theorem (Böhm-Jacopini, 1966)). Every flowchart graph \( G \), can be converted into an equivalent structured program, \( P(G) \).

77. Nassi-Shneiderman diagram

Main Idea Programming is like tiling the plane.

Also Called NSD, and “structograms”

Thought Of As the visual definition of structured programming

Principles:

1. every command is drawn as a rectangle

2. every command has exactly:
   • One entry point
   • One exit point

3. a command may contain other commands

4. a command may be contained in other commands

Figure 6.5: Challenge of understanding spaghetti code
Compound commands are rectangles which have smaller rectangles in them.

Each rectangle may contain one, two, or more rectangles.

Correspond to our familiar command constructors.

Color is not part of the diagram.

But we can add it anyway...

Nassi and Shneiderman did not fully work out the semantics of NSD:

- not in any formal notation;
- not in “legalese”;
- not in mock of “legalese”.

Some notation may be intriguing...

Based on an example provided by the original October 1973 “SIGPLAN Notices” article by Isaac Nassi & Ben Shneiderman.

6.3 Sequencers
83. What are sequencers?

**Definition 6.19 (Sequencers).** Sequencers are atomic commands whose execution alters the “normal” (structural) flow of control.

Examples:
- `goto` from any program point to another
- `return` to the end of an enclosing function
- `break` out of an enclosing iteration
- `continue` to the head of an enclosing iteration
- `throw` exception, that transfers control to a handler in an invoking function

6.4 Exceptions

6.4.1 Robustness (7 frs.)
6.4.2 Policy I: resumption (3 frs.)
6.4.3 Policy II: error rolling (5 frs.)
6.4.4 Policy III: `setjmp/longjmp` of C (6 frs.)
6.4.5 Policy IV: exceptions (1 fr.)
6.4.6 Kinds of exceptions (4 frs.)
6.4.7 Resource acquisition is (or isn’t) initialization (8 frs.)

84. Sometimes... it does not make sense to proceed as usual!

85. When you cannot proceed as usual...

It is tough to write robust programs; sometimes 80% of the code is dedicated to abnormal situations.

There is an “exception”!

**Bugs** The “language runtime environment” must take action, printing error messages, and enforcing graceful termination

**Unusual Environment** programmer must deal with the error; if the programmer fails to detect an “unusual environment”, then it is a bug.

86. Robust PL vs. robust programs

**Definition 6.20 (Robust PL).** The PL’s runtime system recovers gracefully from all program bugs

“the program cannot make the runtime system crash”

**Definition 6.21 (Robust program).** The program recovers gracefully even in the face of weird end-user and execution environment errors

“the user cannot make the runtime system crash”

87. The fundamental theorem of exception handling

**Theorem 6.22 (The robust programs theorem).** % No one really knows how to write robust programs

Proof. Exceptions are detected at lower level of abstraction:
- Wrong keyboard input
- Missing file
- Internet problem
- Low battery
- Out of memory
but they must be handled in higher levels of abstraction.

88. Handling exceptions

Handling must be done at a high abstraction level. Challenges include:
- **Consistency**: Deal with similar errors similarly (tough because many details of the errors are lost at higher abstraction level)
- **Coverage**: make sure that all errors are covered appropriately, and that no two dissimilar errors are grouped together for the purpose of handling. (tough, because the programmer does not always know what errors may happen at lower levels of abstraction)
- **Smart Recovery**: Sometimes, by the time the exception is caught, there is nothing useful that can be done.
- **Systematic Testing**: It is tough to systematically generate all exceptions.
Since no one knows how to do it, no one can design for it.

**Corollary 6.23** (Adequate support for exception handling). No PL provides adequate support for exception handling.

Humanity concentrates in what it does well.

**Corollary 6.24** (Graceful termination). Many PLs offer graceful termination in case of bugs.

Humanity cannot do what it does not know how to do.

**Corollary 6.25** (Raréness of robust programs). Very few programs are truly robust.

And since programmers are lazy,

**Corollary 6.26** (Input errors considered bugs). Most programs convert input errors into bugs.

---

**Example: Heron’s formula for the area of the triangle**

Figure 6.11: Hero of Alexandria (10–70 AD); also known as Heron

**Heron of Alexandria (10–70 AD)** Also known as Heron

![Heron of Alexandria](image)

Figure 6.12: A trivial triangle with edges $a$, $b$, and $c$

\[
A = \sqrt{s(s-a)(s-b)(s-c)}
\]  \hspace{1cm} (6.14)

where
\[
s = \frac{a+b+c}{2}
\]  \hspace{1cm} (6.15)

**Exceptions**

1. Cannot read $c$
2. $a < 0$
3. $b < 0$
4. $c < 0$
5. $s < 0$
6. $s - a < 0$
7. $s - b < 0$
8. $s - c < 0$

---

**Non-robust program for the area of the triangle**

Let’s start implementing it. First, essential include files

```c
#include <stdio.h>
#include <math.h>
```

Then, a function to prompt for and read real numbers:

```c
double get(const char *s) {
    double r;
    printf("Please enter %s: ", s);
    scanf("%lg", &r);
    return r;
}
```

Invalid input? End of file? We do not care, so let’s carry on…

```c
double A(double s, double sa, double sb, double sc) {
    return sqrt(s * sa * sb * sc);
}
```

Square root of a negative number? We don’t care! Proceeding…

```c
double A(double a, double b, double c) {
    double s = (a+b+c) / 2;
    return A(s, s-a, s-b, s-c);
}
```

Is either of $s - a$, $s - b$, or $s - c$ negative? We proceed…

```c
int main() {
    double a = get("A");
    double b = get("B");
    double c = get("C");
    printf("Area is: \%g\n", A(a,b,c));
    return 0;
}
```

And, what happens if the three sides do not satisfy the triangle inequality?

**Answer:**

If you check closely, you will find that this case is covered above.

---

**6.4.2 Policy I: resumption**

Frames: Resumption ☐ Pros & cons of resumption ☐ Emulating resumption in Heron’s program
92. Resumption

- The offending command continues as usual after the handler was invoked.
- Found in PL/I and Eiffel.
- In C++, function `set_new_handler()` takes as argument a pointer to a handler function for failure of the new memory allocation operator

Motivating Example: Memory Exhausted Exception
the memory exhaustion handler can:
- free some non-essential memory
- invoke the garbage collector
but, these efforts are not guaranteed to be sufficient

93. Pros & cons of resumption

Pros
- The offending command is not aware of the problem nor of the solution

Cons
- Perplexing Situation: you need the context to deal with the error, but you don’t have it.
- Hardly Ever Works: in most cases, the handler can try, but cannot promise to fix the problem
- Hardly Used: experience shows that resumption policy is not used very often even in PLs that support it

94. Emulating resumption in Heron’s program

Original: Naive Read

```c
double get(const char *s) {
    double r;
    printf("Please enter %s:\n", s);
    scanf("%lg", &r);
    return r;
}
```
- Unpredicted result in case of invalid input
- Repeated unpredictable result

Retrying the Read

```c
double get(const char *s) {
    double r;
    printf("Please enter %s:\n", s);
    do {
        scanf("%lg", &r);
        if (n < 0) { // Error or EOF
            if (errno != 0) perror(s);
            return -1; // error code
        }
        if (n == 0) // Unexpected input
            return -1; // error code
        if (r < 0)
            return -1; // error code
    } while (1 != scanf("%lg", &r));
    return r;
}
```
- loop forever on EOF.
- loop forever in case of invalid input

`scanf` does not consume unexpected characters in the input stream

Correct and useful “corrective” action?
Easier said than done!

95. Explicit error handling: returning error code

Step I: make low-level functions return error code

Reading Doubles

```c
double get(const char *s) {
    double r;
    int n;
    printf("Please enter %s:\n", s);
    n = scanf("%lg", &r);
    if (n < 0) { // Error or EOF
        if (errno != 0) perror(s);
        return -1; // error code
    }
    if (n == 0) // Unexpected input
        return -1; // error code
    if (r < 0)
        return -1; // error code
    return r;
}
```

Issues:
- Which error code to use?
- Which errors to detect?
- Should we report some errors?

96. More error code returned

Step I: another low-level function...

Computing the area

```c
double A(double s, double sa, double sb, double sc) {
    if (s < 0 || sa < 0)
        return -1; // error code
    if (sb < 0 || sc < 0)
        return -1; // error code
    return sqrt(s * sa * sb * sc);
}
```

Step II: dealing with the propagated errors

```c
int main() {
    double a = get("A");
    double b = get("B");
    double c = get("C");
    double area;
    if (a < 0 || b < 0 || c < 0) {
        fprintf(stderr, "Bad input\n");
        return 1;
    }
    area = A(a, b, c);
    if (area < 0) {
        fprintf(stderr, "Bad triangle\n");
        return 1;
    }
    printf("Area is: %g\n", area);
    return 0;
}
```

More Issues:
98. Error rolling: summary

Every procedure returns a special error code

**Assembler** usually the carry flag

**Icon** each function returns its success value in addition to its real value

**C Convention** returns 0 or negative value for integers, NaN for floating point values,

**Go Convention** procedures return a pair of values:

- the actual value, and
- error status or code.

**Pascal** and old **C** do not allow functions returning structure, so a specialized error value was easy to select

99. Summary: using the error code

The invoking procedure checks this code and tries to recover.

**Assembler** call proc; jc error

**C** if ((f = fopen(...) == 0) ... heavy responsibility on the programmer;

- always remember to test error codes;
- propagate errors up sensibly;
- recover from errors gracefully;

most programmers prove to be irresponsible

6.4.4 Policy III: setjmp/longjmp of C

**Frames:** □ How does it work? □ What’s stored in a jump buffer?
□ Step I: low-level functions call longjmp □ More low-level functions calling longjmp □ Step II: catching the longjmp □ Long jump in **C** & **Pascal**

100. How does it work?

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf b;

void g(void) {
    // will print:
    printf("g()\n");
    // jumps back to where setjmp was
    // called and saved current CPU state
    // in b making setjmp now return 1:
    longjmp(b, 1);
}

void f(void) {
    g();
    // will not print
    printf("f()\n");
}

int main() {
    // when executed first,
    // function setjmp returns 0
    if (0 == setjmp(b))
        f();
    else // when longjmp jumps back,
    // function setjmp returns 1
        // will print
        printf("main()\n");
        return 0;
}
```

101. What’s stored in a jump buffer?

- program counter
- stack pointer
- other CPU registers
- any thing else required to restored the CPU state

In file /usr/include/stdio.h on my machine, you will find something like (a more complex version of)

```c
struct jmp_buf {
    long int data[8]; // for registers probably
    int mask_was_saved; // who noz?
    unsigned long mask[16]; // mask for interrupts
};
typedef struct jmp_buf jmp_buf[1];
```

102. Step I: low-level functions call longjmp

**Reading doubles**

```c
double get(const char *s) {
    double r;
    int n;
    printf("Please enter \%s:\n", s);
    n = scanf("%lg", &r);
    if (n < 0) {
        // Error or EOF
        if (errno != 0) {
            perror(s);
            longjmp(b, 1); // exception code
        }
        longjmp(b, 2); // exception code
    }
    if (n == 0) 
        // Unexpected input
        longjmp(b, 3); // exception code
    if (r < 0) 
        // Negative value
        longjmp(b, 4); // exception code
    return r;
}
```

103. More low-level functions calling longjmp

**Computing the area**

```c
double A(double a, double sa, double sb, double sc) {
    if (a < 0) longjmp(b, 5);
    if (sa < 0) longjmp(b, 6);
    if (sb < 0) longjmp(b, 7);
    if (sc < 0) longjmp(b, 8);
    return sqrt(s * sa * sb * sc);
}
```

Issues:
• Managing and passing the jmp_buff record
• Managing exception codes
• Which errors to detect?
• Should we print an error message on some errors?
• Is it possible that we access uninitialized stack variable?

104. Step II: catching the \texttt{longjmp}

```c
int main() {
    switch (setjmp(b)) {
    case 0: {
        double a = get("A");
        double b = get("B");
        double c = get("C");
        printf("Area is: %g\n", area(a, b, c));
        return 0;
    }
    case 1: break;
    case 2: break;
    ...
    case 8: break;
    }
}
```

105. Long jump in \texttt{C} \& \texttt{Pascal}

\texttt{Pascal} can execute a \texttt{goto} to any nesting block;
- Can only \texttt{goto} to deepest occurrence of nesting block
- No fine control in recursive calls

\texttt{C} \texttt{setjmp} and \texttt{longjmp} allow to jump outside an internally invoked function

- \texttt{setjmp(b)} saves all CPU registers in buffer \texttt{b}
- \texttt{longjmp(b, v)} restores the registers from \texttt{b} and make \texttt{setjmp return v}

Lots of unsafe properties

6.4.5 Policy IV: exceptions

\texttt{Frames: Exception language construct}

106. Exception language construct

```c
#include <stdio.h>
#include <math.h>
can throw integers!

double get(const char *s) {
    double r;
    printf("Please enter %s:", s);
    if (1 == scanf("%lg", &r)) return r;
    throw 1;
}

doctor \texttt{v(double v, const char *error)} {
    if (v <= 0) throw error;
}

double A(double s, double sa,
        double sb, double sc) {
    v(a, "Circumference\ must\ be\ positive");
    v(sa, "Side_A\ must\ be\ positive");
    v(sb, "Side_B\ must\ be\ positive");
    v(sc, "Side_C\ must\ be\ positive");
    return sqrt(s * sa * sb * sc);
}
```

6.4.6 Kinds of exceptions

\texttt{Frames: Summary: policies for exception handling}

107. Summary: policies for exception handling

\texttt{Resumption Resume as usual after detection of exception}

\texttt{Explicit} Explicit error handling: every procedure returns an error code which the invoking procedure has to check

\texttt{Long Jump} moving from a low level of abstraction directly to a higher level of abstraction

\texttt{Language Construct} Exception language mechanism

\texttt{Observation}
In \texttt{C++}, you must have exceptions since constructors return no value. You cannot use explicit error handling for constructors.

108. \texttt{Exception type?}

\texttt{What Can be Thrown?}

\texttt{Pascal} The Unit type: goto to a nesting block has no associated value

\texttt{C} Integer (with \texttt{longjmp})

\texttt{C++} Any type

\texttt{Java} Subclass of \texttt{Throwable}

\texttt{Ada} Any enumerated type

\texttt{Tough Language Definition Issue}
Suppose that a function \texttt{f} may throw exceptions of types \(E_1, \ldots, E_n\). Then, should the type of \texttt{f} also include \(E_1, \ldots, E_n\) as the possible exceptions that \texttt{f} may return?

\texttt{Exceptions seem to be the right solution, but, no one knows how to do these right}
The checked exceptions dilemma

No one really knows how to type-check exceptions in a way that should be productive to the programmer.

Java tries at systematic classification of exceptions
- Function types include a list of exceptions
- But not exceptions thought to be bugs such as null pointer and division by zero
- Static type checking on compile type
- But lax checking by the JVM at runtime

C# learned Java’s bitter lesson
- Exceptions are not part of the function type
- No checked exceptions

C++ Similar to C#.
- Function types may include a list of exceptions
- If the list exists, it is checked in runtime
- But, in recent versions of C++, there is no runtime checking

Finally?

Exception Catching in Java

```java
void f(File f1, File f2) {
    try {
        PrintWriter o1 = new PrintWriter(f1)
        try {
            PrintWriter o2 = new PrintWriter(f2);
            ...
            } catch (IOException e) {
                System.out.println(f1 + "\:I/O error");
            } finally {
                o1.close();
            }
        } catch (IOException e) {
            System.out.println(f1 + "\:I/O error");
        }
    }
}
```
a bit cumbersome

Resource acquisition is initialization (RAII)

When a block goes out of scope

Java (and other language with garbage collection) Local variables wait for the GC.

Pascal (and other language which use the stack memory model) Variables allocated on the stack are reclaimed

C++ Variables allocated on the stack are destructed. This occurs in two cases:
- The block terminates normally
- An exception is thrown out of the block

Definition 6.27 (RAII). A common programming idiom, by which resources are allocated by the constructor and deallocated by the destructor.

File Reader: Build Upon C’s FILE *

```c
struct Reader {
    FILE *f;
    Reader(const char *name): f(fopen(name, \"r\")
    { if \((f) throw name; \}
    -Reader() \{ fclose(f) \}
    int eof() \{ return feof(f) \}
    int get() \{
        if (ferror(f) throw \"read\";
        return fgetc(f);
    }
};
```
The file is opened when a Reader object is created; the file is closed when this object is destructed.
The file is opened when a Writer object is created; the file is closed when this object is destroyed.

```
// Files in and out will be closed here
// (if they were opened)
```

Multiple resource allocated in a try will be automatically deallocated, and in the right order; only the resources which were allocated are deallocated; no need for an explicit close in the finally section;

```
private void f() {
    try (FileInputStream in =
        new FileInputStream("from.txt");
        FileOutputStream f =
        new FileOutputStream("to.txt");
    ) {
        String line = null;
        while ((line = in.readLine()) != null) {
            out.newLine();
            out.writeLine(line);
            out.newLine();
        }
    }
}
```

6.5 Functional abstractions

6.5.1 Closures (39 frs.) .......................... 23

6.5.2 Function objects (12 frs.) .................. 30

6.5.3 Generators (11 frs.) ......................... 33

6.5.4 Iterators (17 frs.) .......................... 35

6.5.5 Iterator examples (15 frs.) ................. 38

6.5.6 Coroutines (10 frs.) ......................... 42

6.5.7 Exercises .................................. 43

6.5.1 Closures

Frames: Context of this discussion Discrimination against function values Function values in C Nested functions in C++ Motivation: objects in ML using closures & references Computing your average grade The nest is accessible to nested functions “Inheritance” of arguments with nested functions Body of function pivot() Using a nested function outside of context Function variables & dangling references Reminder: CPU & memory in the classical model A stack frame Activation record What’s the “environment”? Acquired bindings in nested functions Name vs. entity Binding The environment: a slightly more precise definition Environment of a nested function Environment vs. scope More on semantics of the environment Intricacies in static scoping Lexical nesting does not imply call order Convoluted calls Back pointers in the quick sort example Managing static scoping with back pointers Maintaining the back pointer A simplified implementation of static scoping Dynamic scoping Semantics
119. Context of this discussion

- In discussing type constructors, we introduced the mapping type constructor; e.g., in ML we have the type \texttt{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?

120. Discrimination against function values

In Pascal

- Can define variables of almost any type, but not of type function.
- Functions can take function values as parameters, but functions cannot return function values.

121. 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 allow nested functions.

122. 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 \texttt{sqr} is nested in \texttt{isRightTriangle}
- Function \texttt{sqr} is inaccessible outside \texttt{isRightTriangle}

123. Motivation: objects in ML using closures & references

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

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

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

124. Computing your average grade

- create a new accumulator

```ocaml
val grades = makeAccumulator();
(\#add grades)(82.0);
val it = () : unit
(\#add grades)(100.0);
val it = () : unit
(\#add grades)(37.0);
val it = () : unit
(\#count grades) ();
val it = 3 : int
(\#total grades) ();
val it = 219.0 : real
(\#mean grades) ();
val it = 73.0 : real
```

- record grade in CS101

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

- record grade in PL (note that the parenthesis are optional)

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

- record grade in calculus

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

- how many grades so far?

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

- what’s their total?

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

- what’s their average?

```ocaml
#mean grades ();
val it = 73.0 : real
```
Why use nested function?

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

Nesting structure for an implementation of the quick sort algorithm using nested functions:

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

The Gnu-C compiler generates clever code that makes this possible.

```c
int pivot() {
    // "inherits" from sort: a, 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;
}
```

The nested function `swap` escapes the nest:

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

The escaped value is used in another context:

```c
Access to refugee
...
#define SIZEOF(z) (sizeof(z) / sizeof(z[0]))
int main(int argc, char **argv) {
    int a[500];
    int b[200];
    ...
    sort(b, SIZEOF(b));
    ...
    (*exchange)(10,12);
    ...
}
```

- Would this work?
- Does swapping take place in array `a` or `b`?

**Answer:** Undefined behavior due to the realization of “activation record” as a machine stack frame.

Similarly, suppose that PASCAL had function variables...
Pascal

\begin{verbatim}
VAR
  fv: Integer -> Char;
Procedure P;
  VAR
    v: ...);
Function f(n: Integer): Char;
begin
  ...v...
end;
begin
  fv := f;
end
begin
  P;
  ... fv(0) ...
end
\end{verbatim}

- 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 1st class “function values”, the activation record cannot be allocated on the hardware stack.

[fragile]Understanding the machine stack

\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}

How does this function “exist” at runtime?

**Common** sequence of bytes, representing machine code

**Per activation** the “environment”:
- Parameters
- Local variables

\begin{figure}[h]
\centering
\includegraphics[scale=0.5]{stack-frame.png}
\caption{A stack frame}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[scale=0.5]{activation-record.png}
\caption{Activation record (implemented as stack frame) for function `gcd`}
\end{figure}

A function call has two components
- The caller
- The callee

An activation record represents the interface between the two:
- Saved context 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.
Definition 6.28 (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.

135. Acquired bindings in nested functions

What’s the environment of pivot?

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

- Function sort
- Function swap
- Function qsort
- Function pivot
- Arguments to function sort
  - Array a
  - Integer n
- Arguments to function qsort
  - Integer from
  - Integer to

136. 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)

Definition 6.29 (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.

138. The environment: a slightly more precise definition

Definition 6.30 (Environment (second approximation)). The set of bindings made in 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 to labels)
- Functions and procedures defined within the caller (binding of names to functions and procedure values)

In C/C++, there is no “caller environment”

139. Environment of a nested function

What’s the environment of function pivot?

```
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.

140. 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 where the function is used.
When does \( f_2 \) “inherit” bindings made in \( f_1 \)?

**Answer I:** *(static scoping)* If \( f_2 \) is defined inside \( f_1 \),

*By induction, \( f_2 \) inherits bindings of \( f_0 \) if \( f_1 \) is defined inside \( f_0 \).*

*Environment defined by static code structure.* *(sometimes called static binding, or lexical scoping)*

**Answer II:** *(dynamic scoping)* If \( f_2 \) is called by \( f_1 \).

*By induction, \( f_2 \) inherits bindings of \( f_0 \) if \( f_1 \) is called by \( f_0 \).*

*Environment defined by program dynamics.* *(sometimes called dynamic binding)*

---

**142. Intricacies in static scoping**

**Definition 6.31 (Static 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 \( f_1 \)
- Static scoping means the *most recent version* of \( f_1 \).

---

**143. Lexical nesting does not imply call order**

**Who can call function \( f_2() \)?**

- Function \( f_2 \) itself
- Function \( f_3 \) defined in \( f_2 \)
- Function \( f_4 \) defined in \( f_3 \)
- …
- Another function \( g_1 \) defined in \( f_1 \)
- Function \( g_2 \) defined in \( g_1 \)
- Function \( g_3 \) defined in \( g_2 \)
- …

*But not function \( f_0 \) within which \( f_1 \) is defined!*
Consider the following chain of calls:

\[
\text{main}() \rightarrow \text{sort()} \rightarrow \text{qsort()} \rightarrow \text{qsort()} \rightarrow \text{qsort()} \rightarrow \text{pivot()} \rightarrow \text{swap()}
\]

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

- `main`
- `sort`
- `qsort 1`
- `qsort 2`
- `qsort 3`
- `pivot`
- `swap`

146. Managing static 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 caller

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 two hops along the BP list.
- ...

147. Maintaining the back pointer

- Let \( \mathcal{F} \in \{ f_1, f_2, f_3, \ldots, g_1, g_2, g_3, \ldots \} \) be a function ready to call \( f_2 \).
- \( \mathcal{F} \) 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 \( \mathcal{F} = f_1 \), then no hops.
- If \( \mathcal{F} = f_2 \), then one hop.
- If \( \mathcal{F} = f_3 \), then two hops.
- If \( \mathcal{F} = g_1 \), then one hop.
- If \( \mathcal{F} = g_2 \), then two hops.
- ...

148. A simplified implementation of static 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.

149. Dynamic scoping

**Definition 6.32** (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 make this material)
- Found in PERL!
- Found in BASH!
- Found in HTML/CSS!

The de-facto semantics of the C preprocessor.

150. Semantics of dynamic scoping

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

What would work?

- `sort` calling `qsort`? Yes.
- `qsort` calling `qsort`? Yes.
- `qsort` calling `pivot`? Yes.
- `main` calling `swap`? Yes.
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.

Definition 6.33 (Environment). The environment of a function, is the set of bindings available to this function.

Environment could be
- Defined statically (static scoping)
- Defined dynamically (dynamic scoping)

153. Closures

- 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).

154. Closures in JavaScript

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

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

Output:
```
10
15
20
```

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.

155. Closures and lifetime

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)

156. Closures in ML

157. 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.

6.5.2 Function objects

Frames: □ Emulation closures with C++ function objects □ Using C++’s function objects □ Closures in JAVA? □ Function objects with JAVA □ An interface for function objects □ Another use of the interface for function objects □ Using objects of type Function<T,R> □ Functions returning function objects □ Using factory functions □ Smarter function objects with inner classes □ Function objects with inner classes □ Inner class of factory function □
158. 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

159. 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

```java
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:

```java
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

160. Function objects with Java

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

Value type `double` is not 1st class; use reference type `Double` instead:

```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:

```java
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(1024.0));
  }
}
```

Log base 2 of 1024 is 10.0

161. An interface for function objects

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

```java
interface Function {
  int apply(int v);
}
```

```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 work on atomic types such as `int` and `double`
- Use instead equivalent reference types: `Integer`, `Double`,...

162. 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 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:

```java
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(1024.0));
  }
}
```

Log base 2 of 1024 is 10.0

163. Using objects of type `Function<T,R>`

Storing function objects in variables:
Function<Integer, String> inRoman = new IntMap(
    new String[] {
        "i", "ii", "iii", "iv", "v"
    });
Function<Integer, String> inEnglish = new IntMap(
    "one", "two", "three",
); // Constructor is private!

Applying function objects:

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

164. Functions returning function objects

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

Class with factory method

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

165. Using factory functions

Creating two function objects:

IntMap inLatin = mapping(
    new String[] {
        "i", "ii", "iii", "iv", "v"
    });

IntMap inEnglish = mapping(
    new String[] {
        "one", "two", "three",
    });

Using the returned function objects:

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

166. Smarter function objects with inner classes

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

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

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 one 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'',
  – ...

167. Function objects with inner classes

The trick: Use inner classes; inner classes do close on the environment:
A JAVA Local Class

```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

168. Inner class of factory function

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

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

```java
public // should be public to be useful
static // factory methods tend to be static
Function<Integer,String> // Abstract function type
mapping(String[] map) {
    class IntMap // Local class
        implements Function<Integer,String> {
            public String apply(Integer v) {
                ... // as beforee
            }
        }
    return new IntMap();
}
```

Look mom! No hands!

Inner class IntMap does not have a map data member!

6.5.3 Generators

Frames:  A simple PYTHON generator  A generator built on top of another  Functions vs. generators in PYTHON  Use of functions & generators in PYTHON  What’s a generator?  Generators in ICON  Comparisons generators  Control with generators  ICON implements control using generators  Generators in C#  Summary of generators

169. 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 31st, **23:59:50**:

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

we shall obtain

10 ... 9 ... 8 ... 7 ... 6 ... 5 ... 4 ... 3 ... 2 ... 1 ... 0 ... Happy new year! X! X! X!

170. 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,
```

Output:

3 7 11 15 19

171. Functions vs. generators in Python

PYTHON does not make much distinction between functions and generators

```python
% python
Python 2.7.8 (default, Oct 20 2014, 15:05:19)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> range
<built-in function range>
```

```python
>>> def fu(n):
...     return n
... >>> fu
<function fu at 0x7fc0e82f47d0>
```

```python
>>> def countdown(n):
...     while n >= 0:
...         yield n
...         n -= 1
... >>> countdown
<function countdown at 0x7fc0e82f4848>
```

172. Use of functions & generators in Python
173. What’s a generator?

Definition 6.34 (Generator). A generator is a function that produces a (possibly empty, possibly infinite) sequence of results, instead of returning a single result.

174. 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}

“Product” of sequences (1|2|3)*f() + (1|2) * g() is a generator yielding $6 \times n \times m$ where

- $n$ is the number of values that $f$ yields
- $m$ is the number of values that $f$ yields

175. Comparisons generators

Comparison generators simplify 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 the value to which $i$ equals.
- $0 <= i < n == m$ yields $\{m\}$ if the condition holds:
  - $0 <= i$ yields either $\{i\}$ or $\{\}$
  - $0 <= i < n$ yields either $\{n\}$ or $\{\}$
  - $0 <= i < n == m$ yields either $\{m\}$ or $\{\}$

176. Control with generators

Iteration and conditional are both generating contexts:

- if statement:
  - Execute body if a value is yielded
  - Only takes on the first first yielded value
- while statement:
  - Execute body for each value yielded
  - Uses all yielded values

177. Icon implements control using generators

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}

“Product” of sequences (1|2|3)*f() + (1|2) * g() is a generator yielding $6 \times n \times m$ where

- $n$ is the number of values that $f$ yields
- $m$ is the number of values that $f$ yields

178. 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>
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.

### 6.5.4 Iterators

**Frames:** □ Generators vs. iterators □ The “magic” behind a generating context □ Nested generation □ Distinct generating record for each generation context □ Emulating generators with JAVA iterators □ JAVA iterators: poor man’s generators □ Invoking the iterator □ The iterator class: data members □ The iterator class: member functions □ JAVA special syntax for iterating over an “Iterable” □ The two interfaces offered by JAVA for iteration □ Using JAVA’s syntactic sugar for foreach □ Pseudo generator □ The complex Iterator contract □ Less parameter juggling with inner classes □ Even less clutter with anonymous classes □ An even closer imitation of generators

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

Will print twice each integer \( n \), if \( n \) can be written as

\[
   n = 2^p + 2^q
\]

where

\[
   1 \leq p \leq 30
\]

and

\[
   1 \leq q \leq 30
\]

### 180. Generators vs. iterators

<table>
<thead>
<tr>
<th>PL</th>
<th>C#, Python, ICON,…</th>
<th>JAVA</th>
</tr>
</thead>
<tbody>
<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 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>

Table 6.3: Generators vs. iterators
Nested generation: behind the scenes

```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("\n");
    }
}
```

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("\n");
        }
    }
}
```

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);
        }
    }
}
```

In JAVA, the iterator

- is visible to the programmer.
- is managed by the programmer.

184. Emulating generators with Java iterators

Remember that JAVA

- 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

185. 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.

186. Invoking the iterator

```java
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) {
        this.base = base; this.count = count;
    }
    // Save the environment in class variables:
    this.base = base; this.count = count;
}
```

All four variables used by generator `Powers` of C# must be saved in an instance of class `Powers` in JAVA.

187. The iterator class: data members

Generator of C#

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

Saved environment and local state

```java
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) {
        this.base = base; this.count = count;
    }
    // Save the environment in class variables:
    this.base = base; this.count = count;
}
```

188. The iterator class: member functions
### Generator of C#

```csharp
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
}
```

### Inverted control

```java
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 */
    }
}
```

To invert the control, one must understand when functions **next** and **hasNext** are called.

#### 189. 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 **thing**, i.e., an instance of class that implements the interface **Iterator<Thing>** is syntactic sugar for

```java
Foreach semantics
Iterator<Thing> it = things.iterator();
while (it.hasNext())
    doSomethingWith(it.next());
```

#### 190. The two interfaces offered by Java for iteration

JAVAs standard library offers two related notions:

**Definition 6.35 (Iterable in JAVA).**

- something on which “iteration” is possible.
- e.g., a list, a set,
- e.g., an arithmetical progression,
- factory of **Iterators**

**Definition 6.36 (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.,

```java
for every non-atomic type τ, we have,
- type **Iterable<τ>**
- type **Iterator<τ>**
```

#### 191. Using Java’s syntactic sugar for foreach

To use this sugar:

```java
class POWERS implementsIterable<Integer> {
    ...
}
and then write
```

```java
for (Integer p: new POWERS(2,30))
    System.out.println(p);
```

#### 192. Pseudo generator

**Definition 6.37 (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
A pseudo generator
// pseudo generators are almost always public static:
public static // a pseudo generator must return an Iterable:
    Iterable<Integer>
powers(Integer Base, Integer count) {
        return new POWERS(base, count);
    }
```

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

```java
193. The complex Iterable vs. Iterator contract
1. The only thing an **Iterable** does:
   generates **Iterators**.
```

```java
Using a pseudo generator
for (Integer p: powers(2,30))
    System.out.println(p);
```
2. An **Iterator** is the ultimate *one-time-use* 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

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


An anonymous **Iterator<Integer>**

```java
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 Iterator<Integer>() {
            // Same as before; nothing new here
        };
    }
}
```

**194. Less parameter juggling with inner classes**

195. **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
class POWERS implements Iterable<Integer> {
    final Integer base, count; // Saved environment:
    // Constructor saves the environment
    public POWERS(Integer base, Integer count) {
        this.base = base; this.count = count;
    }
    // Can access base and count of enclosing class
    public Iterator<Integer> iterator() {
        if (i < count) {
            result *= base;
            ++i;
            return result;
        }
        return new Iterator<Integer>() {
            public boolean hasNext() { return i < count; }
            public void remove() { /* empty */ }
        };
    }
    public Iterator<Integer> iterator() {
        return new POWERS(base, count);
    }
}
```

**196. An even closer imitation of generators**

A JAVA function returning **anonymous Iterable<Integer>**:

- is a pseudo generator.
- gets rid of class **POWERS**
- obtains a closer imitation of generators.

**Function powers returning anonymous Iterable<Integer>**

```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 Iterator<Integer>() {
                // Same as before; nothing new here
            };
        }
    };
}
```

**Using a pseudo generator in JAVA**

```java
for (final Integer p: powers(2, 30))
    System.out.println(p);
```

**6.5.5 Iterator examples**

**Frames:**

- Reference types for type `int`
- The actual interfaces
- Using `interface` `Iterator` `Using the Iterator interface`
- Here is the entire code
- A generator for the $3n+1$ sequence
- Refute the Goldbach conjecture with JAVA iterators
- The Goldbach conjecture
- The halting problem
- Iterating over all even numbers
- Check whether an integer refutes the conjecture
- Determine whether an integer is prime
- Maintaining a list of previously found primes for efficiency
- Extending the cache list
- The `primes()` pseudo generator
- Function `next()` in pseudo generator `primes()`

**197. Reference types for type `int`**
198. 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<E> {
  boolean hasNext();
  E next();
  default void remove() {
    throw new UnsupportedOperationException("remove");
  }
}
```

199. Using interface Iterator

To use `interface Iterator`, create a `class` that implements it...

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

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

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

200. Using the Iterator interface

201. Here is the entire code

```java
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();
      }
    };
  }
}
```

202. A generator for the $3n+1$ sequence

The famous $3n+1$ sequence

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

It is conjectured that there is no integer starting value $a_1$ for which the sequence diverges to infinity.
Definition 6.38 (The Goldbach conjecture). 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 {

    public static void main(String[] args) {
        System.out.println("Goldbach
conjecture refuted by " + refutationOfGoldbach());
    }
}
```

203. Refute the Goldbach conjecture with Java iteraters

204. 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.

205. Iterating over all even numbers

Our main loop:

```java
public class NaiveNumberTheory {
    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 class NaiveNumberTheory {
    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));
    }
}
```

206. 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 false;
    }
    return true;
}
```

207. 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
// 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;
        else if (n.mod(p).compareTo(big(0)) == 0)
            return false; // Divisor found
    }
    // No divisor found
    return true;
}

208. Maintaining a list of previously found primes for efficiency

Figure 6.21: Cache of primes (initial state)

Class Node

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));
    }
}

Figure 6.22: Reminder: initial state of the cache of primes

Class Cache

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));
    }
}

210. The \( \text{primes()} \) pseudo generator

Let’s begin with the trivial, technical parts

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("remove");
                }
            };
        }
    };
}

The challenge is in the iterator function \( \text{next()} \)

211. Function \( \text{next()} \) in pseudo generator \( \text{primes()} \)

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

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

```java
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 $;
    }
};
```

6.5.6 Coroutines

**Frames:** □ What are coroutines? □ Unconvincing example of coroutines □ Issues with the pigsty □ Coroutines vs. threads □ The sad story of coroutines □ Killer application of coroutines □ Continuations □ Data stored in a continuation □ Operations on a continuation □ Things you can do with continuations:

**Definition 6.39 (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

(coincidentally, two overloaded meanings of the word “yield” are “to produce” and “to surrender to another the physical control of something”)

**213. Unconvincing example of coroutines**

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

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

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

If `piglet2` is not hungry

- **cow** may die
- **piglet1** will then die
- **piglet2** himself will eventually die

**214. Issues with the pigsty**

It is not clear at all

- how the coroutines are started?
- how each coroutine gains access to the other?
- how can one create multiple instances of the same coroutine? (generate, e.g., several active invocation of `piglet2()`)
- why coroutines are better than plain `threads`?

So, why the heck do we need these?

**215. Coroutines vs. threads**

Both offer multi-tasking:

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

**Coroutine** cooperative; control is

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.

**216. 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:

```csh
void mainWindow() {
  for (;;) {
    Message m = getMessage();
    Area a = findScreenArea(a);
    Windows ws = findWindows(as);
    push(m);
    for (Window w: ws) {
      yield(w);
      if (!there(m))
        break;
    }
  }
}
```

```csh
void anyWindow() {
  for (;;) {
    switch (Message m = getMessage()) {
      case M1: ...
      case M2: ...
      ...
      default:
        push(m)
    }
    yield mainWindow;
  }
}
```

Useful in browser applications such as Gmail

Continuations

Quite an obscure and abstract definition:

**Definition 6.40 (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 a continuation

\[ c \leftarrow \text{start}(f(\ldots)) \] create a new continuation for the call \( f(\ldots) \)

\[ c' \leftarrow \text{clone}(c) \] save execution state for later

\text{resumable}(c) \] determine whether resumption is possible.

\[ c \leftarrow \text{resume}(c) \] resume execution of continuation \( c \)

\text{retrievable}(c) \] determine whether computation has generated a result

\[ r \leftarrow \text{retrieve}(c) \] obtain the pending returned (yielded) value of \( c \)

In fact, continuations are nothing but "generation records" with slightly more operations.

Things you can do 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)

6.5.7 Exercises

1. Which items stored in activation records are not part of the data stored with continuations?
2. How are closures different than anonymous functions?
3. How would you argue that C supports first class functions?
4. What’s non-preemptive multitasking? How is it related to coroutines? Which is more general?
5. Why is C++ forced to support nested functions?
6. Enumerate the data items that continuations have to store.
7. What’s the difference between static- and non-static nested classes?
8. What’s the difference between generators and iterators?
9. Enumerate all operations allowed on first-class functions.
10. Can you use Java’s nested classes to implement anonymous functions?
11. Discuss the restrictions placed on nested classes in C++.
12. How are closures different than lambda functions?
13. What’s the difference, if any, between Java’s local classes, and non-static nested classes?
14. Discuss the restrictions placed on anonymous functions in Gnu-C.

15. Given is a language which only has generators, can you use these to implement coroutines?

16. What’s the etymology of the name “lambda” functions?

17. How are C++ nested classes different than those of JAVA?

18. Why do continuations require GC?

19. Why does PASCAL refuse to give equal rights to functions?

20. Can closures be implemented with activation records? Explain.

21. What are the two meanings of the word “yield”? How are these meaning used in generators and

22. Which stored in activation records are not part of the data stored with closures?

23. Explain why C does not have nested functions.

24. Are data items stored with activation records different in C and in PASCAL?

25. Enumerate the data items stored in an activation record?

26. Are nested functions in C++ first class?

27. Enumerate the data items that closures have to store.

28. Can you use JAVA’s nested classes to implement closures?

29. When are closures planned for JAVA? What’s the proposed syntax? Any restrictions on the implementation?

30. Why do closures require GC?

31. Which data items stored with closures do not occur in activation records?

32. How does JAVA implement generators?

33. What information is contained in coroutines and is missing in closures?

34. When are closures planned for C++? What’s the proposed syntax? Any restrictions on the implementation?

35. Can one emulate closures with coroutines? Explain.

36. Are goroutines a kind of coroutines?

37. In C the calling function is responsible for removing the pushed arguments from the stack. Why is this the only reasonable implementation?

38. Describe the typical situation in which you want your closure to be anonymous?

39. Can you use JAVA’s nested classes to implement coroutines? If yes, explain how; if no, explain why not.

40. Experts claim that all “implementation of all web services are poor-man’s continuations”. Explain and elaborate.

41. Can coroutines be implemented with closures? Explain.

42. Explain why letting the called function to remove the arguments from the stack is more efficient than having the caller do so.

43. What’s the difference between coroutines and generators?

44. Does JAVA make distinction between generators and iterators? How?