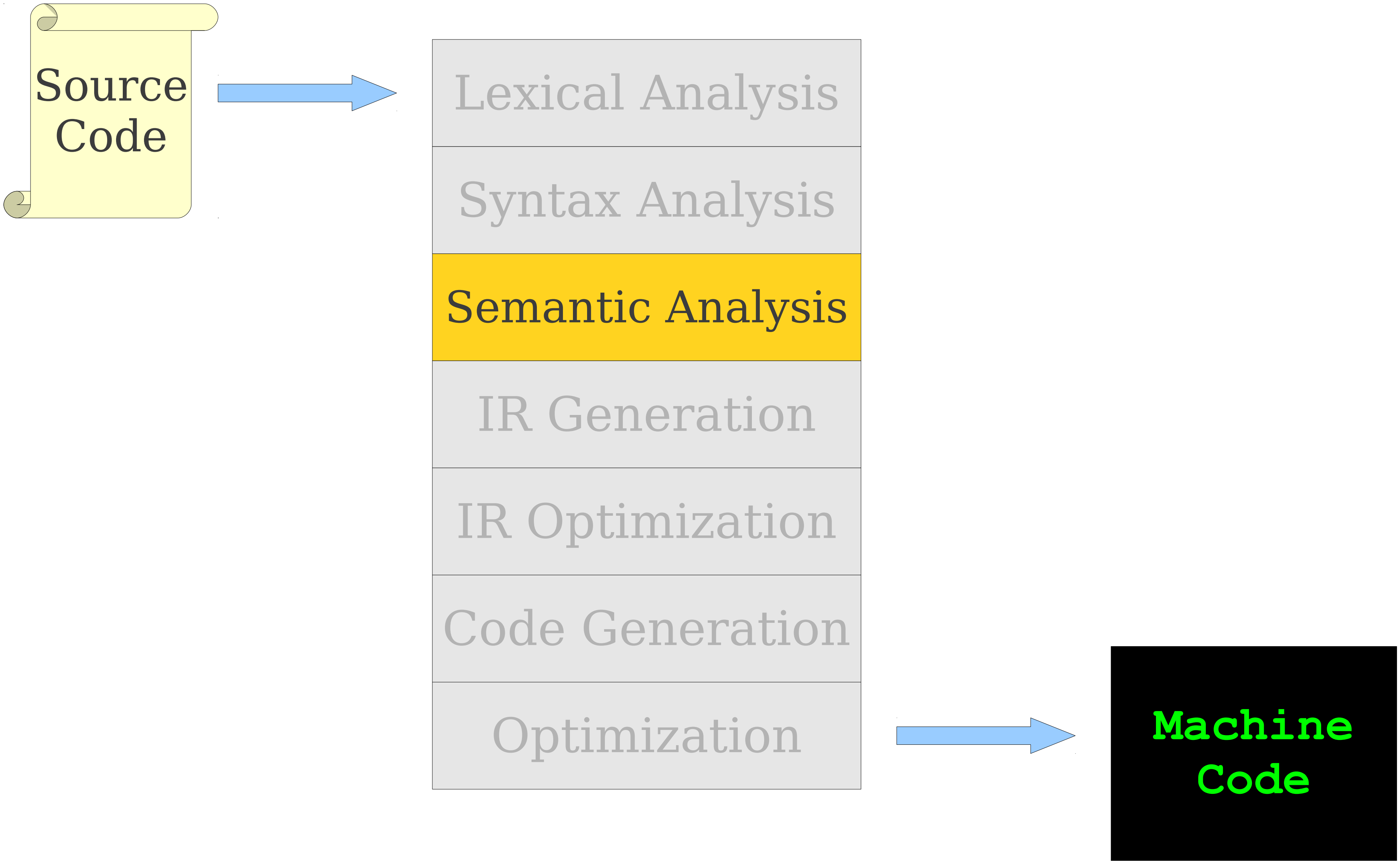
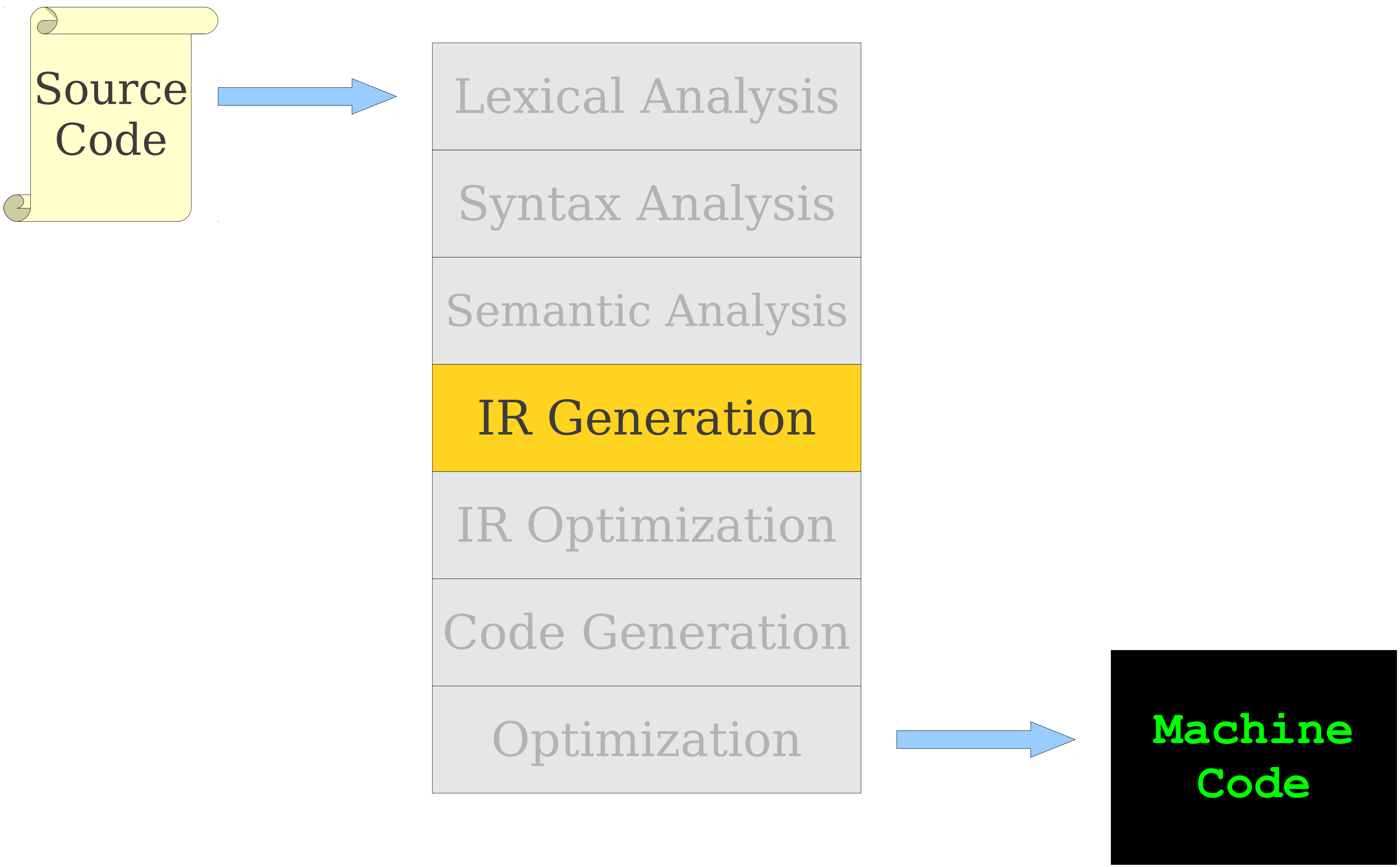
Where We Are



Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

What is IR Generation?

- **Intermediate Representation Generation**.
- The final phase of the compiler front-end.
- Goal: Translate the program into the format expected by the compiler back-end.
- Generated code need not be optimized; that's handled by later passes.
- Generated code need not be in assembly; that can also be handled by later passes.

Why Do IR Generation?

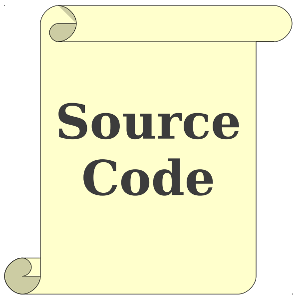
- **Simplify certain optimizations.**
  - Machine code has many constraints that inhibit optimization. (Such as?)
  - Working with an intermediate language makes optimizations easier and clearer.
- **Have many front-ends into a single back-end.**
  - **gcc** can handle C, C++, Java, Fortran, Ada, and many other languages.
  - Each front-end translates source to the GENERIC language.
- **Have many back-ends from a single front-end.**
  - Do most optimization on intermediate representation before emitting code targeted at a single machine.

Designing a Good IR

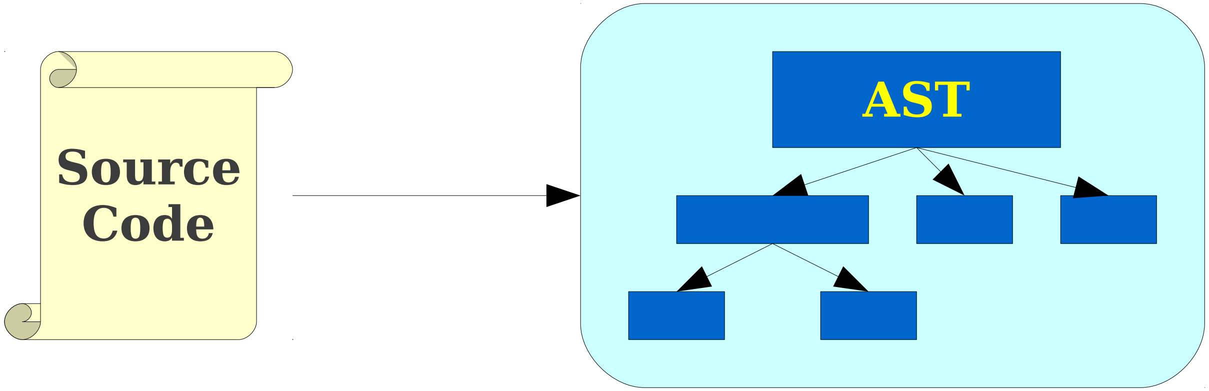
- IRs are like type systems – they're extremely hard to get right.
- Need to balance needs of high-level source language and low-level target language.
- Too high level: can't optimize certain implementation details.
- Too low level: can't use high-level knowledge to perform aggressive optimizations.
- Often have multiple IRs in a single compiler.

Architecture of `gcc`

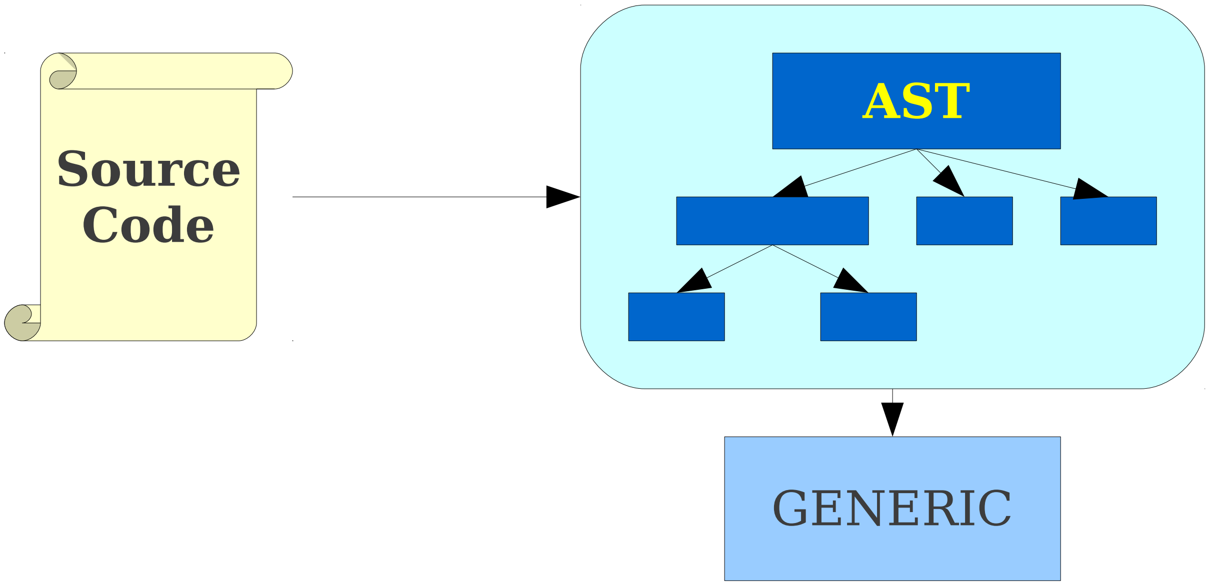
Architecture of gcc



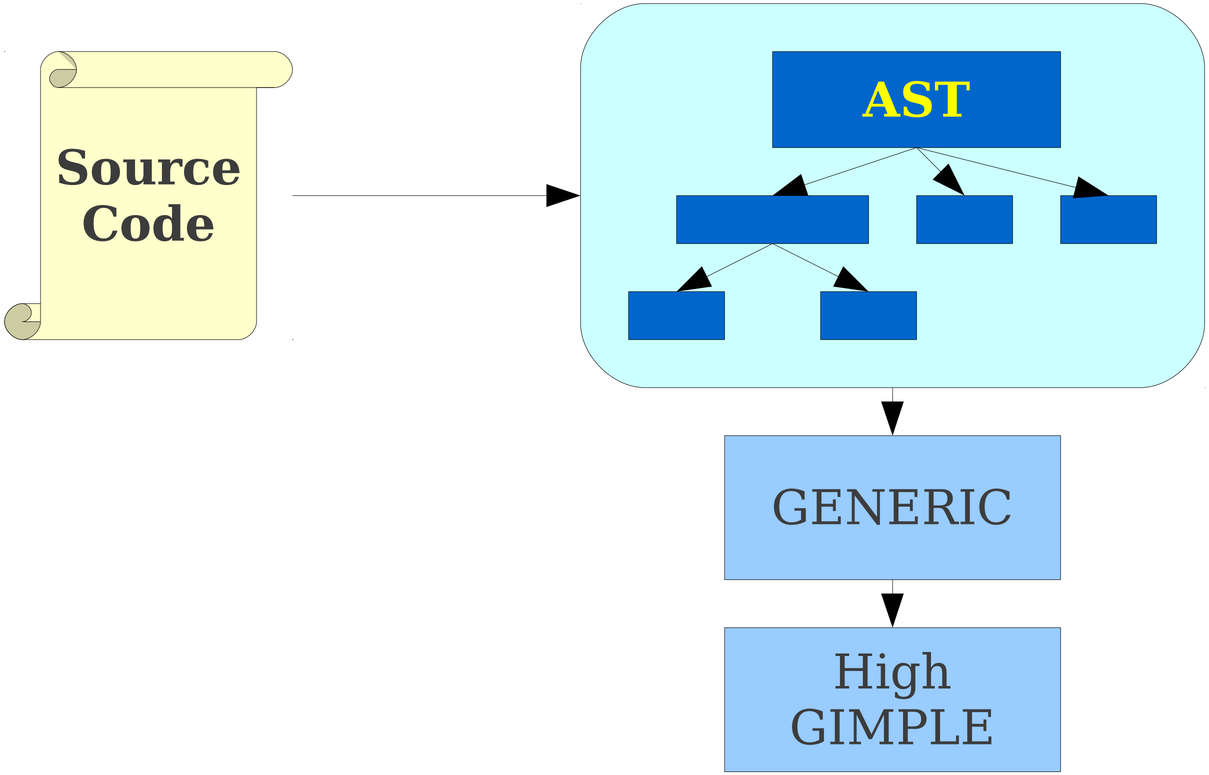
Architecture of gcc



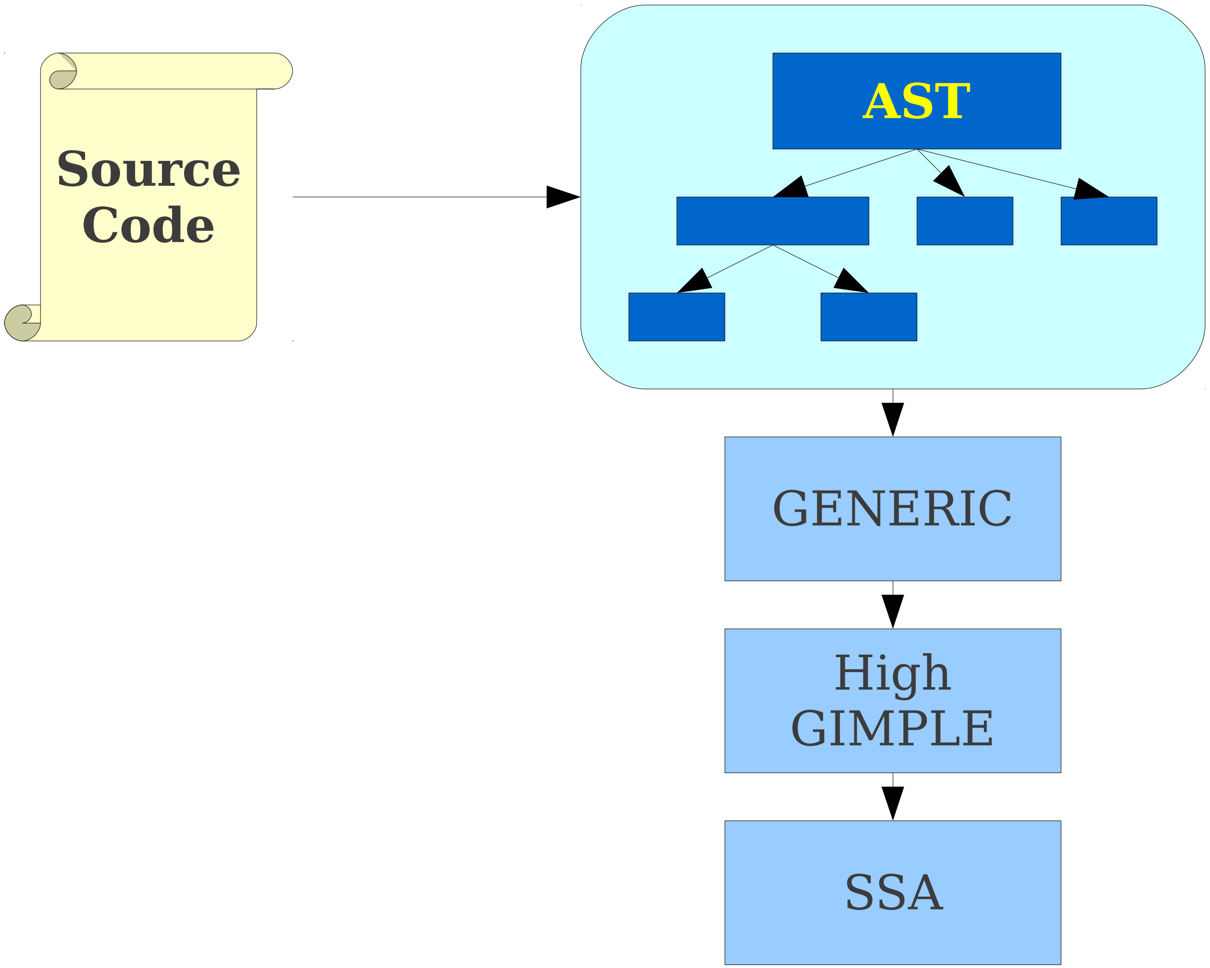
Architecture of gcc



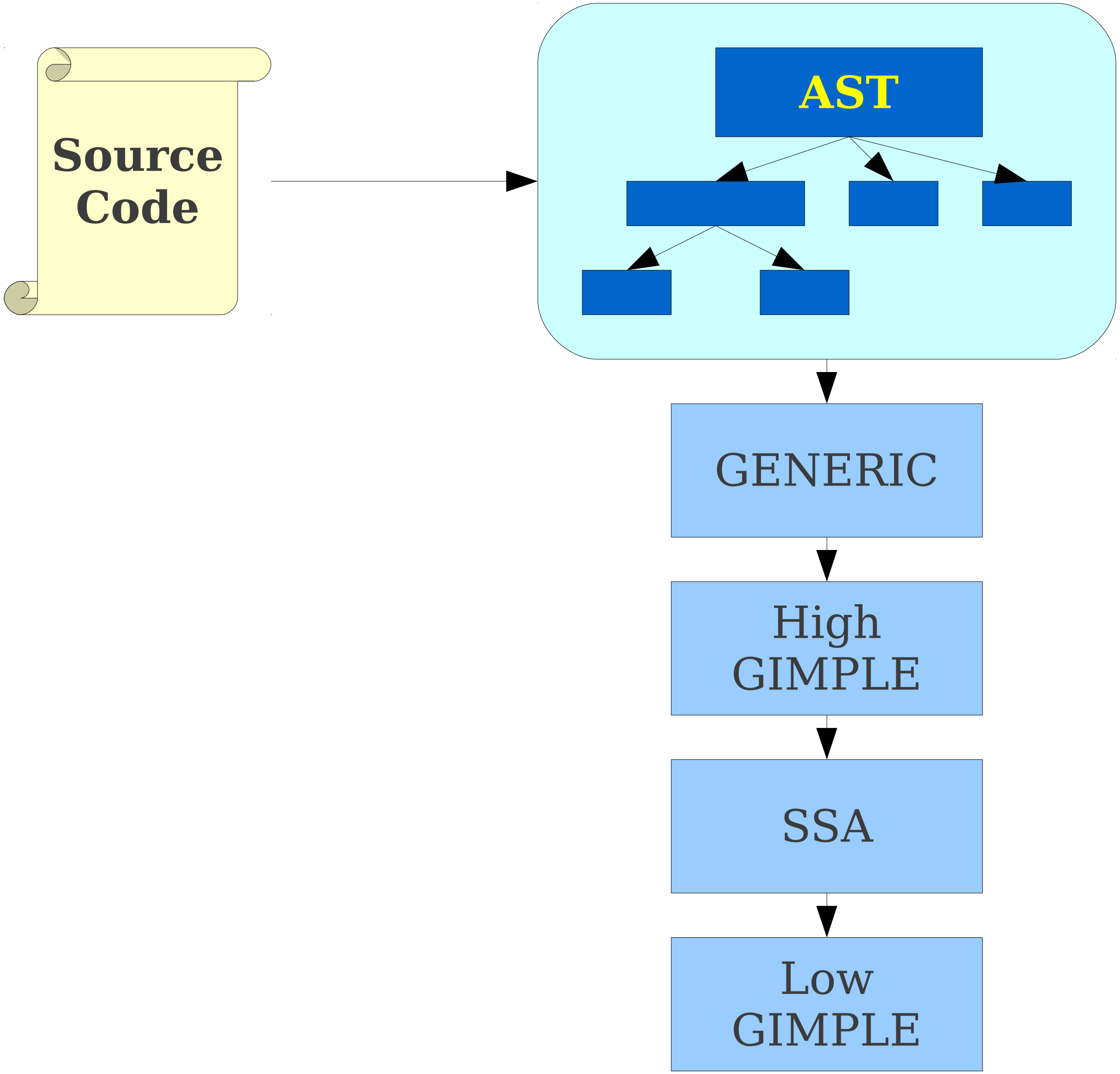
Architecture of gcc



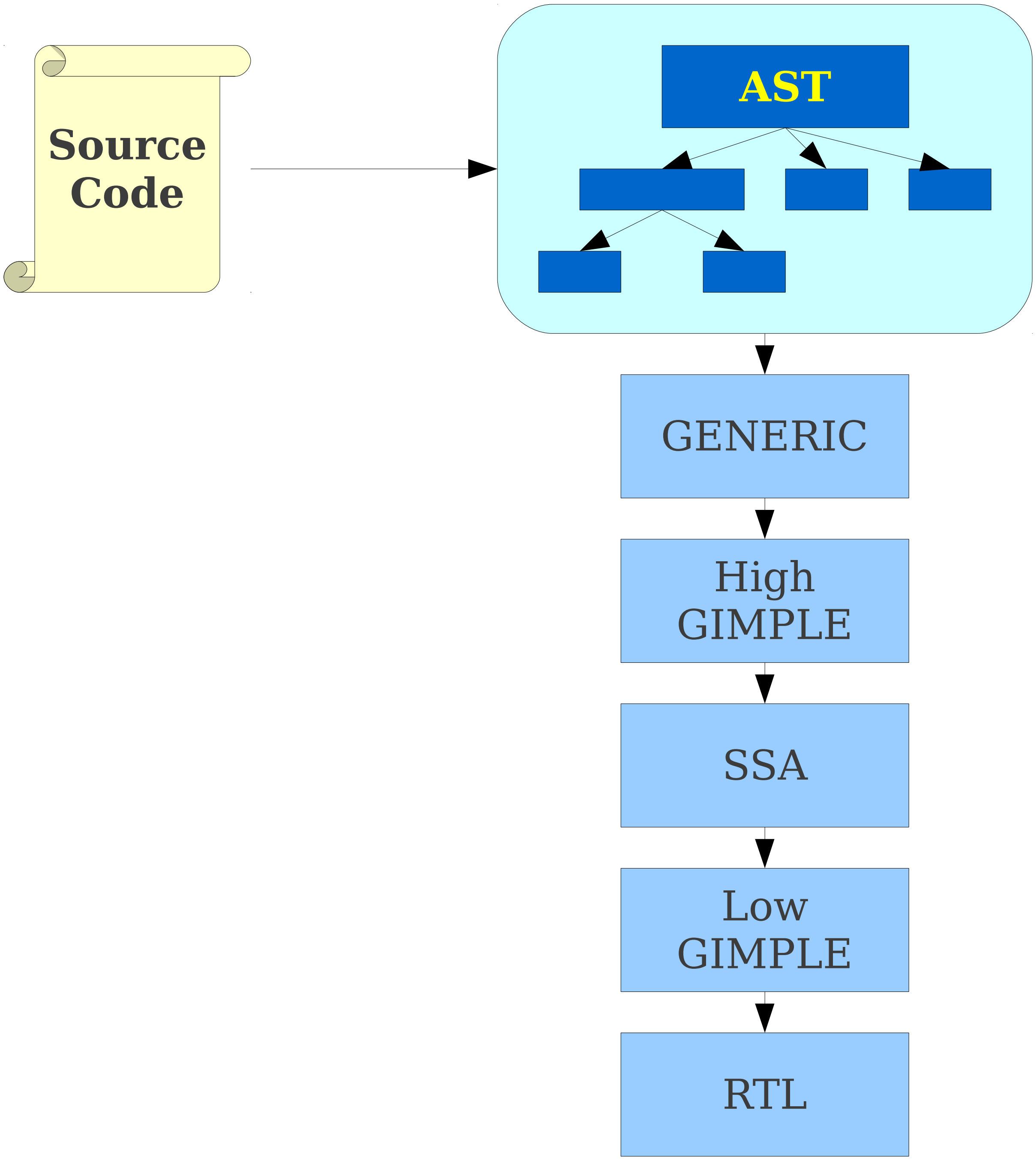
Architecture of gcc



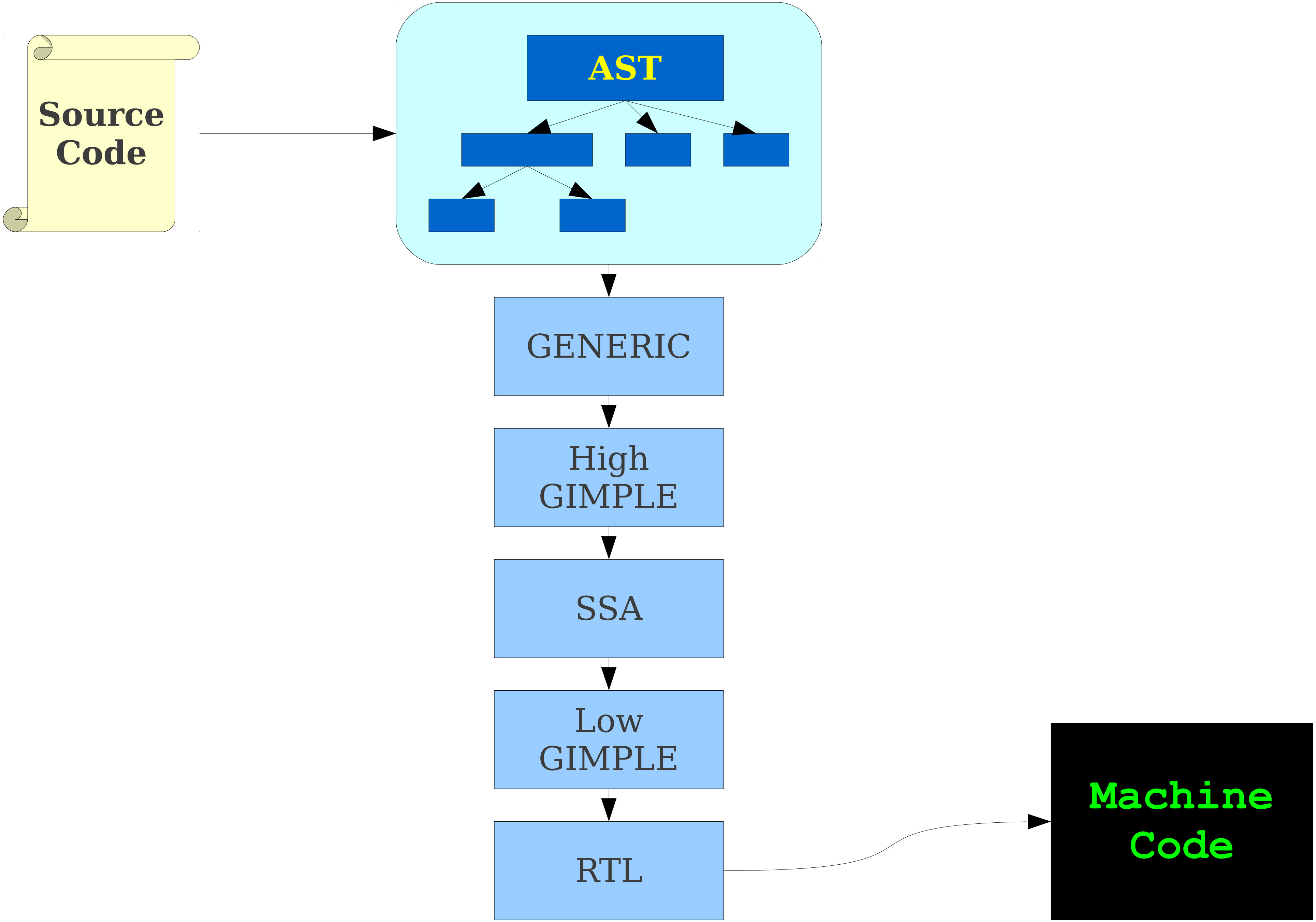
Architecture of `gcc`



Architecture of `gcc`



Architecture of `gcc`



Another Approach: High-Level IR

- Examples:
  - Java bytecode
  - CPython bytecode
  - LLVM IR
  - Microsoft CIL.
- Retains high-level program structure.
  - Try playing around with **`javap`** vs. a disassembler.
- Allows for compilation on target machines.
- Allows for JIT compilation or interpretation.

Runtime Environments

An Important Duality

- Programming languages contain high-level structures:
  - Functions
  - Objects
  - Exceptions
  - Dynamic typing
  - Lazy evaluation
  - (etc.)
- The physical computer only operates in terms of several primitive operations:
  - Arithmetic
  - Data movement
  - Control jumps

Runtime Environments

- We need to come up with a representation of these high-level structures using the low-level structures of the machine.
- A **runtime environment** is a set of data structures maintained at runtime to implement these high-level structures.
  - e.g. the stack, the heap, static area, virtual function tables, etc.
- Strongly depends on the features of both the source and target language. (e.g compiler vs. cross-compiler)
- Our IR generator will depend on how we set up our runtime environment.

Data Representations

- What do different types look like in memory?
- Machine typically supports only limited types:
  - Fixed-width integers: 8-bit, 16-bit- 32-bit, signed, unsigned, etc.
  - Floating point values: 32-bit, 64-bit, 80-bit IEEE 754.
- How do we encode our object types using these types?

Encoding Primitive Types

- Primitive integral types (**`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`unsigned`**, **`uint16_t`**, etc.) typically map directly to the underlying machine type.
- Primitive real-valued types (**`float`**, **`double`**, **`long double`**) typically map directly to underlying machine type.
- Pointers typically implemented as integers holding memory addresses.
  - Size of integer depends on machine architecture; hence 32-bit compatibility mode on 64-bit machines.

Encoding Arrays

Encoding Arrays

- C-style arrays: Elements laid out consecutively in memory.

`Arr[0]`	**`Arr[1]`**	**`Arr[2]`**	**`...`**	**`Arr[n-1]`**

Encoding Arrays

- C-style arrays: Elements laid out consecutively in memory.

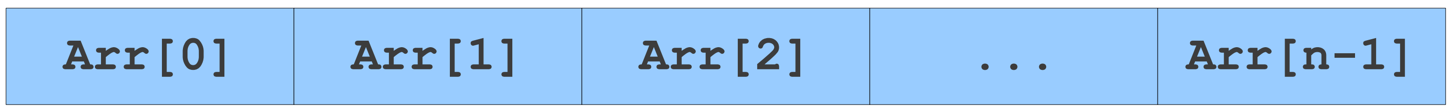
Arr[0]	**Arr[1]**	**Arr[2]**	**...**	**Arr[n-1]**

- Java-style arrays: Elements laid out consecutively in memory with size information prepended.

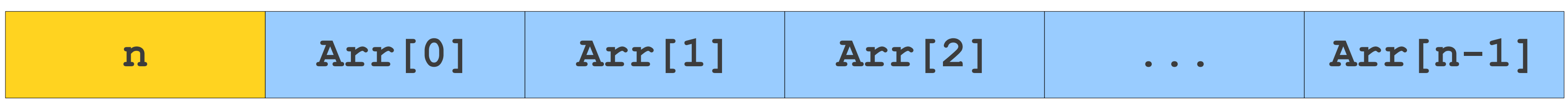
n	**Arr[0]**	**Arr[1]**	**Arr[2]**	**...**	**Arr[n-1]**

Encoding Arrays

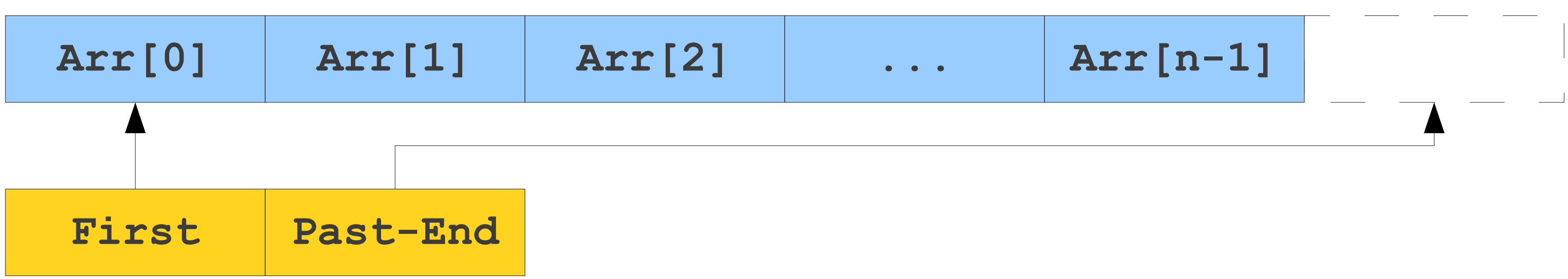
- C-style arrays: Elements laid out consecutively in memory.



- Java-style arrays: Elements laid out consecutively in memory with size information prepended.

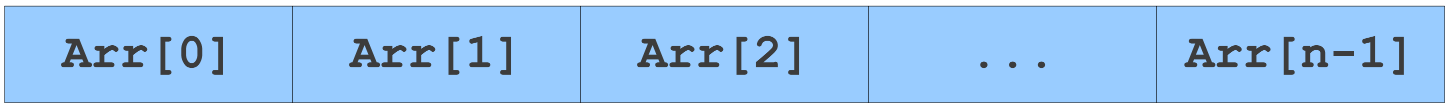


- D-style arrays: Elements laid out consecutively in memory; array variables store pointers to first and past-the-end elements.

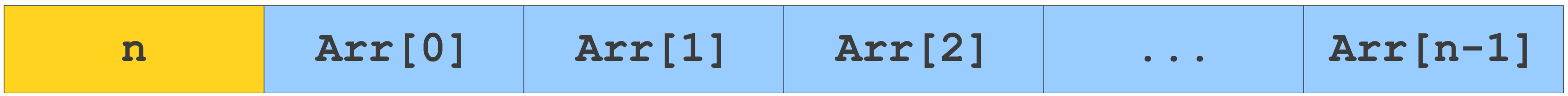


Encoding Arrays

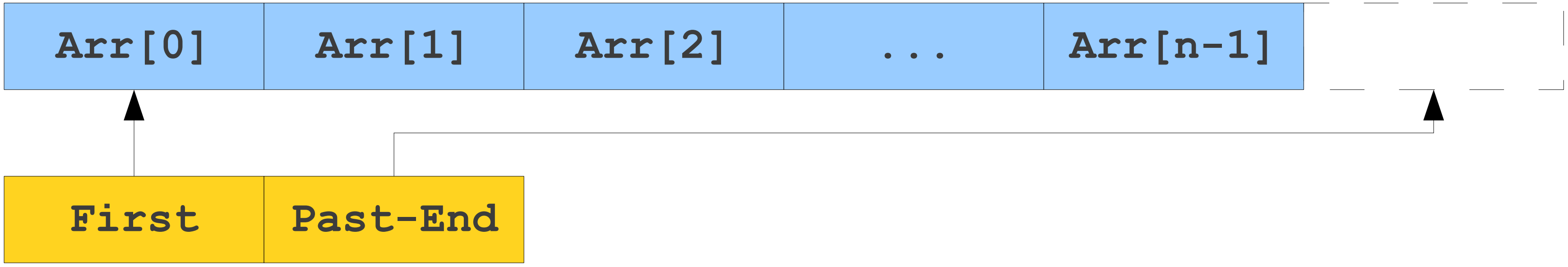
- C-style arrays: Elements laid out consecutively in memory.



- Java-style arrays: Elements laid out consecutively in memory with size information prepended.



- D-style arrays: Elements laid out consecutively in memory; array variables store pointers to first and past-the-end elements.



- (Which of these works well for Decaf?)

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

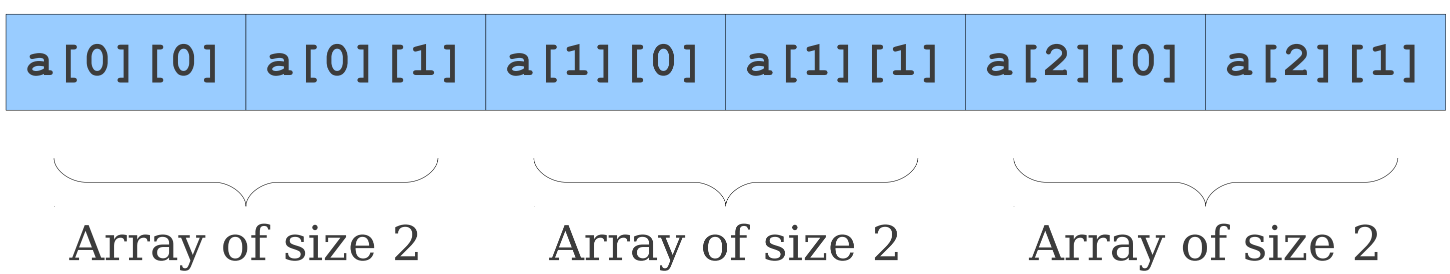
```
int a[3][2];
```

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```

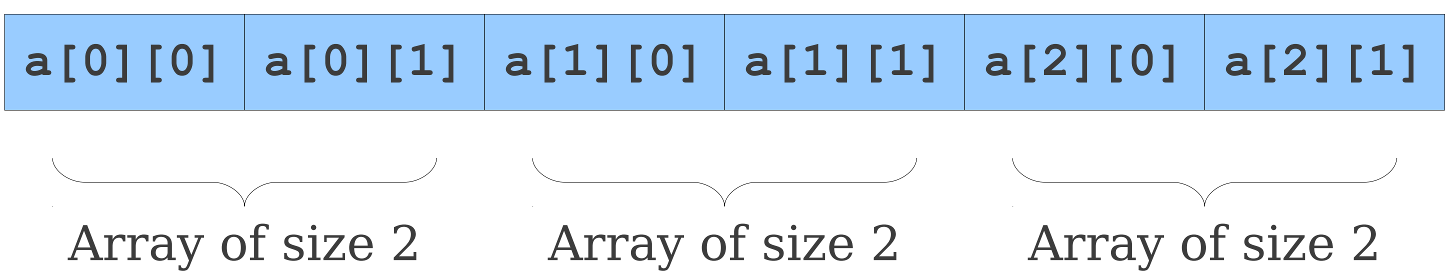


Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```

How do you know where to look for an element in an array like this?



Encoding Multidimensional Arrays

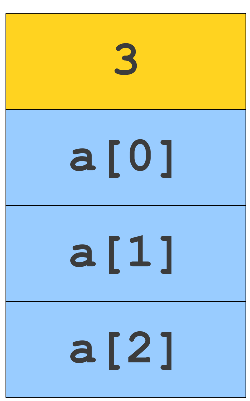
- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

```
int[][] a = new int [3][2];
```

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

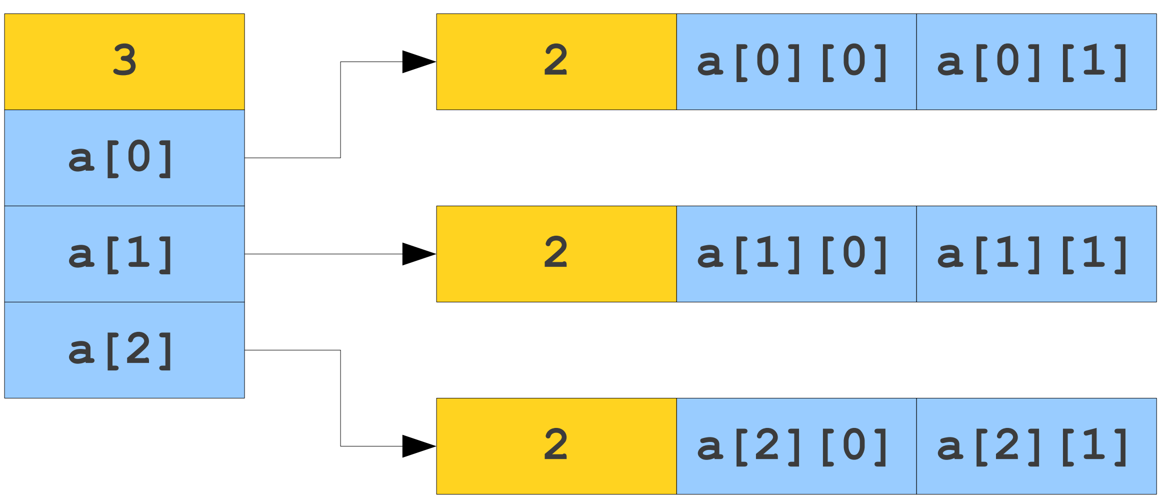
```
int[][] a = new int [3][2];
```



Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

```
int[][] a = new int [3][2];
```



Encoding Functions

- Many questions to answer:
  - What does the dynamic execution of functions look like?
  - Where is the executable code for functions located?
  - How are parameters passed in and out of functions?
  - Where are local variables stored?
- The answers strongly depend on what the language supports.

Review: The Stack

- Function calls are often implemented using a stack of **activation records** (or **stack frames**).
- Calling a function pushes a new activation record onto the stack.
- Returning from a function pops the current activation record from the stack.
- Questions:
  - **Why** does this work?
  - Does this **always** work?

Activation Trees

- An **activation tree** is a tree structure representing all of the function calls made by a program on a particular execution.
  - Depends on the runtime behavior of a program; can't always be determined at compile-time.
  - (The static equivalent is the **call graph**).
- Each node in the tree is an activation record.
- Each activation record stores a **control link** to the activation record of the function that invoked it.

Activation Trees

Activation Trees

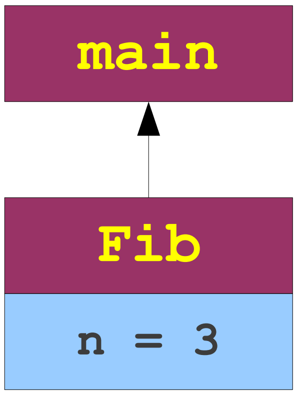
```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

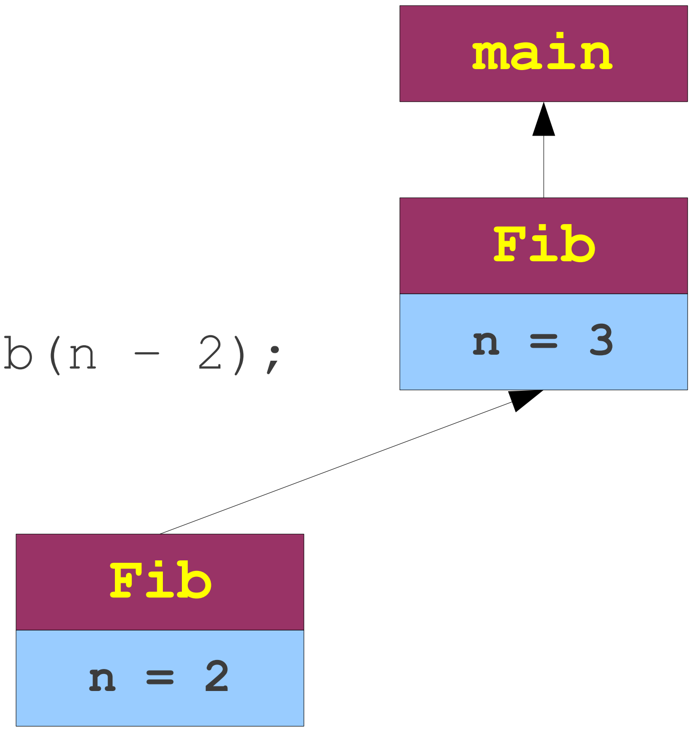
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



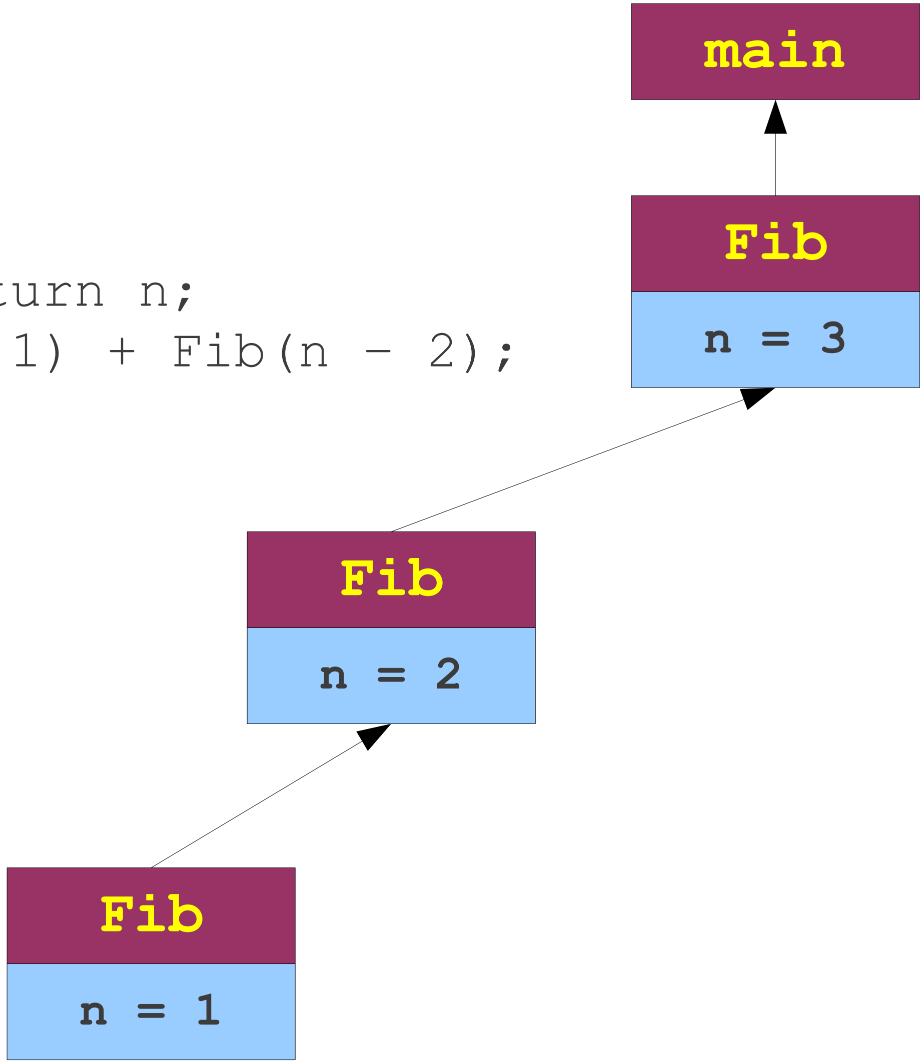
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



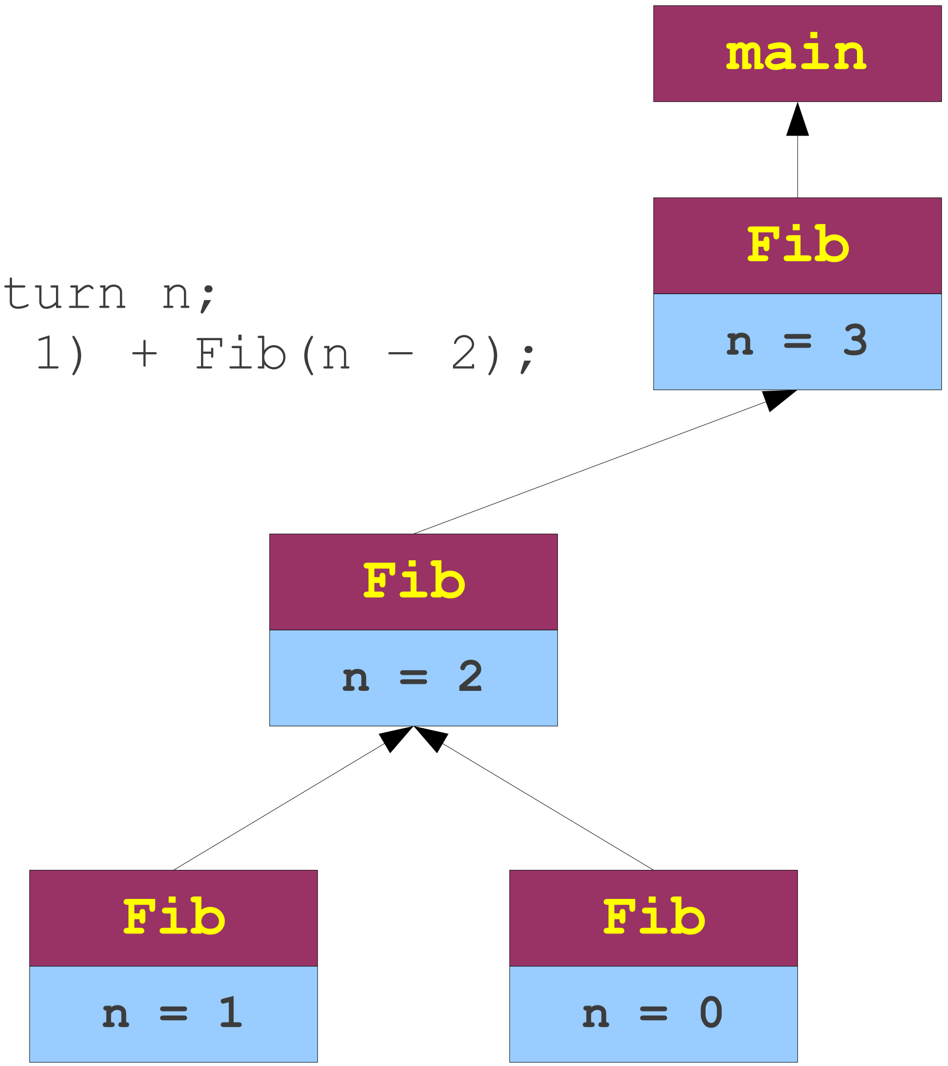
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



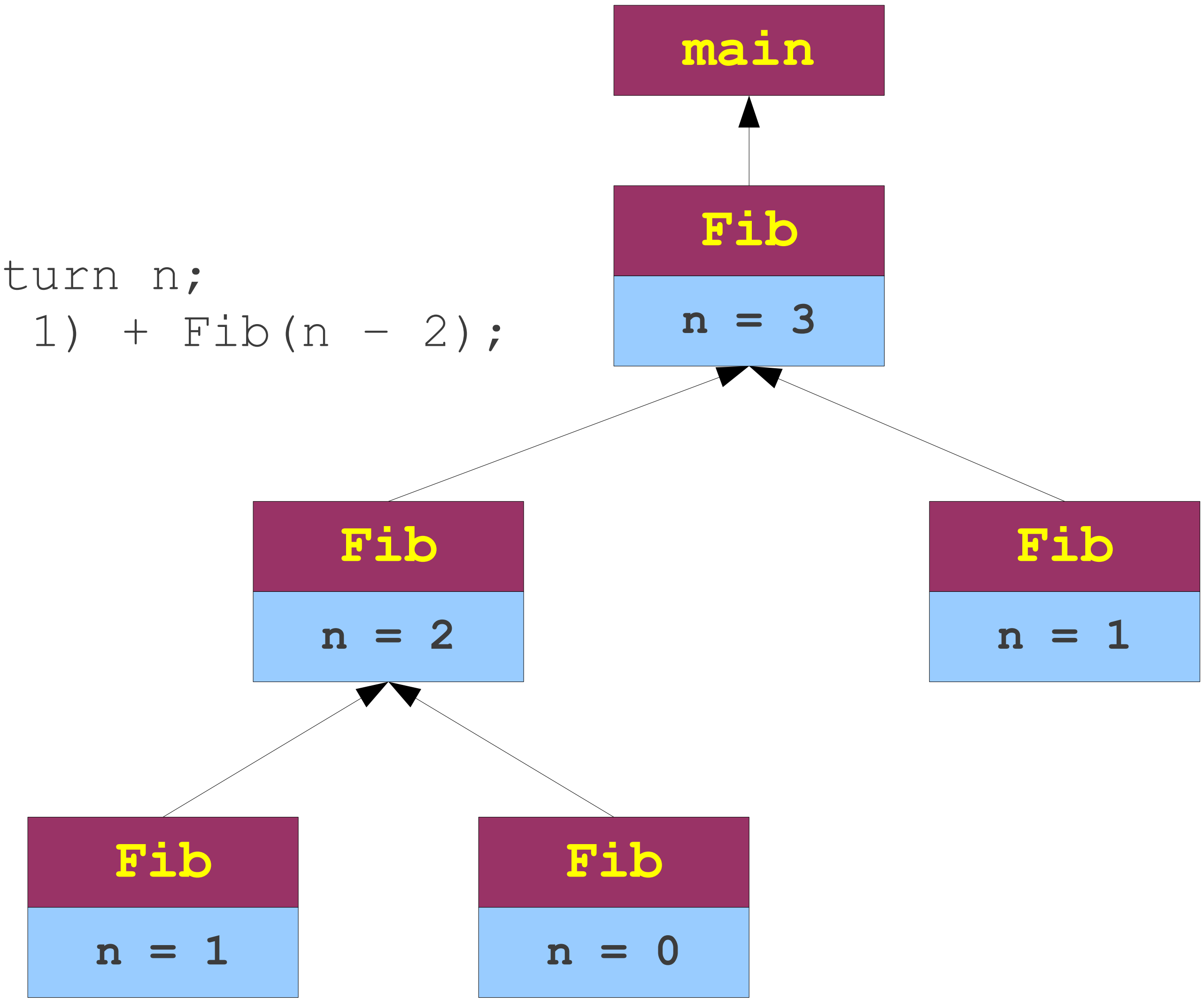
Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



Activation Trees

```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



An activation tree is a **spaghetti stack**.

The runtime stack is an **optimization** of this spaghetti stack.

Why Can We Optimize the Stack?

- Once a function returns, its activation record cannot be referenced again.
  - We don't need to store old nodes in the activation tree.
- Every activation record has either finished executing or is an ancestor of the current activation record.
  - We don't need to keep multiple branches alive at any one time.
- **These are not always true!**

Breaking Assumption 1

- **“Once a function returns, its activation record cannot be referenced again.”**
- Any ideas on how to break this?

Breaking Assumption 1

- **“Once a function returns, its activation record cannot be referenced again.”**
- Any ideas on how to break this?
- One option: **Closures**

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}
```

Breaking Assumption 1

- **"Once a function returns, its activation record cannot be referenced again."**
- Any ideas on how to break this?
- One option: **Closures**

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}
```

Closures

Closures

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```

Closures

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```

```
>
```

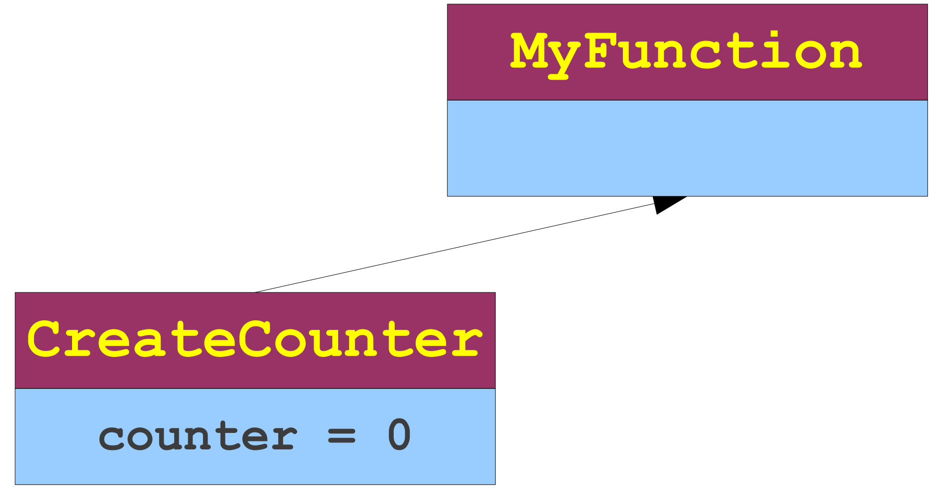
Closures

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```


```
>
```

Closures

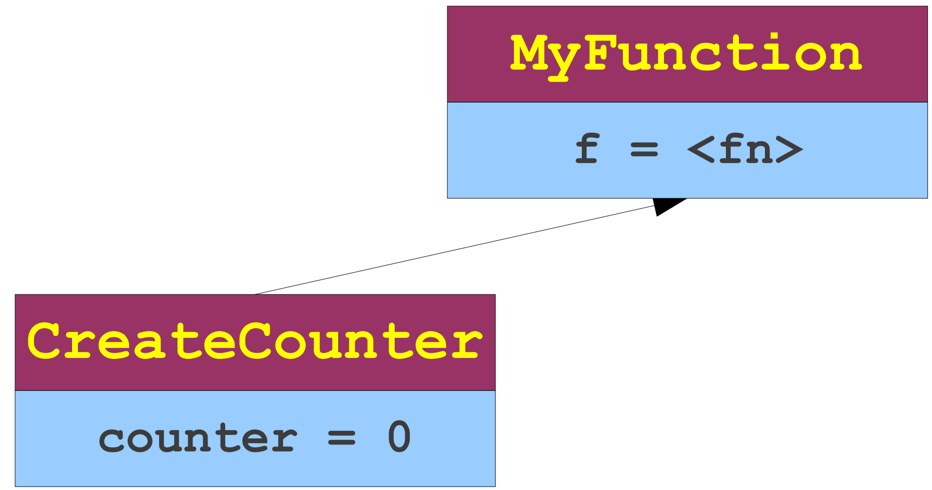
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

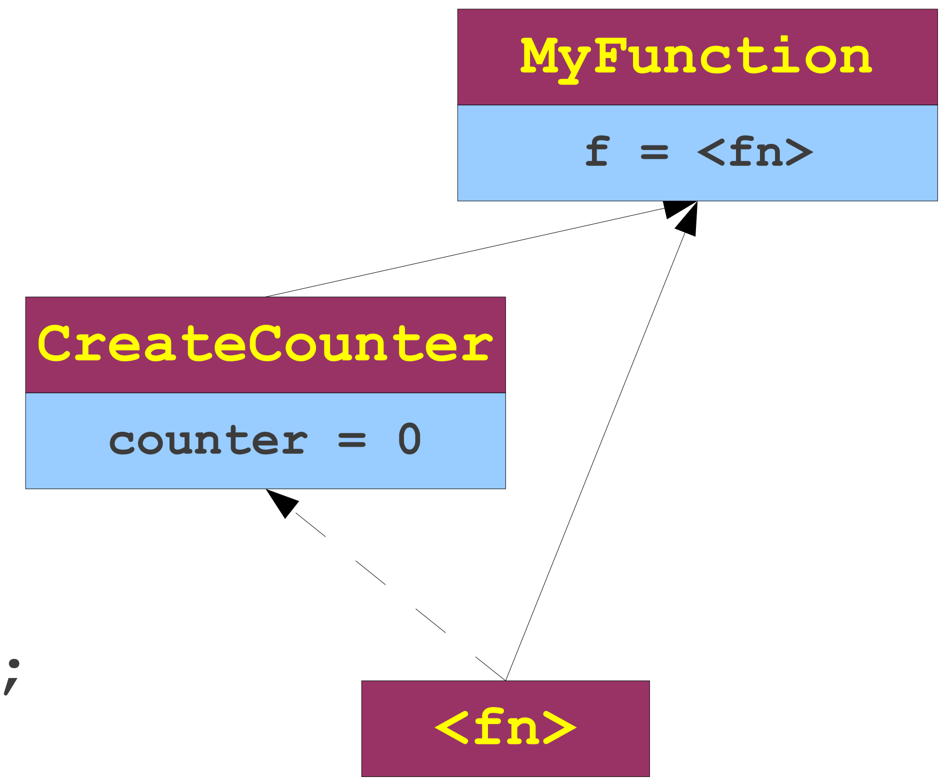
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

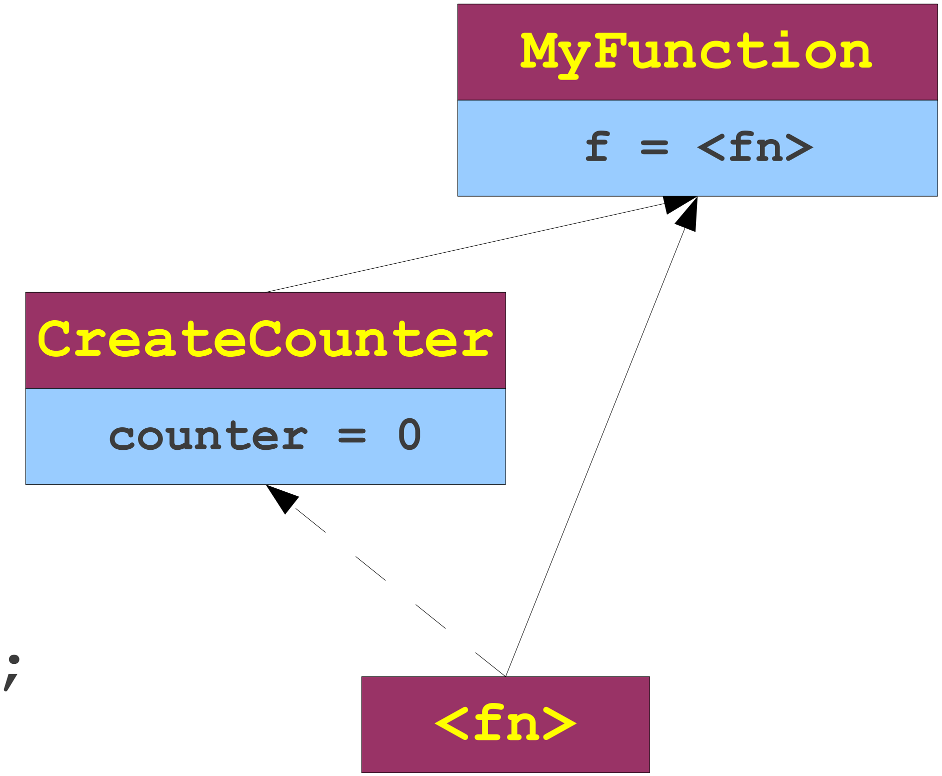
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

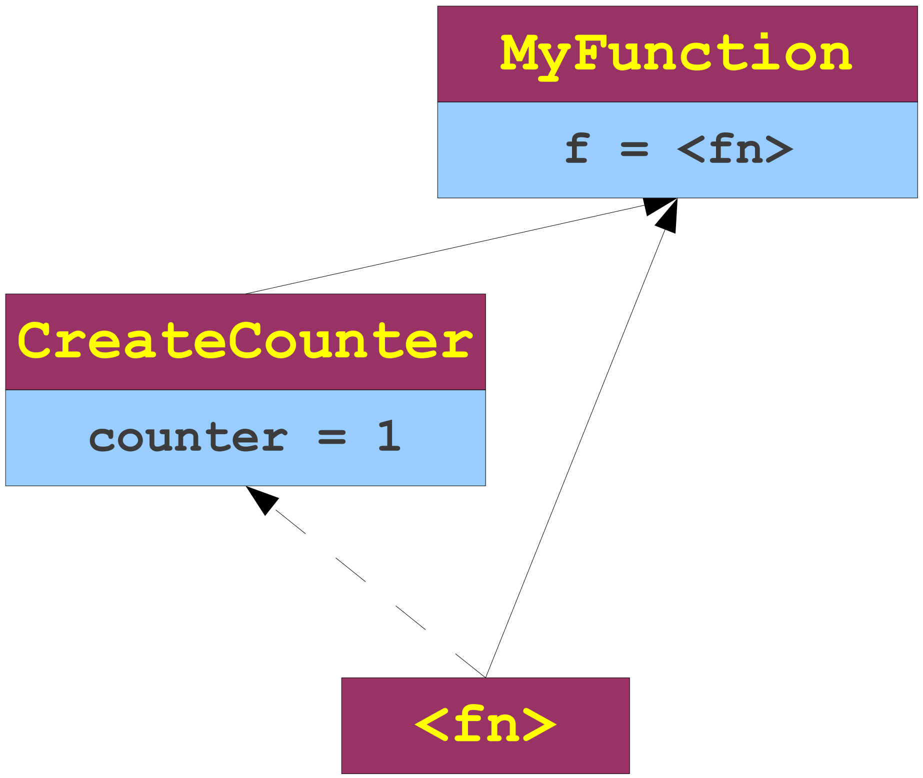
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

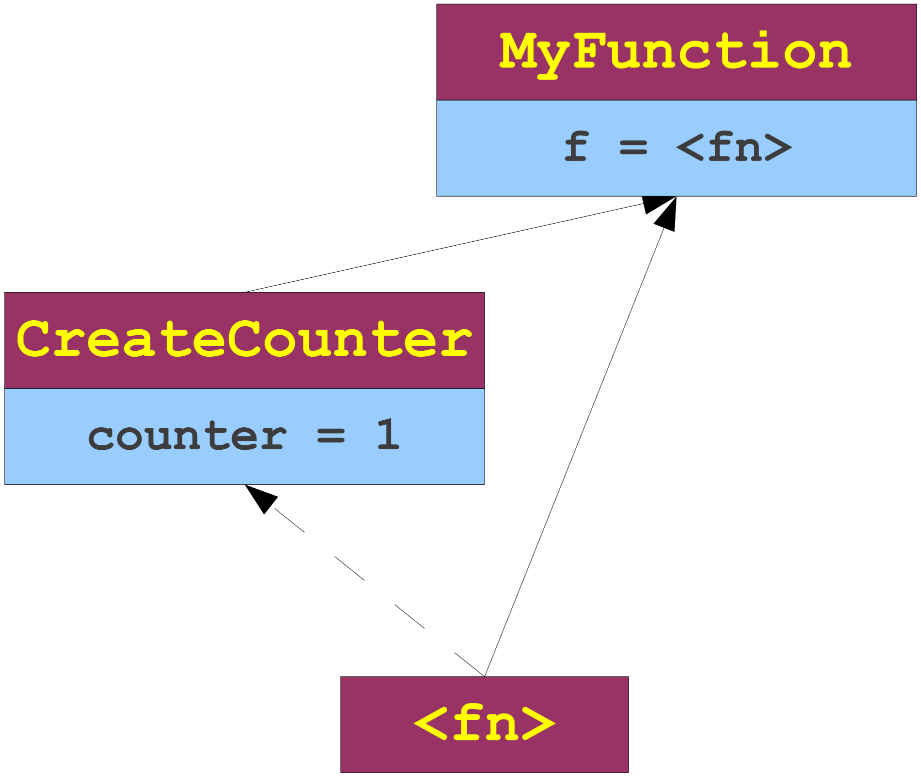
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

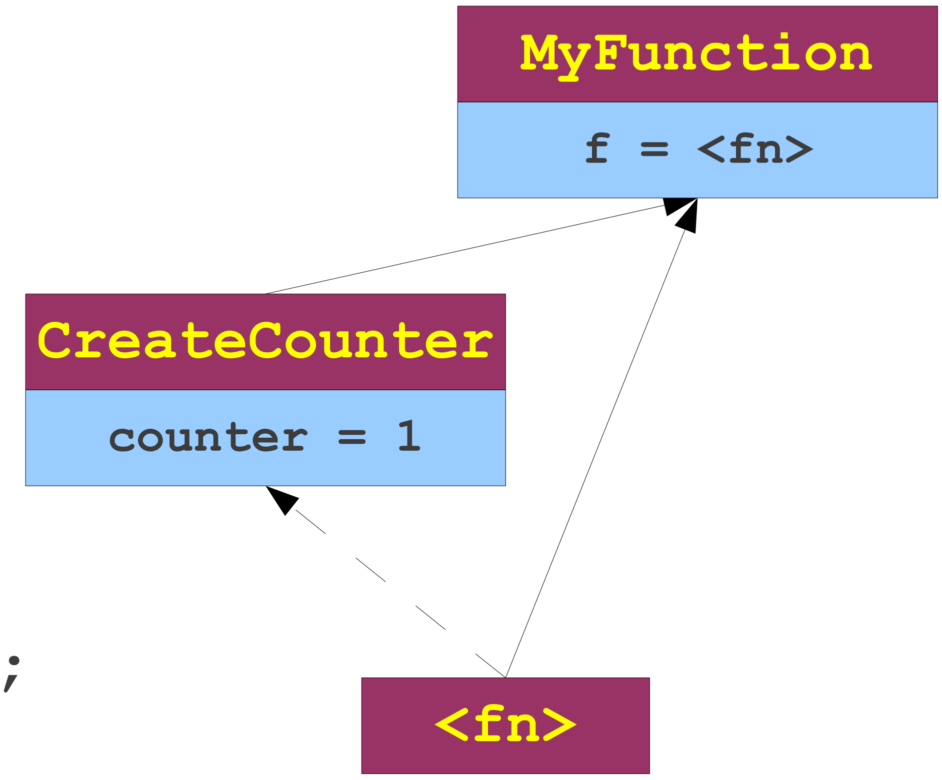
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
>
```

Closures

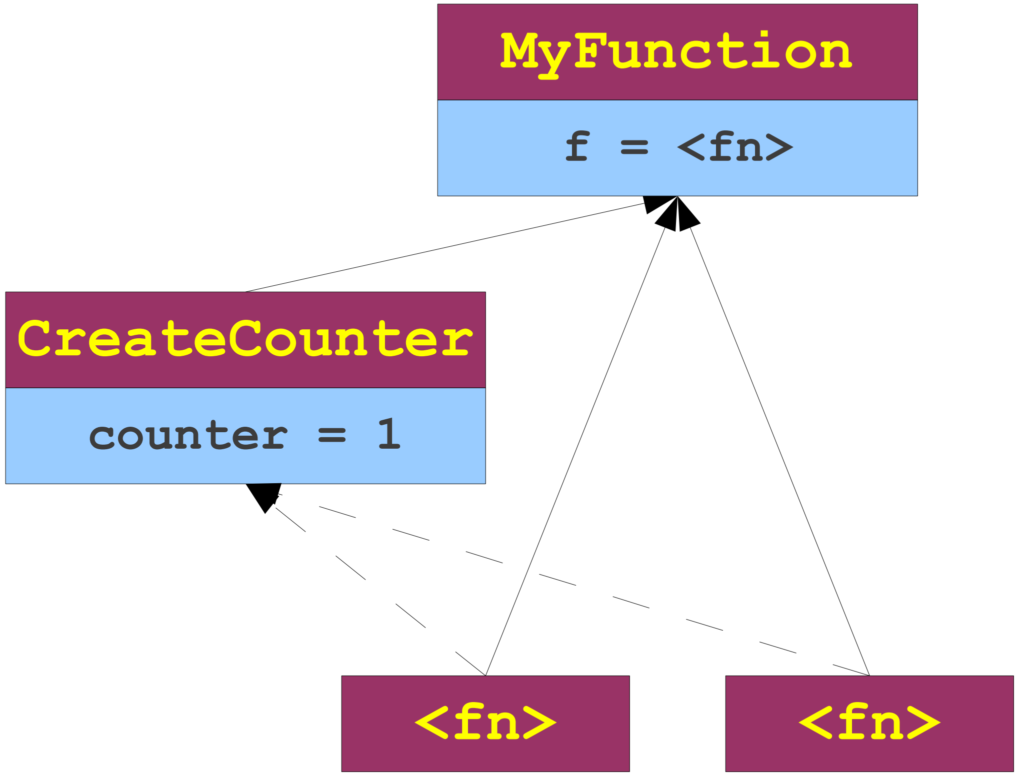
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
```

Closures

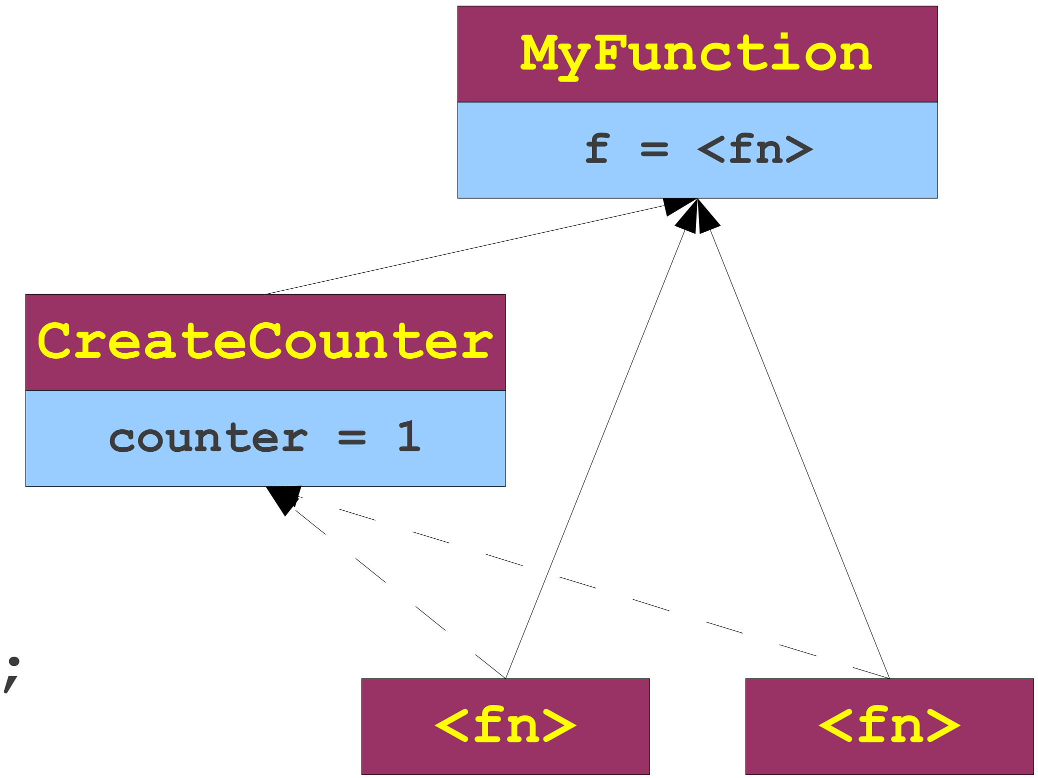
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
```

Closures

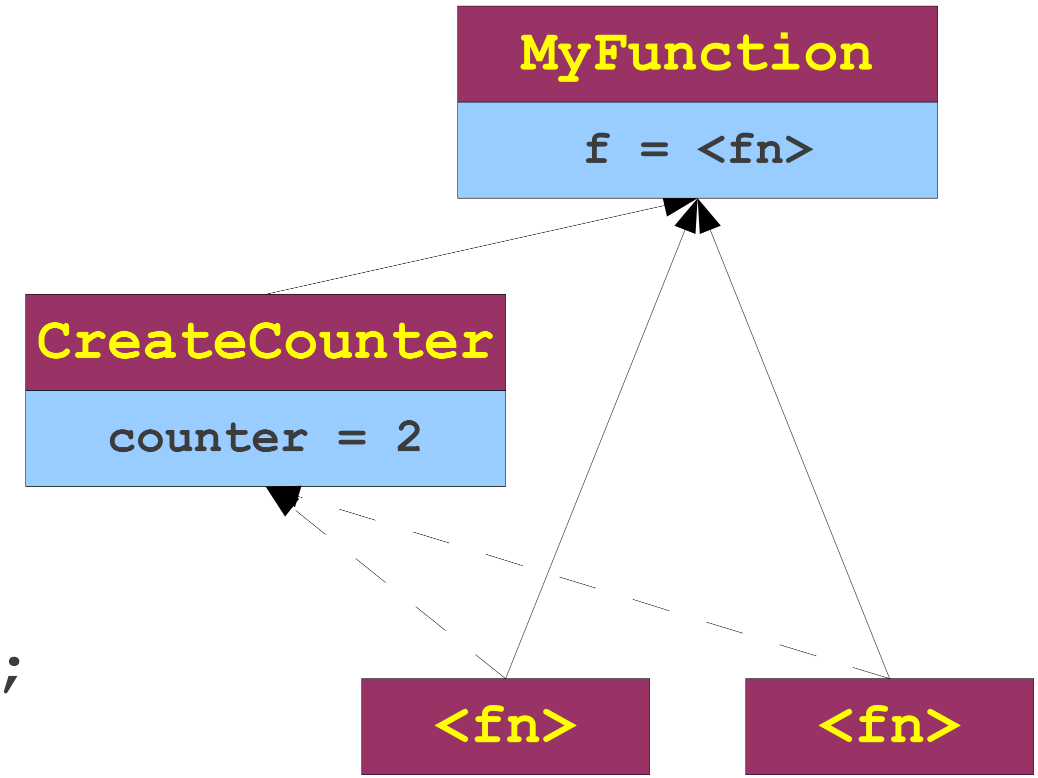
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
```

Closures

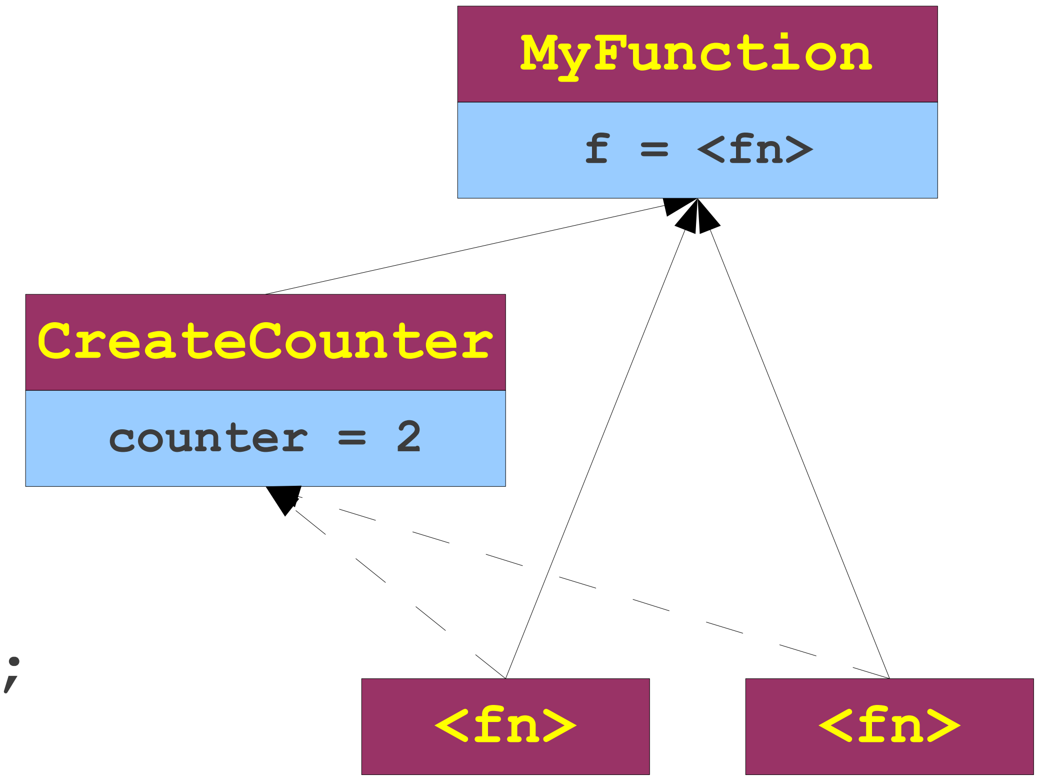
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
```

Closures

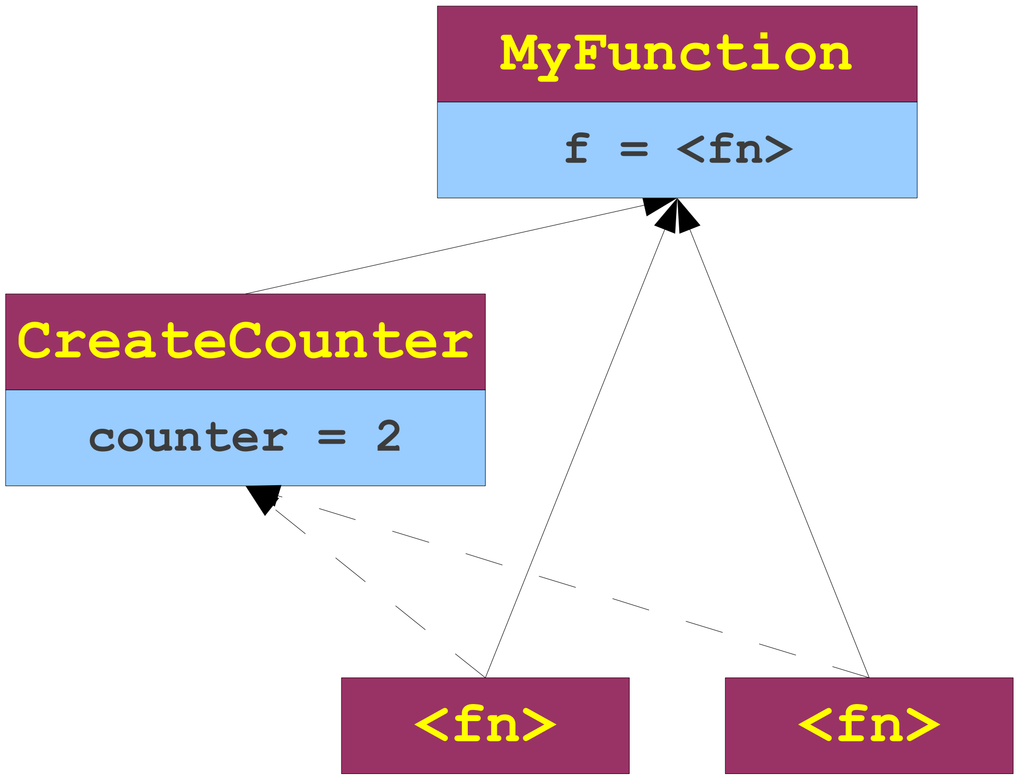
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
```

Closures

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
  2
```

Control and Access Links

- The **control link** of a function is a pointer to the function that called it.
  - Used to determine where to resume execution after the function returns.
- The **access link** of a function is a pointer to the activation record in which the function was created.
  - Used by nested functions to determine the location of variables from the outer scope.

Closures and the Runtime Stack

- Languages supporting closures do not typically have a runtime stack.
- Activation records typically dynamically allocated and garbage collected.
- Interesting exception: `gcc` C allows for nested functions, but uses a runtime stack.
- Behavior is undefined if nested function accesses data from its enclosing function once that function returns.
  - (Why?)

Breaking Assumption 2

- **“Every activation record has either finished executing or is an ancestor of the current activation record.”**
- Any ideas on how to break this?

Breaking Assumption 2

- **“Every activation record has either finished executing or is an ancestor of the current activation record.”**
- Any ideas on how to break this?
- One idea: **Coroutines**

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

Breaking Assumption 2

- **“Every activation record has either finished executing or is an ancestor of the current activation record.”**
- Any ideas on how to break this?
- One idea: **Coroutines**

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

Coroutines

Coroutines

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


def myFunc():
   for i in downFrom(3):
       print i
```

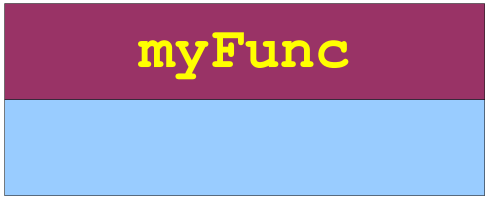
Coroutines

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


def myFunc():
    for i in downFrom(3):
        print i
```

```
>
```

Coroutines

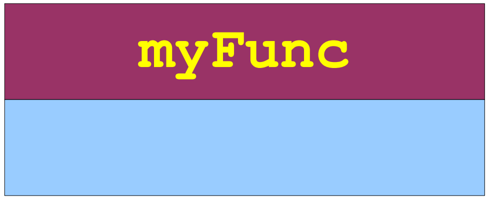
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

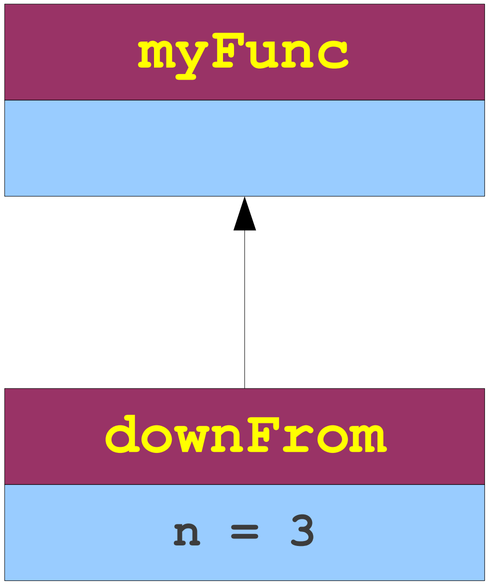
Coroutines

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


def myFunc():
    for i in downFrom(3):
        print i
```



Coroutines

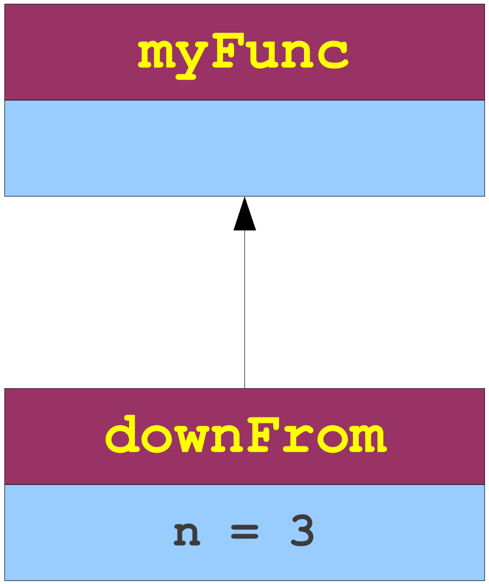
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

Coroutines

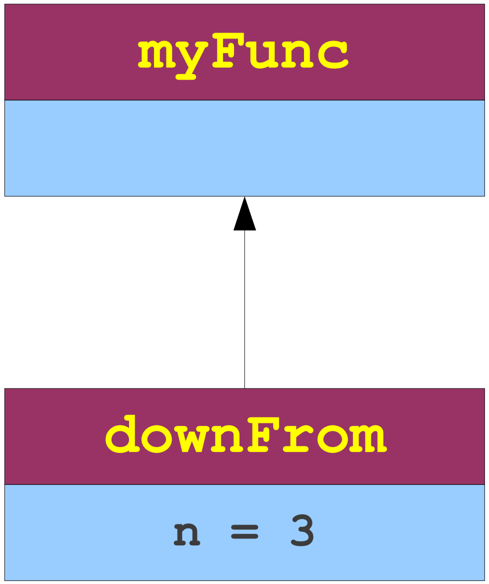
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

Coroutines

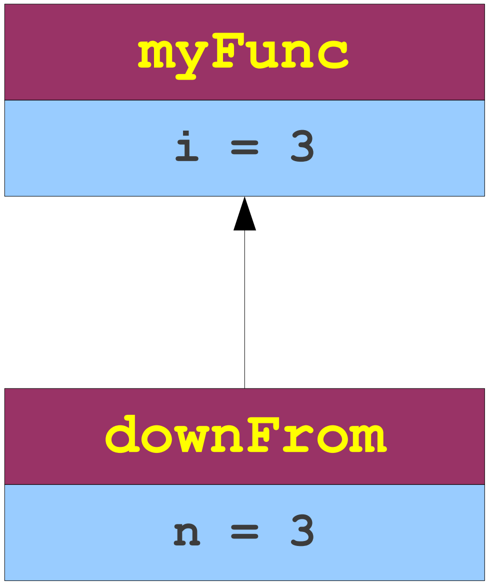
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

Coroutines

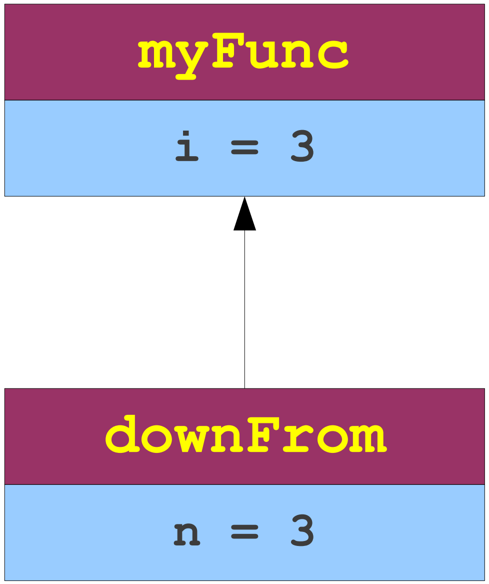
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

Coroutines

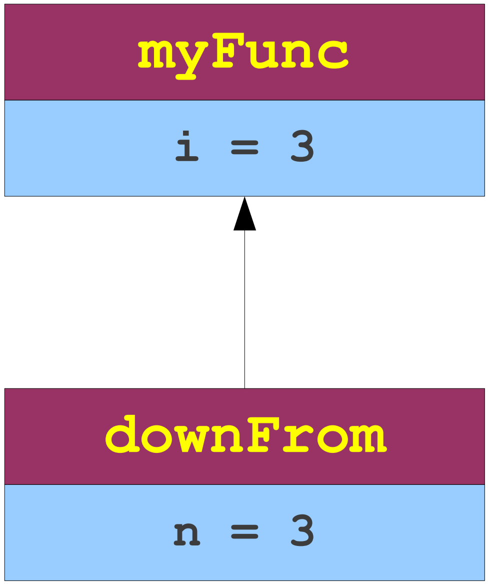
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
>
```

Coroutines

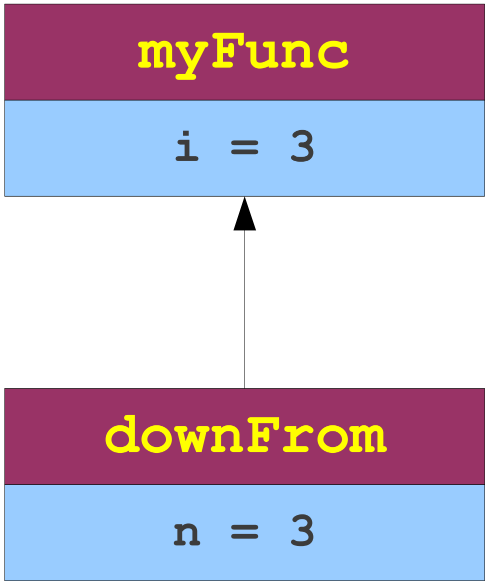
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

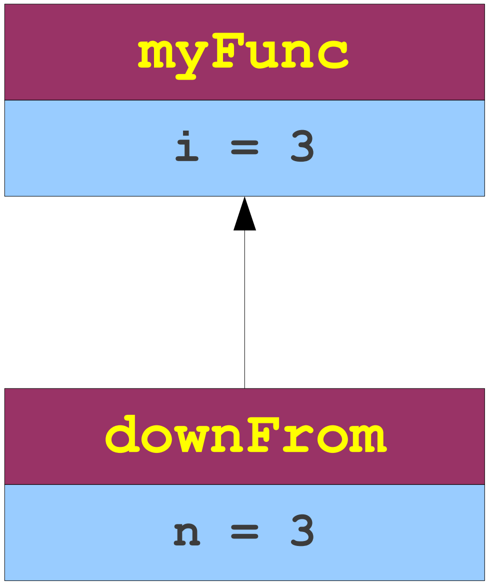
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

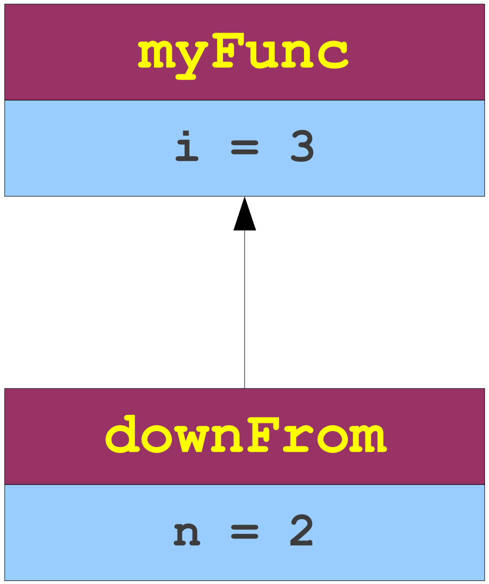
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

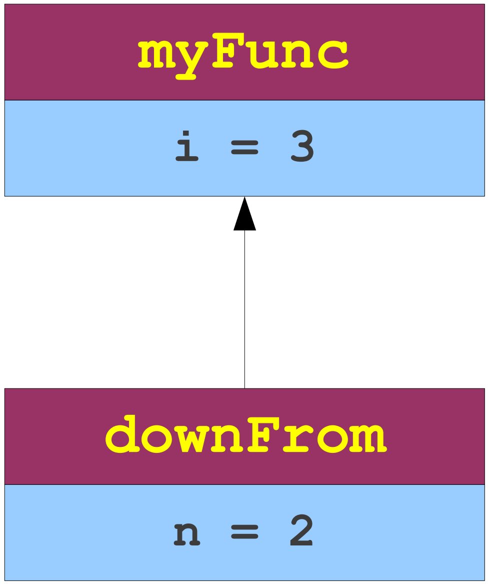
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

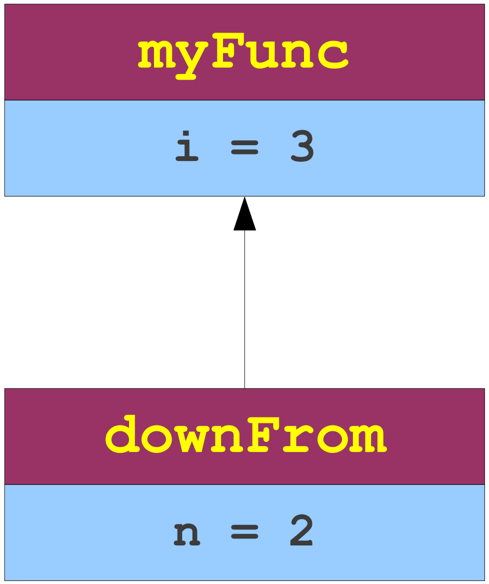
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

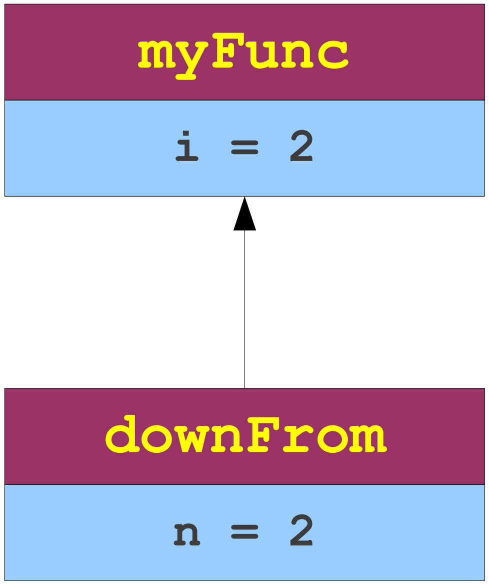
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

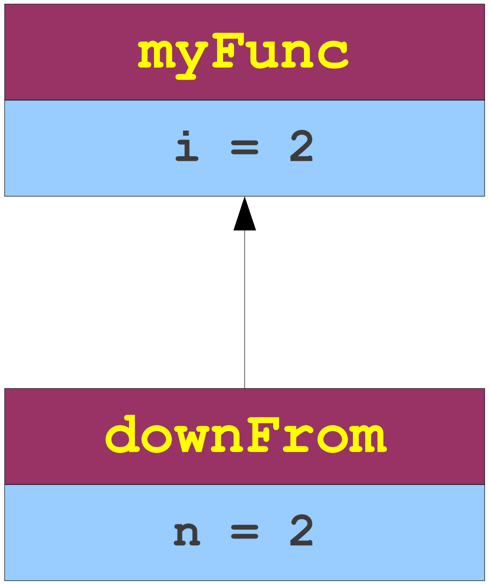
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

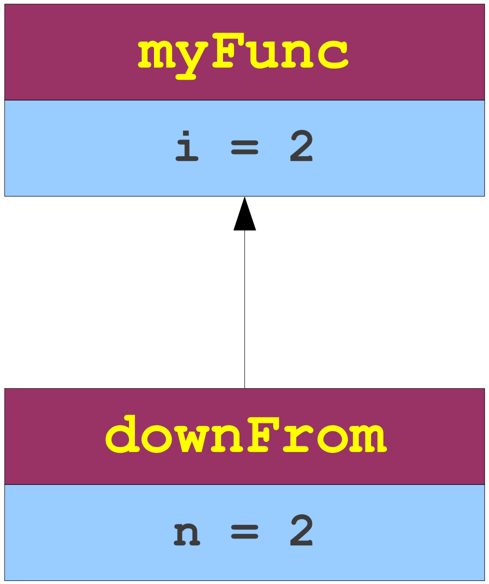
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
```

Coroutines

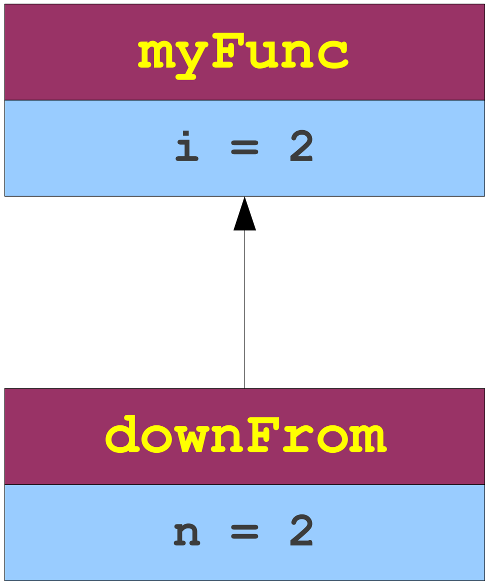
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

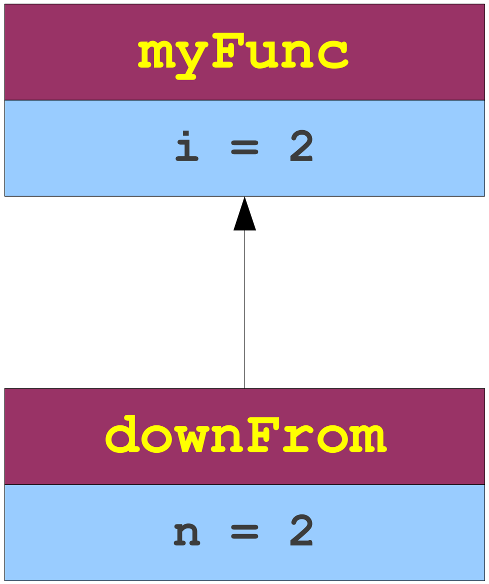
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

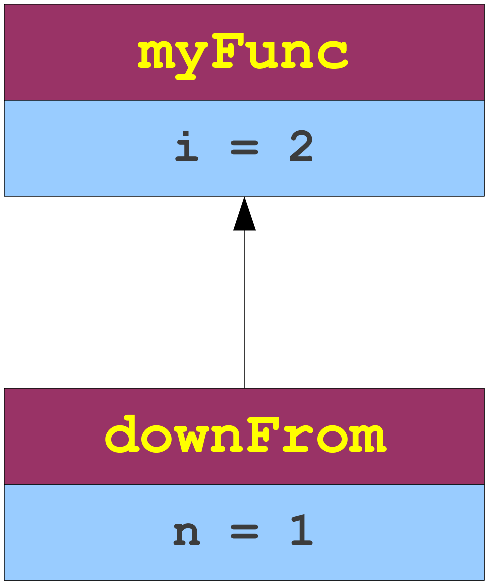
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

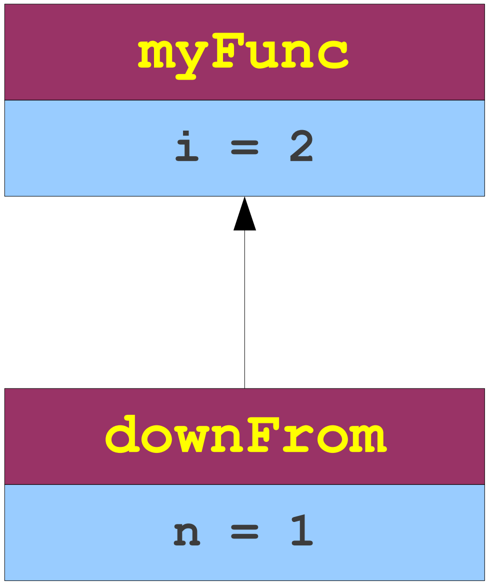
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

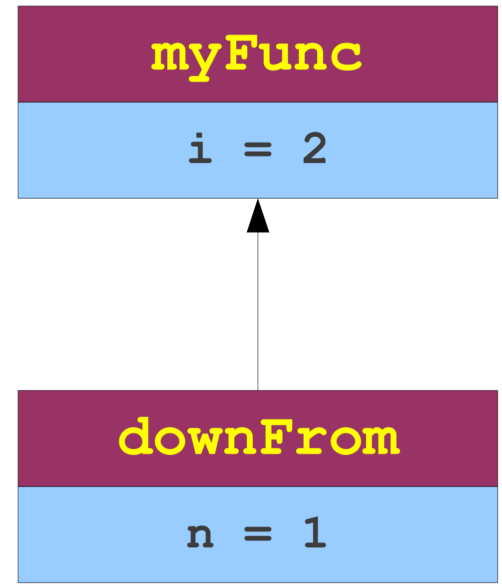
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

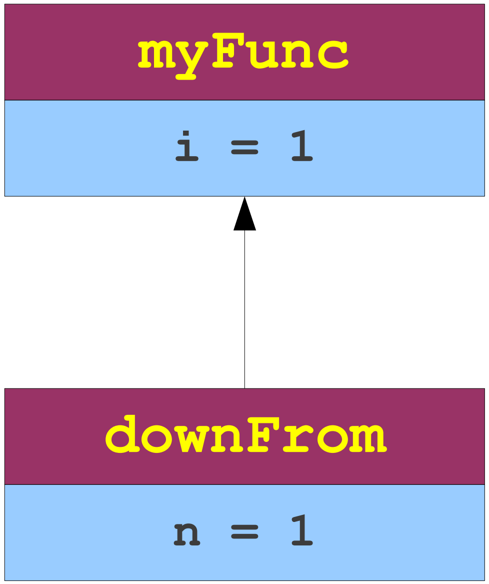
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

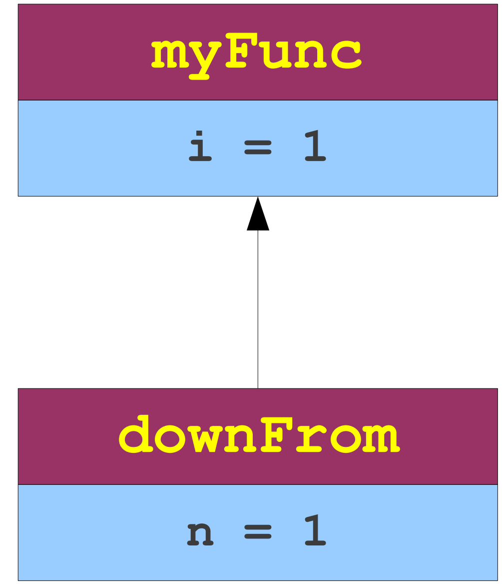
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

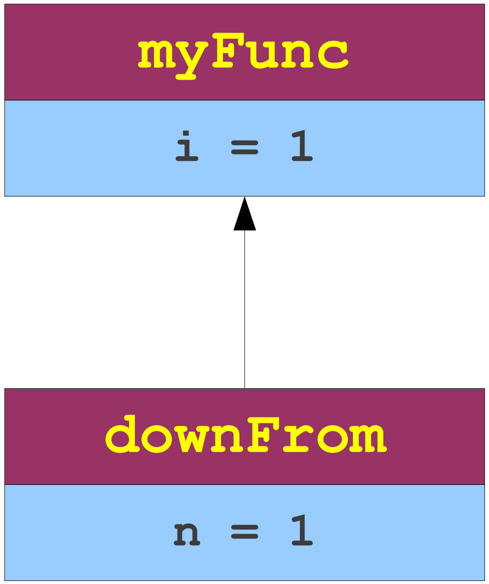
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
```

Coroutines

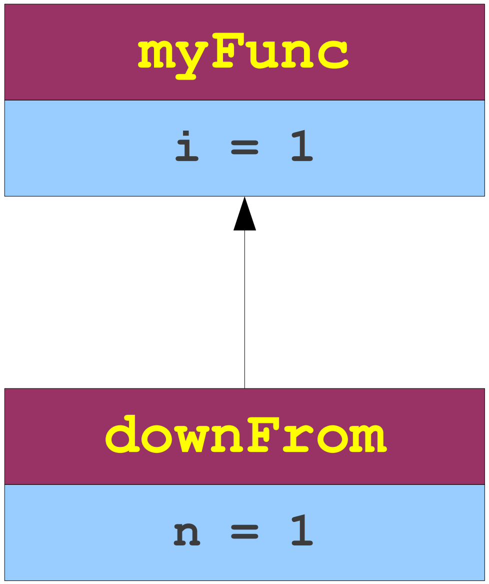
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

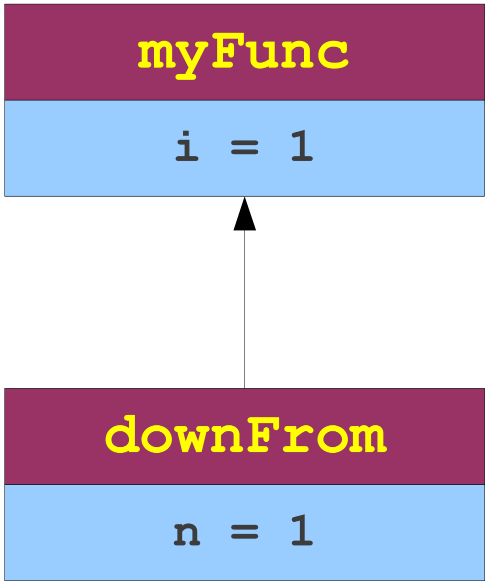
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

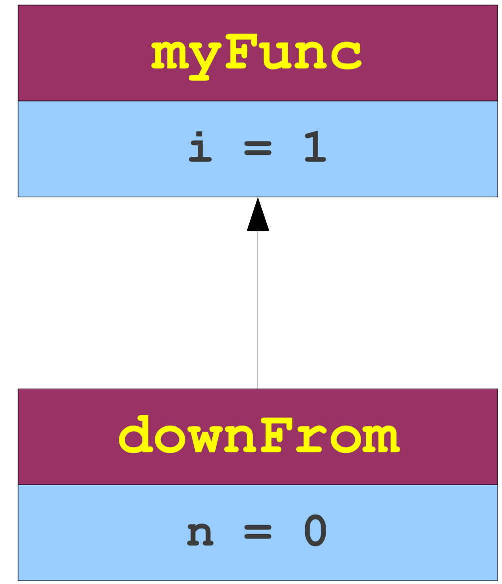
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

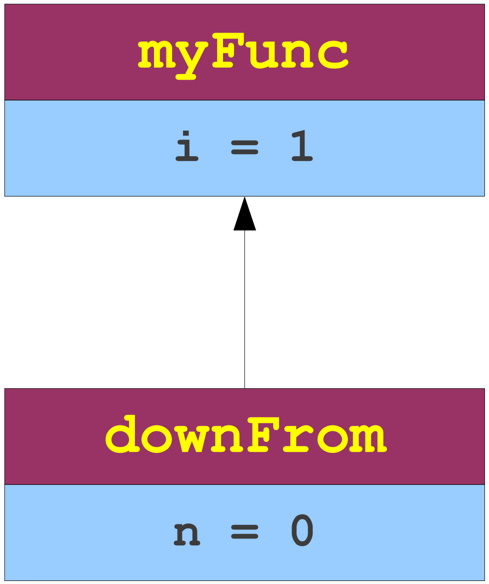
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

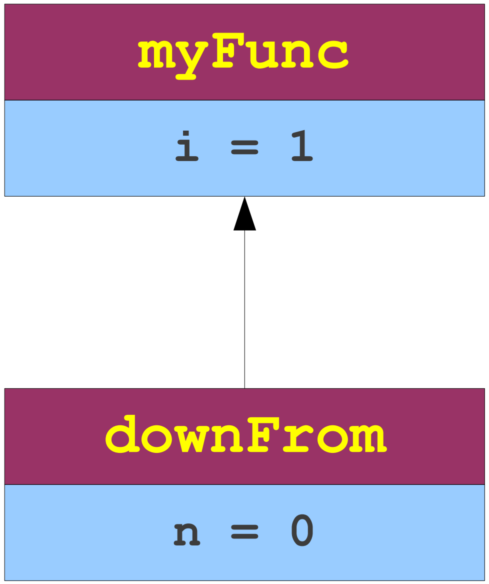
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

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


def myFunc():
    for i in downFrom(3):
        print i
```




```
> 3
  2
  1
```

Coroutines

- A **subroutine** is a function that, when invoked, runs to completion and returns control to the calling function.
  - Master/slave relationship between caller/callee.
- A **coroutine** is a function that, when invoked, does some amount of work, then returns control to the calling function. It can then be resumed later.
  - Peer/peer relationship between caller/callee.
- Subroutines are a special case of coroutines.

Coroutines and the Runtime Stack

- Coroutines often cannot be implemented with purely a runtime stack.
  - What if a function has multiple coroutines running alongside it?
- Few languages support coroutines, though some do (Python, for example).

So What?

- Even a concept as fundamental as “the stack” is actually quite complex.
- When designing a compiler or programming language, you must keep in mind how your language features influence the runtime environment.
- **Always be critical of the languages you use!**