CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 4 Programming Paradigm – Imperative Programming

LECTURE 4: IMPERATIVE PROGRAMMING

DR. ERIC CHOU

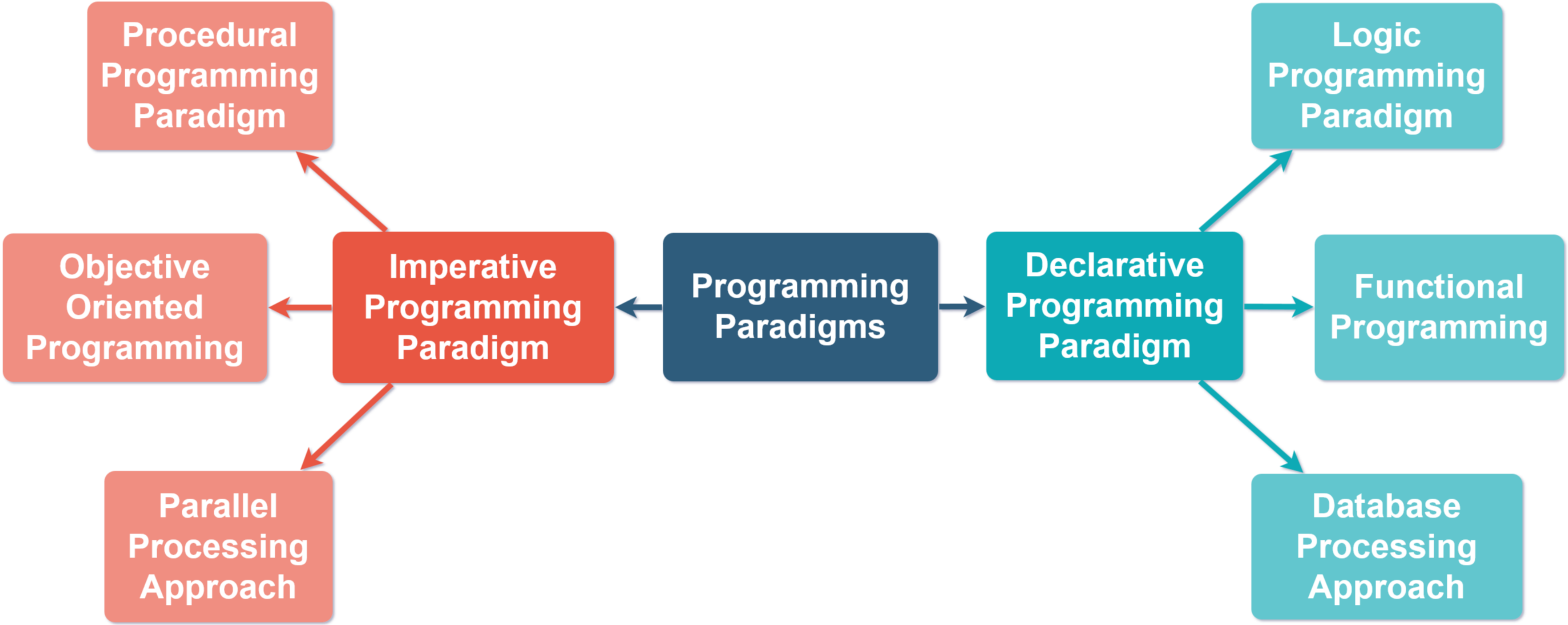
IEEE SENIOR MEMBER

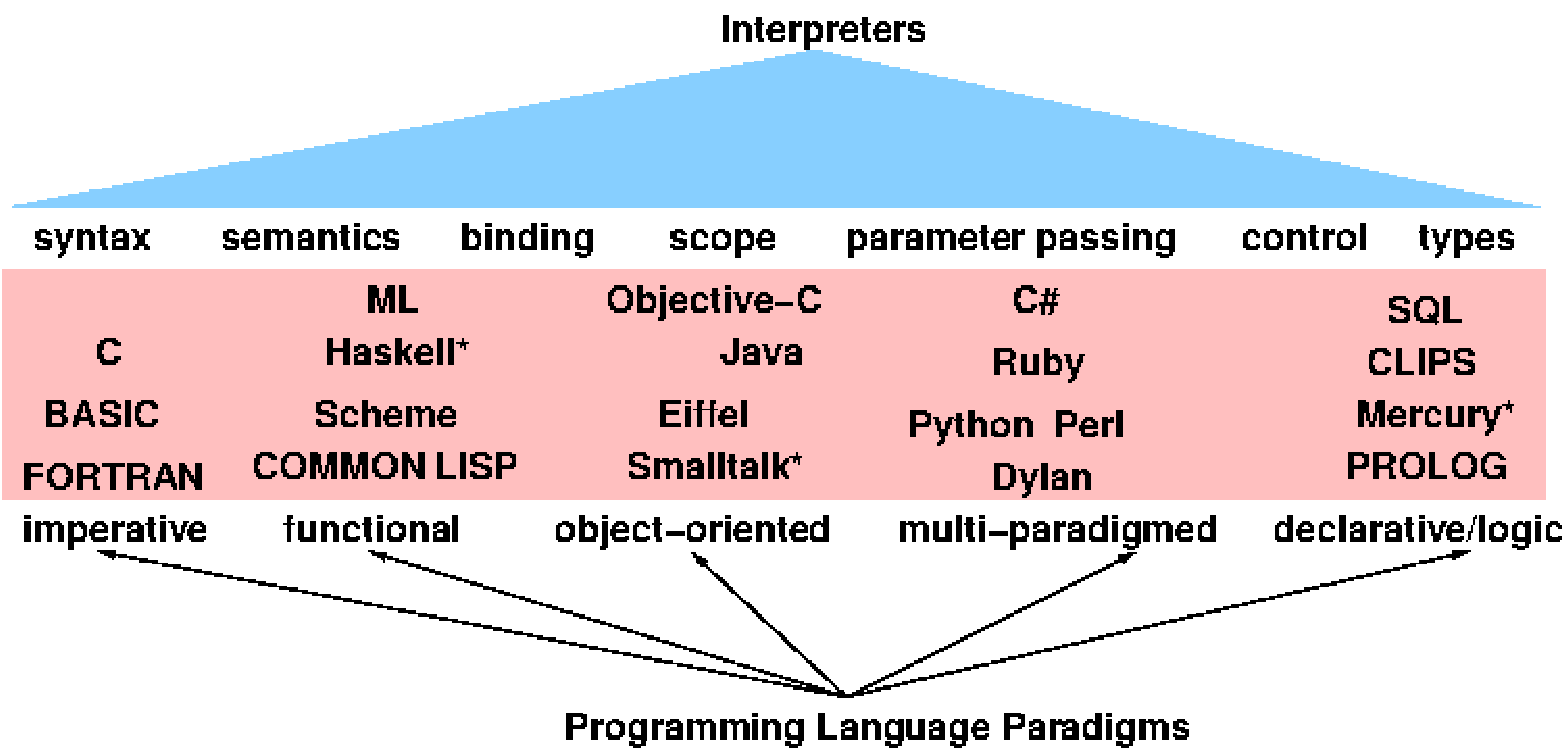
Objectives

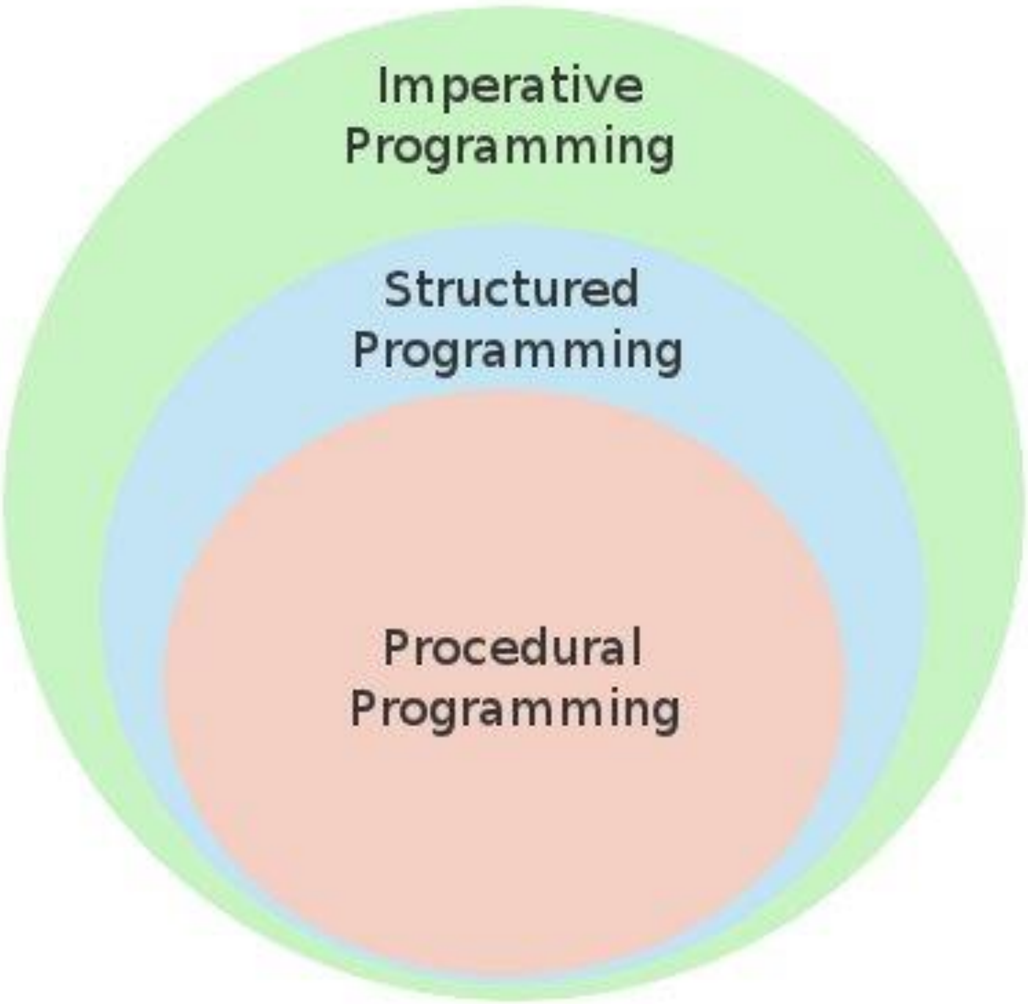
- Program Design Paradigms
- Imperative Programming Paradigm
- Structured Programming
- Stack Diagram
- Call and Return Sequences

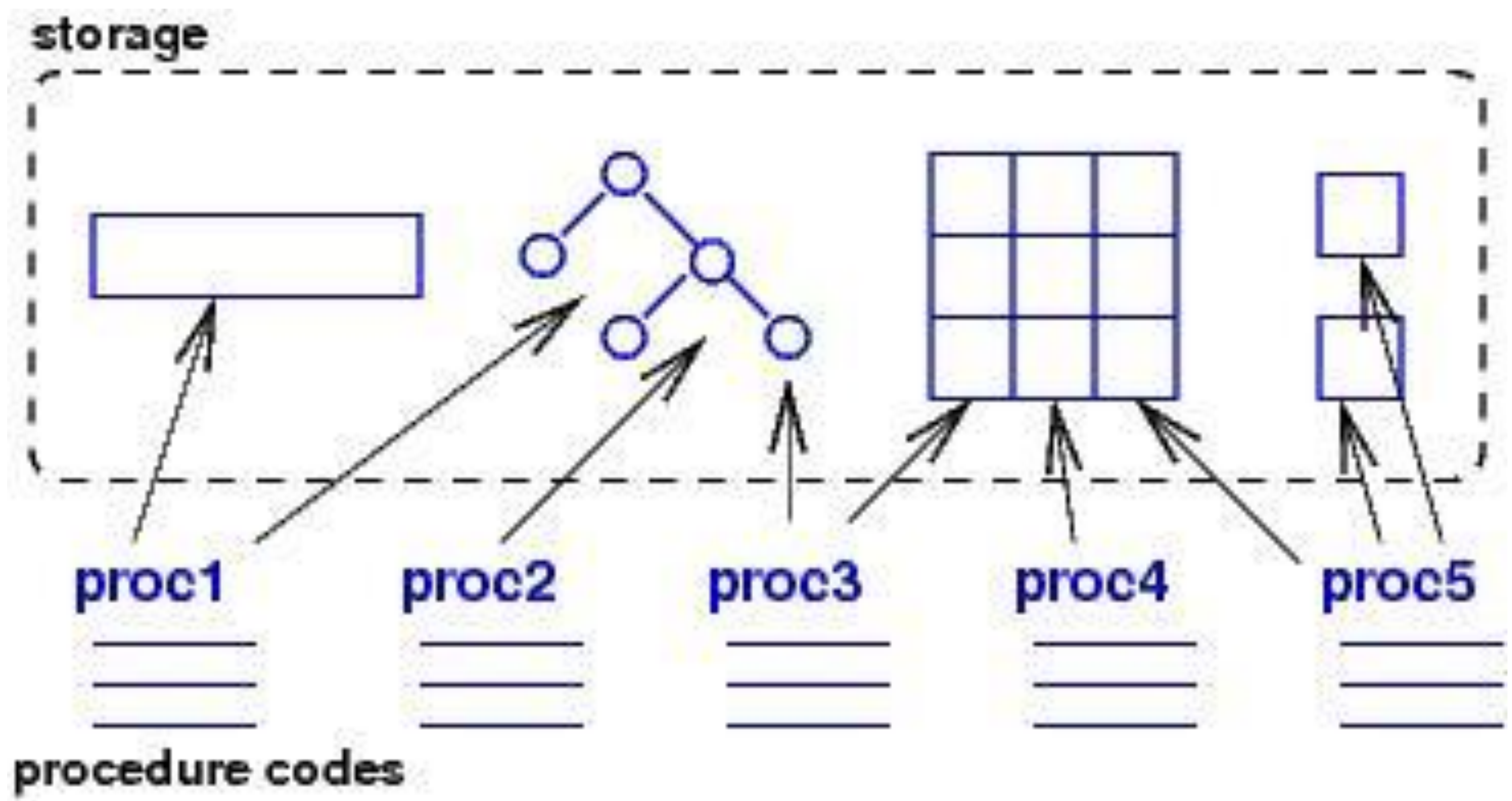
Programming Paradigms

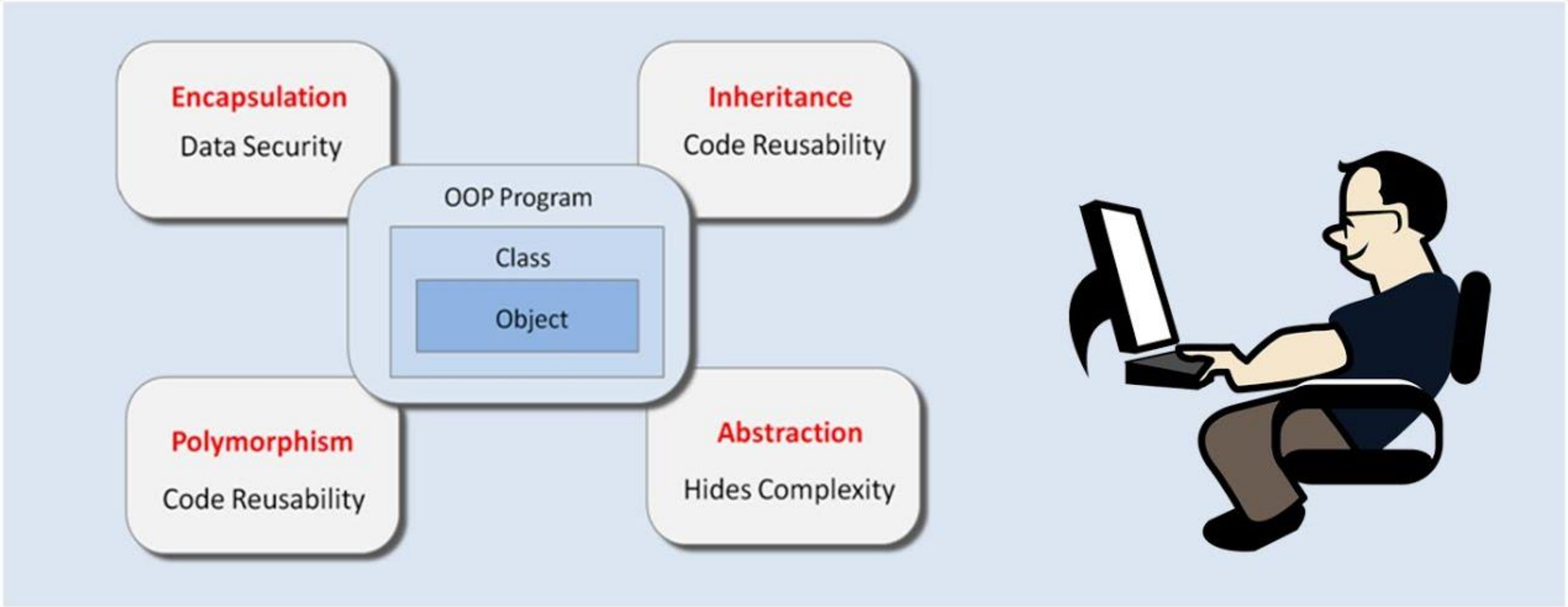
SECTION 1


Procedural Programming Paradigm
Objective Oriented Programming
Imperative Programming Paradigm
Parallel Processing Approach
Programming Paradigms
Declarative Programming Paradigm
Logic Programming Paradigm
Functional Programming
Database Processing Approach

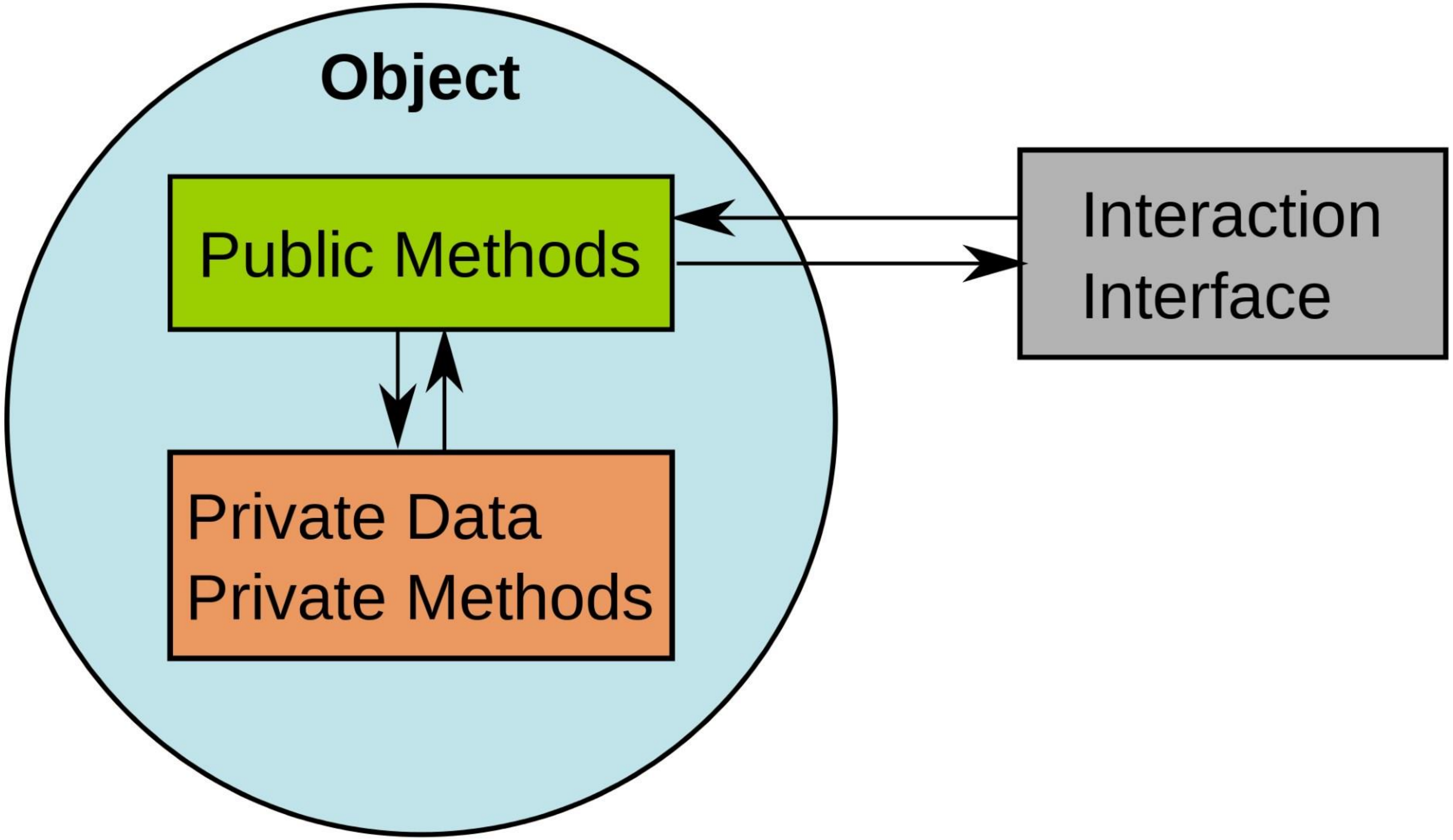

Interpreters
syntax
semantics
binding
scope
parameter passing
control
types
C
BASIC
FORTRAN
ML
Haskell*
Scheme
COMMON LISP
Objective-C
Java
Eiffel
Smalltalk*
C#
Ruby
Python Perl
Dylan
SQL
CLIPS
Mercury*
PROLOG
imperative
functional
object-oriented
multi-paradigmed
declarative/logic
Programming Language Paradigms


Imperative
Programming
Structured
Programming
Procedural
Programming

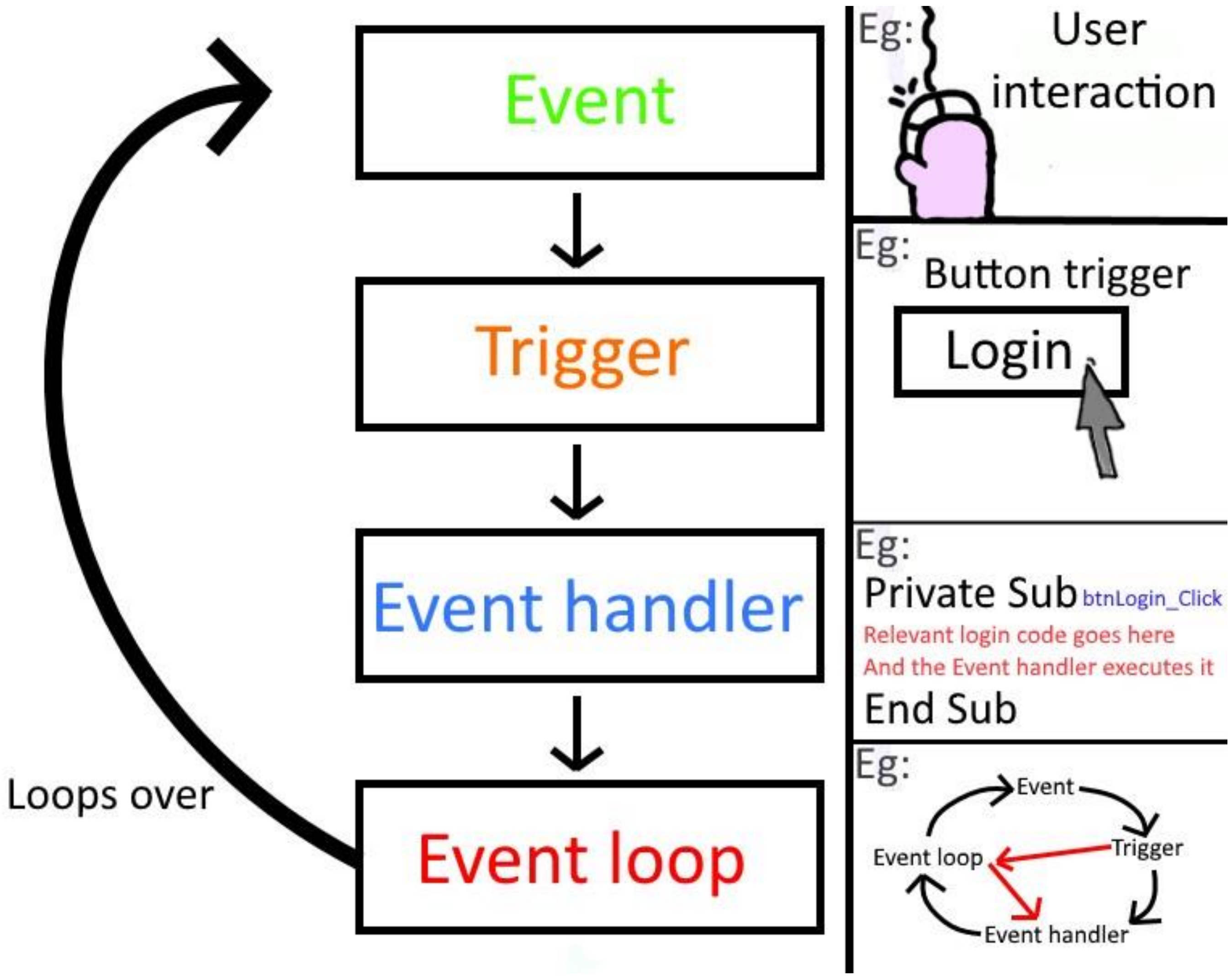

storage
proc1
proc2
proc3
proc4
proc5
procedure codes



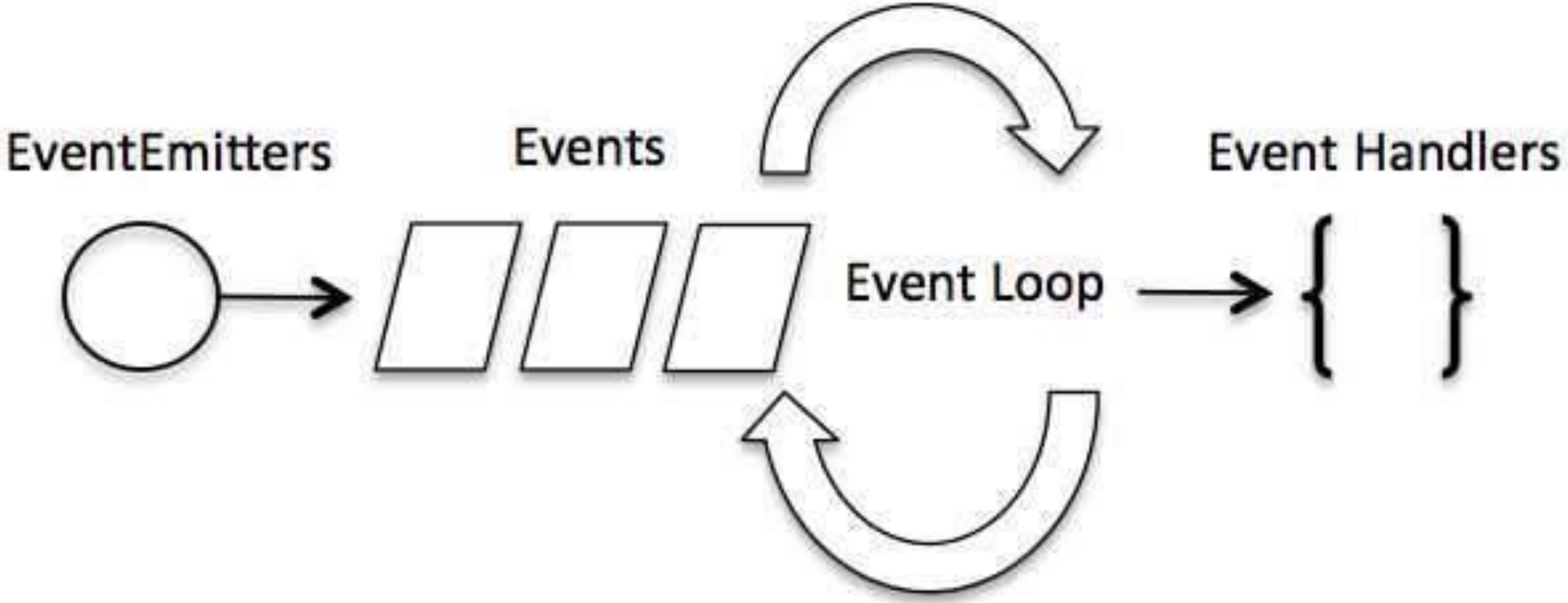
What Is Object Oriented Programming ?



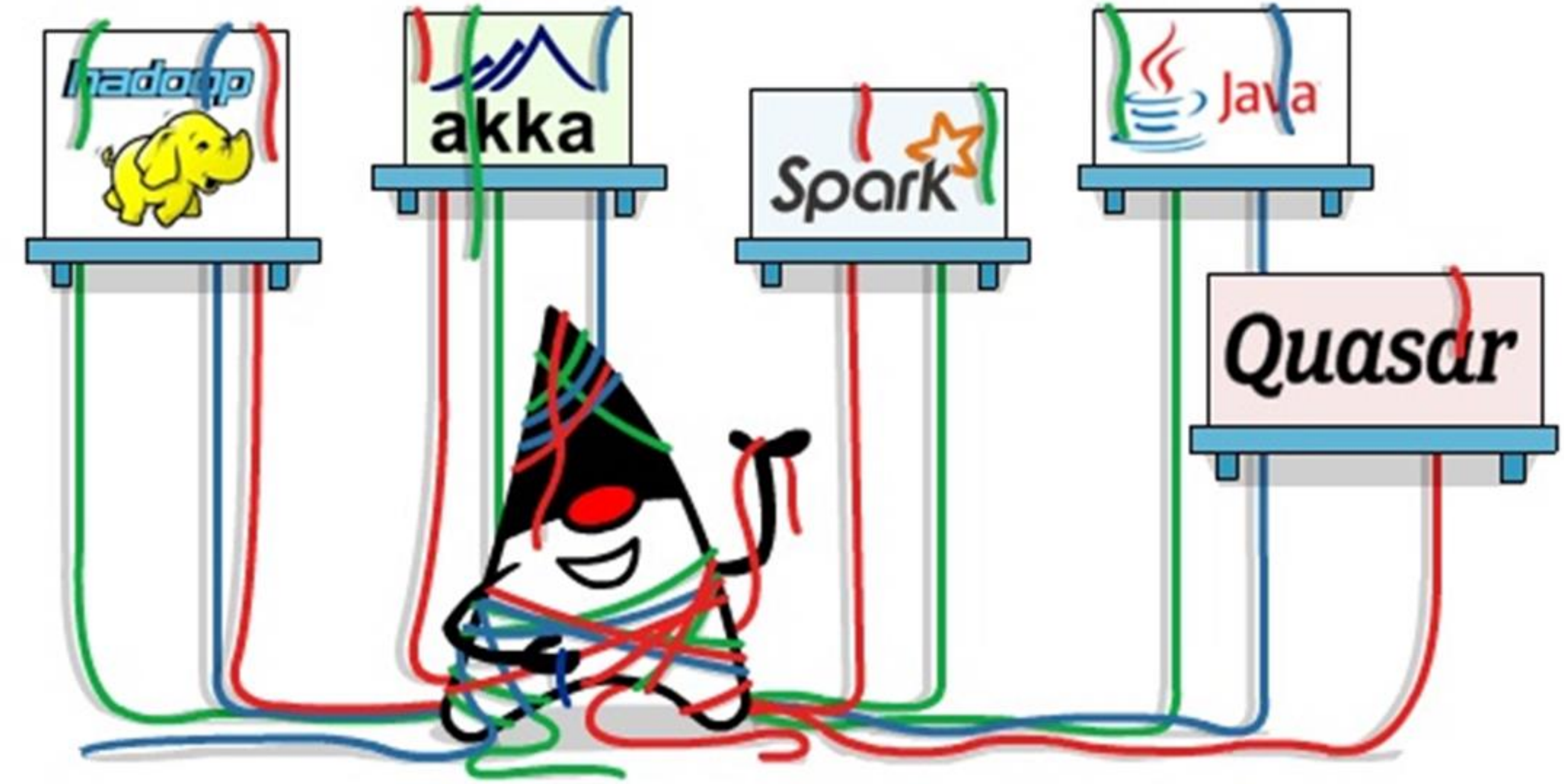
Object-Oriented Programming



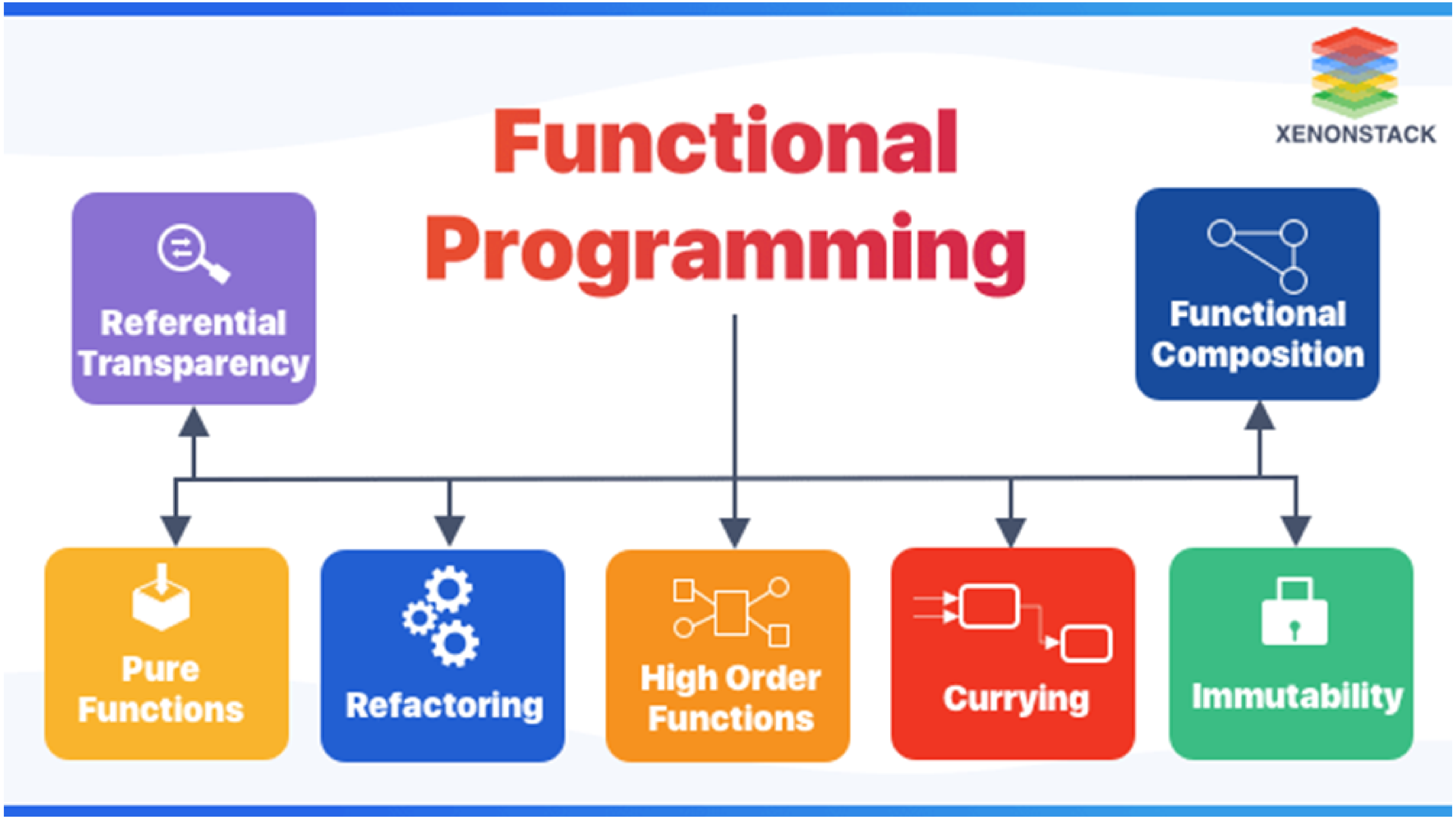
Event-Driven Programming



Event-Driven Programming



Multithreading/Multiprocessing


Functional Programming
XENONSTACK
Referential Transparency
Functional Composition
Pure Functions
Refactoring
High Order Functions
Currying
Immutability

Imperative Programming Paradigm

SECTION 2

Declarative and Imperative

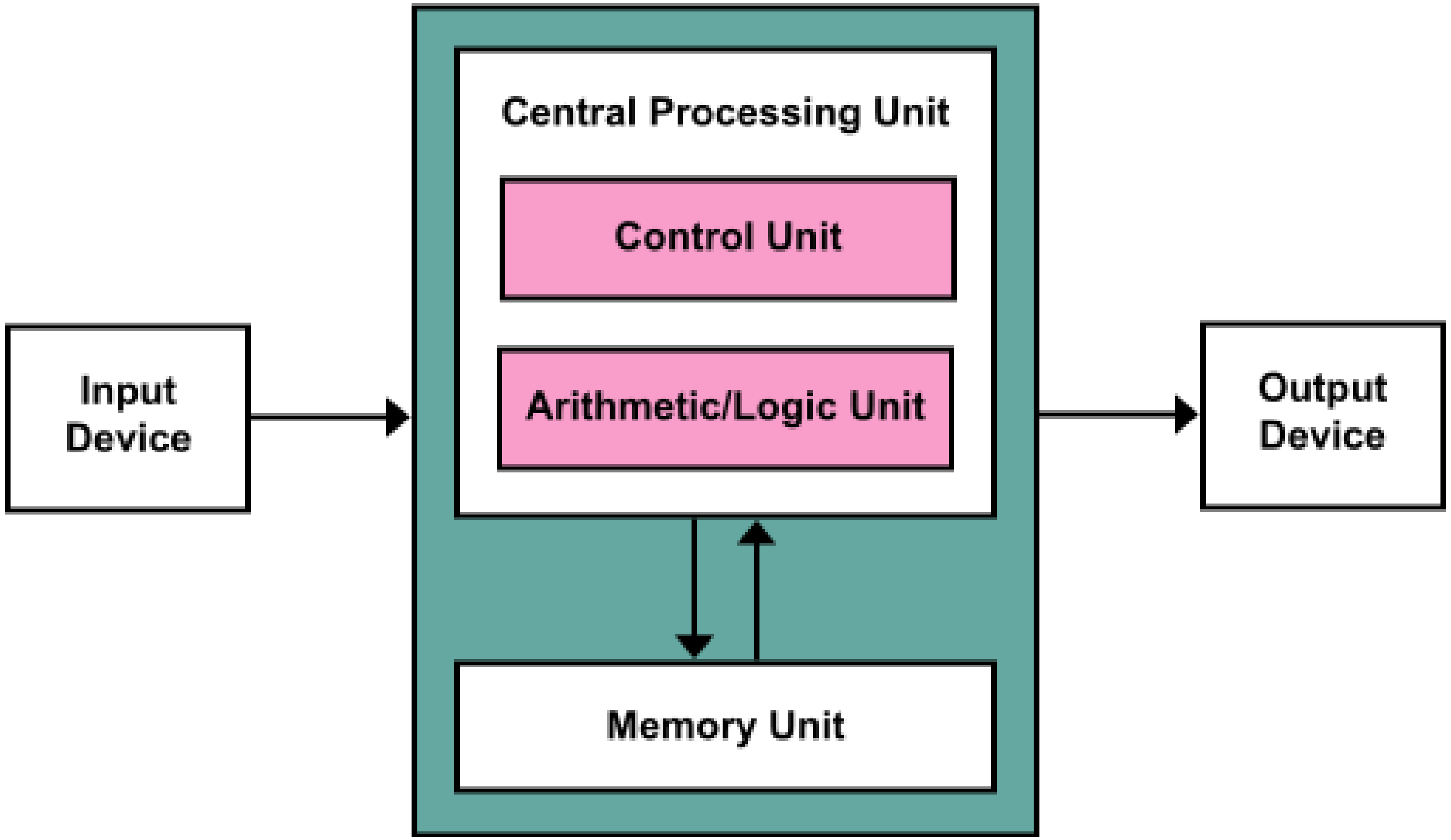
Declarative

- What not how
- Language can figure out how when you tell it what
- No side effect
- No Mutatable variables
- Express data flow

Imperative

- Commands manipulate state of system and variables
- Many side effects
- Mutatable variables
- Control flow

Imperative Programming Paradigm



Sequential Programming

SECTION 3

Control Flow

Basic paradigms for control flow

- **Goto’s (Branch, Jump)**
- **Sequencing**
- Selection
- Iteration
- Procedural Abstraction
- Recursion
- Concurrency
- Exception Handling and Speculation
- Non-determinacy

Unstructured Control Flow

Assembly, COBOL, Fortran

- Sequencing
- Goto’s (Branch, and Jump)
- Code Section or Segment (COBOL)

Unstructured Control Flow

- Unstructured control flow: the use of goto statements and statement labels to implement control flow
  - Generally considered bad
  - Most can be replaced with structures with some exceptions
    - Break from a nested loop (e.g. with an exception condition)
    - Return from multiple routine calls
  - Java has no goto statement (supports labeled loops and breaks)
- Language Feature to support unstructured control flow: Sequencing, Branch on Condition, and Goto Labels.

Sequencing

The execution of statements and evaluation of expressions is usually in the order in which they appear in a program text.

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neumann
- A compound statement is a delimited list of statements
  - A compound statement is called a block when it includes variable declarations
  - C, C++, and Java use **{** and **}** to delimit a block
  - Pascal and Modula use `begin ... end`
  - Ada uses `declare ... begin ... end`

Assembly Jump and C++ goto

Assembly jump	C++ goto

<pre>   mov eax,3
        jmp lemme_outta_here
        mov eax,999 ; <- not executed!
lemme_outta_here:
        ret</pre>
	<pre> int x=3;
	        goto quiddit;
	        x=999;
	quiddit:
	        return x;</pre>

Assembly Branch and Jumps

Here's how to use compare and jump-if-equal ("je"):

```
        mov eax,3
        cmp eax,3 ; how does eax compare with 3?
        je lemme_outta_here  ; if it's equal, then jump
        mov eax,999  ; <- not executed *if* we jump over it
lemme_outta_here:
        ret
```

Here's compare and jump-if-less-than ("jl"):

```
        mov eax,1
        cmp eax,3 ; how does eax compare with 3?
        jl lemme_outta_here  ; if it's less, then jump
        mov eax,999  ; <- not executed *if* we jump over it
lemme_outta_here:
        ret
```

Instruction	Useful to...
jmp	Always jump
ja	Unsigned >
jae	Unsigned >=
jb	Unsigned <
jbe	Unsigned <=
jc	Unsigned overflow, or multiprecision add
jecxz	Compare ecx with 0 (Seriously!?)
je	Equality
jg	Signed >
jge	Signed >=
jl	Signed <
jle	Signed <=
jne	Inequality
jo	Signed overflow

Expression Evaluation I Continue/Exit Condition

SECTION 4.1

Control Flow

Basic paradigms for control flow

- Sequencing
- **Selection**
- **Iteration**
- **Procedural Abstraction**
- **Recursion**
- Concurrency
- Exception Handling and Speculation
- Non-determinacy

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

Expression Evaluation

- Infix, prefix operators
- Precedence, associativity (see Figure 6.1)
  - C has 15 levels - too many to remember
  - Pascal has 3 levels - too few for good semantics
  - Fortran has 8
  - Ada has 6
    - Ada puts *and* & *or* at same level
  - **Lesson**: when unsure, use parentheses!

Expression Evaluation

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Figure 6.1 Operator precedence levels in Fortran, Pascal, C, and Ada. The operator s at the top of the figure group most tightly.

Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - distinguish between commutativity, and (assumed to be safe)
  - associativity (known to be dangerous)

```
(a + b) + c works if a~=maxint and b~=minint
  and c<0
a + (b + c) does not
```

  - inviolability of parentheses

Expression Evaluation

Short-circuiting

- Consider `(a < b) and (b < c)`:
  - If **a >= b** there is no point evaluating whether **b < c** because **(a < b) and (b < c)** is automatically false
- Other similar situations

```
if (b != 0 and a/b == c): ...
```

- Can be avoided to allow for side effects in the condition functions

Expression Evaluation II
Orthogonality

SECTION 4.2

Expression Evaluation

Orthogonality

- Features that can be used in any combination
  - Meaning is consistent

```
if (a/b == c if b != 0 else false): ...
if f or messy(): ...
```

Expression Evaluation

Assignment

- statement (or expression) executed for its side effect
- assignment operators (+=, -=, etc)
  - handy
  - avoid redundant work (or need for optimization)
  - perform side effects exactly once
- C --, ++
  - postfix form

Expression Evaluation

Side Effects

- often discussed in the context of functions
- a side effect is some permanent state change caused by execution of function
  - some noticeable effect of call other than return value
  - in a more general sense, **assignment** statements provide the ultimate example of side effects
    - they change the value of a variable

Expression Evaluation

Side Effects

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING
- In (pure) functional, logic, and dataflow languages, there are no such changes
  - These languages are called SINGLE-ASSIGNMENT languages

Expression Evaluation

Side Effects

- Several languages outlaw side effects for functions
  - easier to prove things about programs
  - closer to mathematical intuition
  - easier to optimize
  - (often) easier to understand
- But side effects can be nice
  - consider **`rand()`**

```
x = 0;
def xSetter(n):
    global x
    x = n
xSetter(5)
xSetter(5)
```

Expression Evaluation

Side Effects

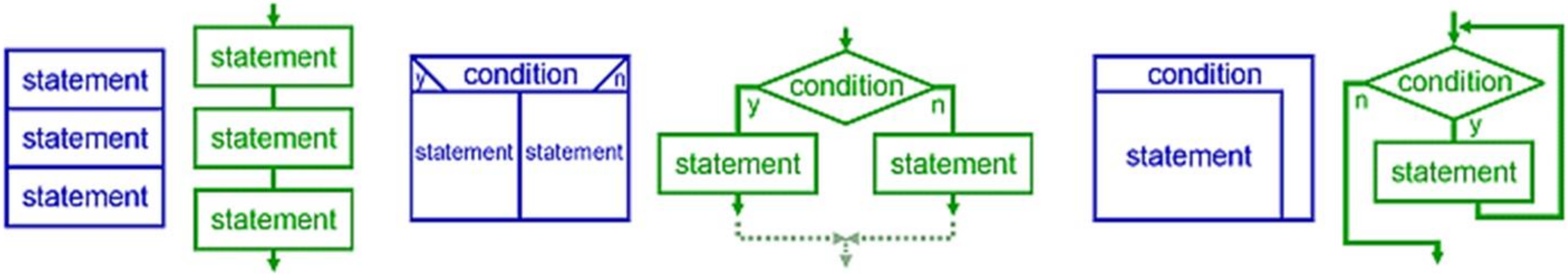
- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - It's nice not to specify an order, because it makes it easier to optimize
  - Fortran says it's OK to have side effects
    - they aren't allowed to change other parts of the expression containing the function call
    - Unfortunately, compilers can't check this completely, and most don't at all

Control Structures I Selection

SECTION 5.1

Structured Programming

- Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of **subroutines**, **block structures**, **for and while loops**—in contrast to using simple tests and jumps such as the go to statement which could lead to "spaghetti code" causing difficulty to both follow and maintain.


statement
statement
statement
statement
statement
statement
y
condition
n
statement
statement
condition
y
n
statement
statement
condition
statement
condition
n
y
statement

Structured control flow

- Statement sequencing
- Selection with “if-then-else” statements and “switch” statements
- Iteration with “for” and “while” loop statements
- Subroutine (function/method) calls (including recursion)
- All of which promotes “structured programming”
- Break levels (pass, continue, break, return, exit(0))

Code Blocks

- Statements;
- Compound Statements;
- Program Structure (loops);
- Procedure or Functions;

Selection

Condition, Switch, and if-elif-else

- sequential if statements

```
if ... :
    ...
else:
    ...
```

```
if ... :
    ...
elif ... :
    ...
else:
    ...
```

Control Structures II Iteration

SECTION 5.2

Loops

- while-loop
- do-while-loop (repeat-until)
- for-loop
- for-each-loop

Iteration

Enumeration-controlled (indexed loop)

- Pascal or Fortran-style **for** loops
  - scope of control variable
  - changes to bounds within loop
  - changes to loop variable within loop
  - value after the loop
- Can iterate over elements of any well-defined set
- repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop

While-loop

```
i = 0
while i<10:
  print(i)
  I += 1
```

for-loop

1 First step

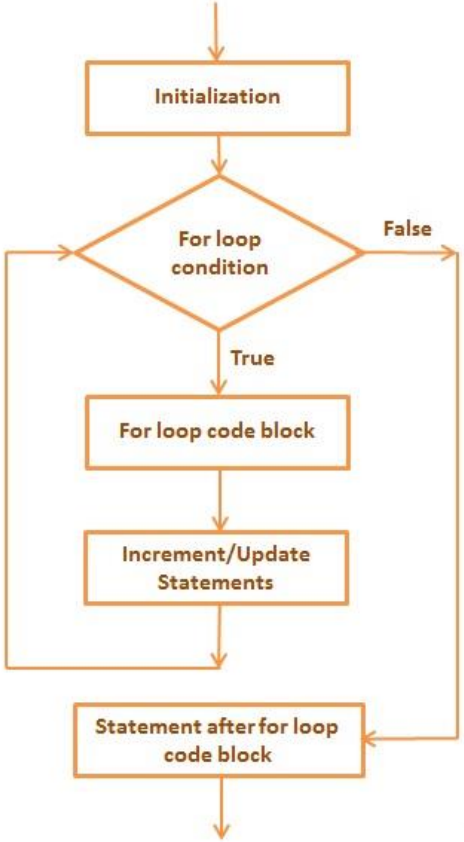
2 Second step

```
for(expression 1; expression 2; expression 3)
{
        statement 1;
        statement 2;
}
```

3 Third step

4 Fourth step

For Loop Flow Diagram



for-loop

```
for(int a = 10; a < 20; a = a + 1 ){ // C
       printf("value of a: %d\n", a);
    }
```

for-each-loop

Algorithm

{Sum first *n* integers}
begin
1. **input** *n*;
2. *sum* := 0;
3. **for** *i* := 1 to *n* **do**
4. *sum* := *sum* + *i*;
5. **output** *sum*;
end

Python

```
# sum first n integers

n = int(argv[1])
sum = 0
for i in range(1,n):
    sum = sum + i
print(sum)
```

For each loop

```
for i in range(10):
  print(i)
```

Logically-Controlled Loop

Pre-Test Loops(P):

```
readln(line)
while line[1] <> '$' do
      readln(line);
```

Post-Test Loops(P):

```
repeat
  readln(line)
until line[1]='$';
```

Post-Test Loops(C):

```
do{
   line = read_line(stdin);
} while (line[0] != '$');
```

Mid-Test Loops(C):

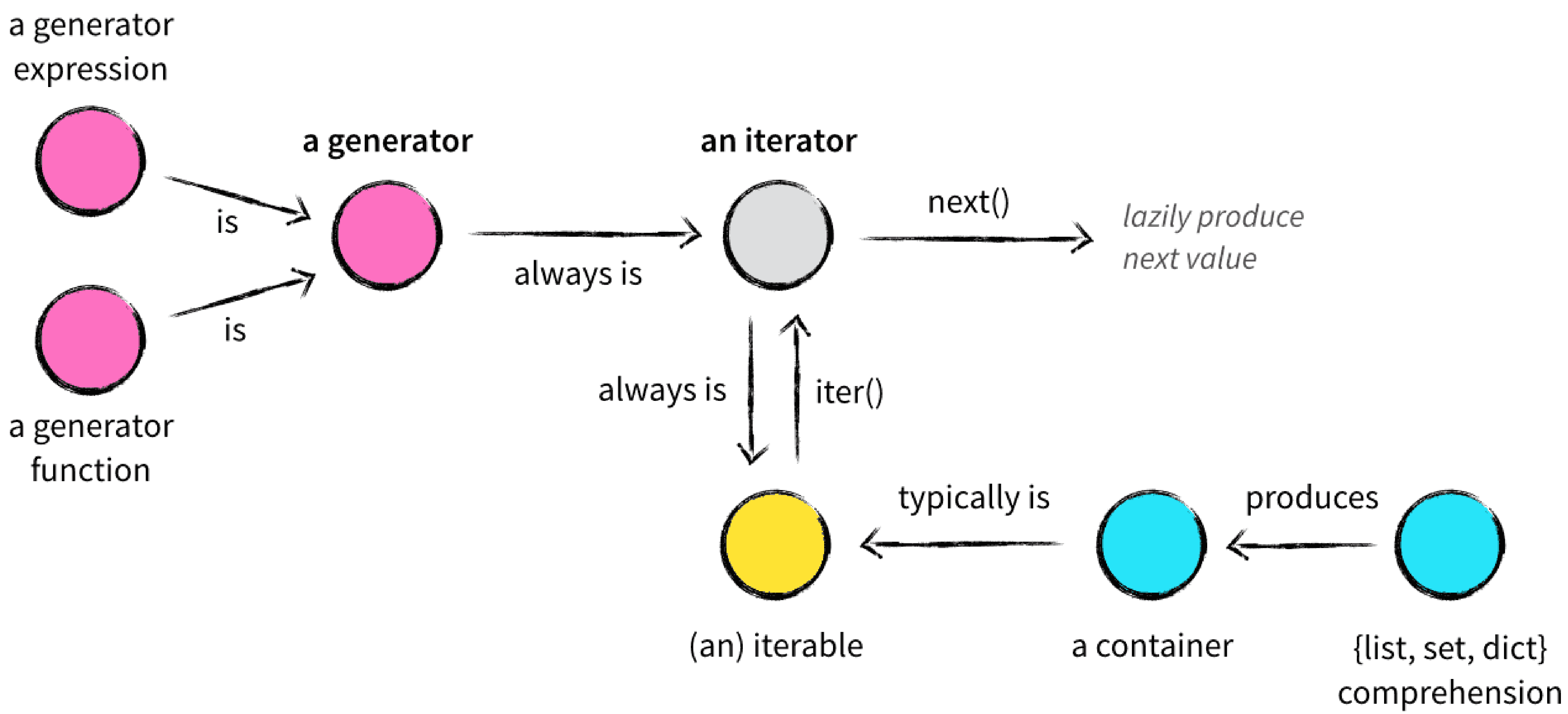
```
for (;;){
   line = read_line(stdin);
   if (all_blanks(line)) break;
   consum_line(lin);
}
```

Iterables vs. Iterators vs. Generators

A little pocket reference on iterables, iterators and generators.

The following related concepts in Python are very confusing:

- a container
- an iterable
- an iterator
- a generator
- a generator expression
- a {list, set, dict} comprehension


a generator expression
a generator function
is
is
a generator
always is
an iterator
next()
lazily produce next value
always is
iter()
(an) iterable
typically is
a container
produces
{list, set, dict} comprehension

Containers

Containers are data structures holding elements, and that support membership tests. They are data structures that live in memory, and typically hold all their values in memory, too. In Python, some well known examples are:

- **list**, deque, …
- **set**, frozensets, …
- **dict**, defaultdict, OrderedDict, Counter, …
- **tuple**, namedtuple, …
- **str**

Containers are easy to grasp, because you can think of them as real life containers: a box, a cubboard, a house, a ship, etc.

Iterables

- As said, most containers are also iterable. But many more things are iterable as well. Examples are open files, open sockets, etc. Where containers are typically finite, an iterable may just as well represent an infinite source of data.
- An iterable is any object, not necessarily a data structure, that can return an iterator (with the purpose of returning all of its elements). That sounds a bit awkward, but there is an important difference between an iterable and an iterator. Take a look at this example:

container4.py

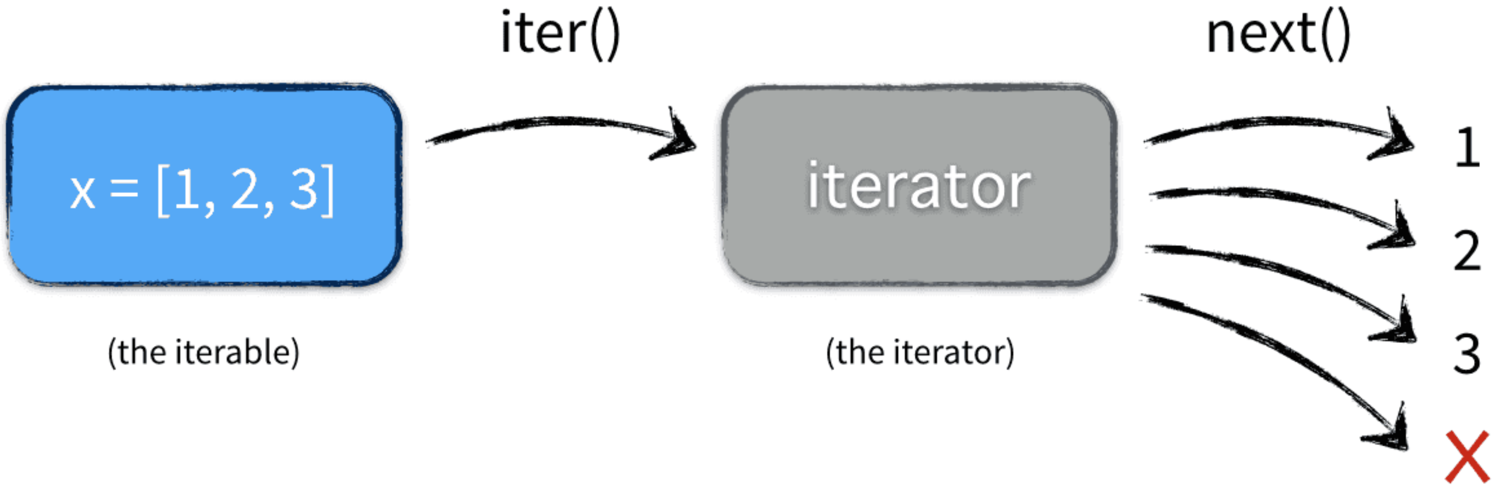
```
>>> x = [1, 2, 3]
>>> y = iter(x)
>>> z = iter(x)
>>> next(y)
1
>>> next(y)
2
>>> next(z)
1
>>> type(x)
<class 'list’>
>>> type(y)
<class 'list_iterator'>
```

Here, x is the iterable, while y and z are two individual instances of an iterator, producing values from the iterable x. Both y and z hold state, as you can see from the example. In this example, x is a data structure (a list), but that is not a requirement.

NOTE:
Often, for pragmatic reasons, iterable classes will implement both __iter__() and __next__() in the same class, and have __iter__() return self, which makes the class both an iterable and its own iterator. It is perfectly fine to return a different object as the iterator, though.

For-Each Loop

```
x = [1, 2, 3]
for elem in x:
    ...
```



Disassemble Python Code for Iterator

- When you disassemble this Python code, you can see the explicit call to GET_ITER, which is essentially like invoking iter(x). The FOR_ITER is an instruction that will do the equivalent of calling next() repeatedly to get every element, but this does not show from the byte code instructions because it's optimized for speed in the interpreter.

Disassemble Python Code for Iterator

disassemble.py

```
>>> import dis
>>> x = [1, 2, 3]
>>> dis.dis('for _ in x: pass')
  1           0 SETUP_LOOP              14 (to 17)
              3 LOAD_NAME                0 (x)
              6 GET_ITER
        >>    7 FOR_ITER                 6 (to 16)
             10 STORE_NAME               1 (_)
             13 JUMP_ABSOLUTE            7
        >>   16 POP_BLOCK
        >>   17 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

Iterators

- So, what is an iterator then? It's a stateful helper object that will produce the next value when you call **next()** on it. Any object that has a **__next__()** method is therefore an iterator. How it produces a value is irrelevant.
- So, an iterator is a value factory. Each time you ask it for "the next" value, it knows how to compute it because it holds internal state.

Iterators

- There are countless examples of iterators. All of the itertools functions return iterators. Some produce infinite sequences:

```
>>> from itertools import count
>>> counter = count(start=13)
>>> (counter)
13
>>> (counter)
14
```

Iterators

```
from itertools import count
counter = count(start=13)
print((counter))
print(next(counter))
print(next(counter))
```

iterator2.py

```
count(13)
13
14
```

Cyclic Iterator

- Some produce infinite sequences from finite sequences:

iterator3.py

```
from itertools import cycle
colors = cycle(['red', 'white', 'blue'])
print(next(colors))
print(next(colors))
print(next(colors))
print(next(colors))
```

```
red
white
blue
red
```

Cyclic Iterator

- Some produce finite sequences from infinite sequences:

```
from itertools import islice, cycle
colors = cycle(['red', 'white', 'blue'])  # infinite
limited = islice(colors, 0, 7)            # finite
for x in limited:                         # so safe to use for-loop on
    print(x)
```

iterator4.py

```
red
white
blue
red
white
blue
red
```

Example for Iterable and Iterator

iterator5.py

```
from itertools import islice
class fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1
    def __iter__(self):
        return self
    def __next__(self):
        value = self.curr
        self.curr += self.prev
        self.prev = value
        return value

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Example for Iterable and Iterator

- Note that this class is both an iterable (because it sports an `__iter__()` method), and its own iterator (because it has a `__next__()` method).

- The state inside this iterator is fully kept inside the prev and curr instance variables, and are used for subsequent calls to the iterator. Every call to next() does two important things:
    1. Modify its state for the next next() call;
    2. Produce the result for the current call.

Generators

- Finally, we've arrived at our destination! The generators are my absolute favorite Python language feature. **A generator is a special kind of iterator**—the elegant kind.
- A generator allows you to write iterators much like the Fibonacci sequence iterator example above, but in an elegant succinct syntax that avoids writing classes with **__iter__()** and **__next__()** methods.
- Let's be explicit:
  - Any generator also is an iterator (not vice versa!);
  - Any generator, therefore, is a factory that lazily produces values.

Central idea: a lazy factory

From the outside, the iterator is like a lazy factory that is idle until you ask it for a value, which is when it starts to buzz and produce a single value, after which it turns idle again.

Generating Function

generator1.py

```
from itertools import islice
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Generating Function

- Wow, isn't that elegant? Notice the magic keyword that's responsible for the beauty:

```
yield
```

- Let's break down what happened here: first of all, take note that fib is defined as a normal Python function, nothing special. Notice, however, that there's no return keyword inside the function body. The return value of the function will be a generator (read: an iterator, a factory, a stateful helper object).

Generating Function

- Now when `f=fib()` is called, the generator (the factory) is instantiated and returned. No code will be executed at this point: the generator starts in an idle state initially. To be explicit: the line `prev, curr = 0, 1` is not executed yet.
- Then, this generator instance is wrapped in an `islice()`. This is itself also an iterator, so idle initially. Nothing happens, still.
- Then, this iterator is wrapped in a `list`(), which will consume all of its arguments and build a list from it. To do so, it will start calling `next()` on the `islice()` instance, which in turn will start calling `next()` on our f instance.

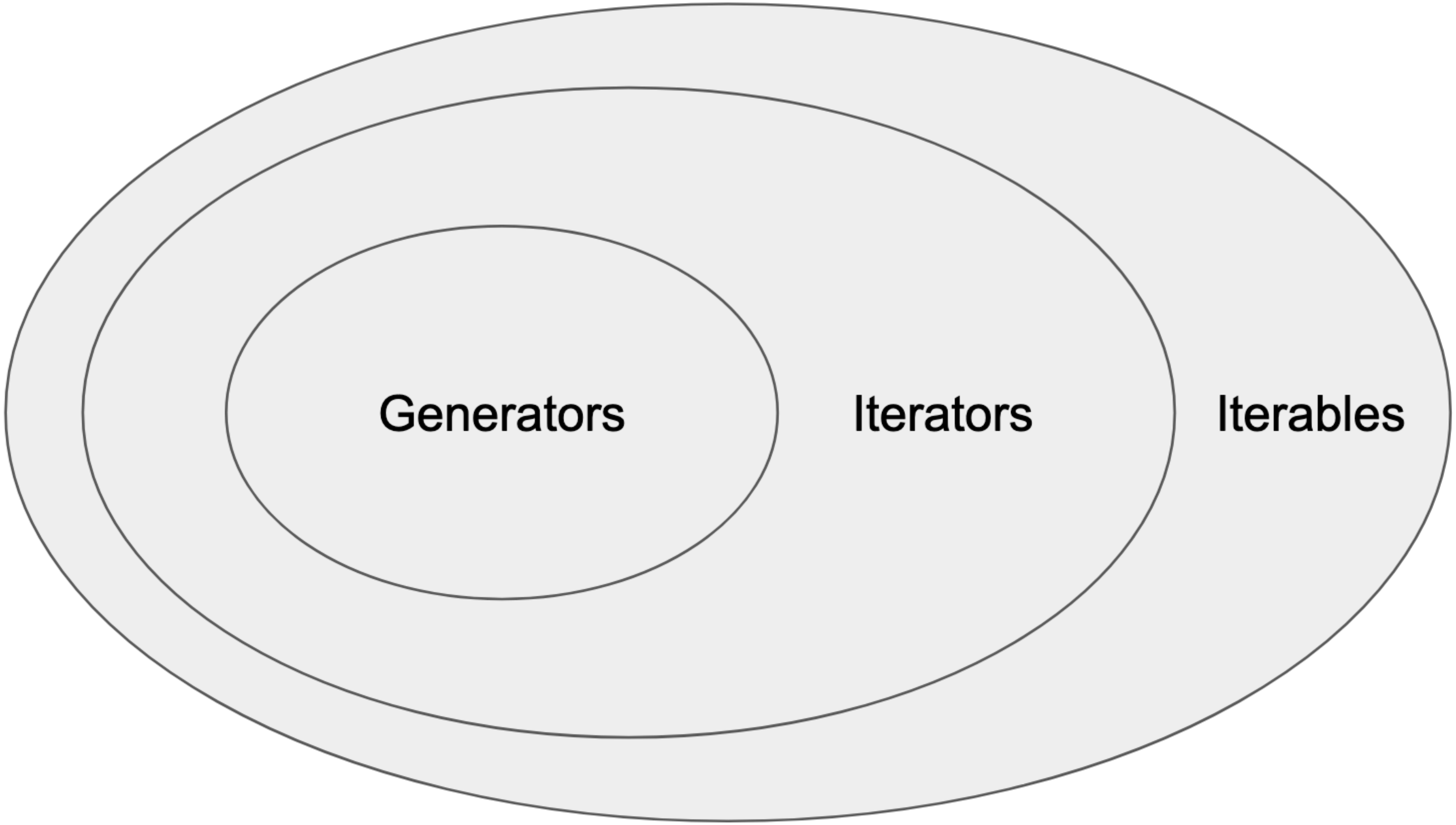
Generating Function

- But one step at a time. On the first invocation, the code will finally run a bit: `prev, curr = 0, 1` gets executed, the while True loop is entered, and then it encounters the yield curr statement. It will produce the value that's currently in the curr variable and become idle again.
- This value is passed to the `islice()` wrapper, which will produce it (because it's not past the 10th value yet), and list can add the value 1 to the list now.

Generating Function

- Then, it asks `islice()` for the next value, which will ask f for the next value, which will "unpause" f from its previous state, resuming with the statement `prev, curr = curr, prev + curr`. Then it re-enters the next iteration of the while loop, and hits the `yield curr` statement, returning the next value of `curr`.

- This happens until the output list is 10 elements long and when `list()` asks `islice()` for the 11th value, `islice()` will raise a **`StopIteration`** exception, indicating that the end has been reached, and list will return the result: a list of 10 items, containing the first 10 Fibonacci numbers. Notice that the generator doesn't receive the 11th `next()` call. In fact, it will not be used again, and will be garbage collected later.


Generators
Iterators
Iterables

Types of Generators

- There are two types of generators in Python: generator functions and generator expressions. A generator function is any function in which the keyword yield appears in its body. We just saw an example of that. The appearance of the keyword yield is enough to make the function a generator function.
- The other type of generators are the generator equivalent of a list comprehension. Its syntax is really elegant for a limited use case.

Python Generators

A Quick Guide for Beginners

```
def gen_func():
    ...
    while <cond>:
        ...
        yield num

next(gen_func())
```

```
gen_expr = (a**(1/2)
for a in alist)
```

```
pipeline
gen_fn1()
    => gen_fn2()
        => gen_fn3()
```

```
for item in
gen_func(args):
    print(item)
```

www.techbeamers.com

Comprehension Generators

generator2.py

```
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
b = [x*x for x in numbers]      # comprehensive list
c = {x*x for x in numbers}      # comprehensive set
d = {x:x*x for x in numbers}    # comprehensive dict
print(a)
print(b)
print(c)
print(d)
```

```
<generator object <genexpr> at 0x000001BE7126F048>
[1, 4, 9, 16, 25, 36]
{1, 4, 36, 9, 16, 25}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Generator expressions

- **A comprehension-based expression that results in an iterator object**
  - **Does not result in a container of values**
  - **Must be surrounded by parentheses unless it is the sole argument of a function**
  - **May be returned as the result of a function**

```
numbers = (random() for _ in range(42))
sum(numbers)
```

```
sum(random() for _ in range(42))
```

Generator Expression

note: this is not a tuple comprehension

generator3.py

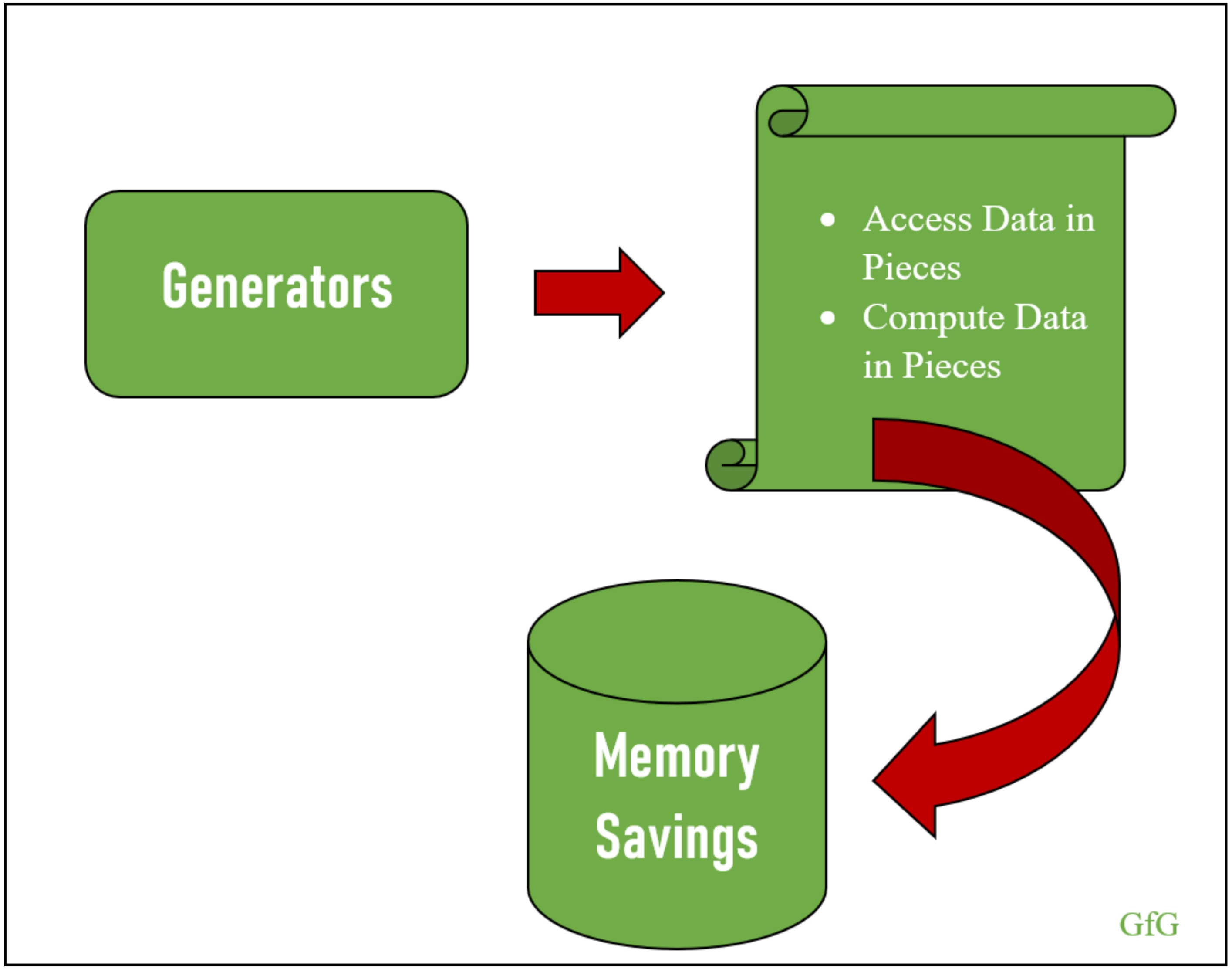
```
# Generator Expression
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

```
1
4
9
16
```

Generator Expression

note: this is not a tuple comprehension

- Note that, because we read the first value from lazy_squares with next(), it's state is now at the "second" item, so when we consume it entirely by calling list(), that will only return the partial list of squares. (This is just to show the lazy behaviour.) This is as much a generator (and thus, an iterator) as the other examples above.


Generators
Access Data in Pieces
Compute Data in Pieces
Memory Savings
GfG

Summary

- Generators are an incredible powerful programming construct. They allow you to write streaming code with fewer intermediate variables and data structures. Besides that, they are more memory and CPU efficient. Finally, they tend to require fewer lines of code, too.

Control Structures III
Function

SECTION 5.3

Functional Abstraction

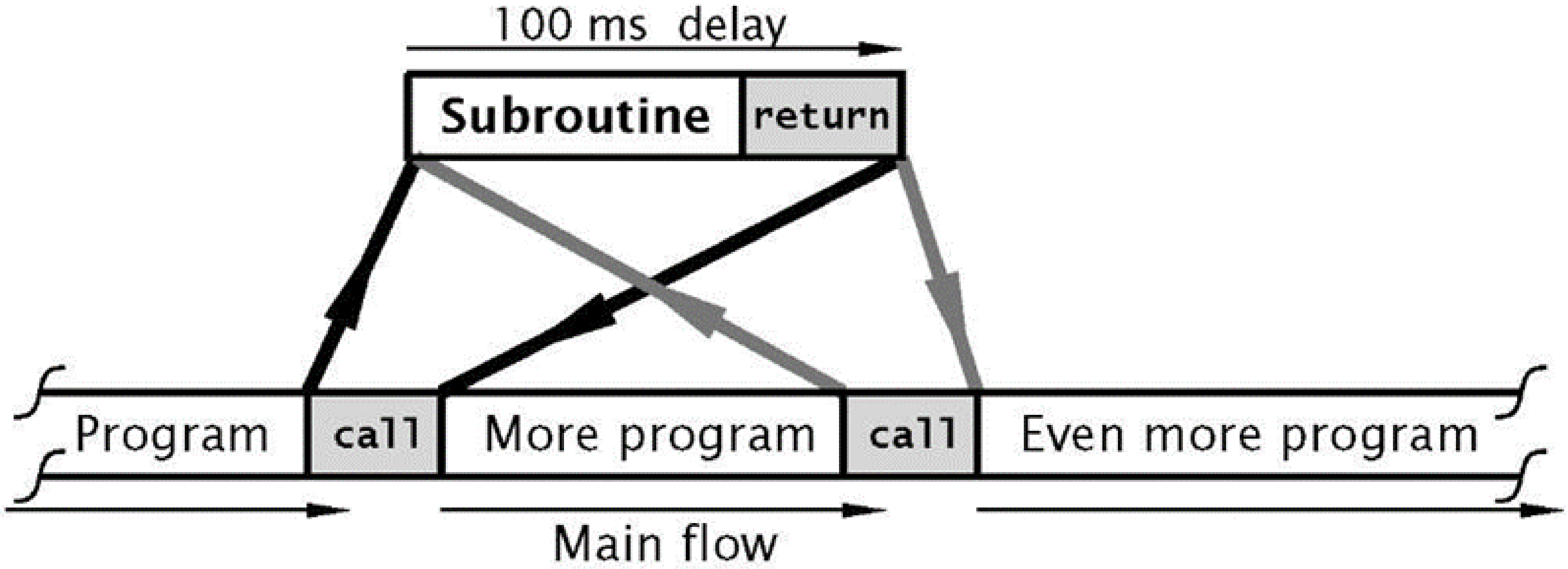
- A function performs a specified task, given stated preconditions and postconditions
- It has a name, parameters and a return value
- It may be used “by name” as long as …
  - appropriate values or objects are passed to it as parameters,
  - its preconditions are met
  - its return value is used in an appropriate context
- In this sense we have “abstracted” the function and “hidden” its implementation details

Advantages

- **Modularization:** Decomposing a complex programming task into simpler steps
- **Code Reuse:** Reducing duplicate code within a program
- **Packaging:** Enabling reuse of code across multiple programs
- **Team Work:** Dividing a large programming task among various programmers, or various stages of a project
- **Abstraction:** Hiding implementation details
- **Readability:** Improving traceability

A Set of Rules

- Code Section
- Subroutines
- Procedure
- Operation
- Function
- Method
- Lambda Expression

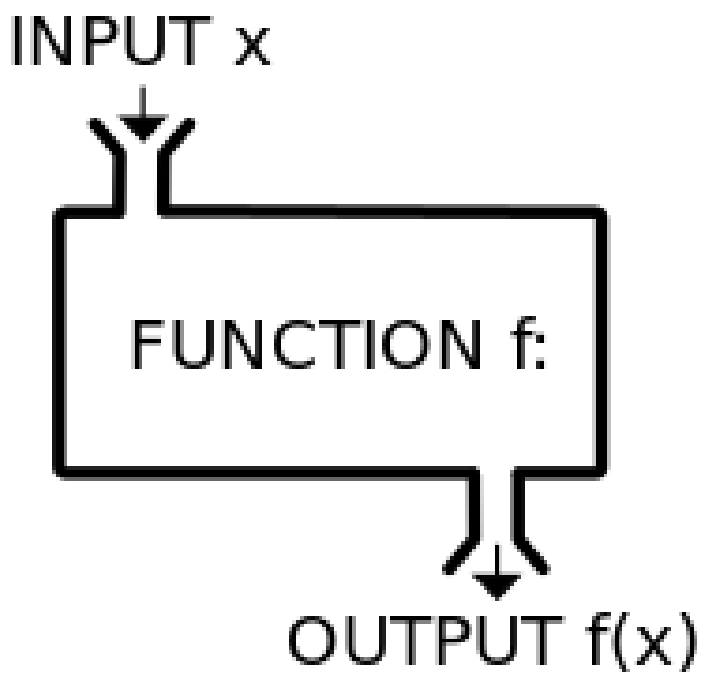

100 ms delay
Subroutine
return
Program
call
More program
call
Even more program
Main flow

Control Structure III

- **Procedural abstraction**: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
- **Recursion**: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
- **Concurrency**: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
- **Non-determinacy**: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result. (function as pointer, randomized functional calls)

Sub-Routine, Function, and Methods

- Abstraction
- Reusability
- Library



Function Signature
Python

functional signature

function name

parameter list

return type hint

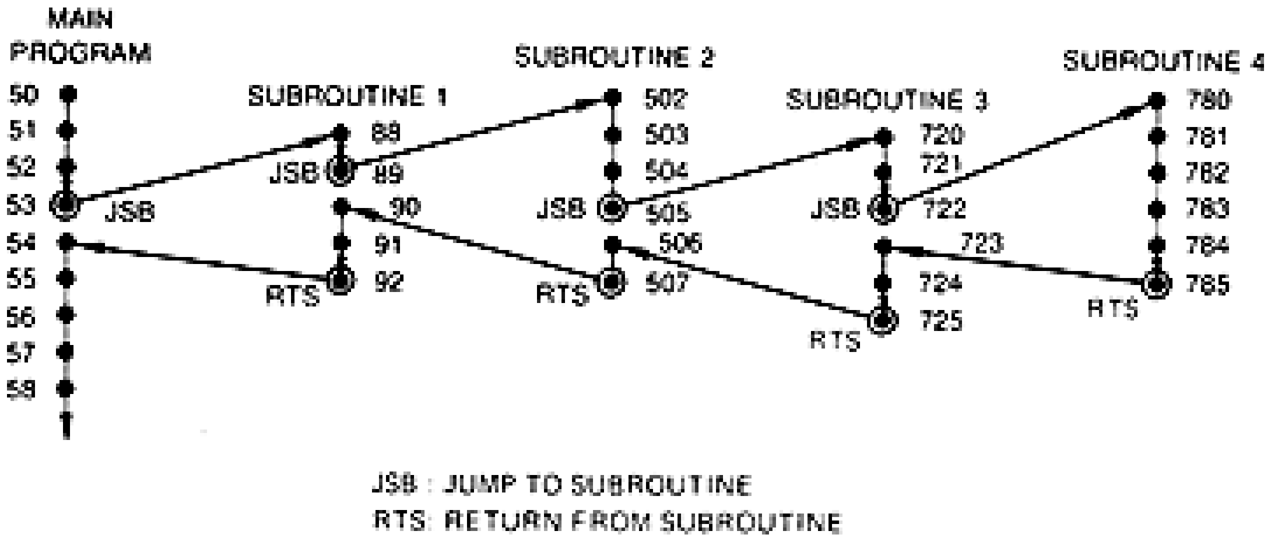
```python
def func(a=0, b=0, *args) -> int:
    y = 0
    for x in args:
        y += a * x + b
    return y
```

local variable

return statement

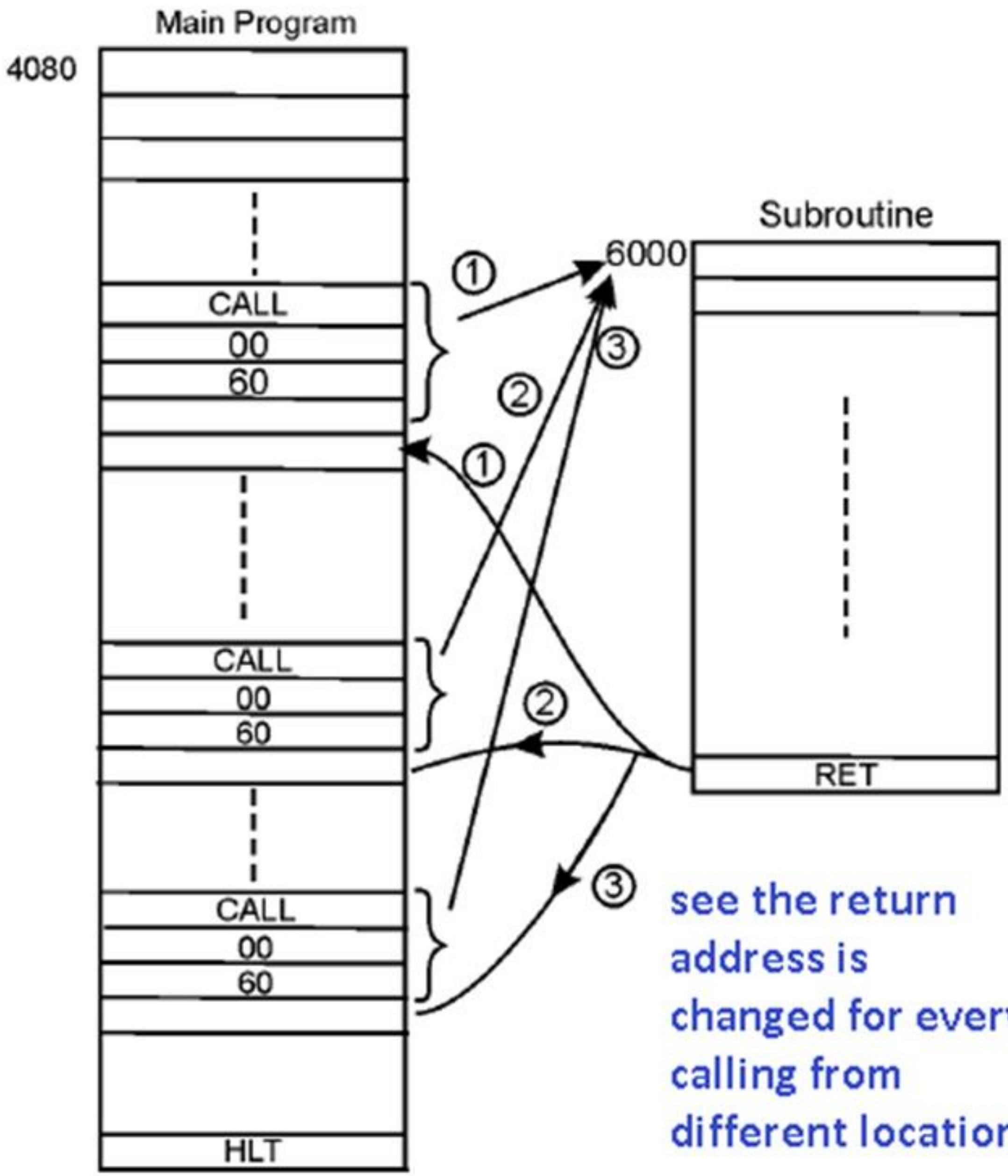
function1.py

Call Stack



START	AFTER 53	AFTER 89	AFTER 505	AFTER 722
	54	90	506	723
		54	90	506
			54	90
				54

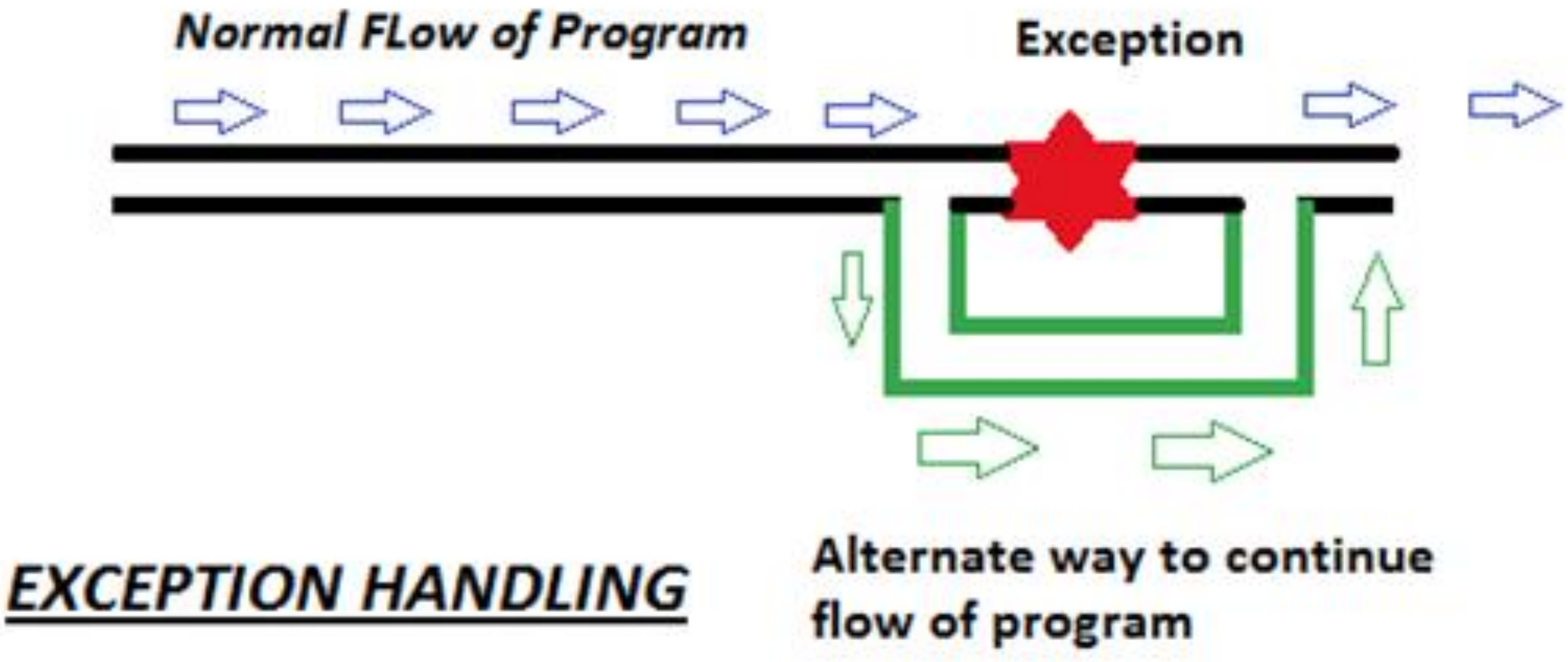
LIFO STACK CONTENTS



see the return address is changed for every calling from different location

Control Structures IV Exception

SECTION 5.4


Normal FLow of Program
Exception
EXCEPTION HANDLING
Alternate way to continue
flow of program


Problem Occurs
Create Exception
Throw Exception
Handle Exception

Exception Handling

Propagate to the Level with Handler

Raise Exception Object

exception1.py

```
x = 0
done = False
while not done:
    try:
        x = int(input("Enter an even number: "))
        if x%2 !=0: raise BaseException("Not Even")
        done = True
    except BaseException as e:
        print(e)
    except:
        print("Wrong Input Format")
```

Exception Type

Incoming Exception Object as Parameter

Control Structures V
Recursion

SECTION 5.5

Recursion

- equally powerful to iteration
- mechanical transformations back and forth
- often more intuitive (sometimes less)
- *naïve* implementation less efficient
  - no special syntax required
  - fundamental to functional languages like Scheme

Recursion

- **Recursion:** subroutines that call themselves directly or indirectly (mutual recursion)
- Typically used to solve a problem that is defined in terms of simpler versions, for example:
  - To compute the length of a list, remove the first element, calculate the length of the remaining list in *n*, and return *n*+1
  - Termination condition: if the list is empty, return 0

Recursion

- Iteration and recursion are equally powerful in theoretical sense
  - Iteration can be expressed by recursion and vice versa
- Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm
- Recursion can be less efficient, but most compilers for functional languages are often able to replace it with iterations

Tail recursion

- No computation follows recursive call

```
def gcd2(m, n):
    if (m==n): return m
    if (m>n): return gcd2(m-n, n)
    return gcd2(m, n-m)
```

recursion1.py

```python
def gcd1(m, n):
    if n==0: return m
    return gcd1(n, m%n)

def gcd2(m, n):
    if (m==n): return m
    if (m>n): return gcd2(m-n, n)
    return gcd2(m, n-m)

print(gcd1(48, 36))
print(gcd2(48, 36))
print(gcd1(36, 48))
print(gcd2(36, 48))
```

Tail-Recursive Functions

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

tail-recursive

```
def trfun():
    …
    return trfun();
```

not tail-recursive

```
def rfun():
    …
    return rfun()+1;
```

Tail-Recursion Optimization

- A tail-recursive call could ***reuse*** the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed
  - Simply eliminating the push (and pop) of the next frame will do
- In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function

Iterative Counterpart

- It is not hard to find a more efficient iterative counterpart for all recursive functions.

recursion2.py

```python
def gcd3(m, n):
    while (m!=n):
        if (m>n): m = m-n
        else: n = n-m
    return m

def gcd4(m, n):
    while n!=0 and m!=0:
        if m>n: m = m%n
        else: n = n%m
    return n if m==0 else m

print(gcd3(48, 36))
print(gcd4(48, 36))
print(gcd3(36, 48))
print(gcd4(36, 48))
```

When Recursion is inefficient

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

```
def fib(n):
    if n==0 or n==1: return 1
    return fib(n-1)+fib(n-2)

for i in range(1, 6):
    print("fib(%d)=%d" % (i, fib(i)))
```

```python
def fib(n):
    if n==0 or n==1: return 1
    return fib(n-1)+fib(n-2)

def fib2(n):
    if n==0 or n==1: return 1
    i = 2; f = 1; f1 = 1; f2 = 1
    while i<=n:
        f = f1 + f2
        i += 1
        f2 = f1
        f1 = f
    return f
for i in range(1, 6):
    print("fib(%d)=%d" % (i, fib(i)))
for i in range(1, 6):
    print("fib2(%d)=%d" % (i, fib2(i)))
```

recursion3.py

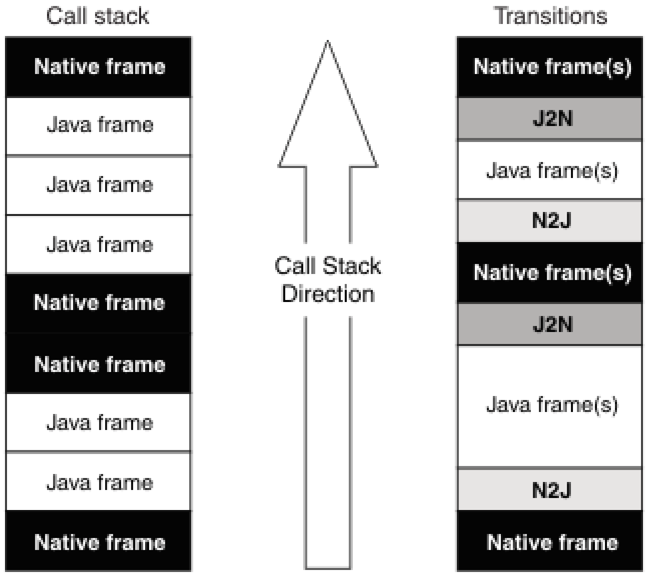
```
fib(1)=1
fib(2)=2
fib(3)=3
fib(4)=5
fib(5)=8
fib2(1)=1
fib2(2)=2
fib2(3)=3
fib2(4)=5
fib2(5)=8
```

Stack Layout

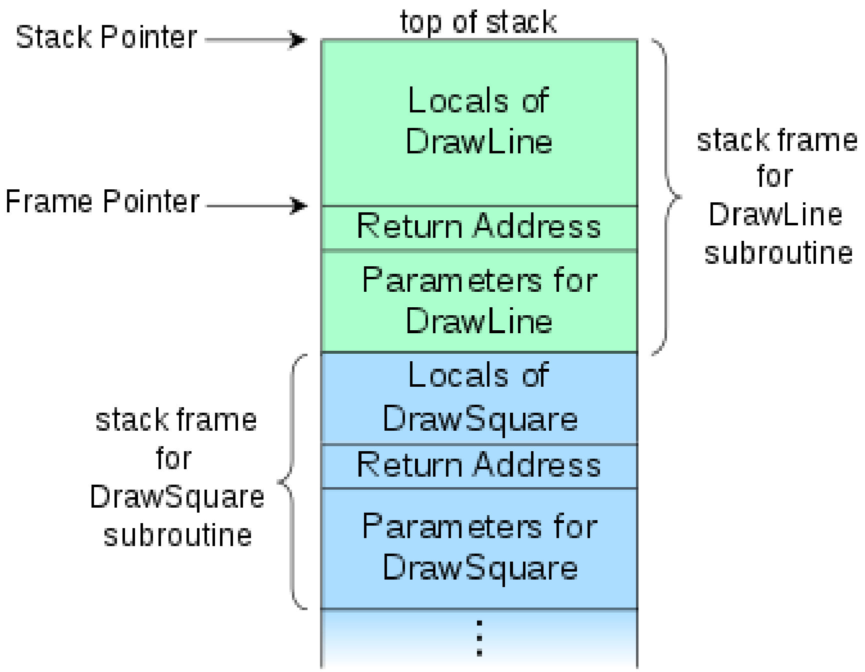
SECTION 6

Class Method Area In Stack (Dynamic)

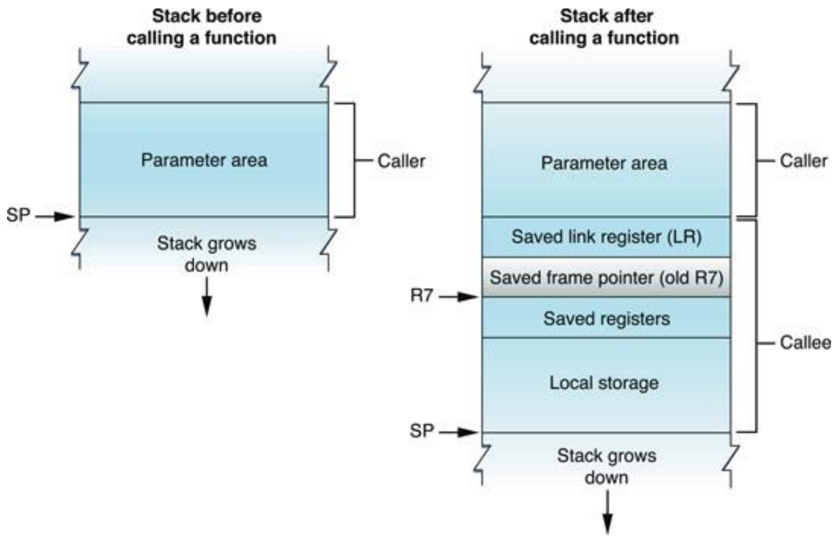
Growth of Call Stack



C Call Stack



Assembly Call Stack



Review Of Stack Layout

Allocation strategies

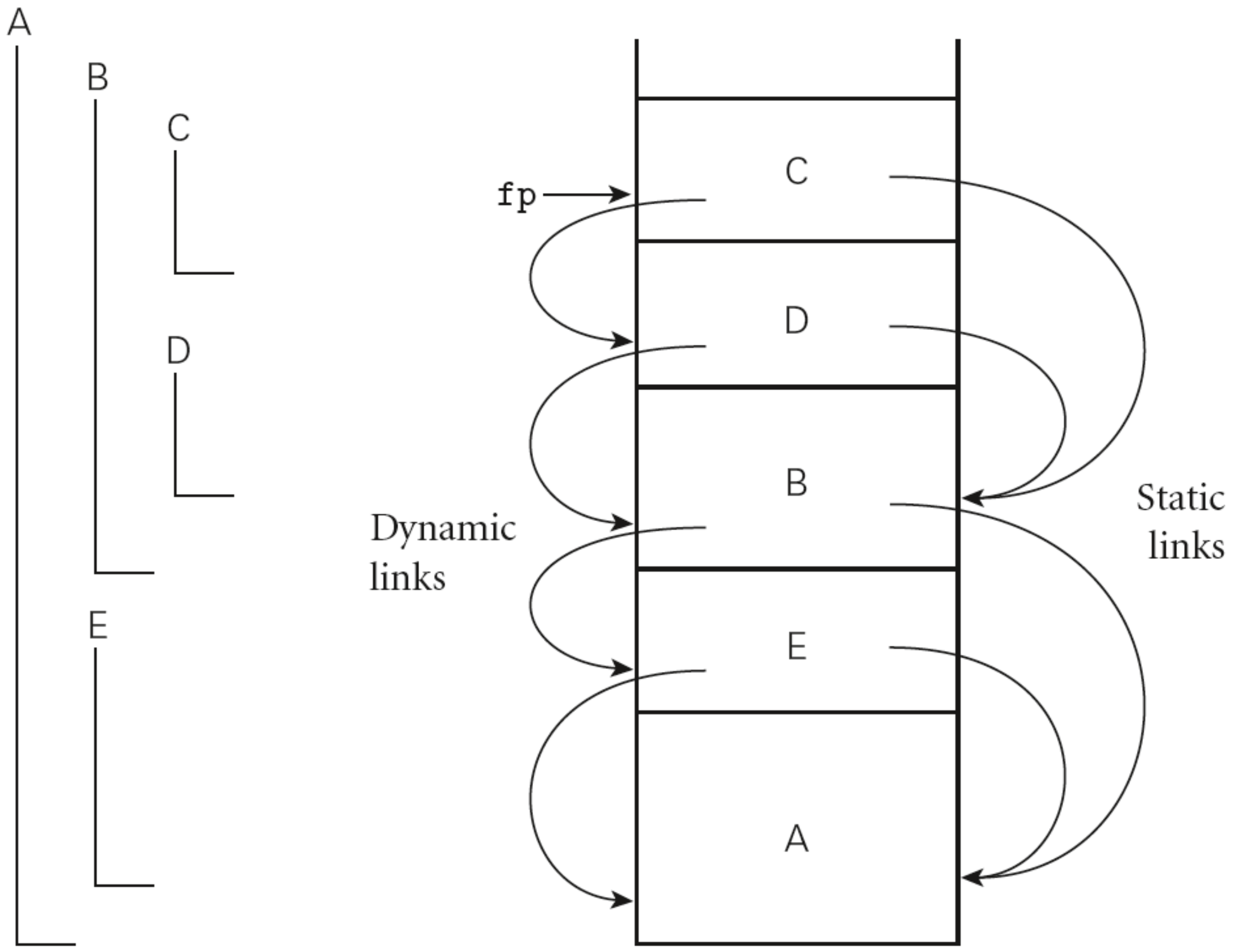


Figure 9.1 **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Review Of Stack Layout

Allocation strategies (2)

- Stack
  - parameters
  - local variables
  - temporaries
  - bookkeeping information
- Heap
  - dynamic allocation

Review Of Stack Layout

Contents of a Stack Frame

- bookkeeping
  - return PC (dynamic link)
  - saved registers
  - line number
  - saved display entries
  - static link
- arguments and returns
- local variables
- temporaries

Calling Sequences

SECTION 7

Calling Sequences

- Maintenance of stack is responsibility of **calling sequence** and **subroutine prolog (call)** and **epilog (return)**
  - space is saved by putting as much in the prolog and epilog as possible
  - time **may** be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee

Task Must be Done

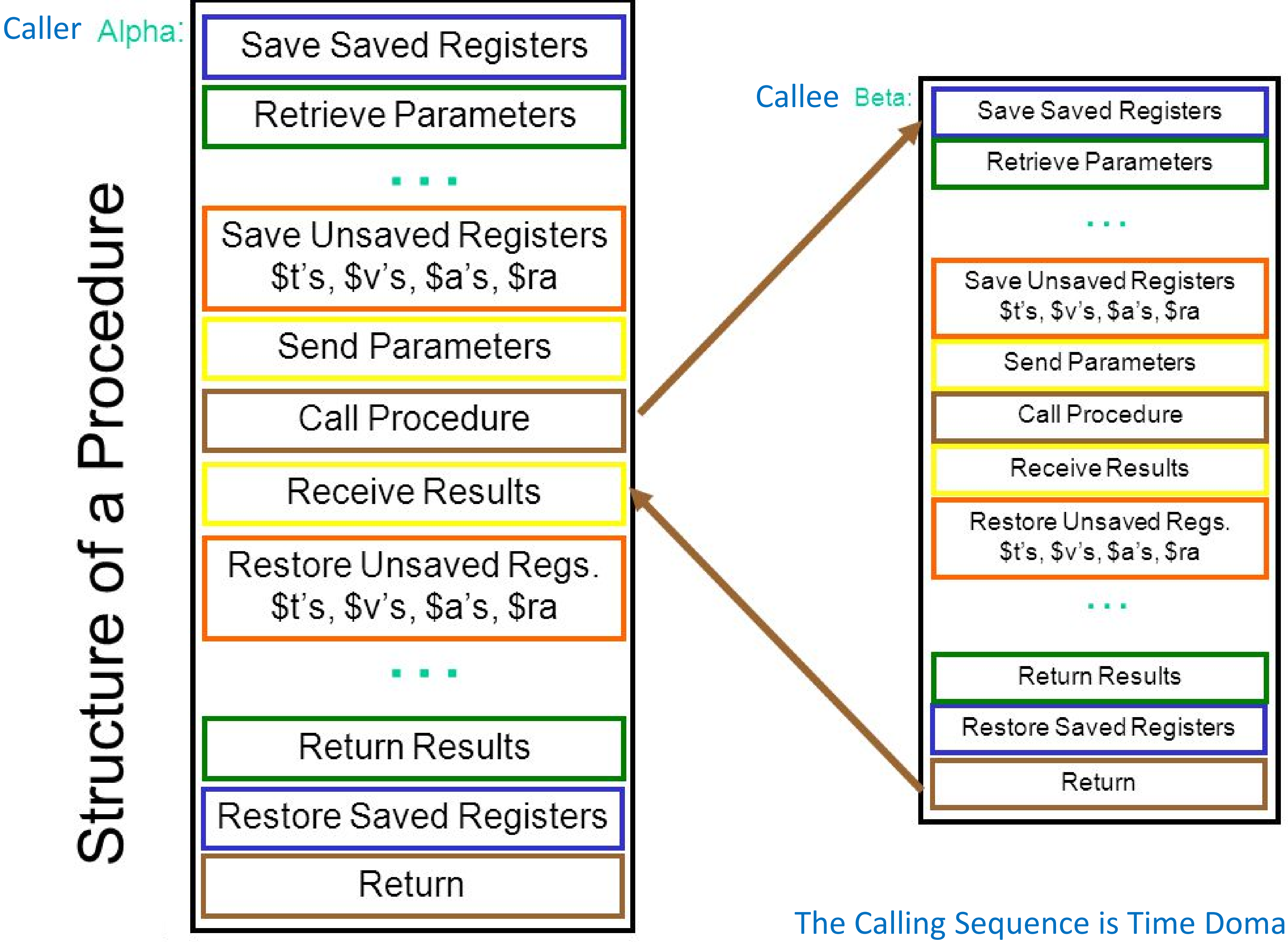
Prologue (Call):

- Passing Parameters
- Saving the Return Address
- Changing the Program Counters
- Changing the Stack Pointer (Call Stack)
- Saving Registers
- Changing Frame Pointer to Refer to the New Frame
- Executing the Initialization Code for New Objects

Epilog (Return):

- Passing the Return Parameters or Function Values
- Executing the Finalization Code for the Local Objects
- Deallocating the Stack Frame
- Restoring other Stored Registers
- Restoring Program Counter

Structure of a Procedure

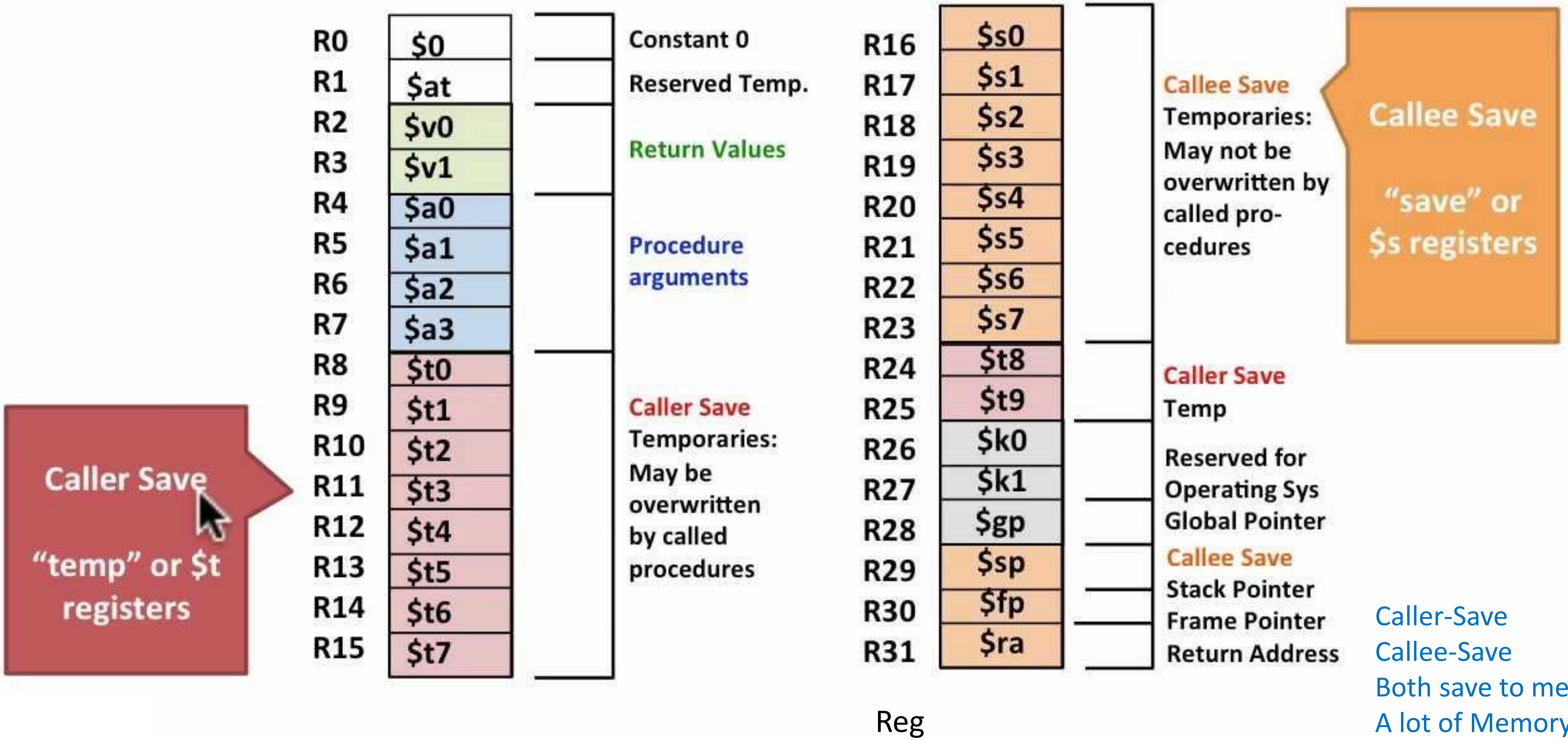


The Calling Sequence is Time Domain not Spatial Domain.

Calling Sequences

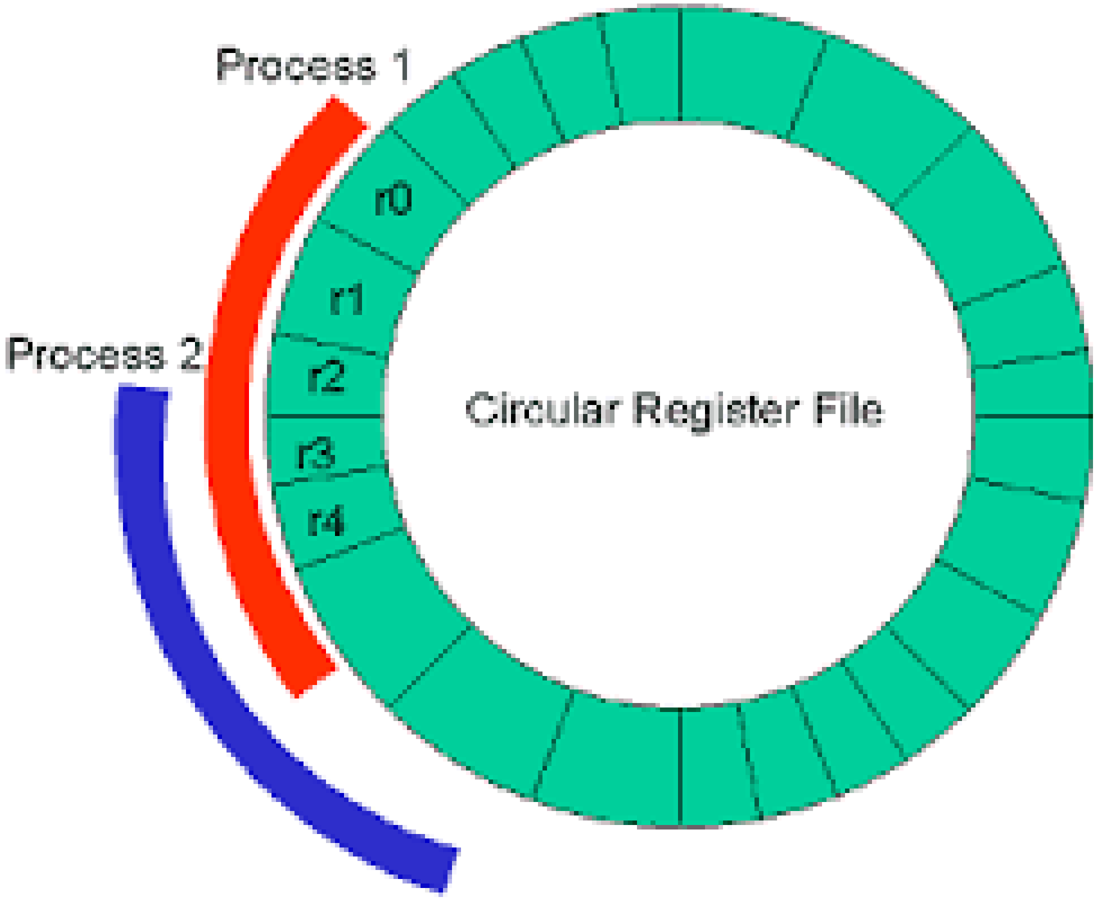
- The ideal approach is to save those registers that are both live in the caller and needed for other purpose in the Callee.
- Hard to determine this intersecting set.
- Common strategy is to divide registers into **caller-saves** and **callee-saves** sets. (Of equal size)
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
  - some storage layouts use a separate arguments pointer
  - the VAX architecture encouraged this

More convenient names for registers



Caller-Save
Callee-Save
Both save to memory (Stack)
A lot of Memory Cycles

Register Windows on RISC Machine



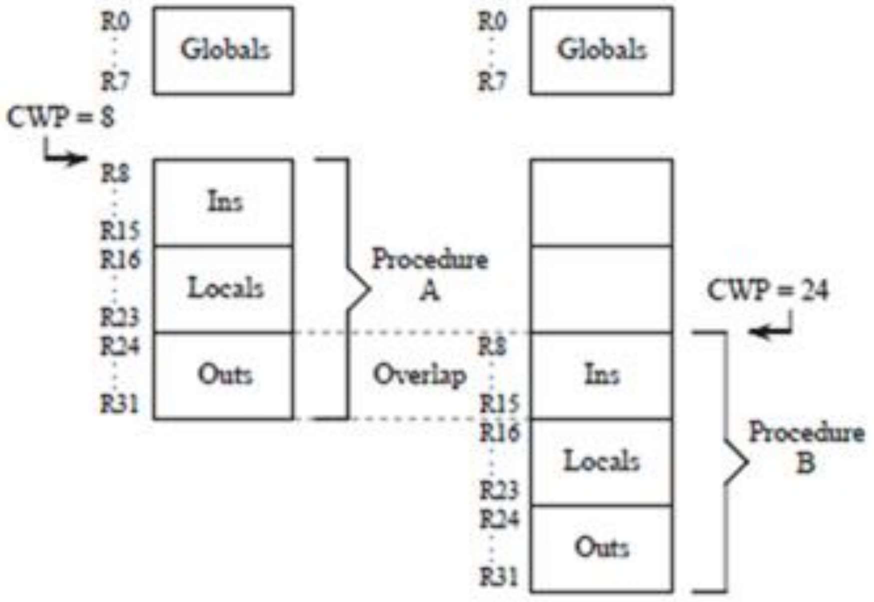
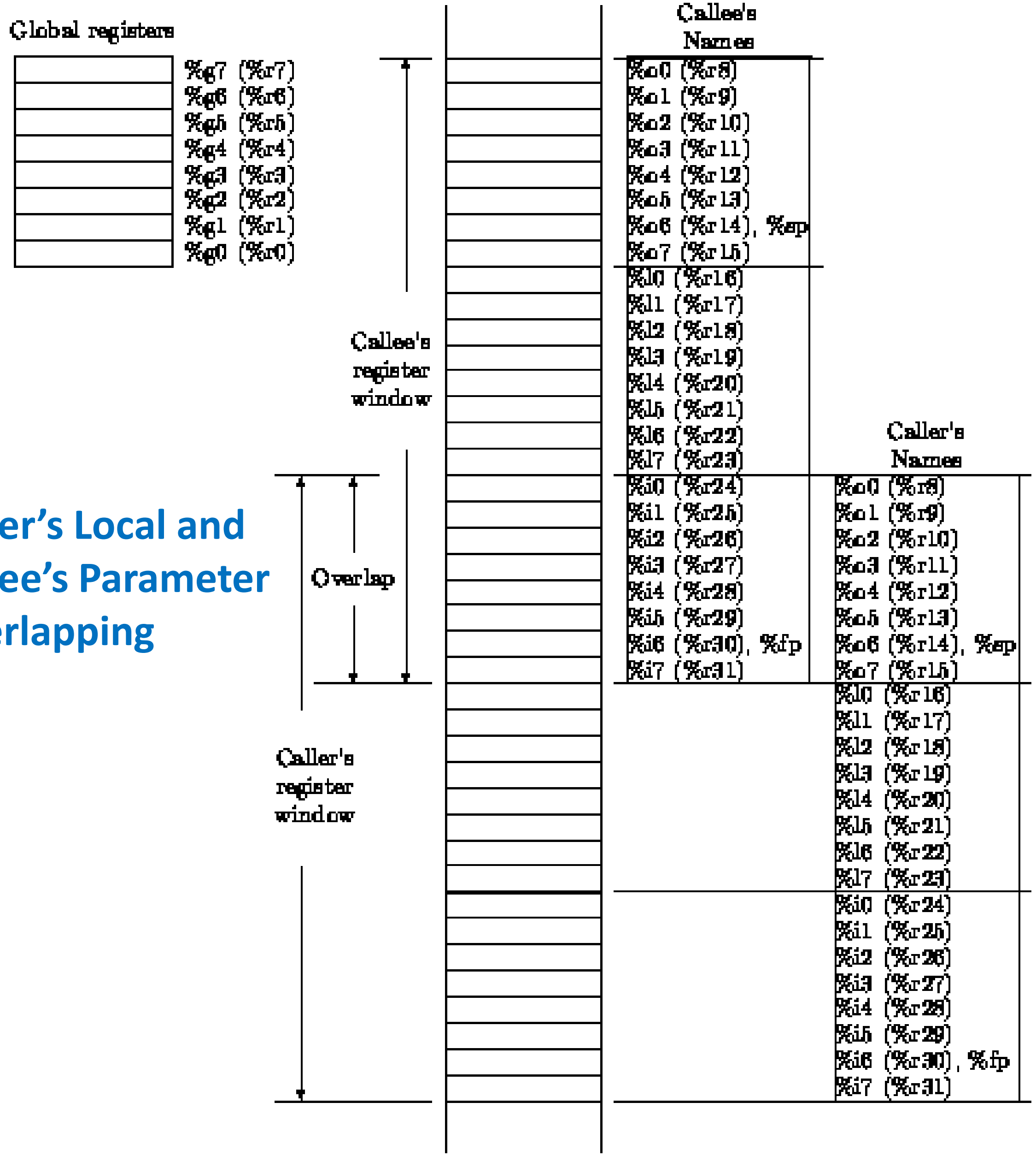


Figure 10-8 **Overlapping register windows.**

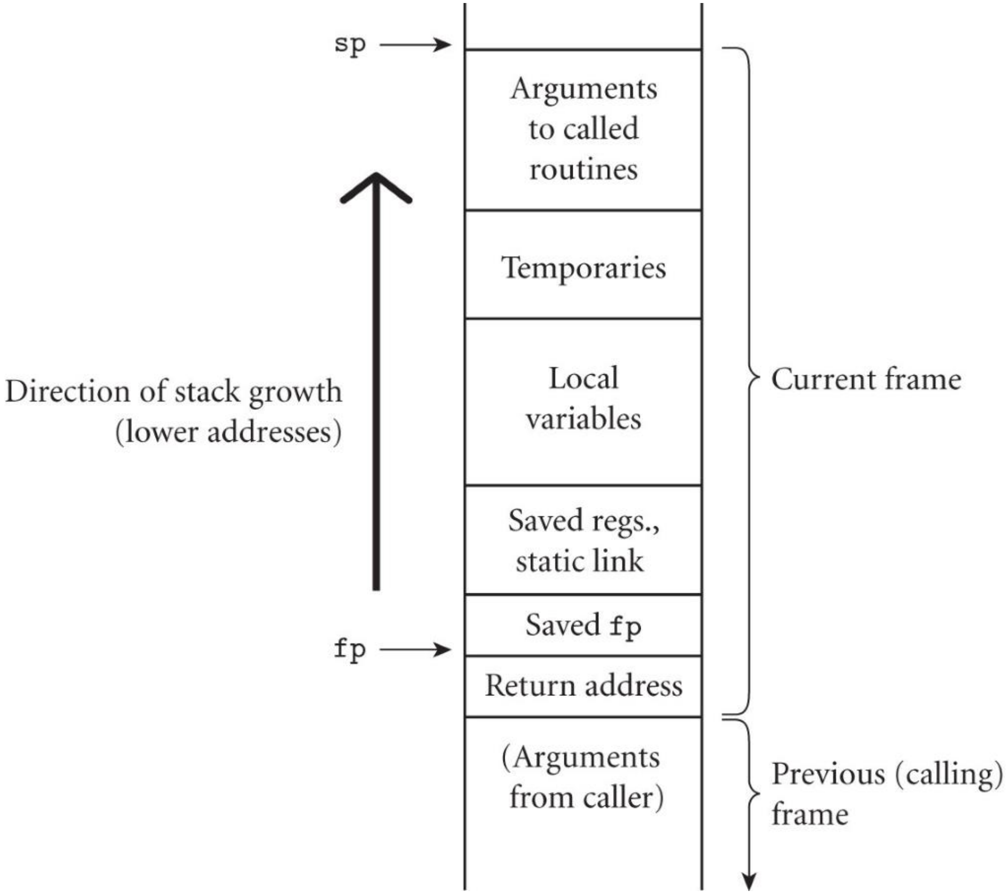
Caller’s Local and Callee’s Parameter Overlapping



Operation	Syntax	Operation implemented
save caller's register window	save rs_1, rs_2, rd	res = reg[rs_1]+reg[rs_2] CWP = (CWP−1) % NWINDOWS reg[rd] = res
	save rs_1, $siconst_{13}$, rd	res = reg[rs_1]+$siconst_{13}$ CWP = (CWP−1) % NWINDOWS reg[rd] = res
restore caller's register window	restore rs_1, rs_2, rd	res = reg[rs_1]+reg[rs_2] CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore rs, $siconst_{13}$, rd	res = reg[rs]+$siconst_{13}$ CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore	CWP = (CWP+1) % NWINDOWS

A Typical Calling Sequence

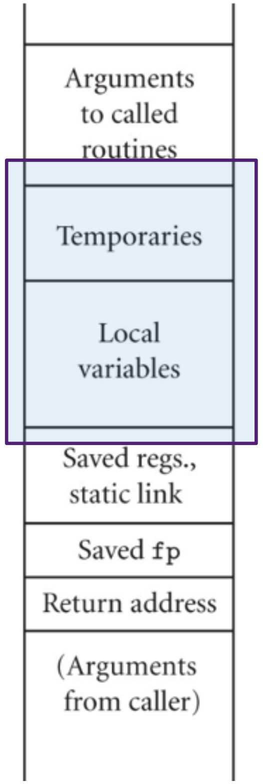
To maintain this stack layout, the calling sequence might operate as follows.
The caller
1. saves any caller-saves registers whose values will be needed after the call
2. computes the values of arguments and moves them into the stack or registers
3. computes the static link (if this is a language with nested subroutines), and passes it as an extra, hidden argument
4. uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register
In its prologue, the callee
1. allocates a frame by subtracting an appropriate constant from the sp
2. saves the old frame pointer into the stack, and assigns it an appropriate new Value
3. saves any callee-saves registers that may be overwritten by the current routine (including the static link and return address, if they were passed in registers)
After the subroutine has completed, the epilogue
1. moves the return value (if any) into a register or a reserved location in the stack
2. restores callee-saves registers if needed
3. restores the fp and the sp
4. jumps back to the return address
Finally, the caller
1. moves the return value to wherever it is needed
2. restores caller-saves registers if needed



Calling Sequences (LLVM on ARM)

Caller

- saves into the “local variable and temporaries” area any caller-saves registers whose values are still needed
- puts up to 4 small arguments into registers r0-r3
- puts the rest of the arguments into the argument build area at the top of the current frame
- does b1 or b1x, which puts return address into register lr, jumps to target address, and (optionally) changes instruction set coding



Low Level Virtual Machine (LLVM)
LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala, and Swift.

Calling Sequences (LLVM on ARM)

- **In prolog, Callee**
  - pushes necessary registers onto stack
  - initializes frame pointer by adding small constant to the sp placing result in r7
  - subtracts from sp to make space for local variables, temporaries, and arg build area at top of stack
- **In epilog, Callee**
  - puts return value into r0-r3 or memory (as appropriate)
  - subtracts small constant from r7, puts result in sp (effectively deallocates most of frame)
  - pops saved registers from stack, pc takes place of lr from prologue (branches to caller as side effect)

Calling Sequences (LLVM on ARM)

- After call, Caller
  - moves return value to wherever it's needed
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory

Calling Sequences (LLVM on ARM)

- This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope

Calling Sequences (LLVM on ARM)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - particularly LEAF routines (those that don't call other routines)
    - leaving things out saves time
    - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast

End of Chapter 4