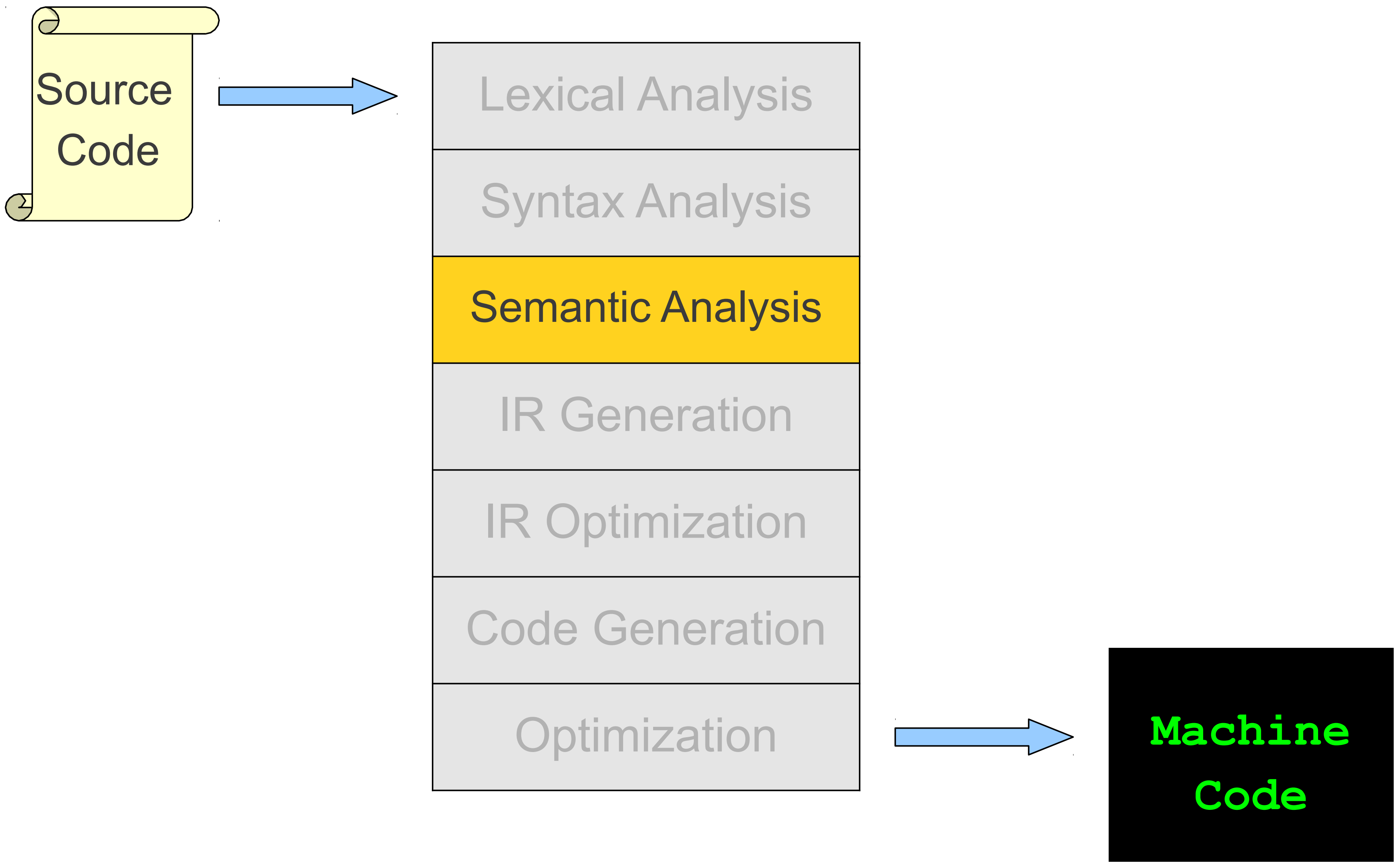
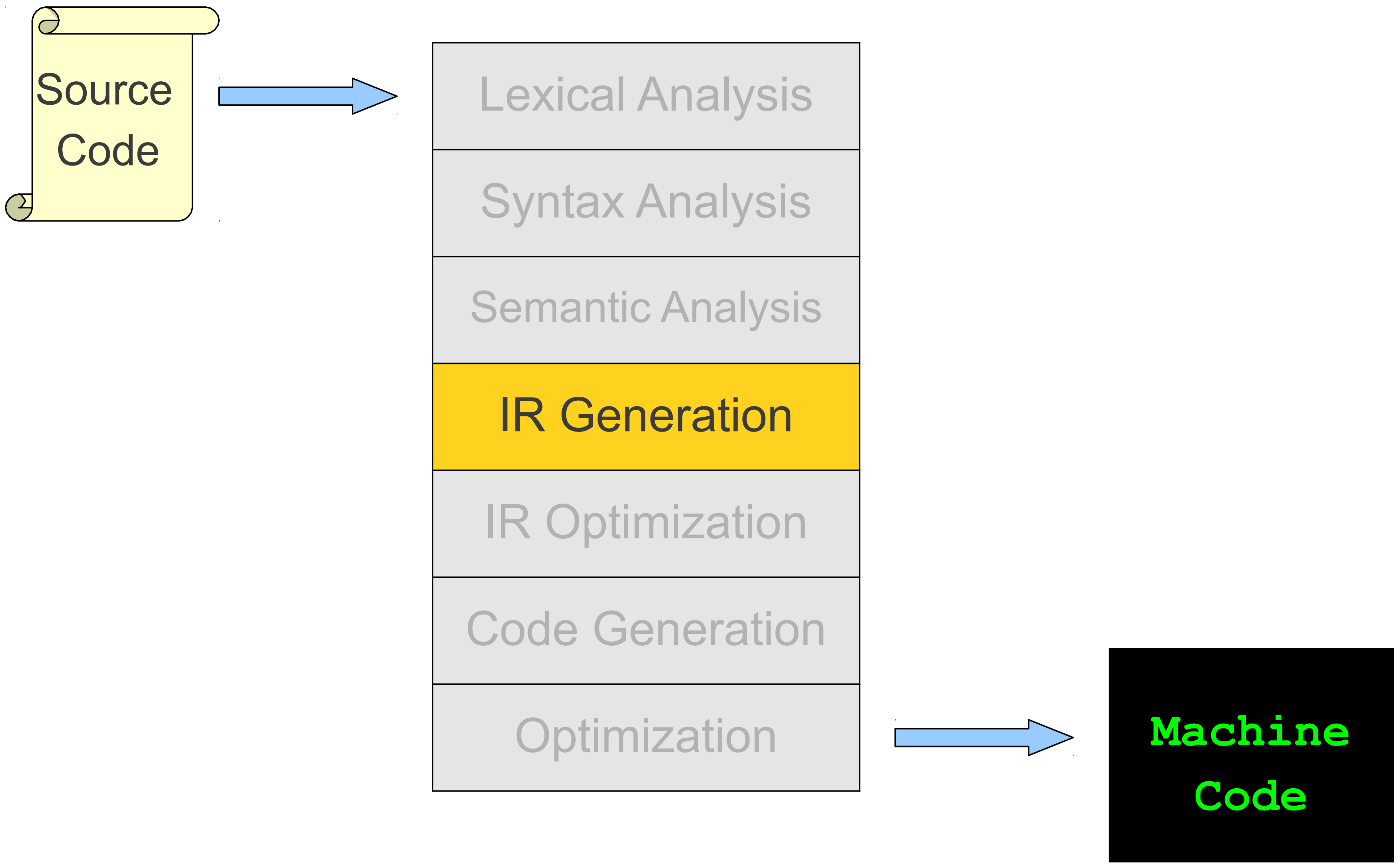
بسم الله الرحمن الرحيم

Semantic Analysis, Runtime Environments

Where We Were



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.

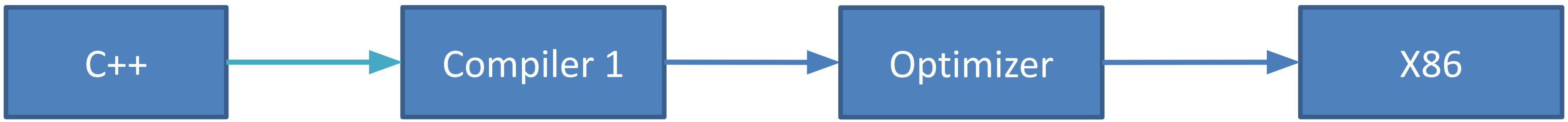
The final phase of the compiler front-end.

Goal: Translate the program into the format expected by the compiler back-end.

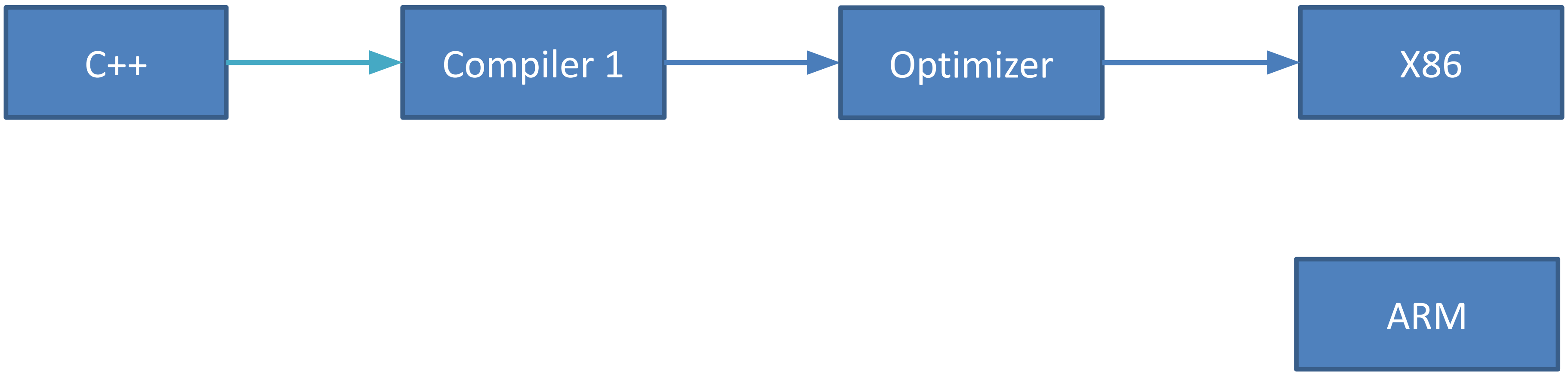
Generated code need not be optimized; that's handled by later passes.

Generated code need not be in assembly; that can also be handled by later passes.

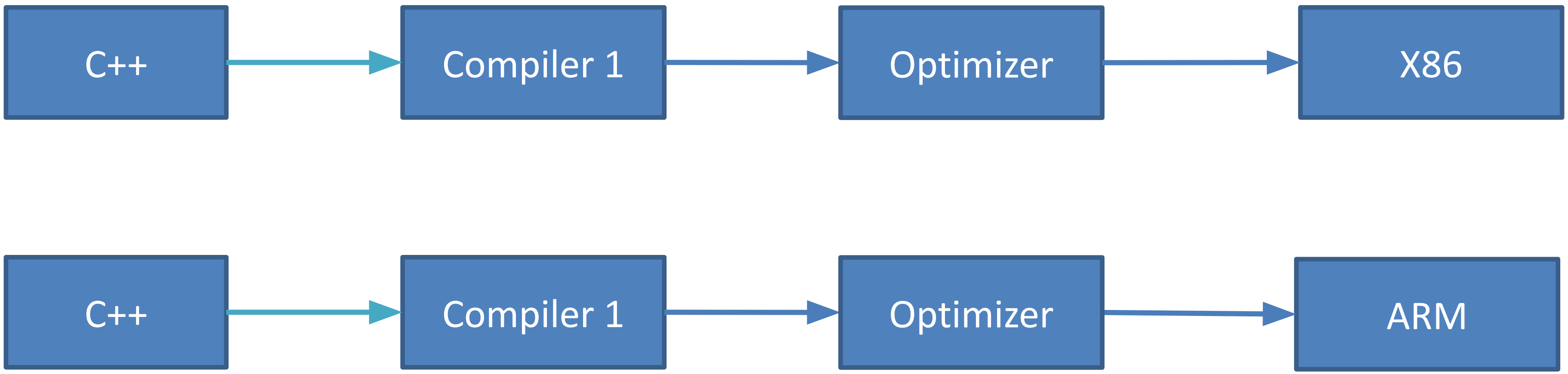
Why Do IR Generation?



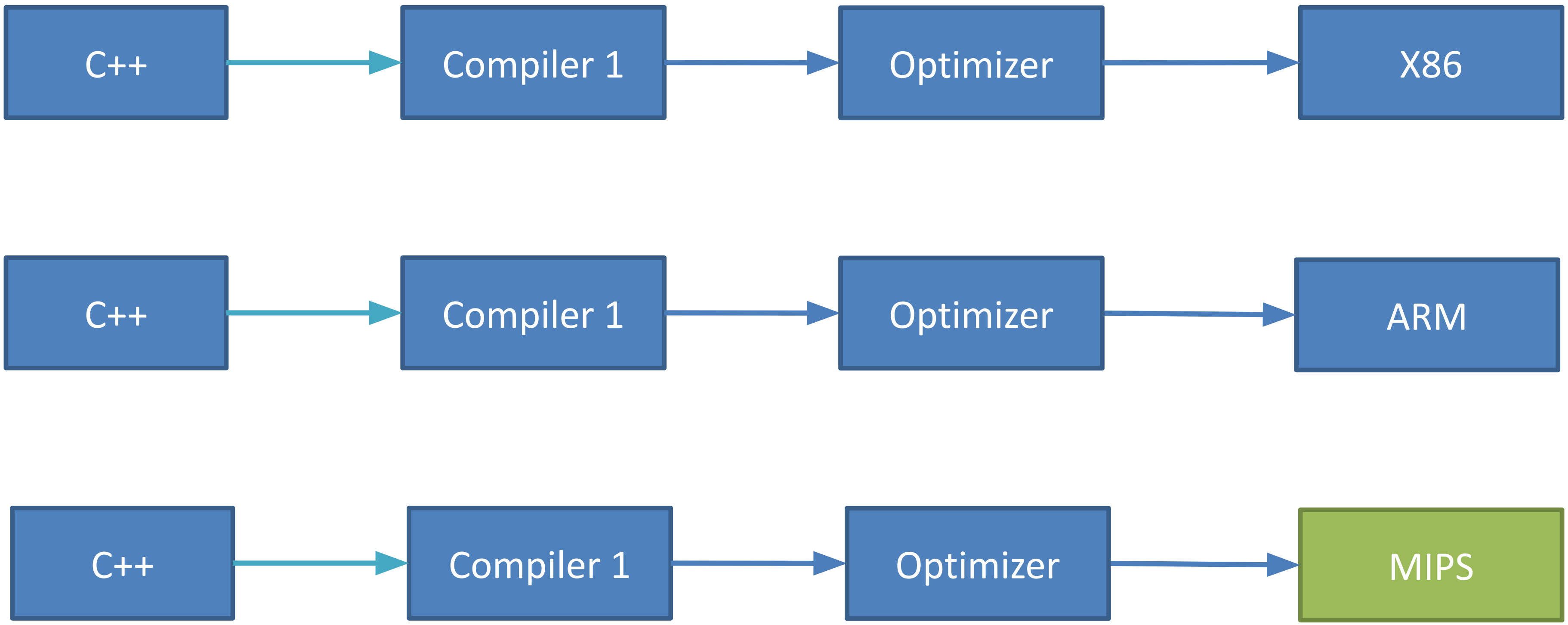
Why Do IR Generation?



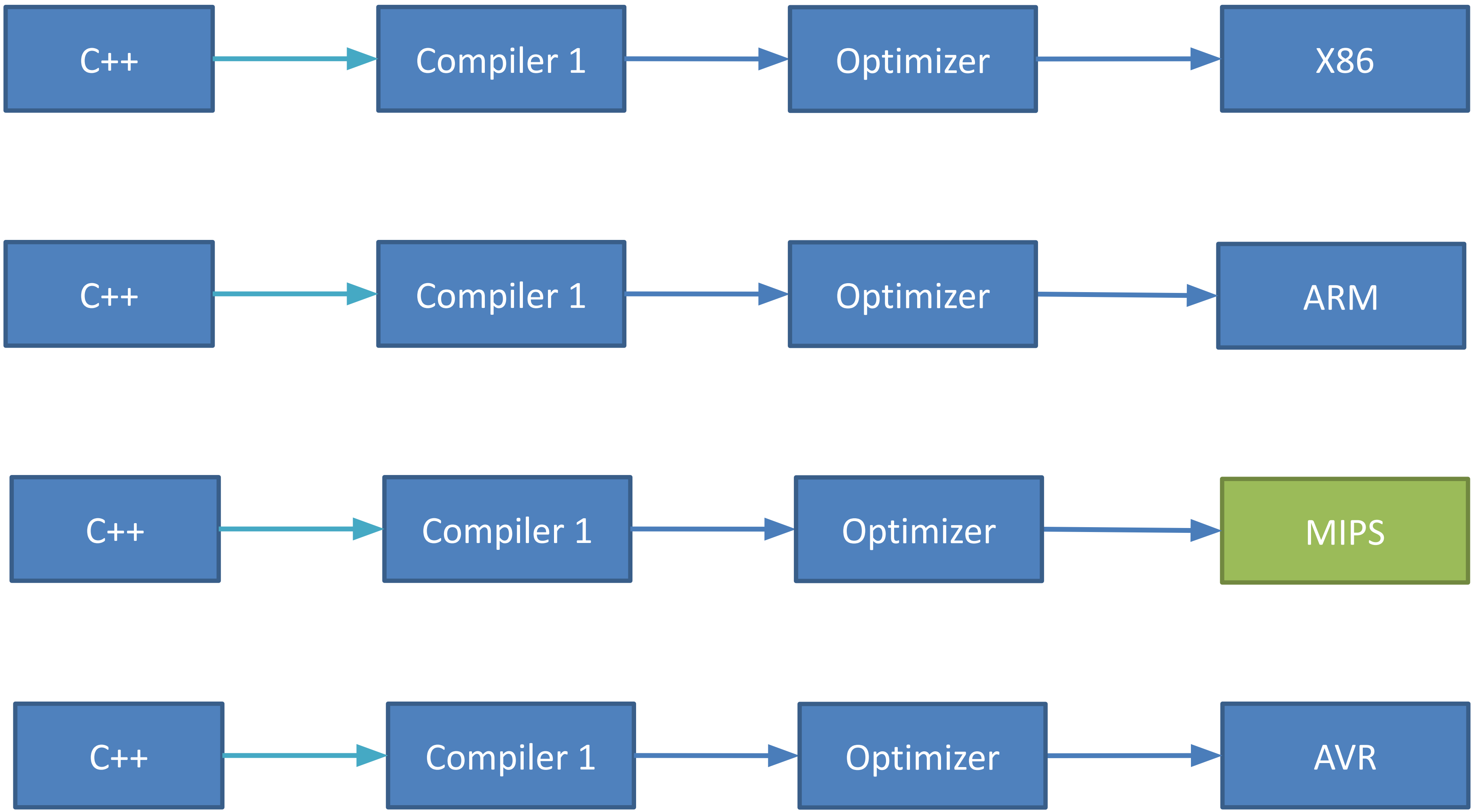
Why Do IR Generation?



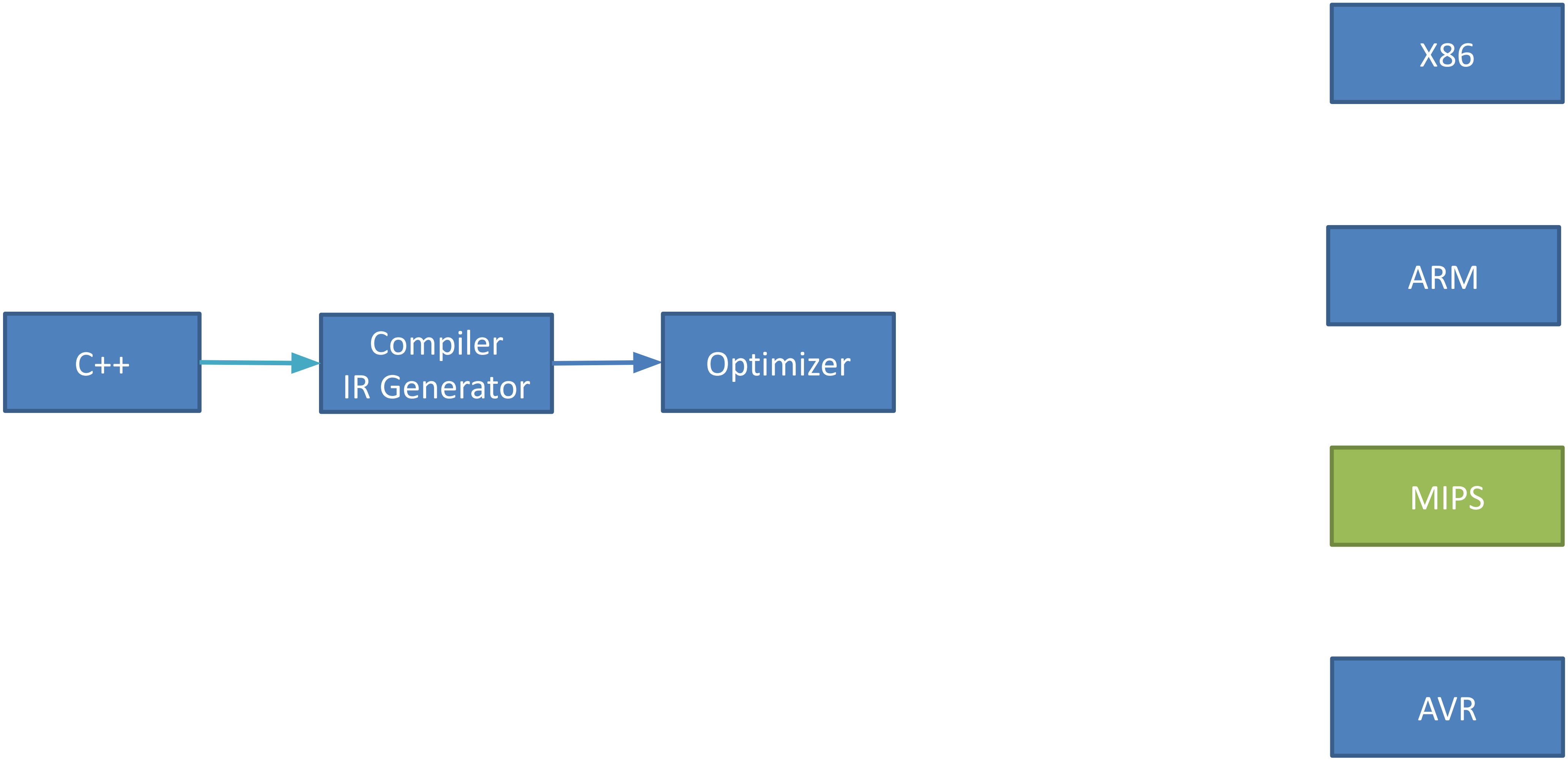
Why Do IR Generation?



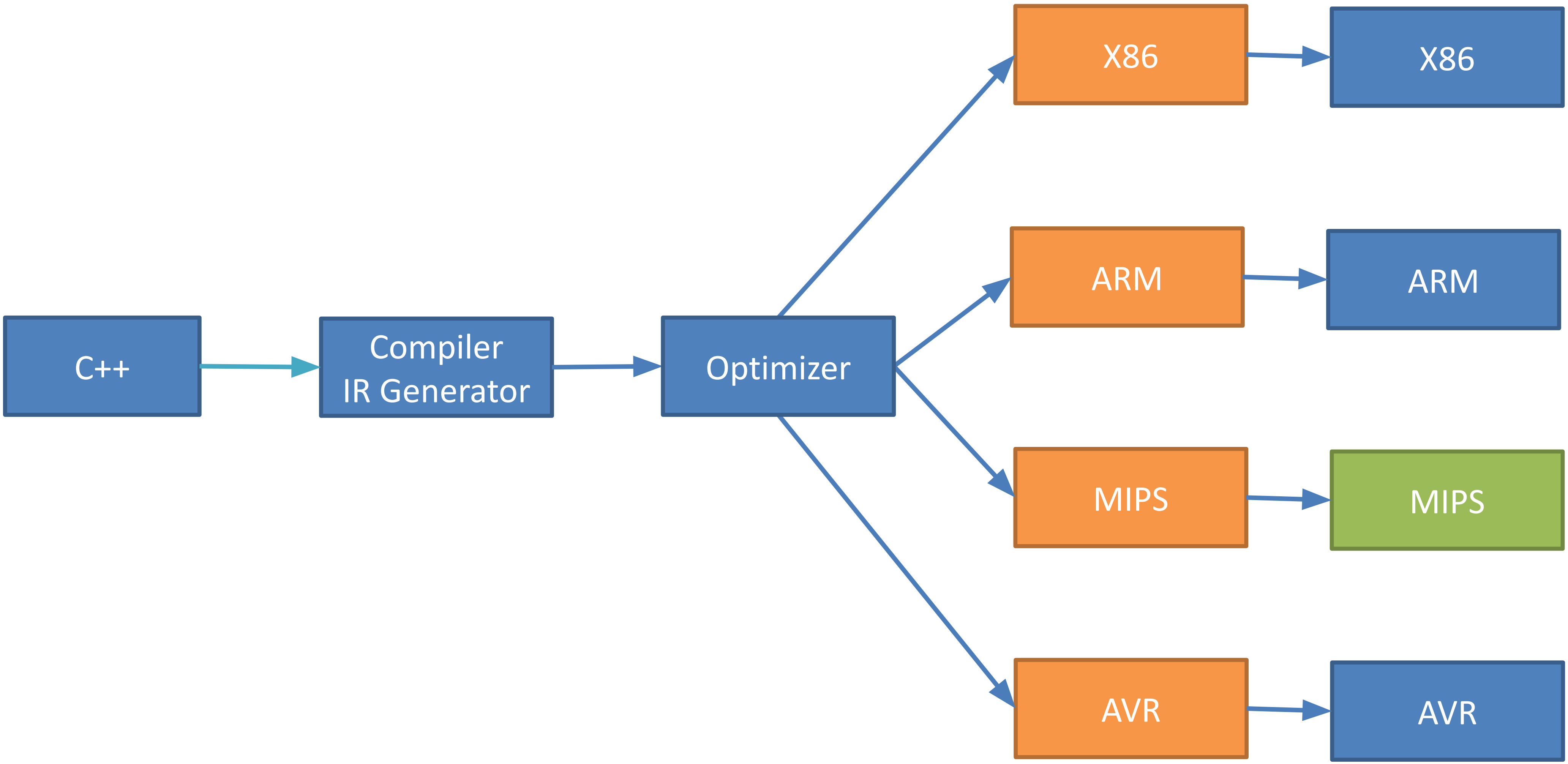
Why Do IR Generation?



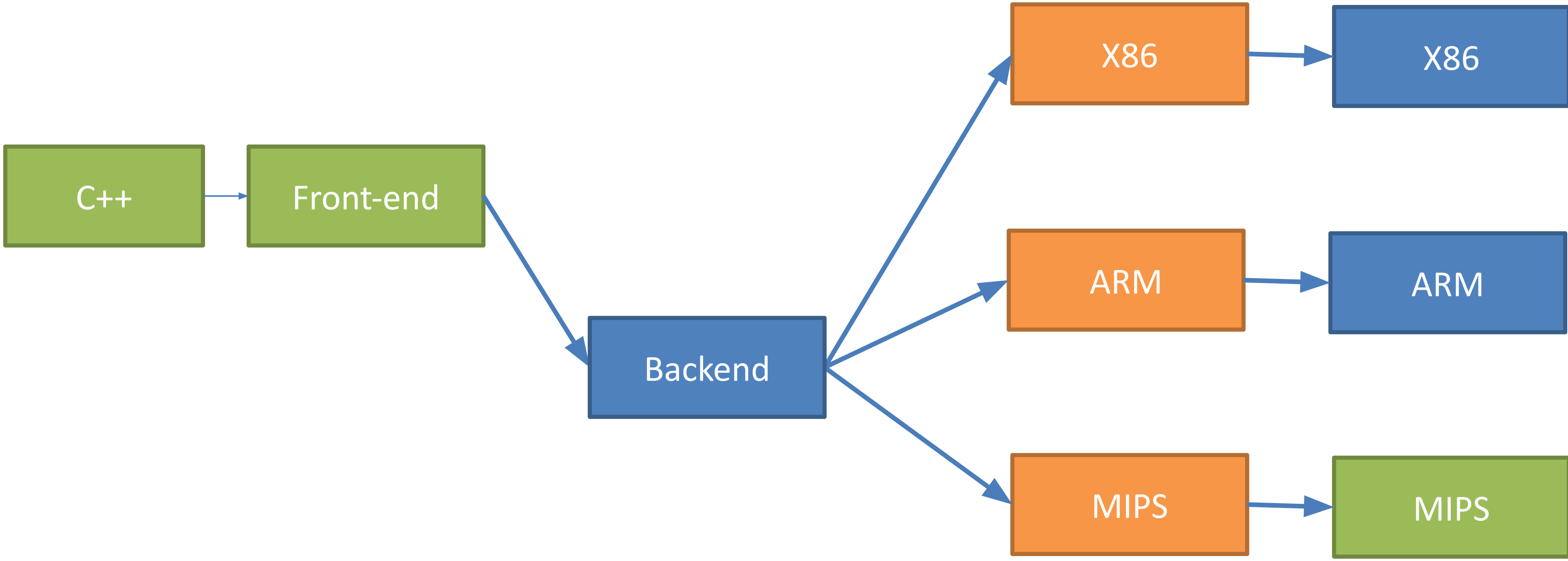
Why Do IR Generation?



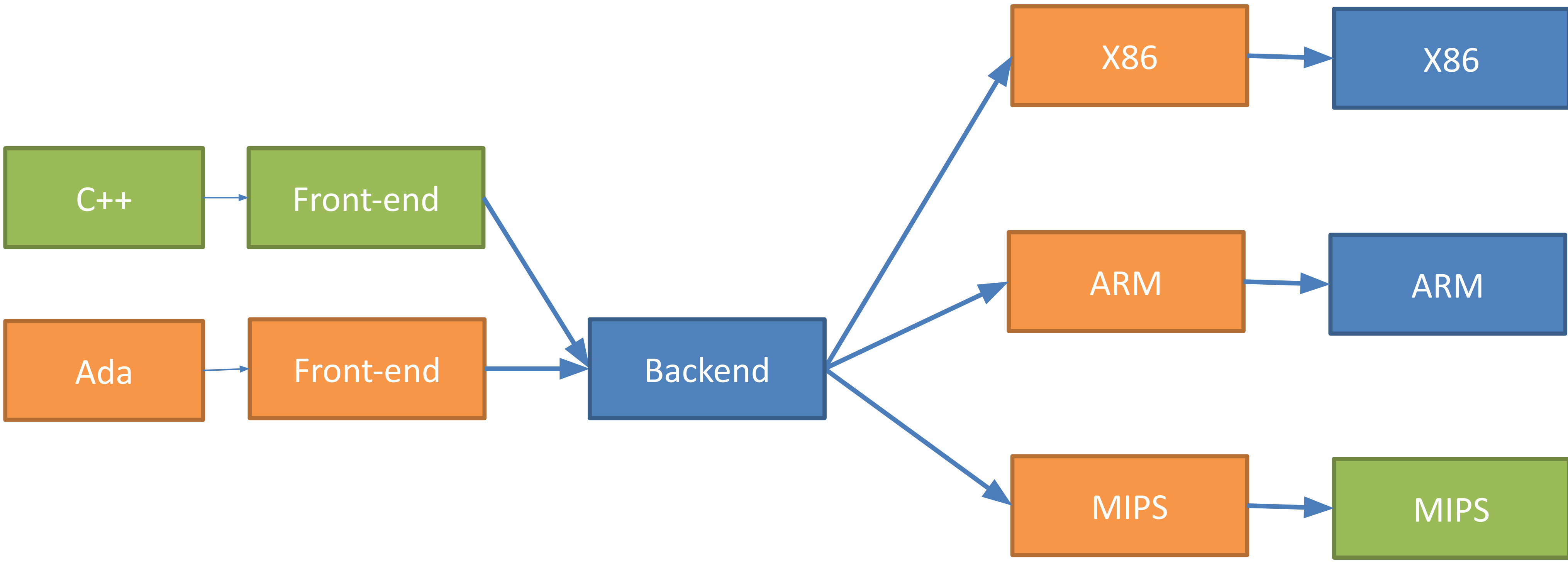
Why Do IR Generation?



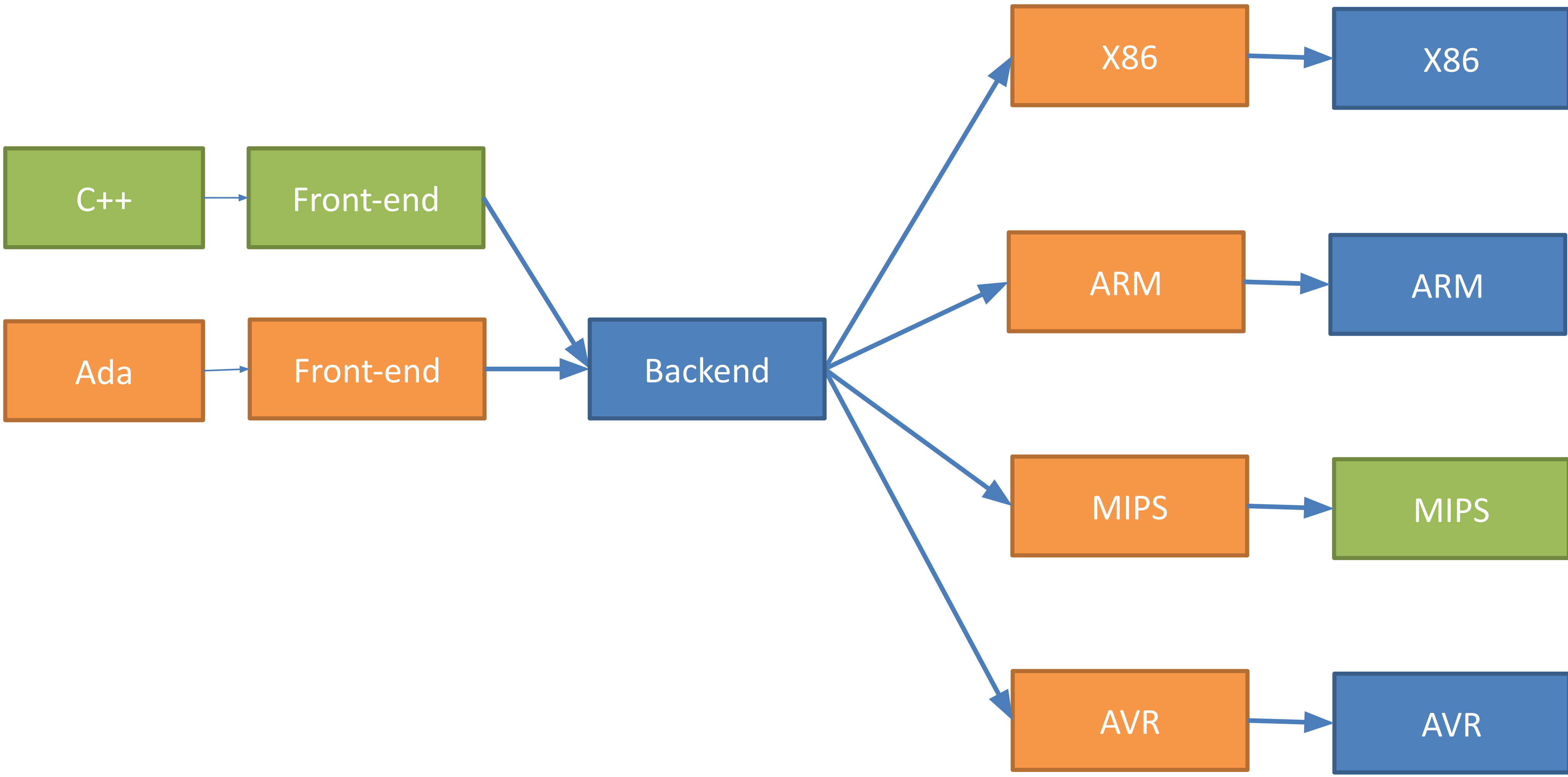
Why Do IR Generation?



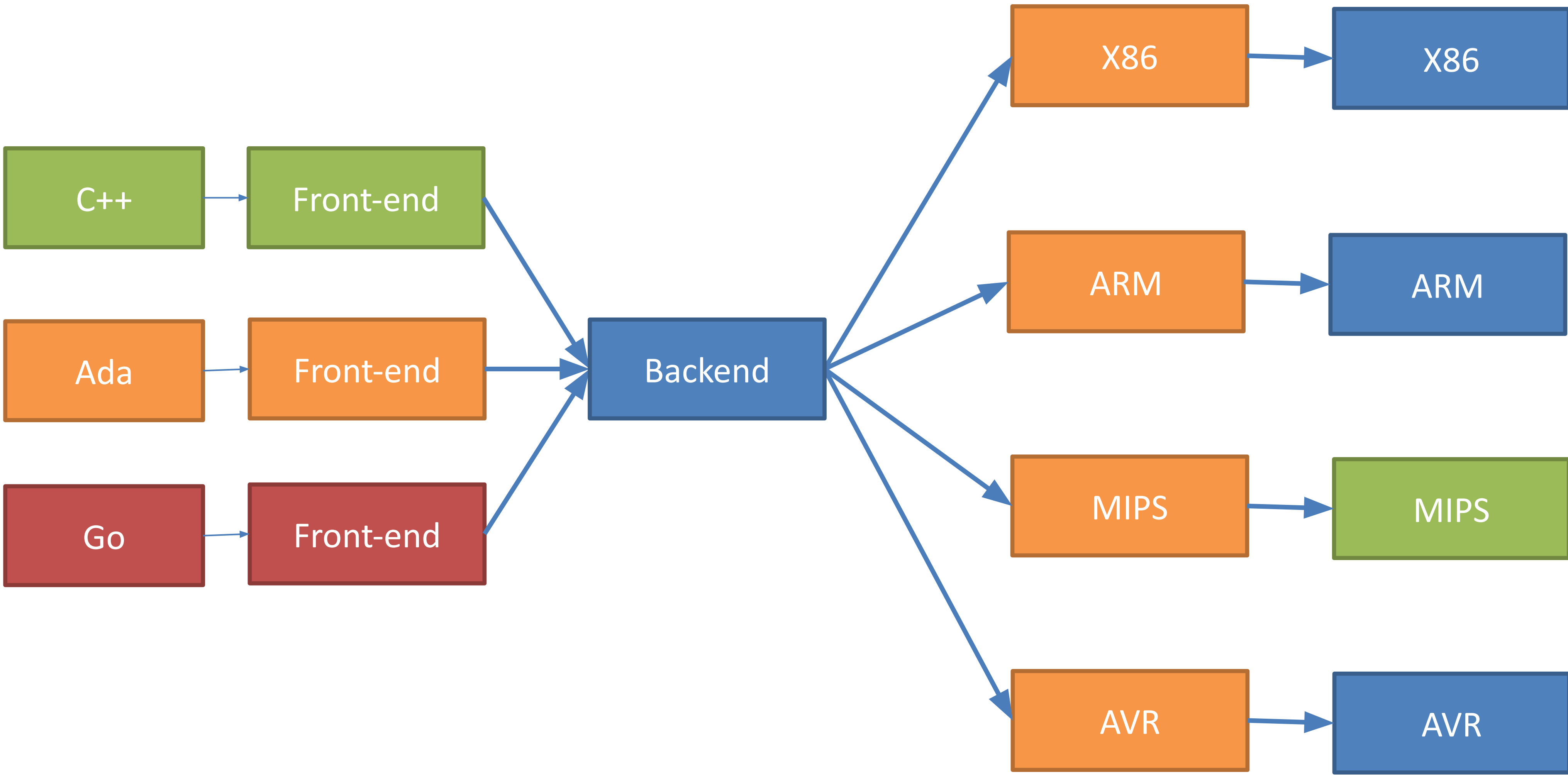
Why Do IR Generation?



Why Do IR Generation?



Why Do IR Generation?



Why Do IR Generation?

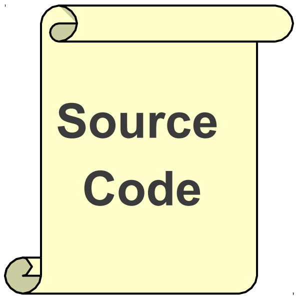
- **Simplify certain optimizations.**
  - Machine code has many constraints that inhibit optimization. (Such as?)
  - Working with an intermediate language makes optimizations easier and clearer.
- **Have many front-ends into a single back-end.**
  - `gcc` can handle C, C++, Java, Fortran, Ada, and many 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

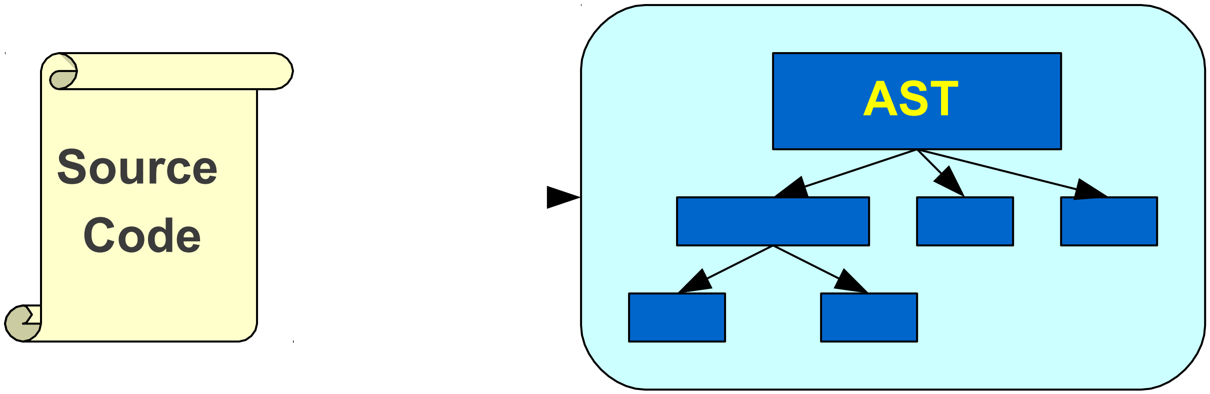
- IRs are like type systems – they're extremely hard to get right.
- Need to balance needs of high-level source language and low-level target language.
- Too high level: can't optimize certain implementation details.
- Too low level: can't use high-level knowledge to perform aggressive optimizations.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

Architecture of `gcc`

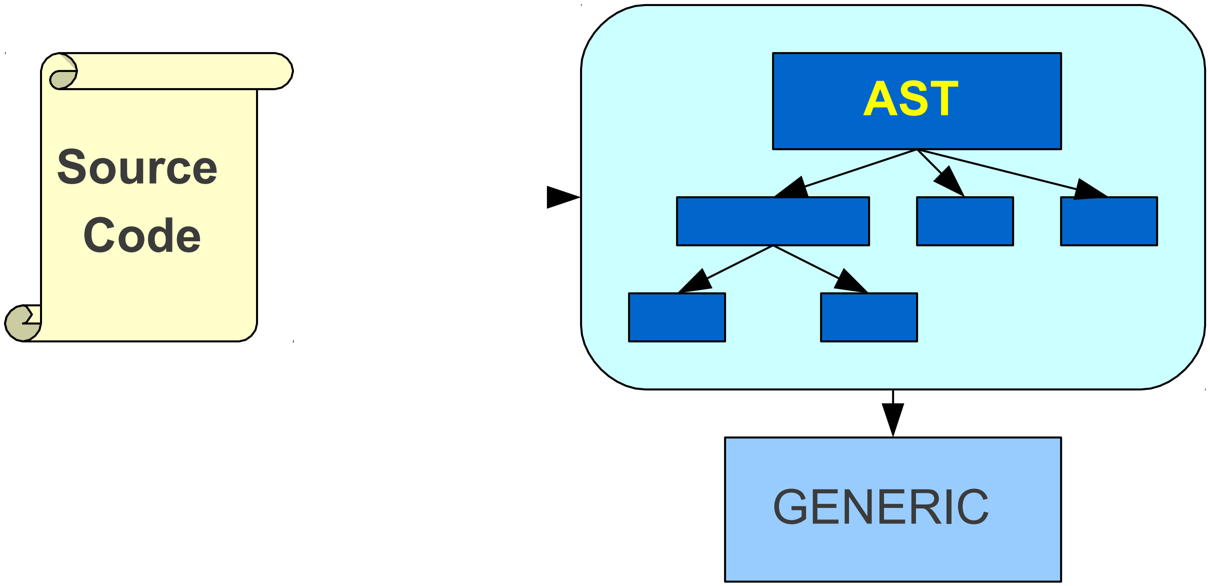
Architecture of `gcc`



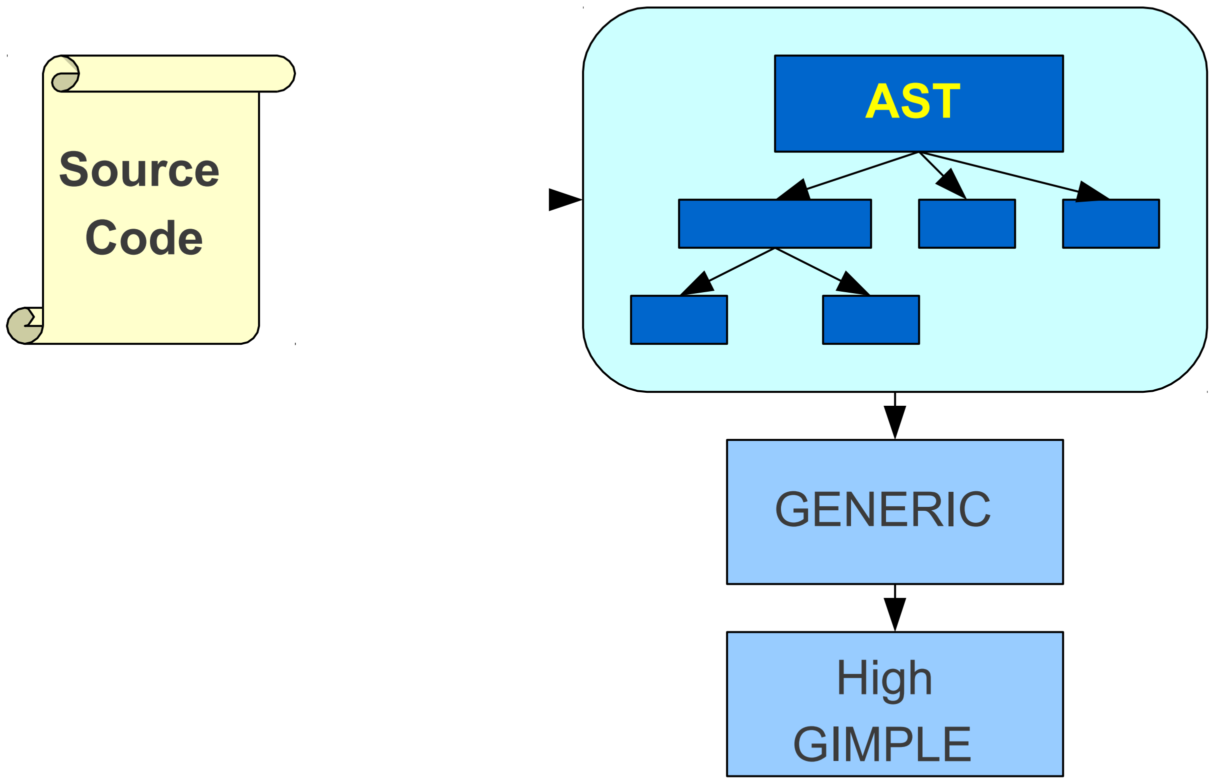
Architecture of gcc



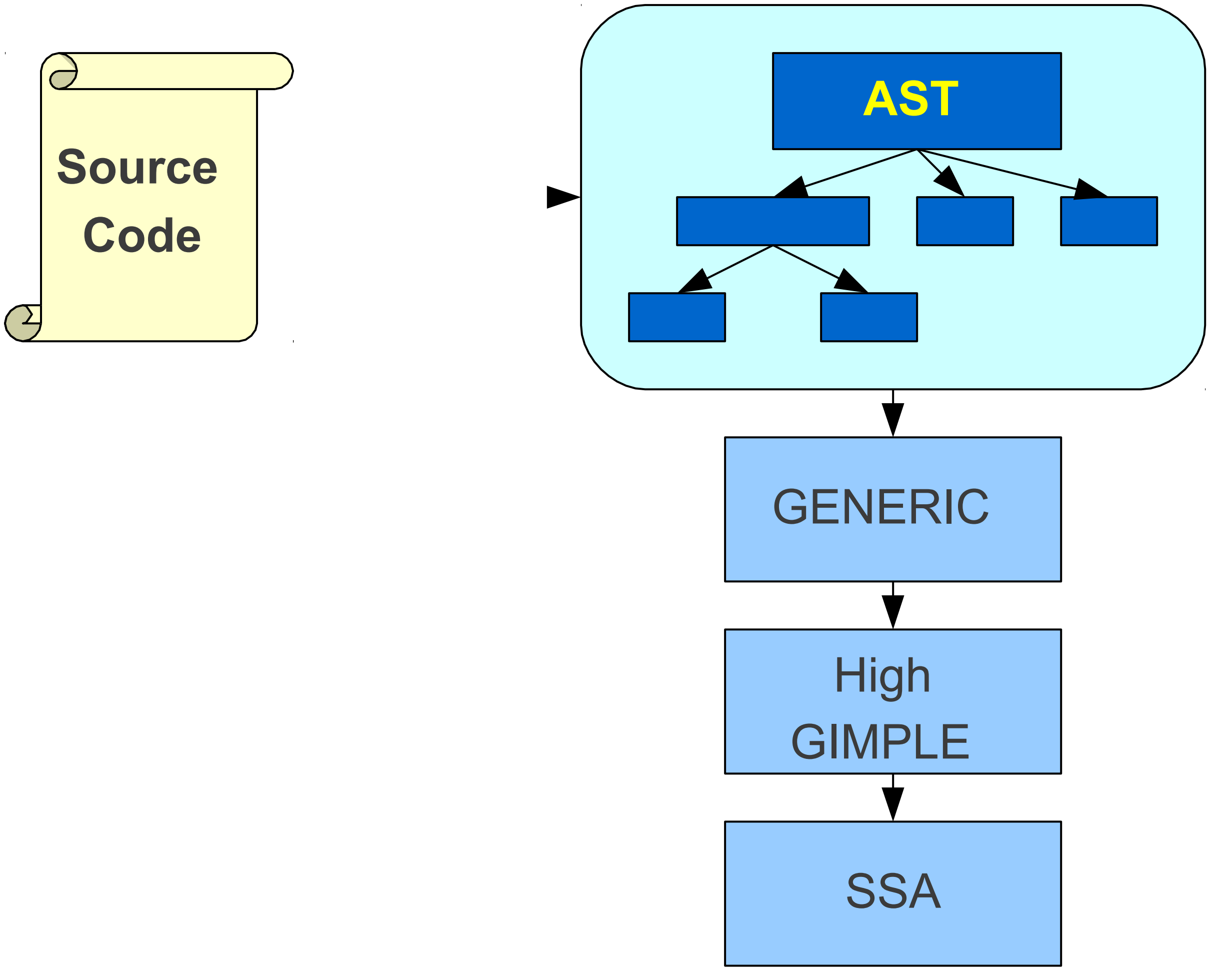
Architecture of `gcc`



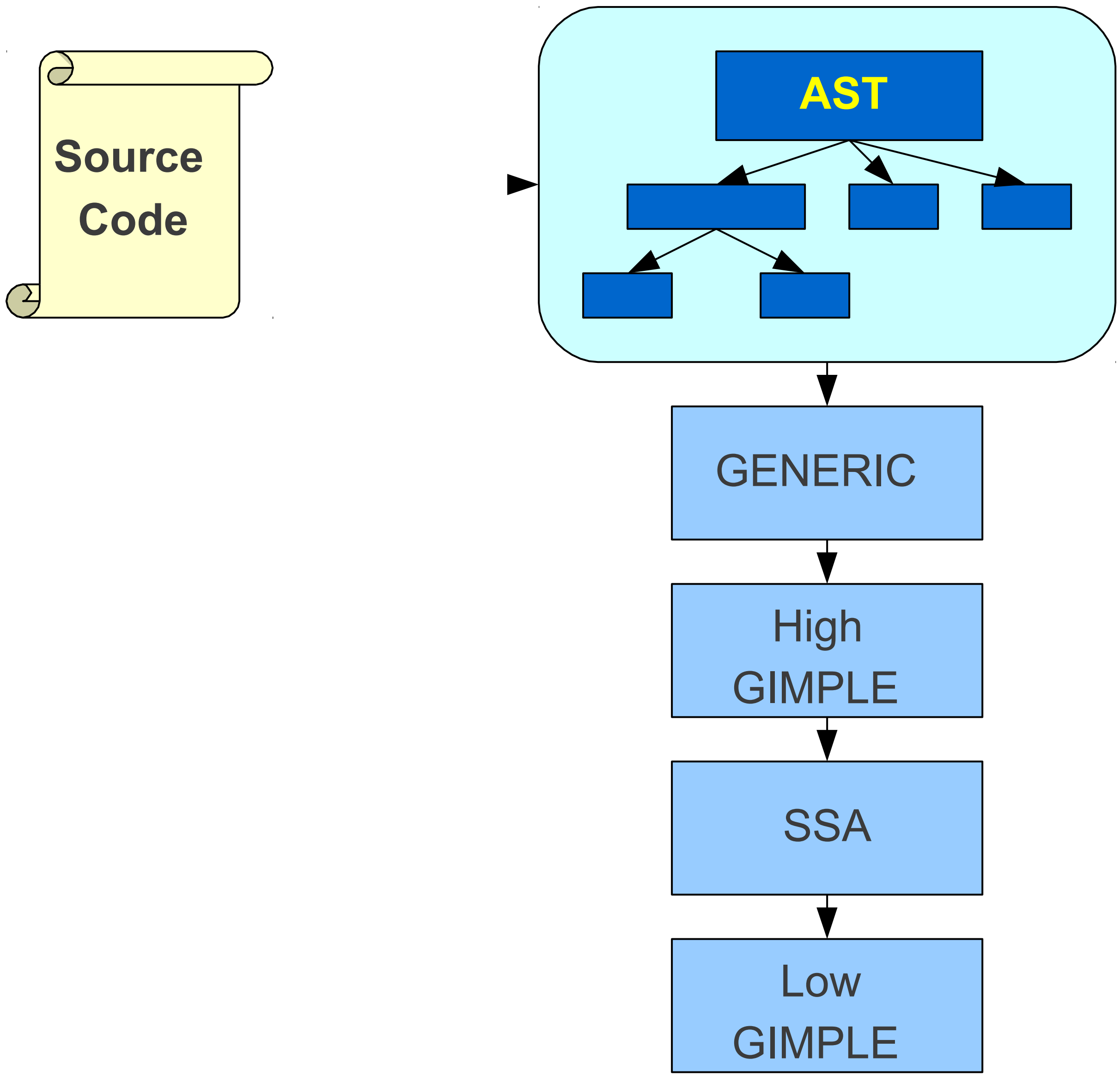
Architecture of gcc



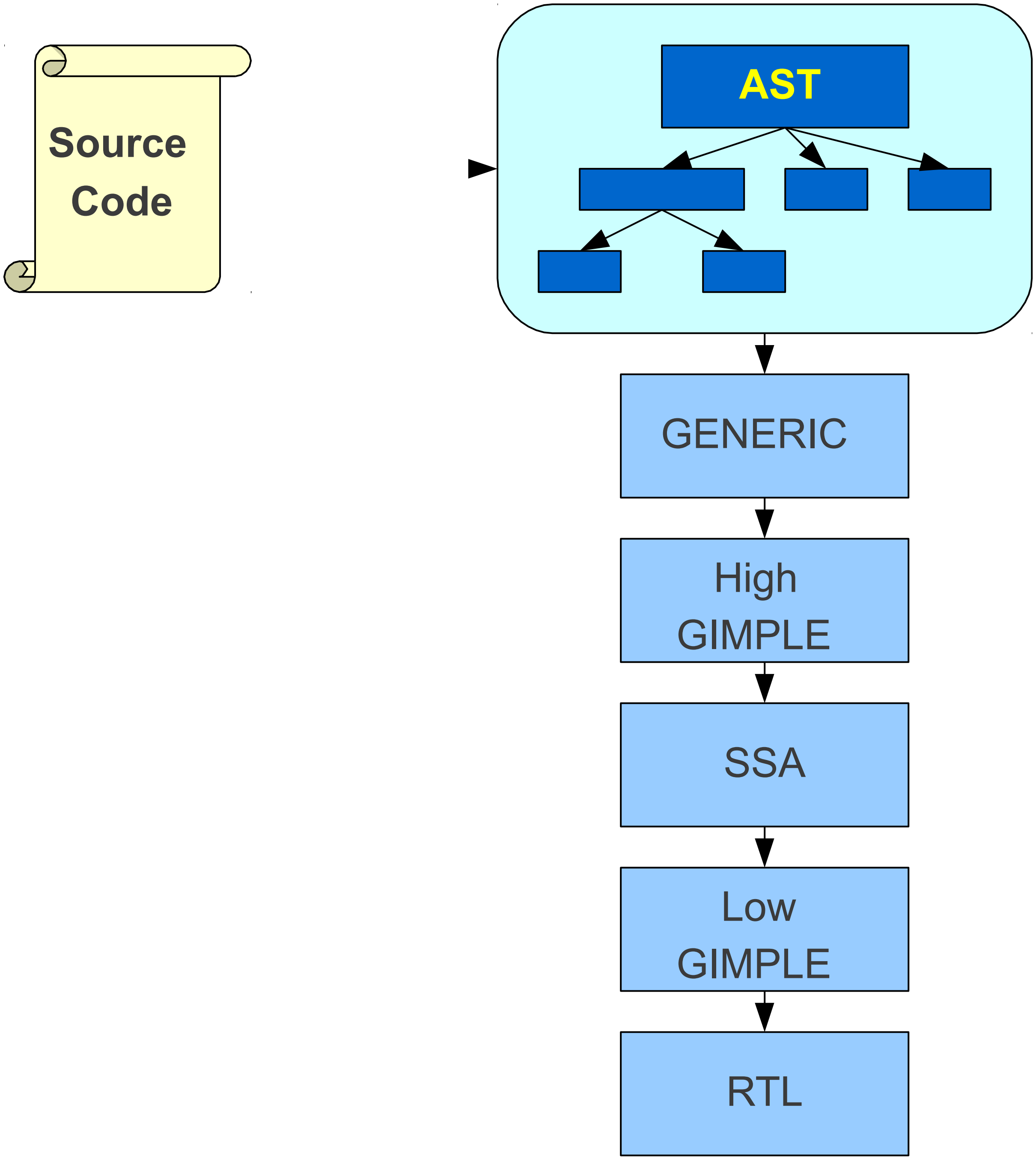
Architecture of gcc



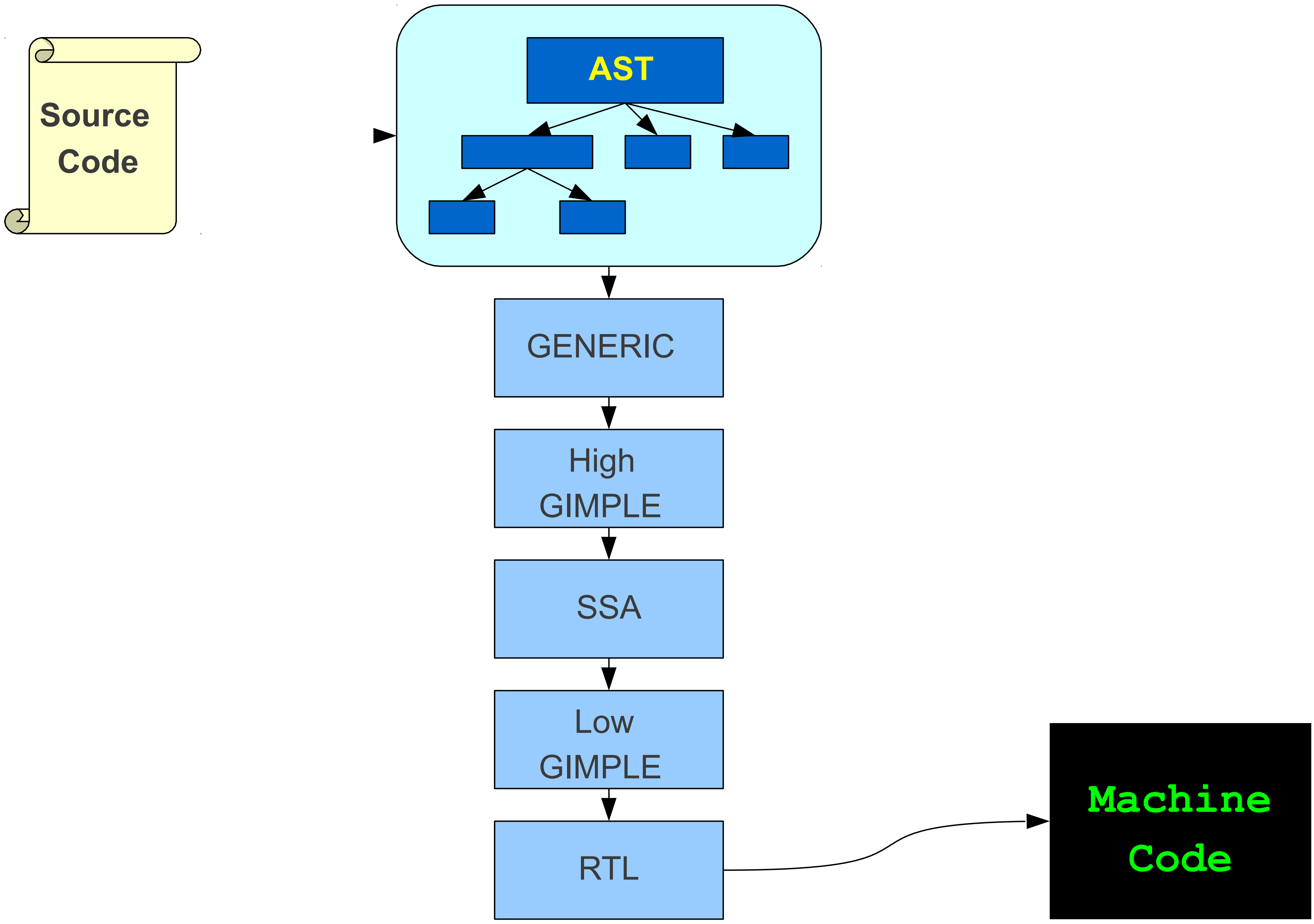
Architecture of gcc



Architecture of gcc



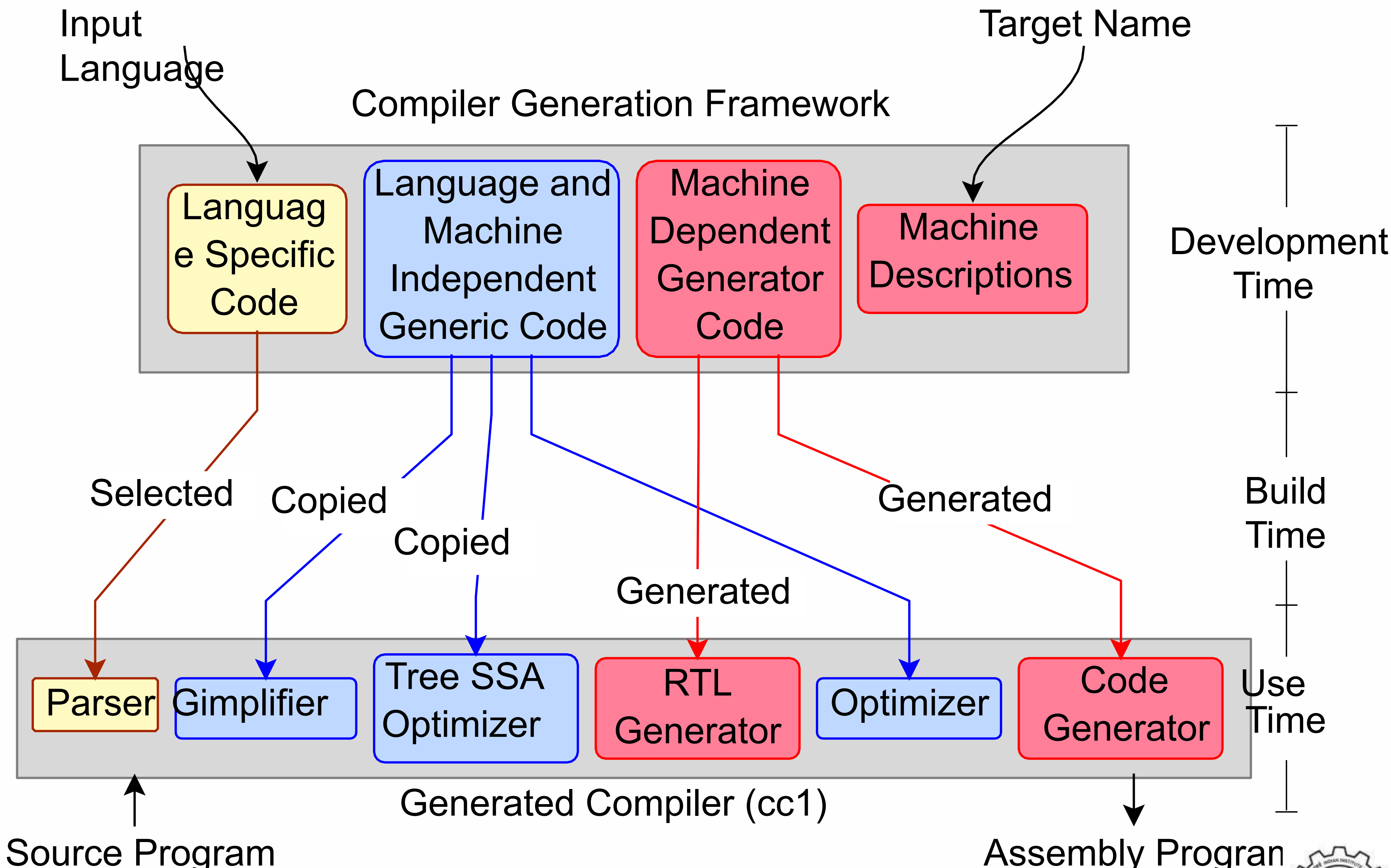
Architecture of gcc



Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200 MN10300, Motorola 88000, NS32K, ROMP, Stormy16 V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant), Z8000, PIC24/dsPIC, NEC SX architecture

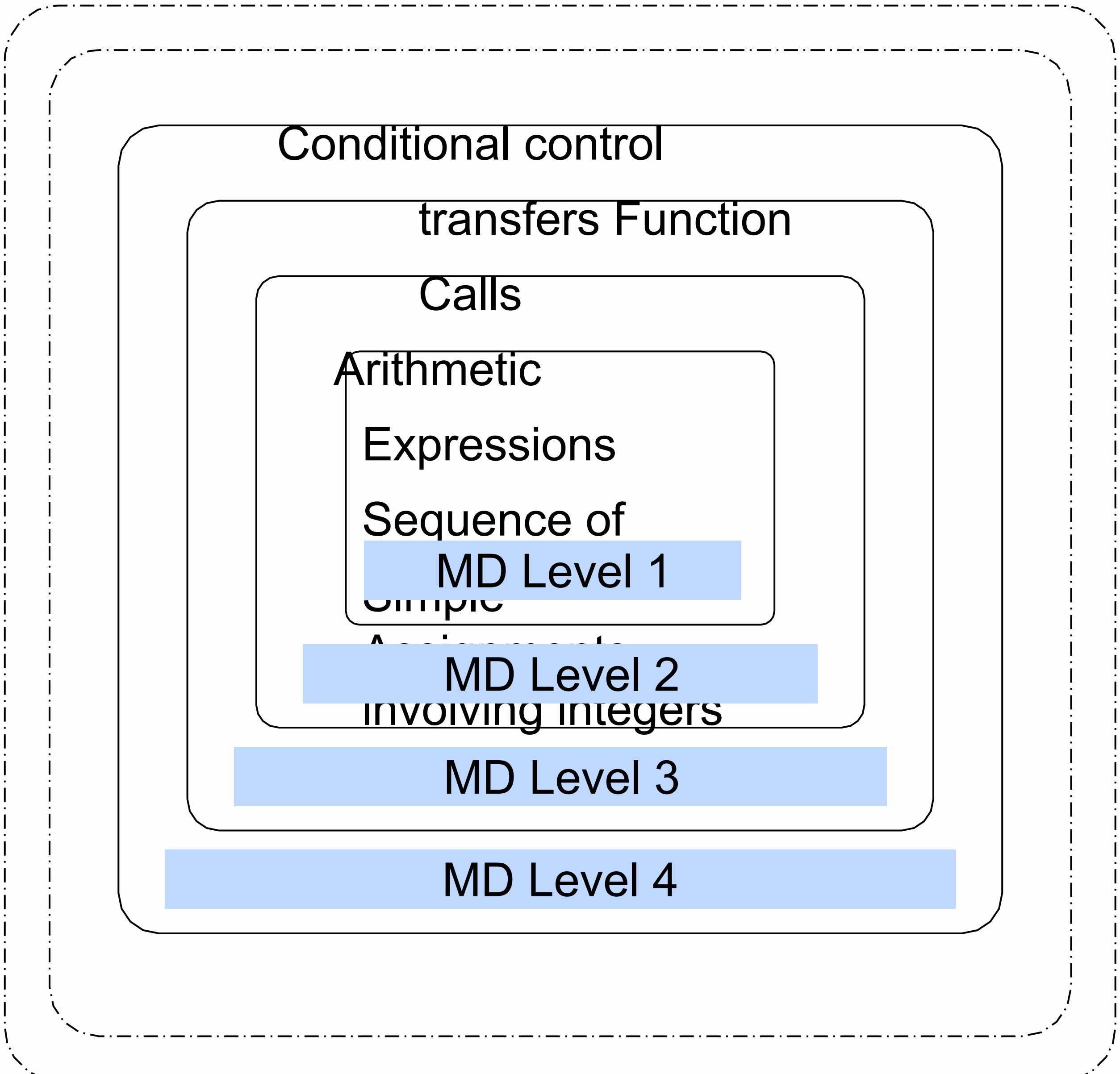
The Architecture of GCC



Incremental Construction of Machine Descriptions

- Define different levels of source language
- Identify the minimal set of features in the target required to support each level
- Identify the minimal information required in the machine description to support each level
  - Successful compilation of any program, and
  - correct execution of the generated assembly program
- Interesting observations
  - It is the increment in the source language which results in understandable increments in machine descriptions rather than the increment in the target architecture
  - If the levels are identified properly, the increments in machine descriptions are monotonic

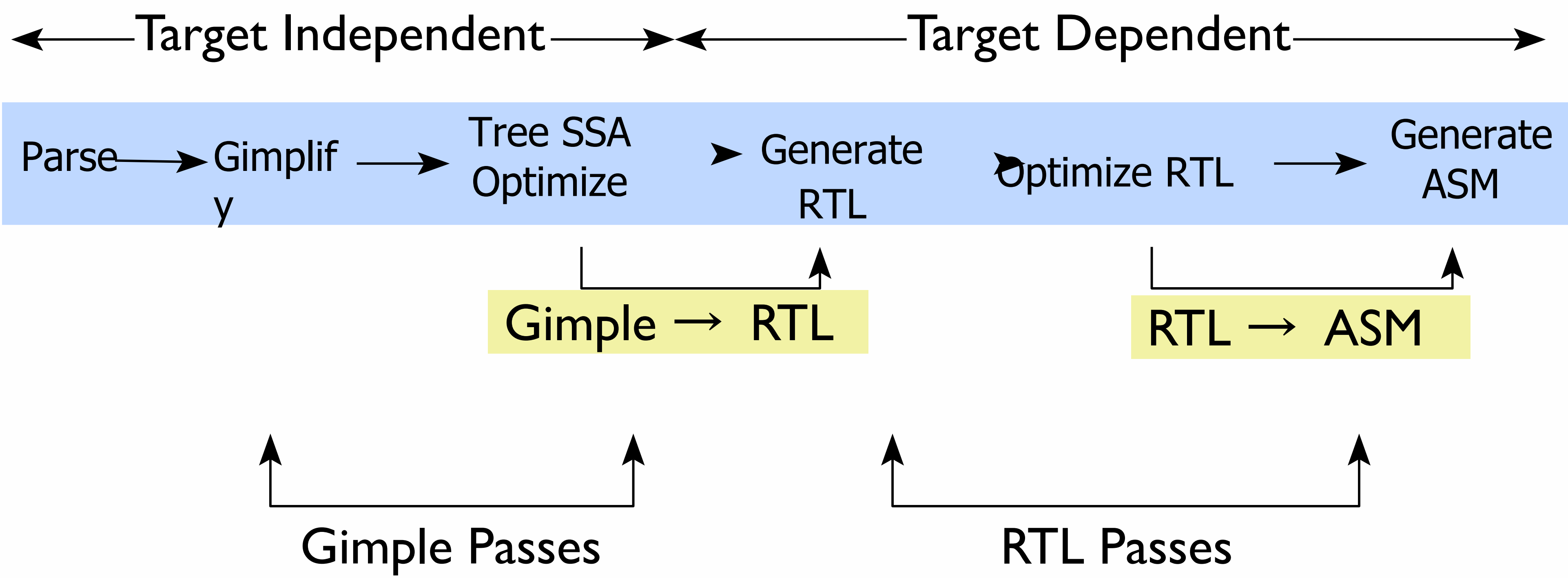
Incremental Construction of Machine Descriptions



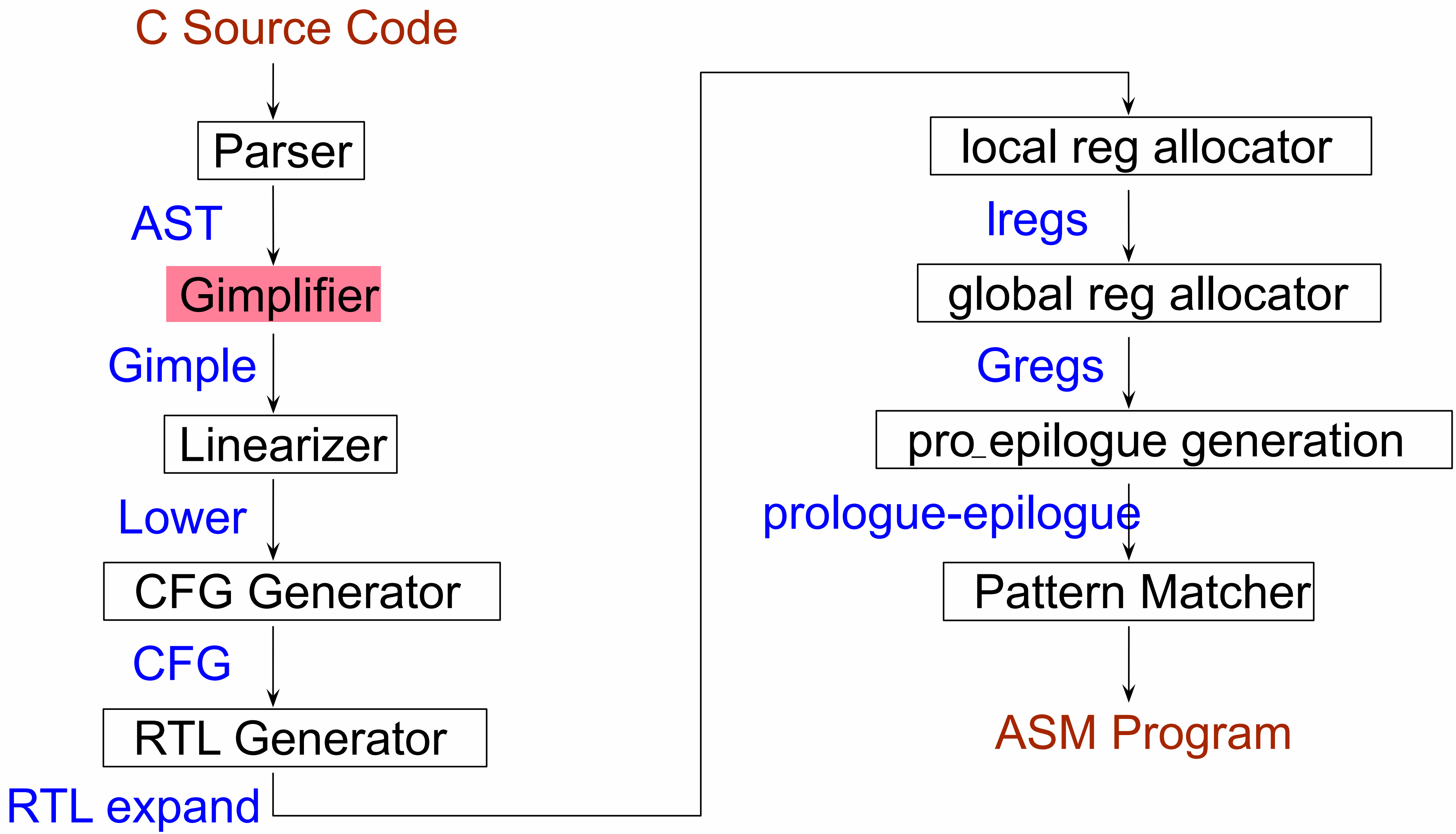
Transformation Passes in GCC

- A total of 196 unique pass names initialized in ${SOURCE}/gcc/passes.c
  - Some passes are called multiple times in different contexts Conditional constant propagation and dead code elimination are called thrice
  - Some passes are only demo passes (eg. data dependence analysis)
  - Some passes have many variations (eg. special cases for loops) Common subexpression elimination, dead code elimination
- The pass sequence can be divided broadly in two parts
  - Passes on Gimple
  - Passes on RTL
- Some passes are organizational passes to group related passes

Basic Transformations in GCC



Important Phases of GCC



What is Generic ?

What?

- Language independent IR for a complete function in the form of trees
- Obtained by removing language specific constructs from ASTs
- All tree codes defined in $(SOURCE)/gcc/tree.def

Why?

- Each language frontend can have its own AST
- Once parsing is complete they must emit Generic

Gimple: Translation of Composite Expressions

Dump file: test.c.004t.gimple

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
     if (a<=12)
          a =a+b+c;

}
```

```
if (a <= 12)
{
      D.1199 = a + b;
      a = D.1199 + c;
}
else
{
}
```

Gimple: Translation of Higher Level Control Constructs

Dump file: test.c.004t.gimple

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
        a = a+b+c;
}
```

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
{
        goto <D.1196>;
}
else
{
        goto <D.1198>;
}
<D.1198>:;
```

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.

Outline

- **Runtime Environments**
  - How do we implement language features in machine code?
  - What data structures do we need?
- **Three-Address Code IR**
  - What IR are we using in this course?
  - What features does it have?

Runtime Environments

Outline

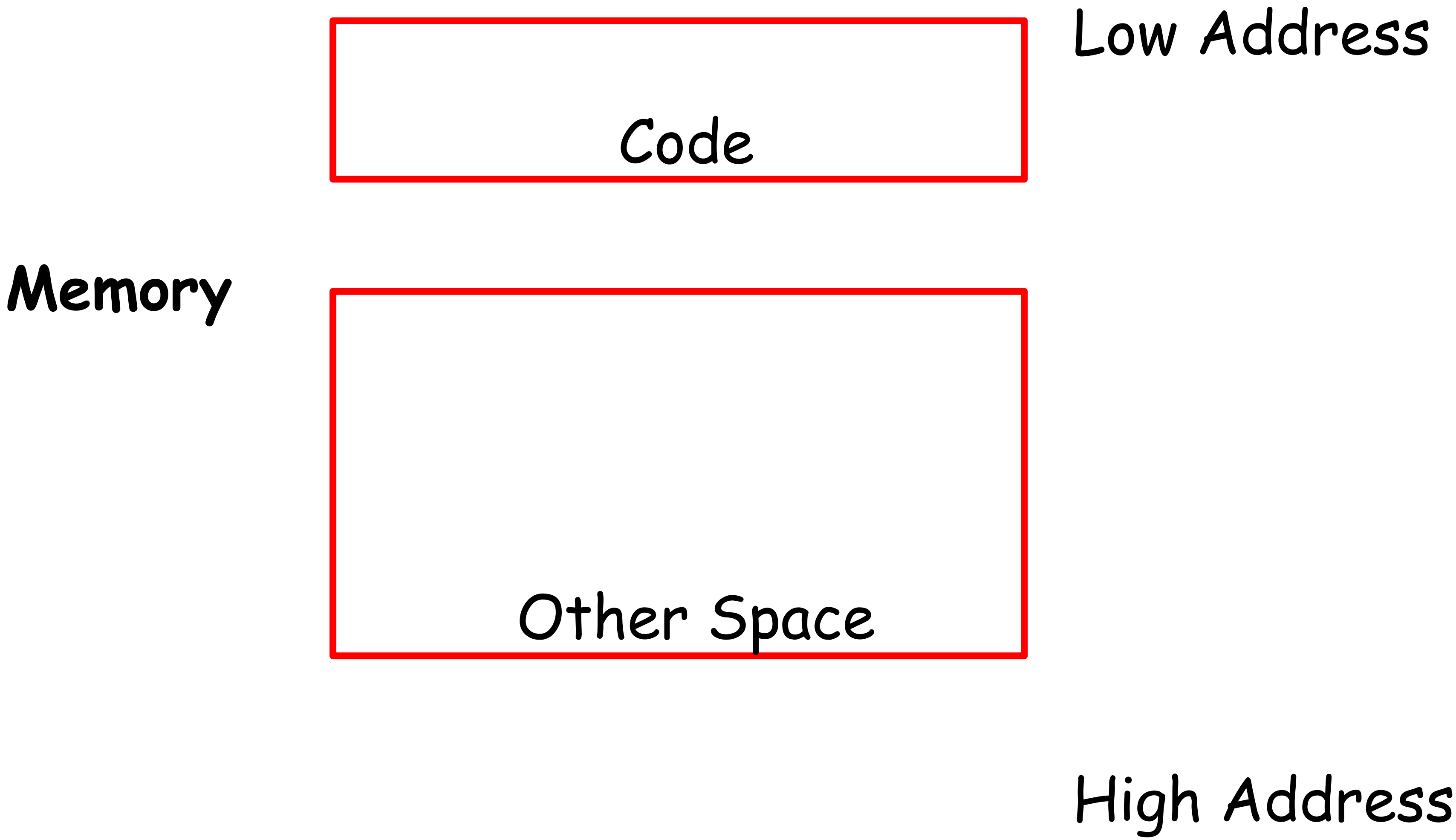
- Management of run-time resources
- Correspondence between
  - static (compile-time) and
  - dynamic (run-time) structures
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system

- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



Notes

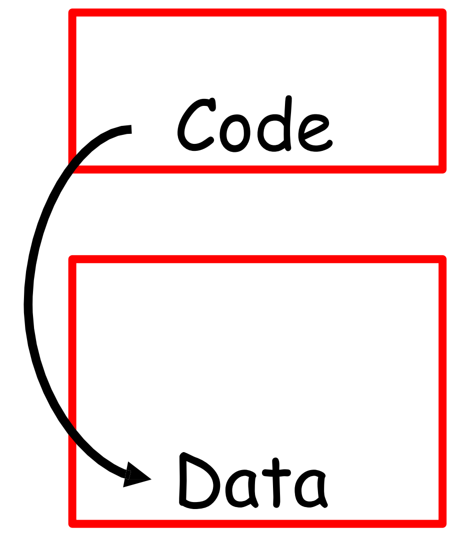
- By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

- These pictures are simplifications
  - E.g., not all memory need be contiguous

What is Other Space?

- Holds all data for the program
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area



Code Generation Goals

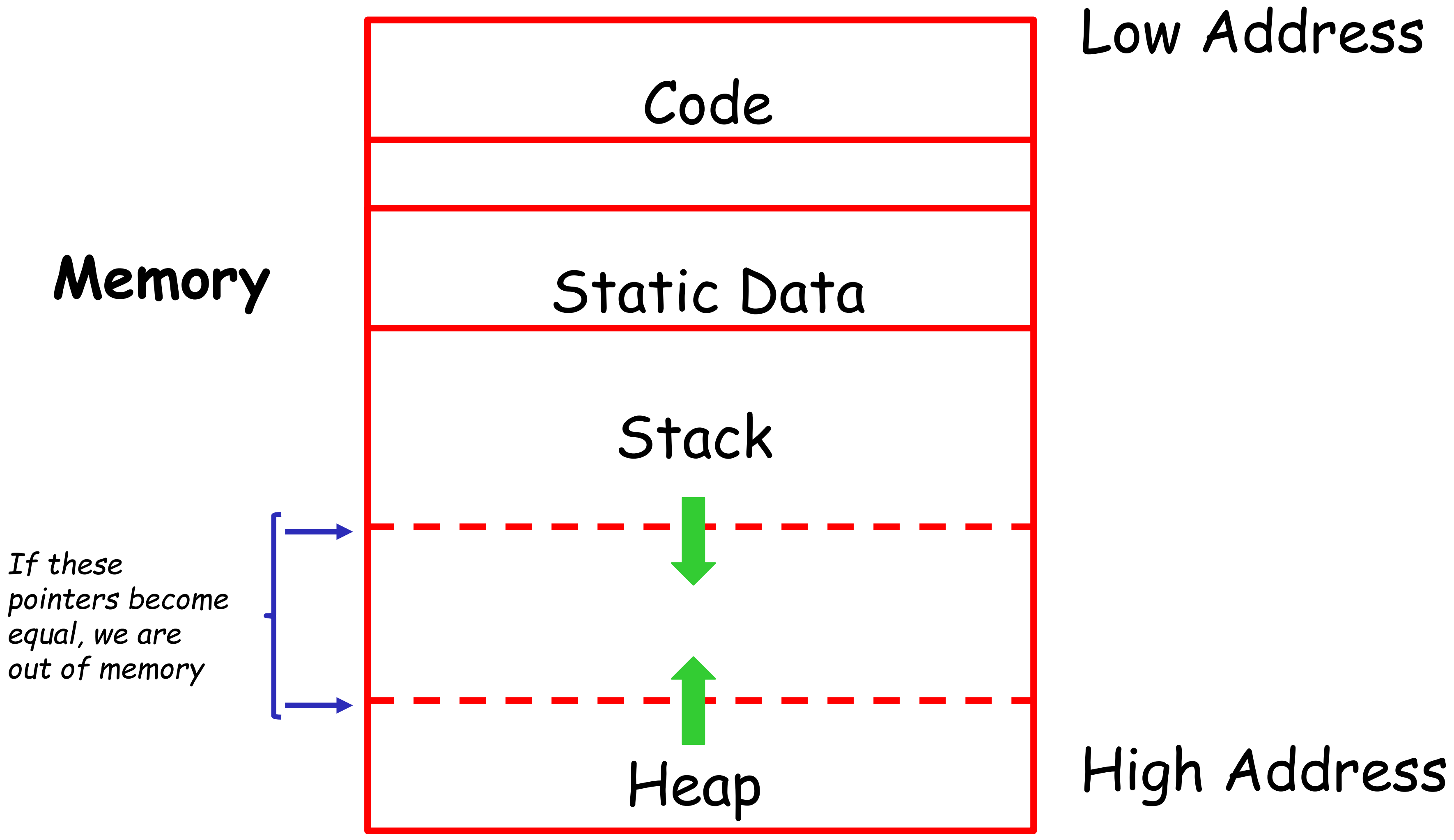
- Two goals:
  - Correctness
  - Speed

- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

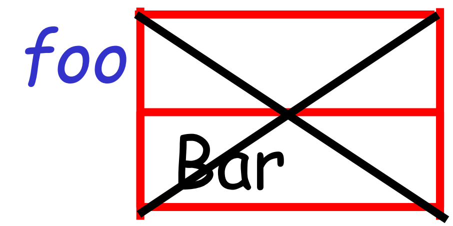
1. Execution is sequential; control moves from one point in a program to another in a well-defined order
   - No concurrency
2. When a procedure is called, control eventually returns to the point immediately after the call
   - No exceptions

Memory Layout with Heap



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR



method foo() { new Bar }

The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use a *heap* to store dynamic data

Notes

- The code area contains object code
  - For many languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

Notes (Cont.)

- Both the heap and the stack grow

- Must take care that they don't grow into each other

- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

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.

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.

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.

The Decaf Runtime Environment

- Need to consider
  - What do objects look like in memory?
  - What do functions look like in memory?
  - Where in memory should they be placed?
- **There are no right answers to these questions**.
  - Many different options and tradeoffs.
  - We will see several approaches.

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

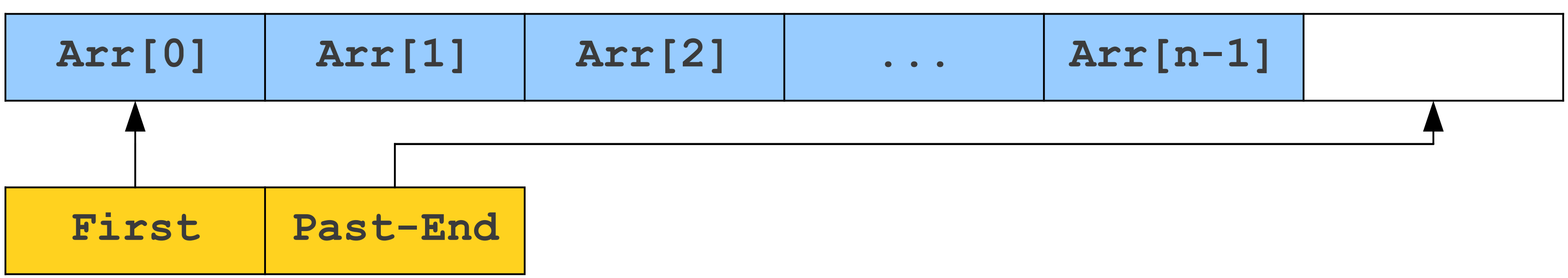
- C-style arrays: Elements laid out consecutively in memory.

`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]`

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



Encoding Arrays

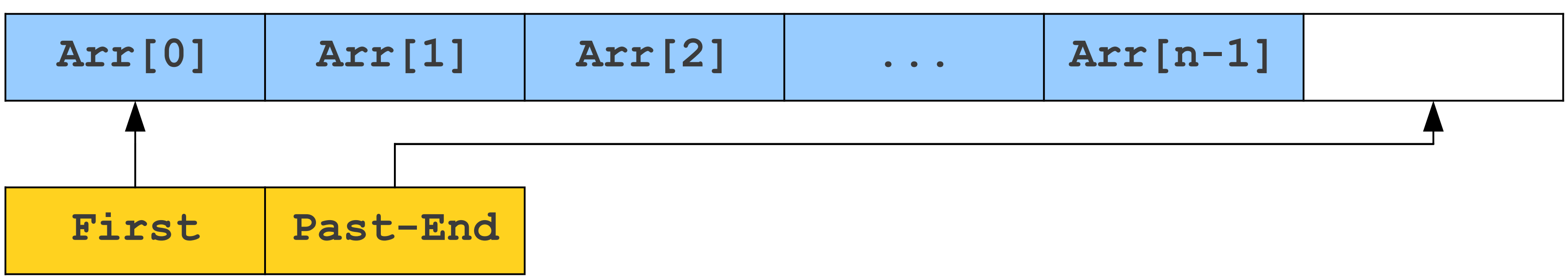
- C-style arrays: Elements laid out consecutively in memory.

`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]`

- 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];
```

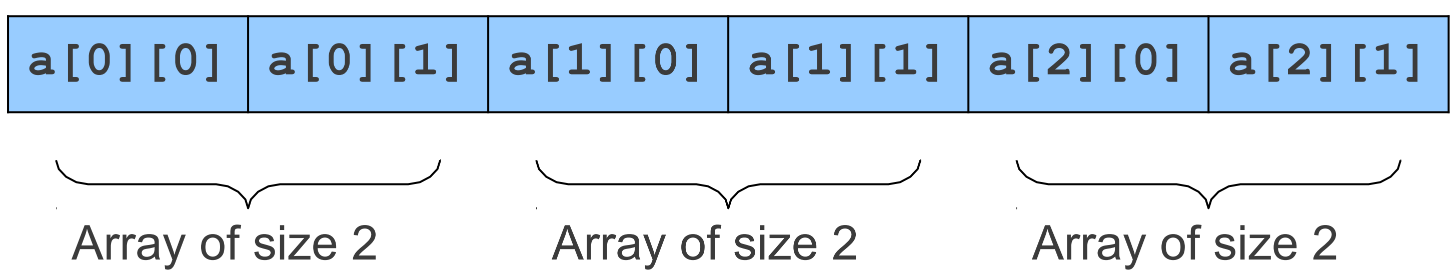
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

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?

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

Array of size 2 Array of size 2 Array of size 2

Encoding Multidimensional Arrays

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

- Java-style arrays:

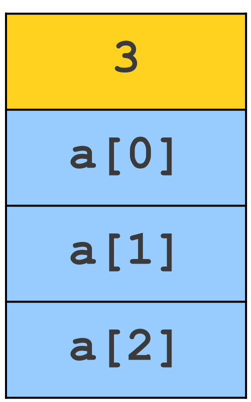
```
int[][] a = new int [3][2];
```

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];
```

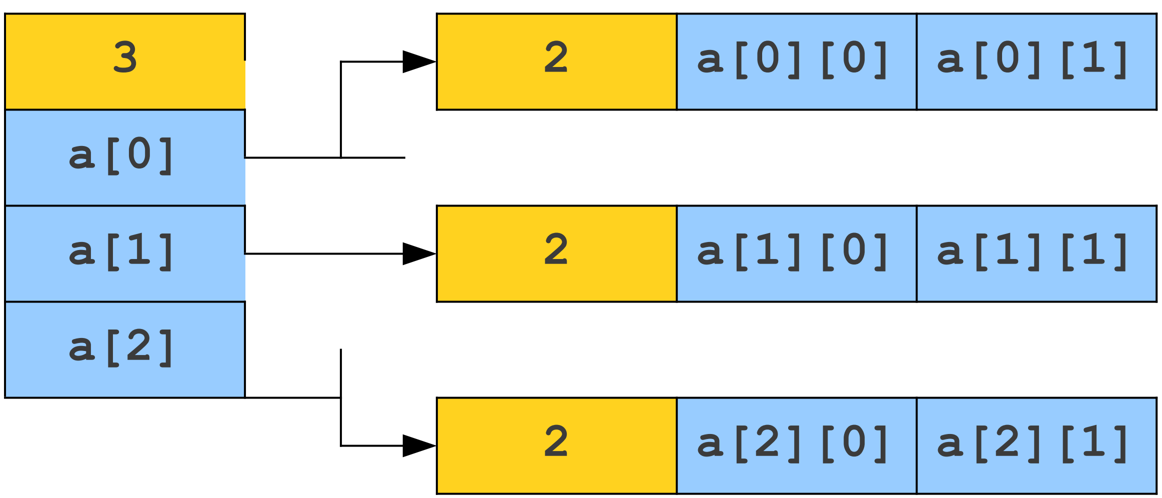


Encoding Multidimensional Arrays

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

- Java-style arrays:

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



Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is *alignment*

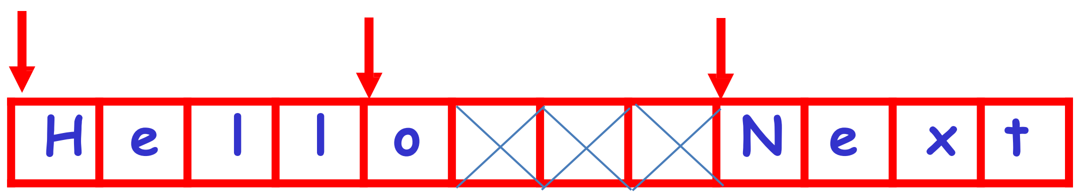
Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Alignment (Cont.)

- Example: A string "Hello"

  Takes 5 characters (without a terminating \0)



- To word align next datum, add 3 "padding" characters to the string
- The padding is not part of the string, it's just unused memory

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.

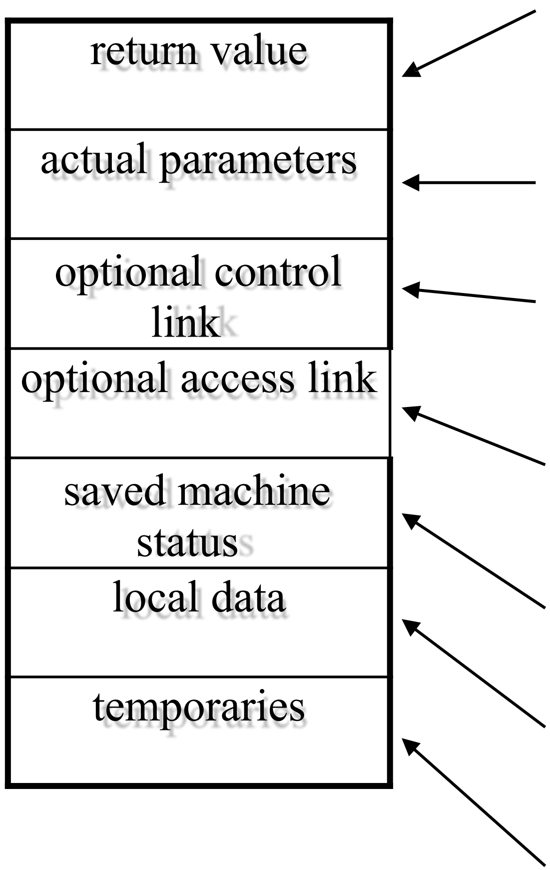
Review: The Stack

- 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 Record

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Activation Records (more details)



The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

The field for saved machine status holds information about the state of the machine before the procedure is called.

The field of local data holds data that local to an execution of a procedure..

Temporary variables is stored in the field of temporaries.

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

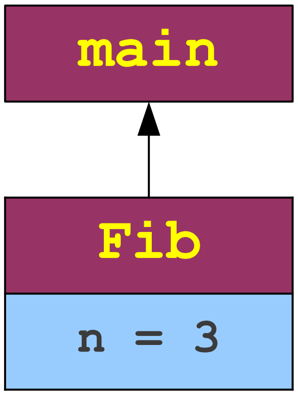
```
int main() {
    Fib(3);
}

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);
}
```

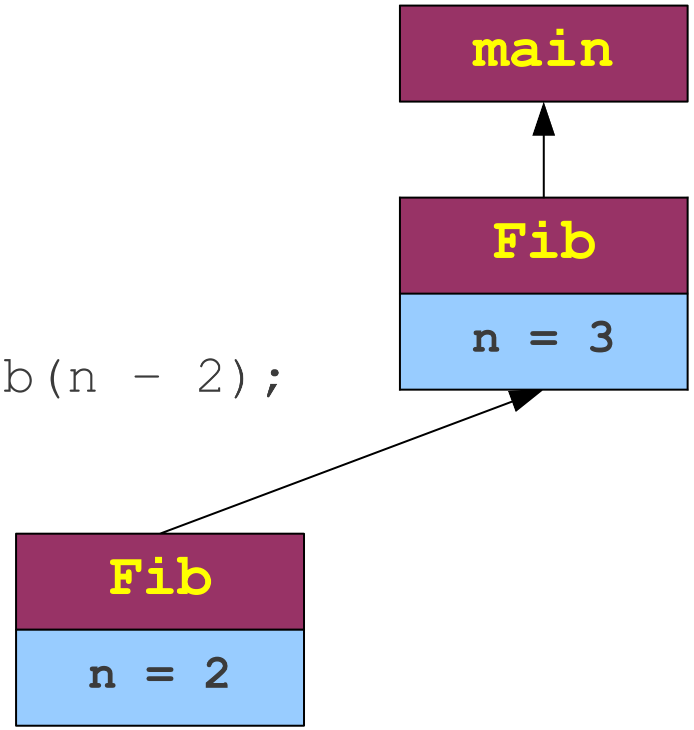
Activation Trees

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

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



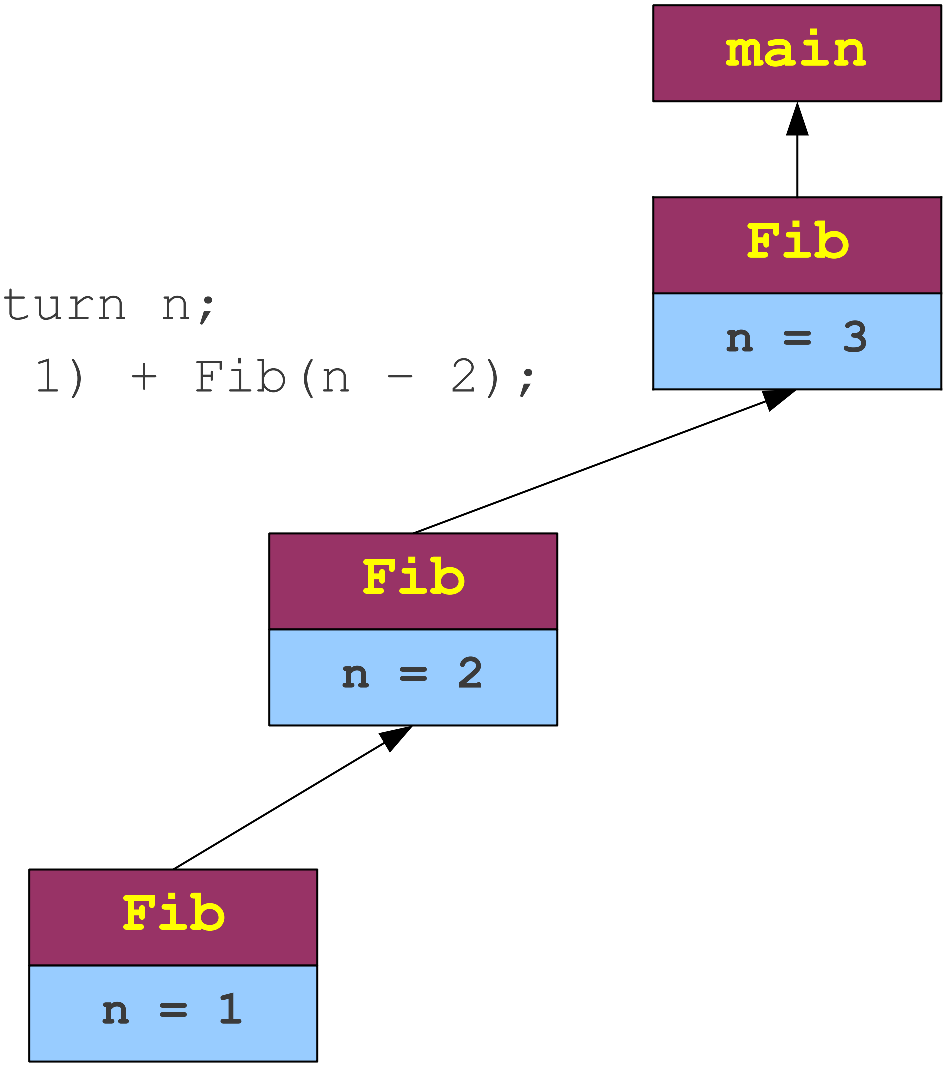
Activation Trees

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

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



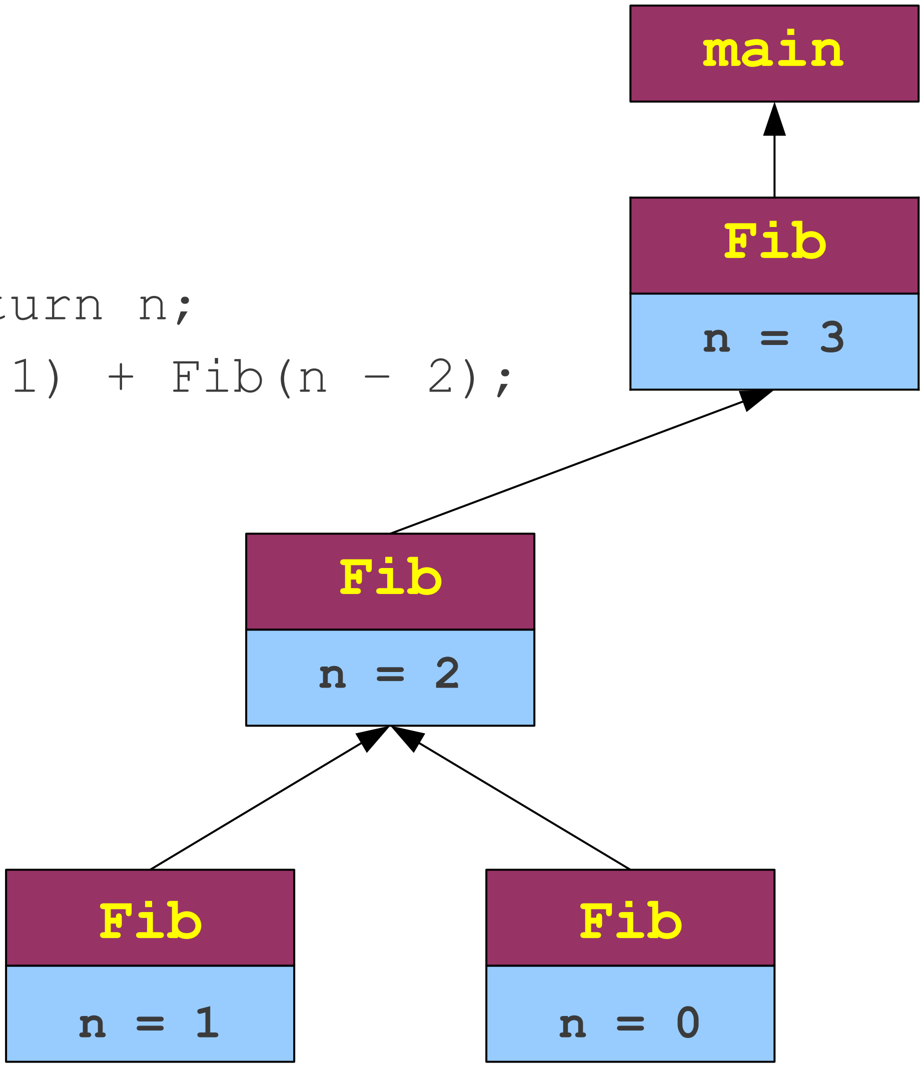
Activation Trees

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

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



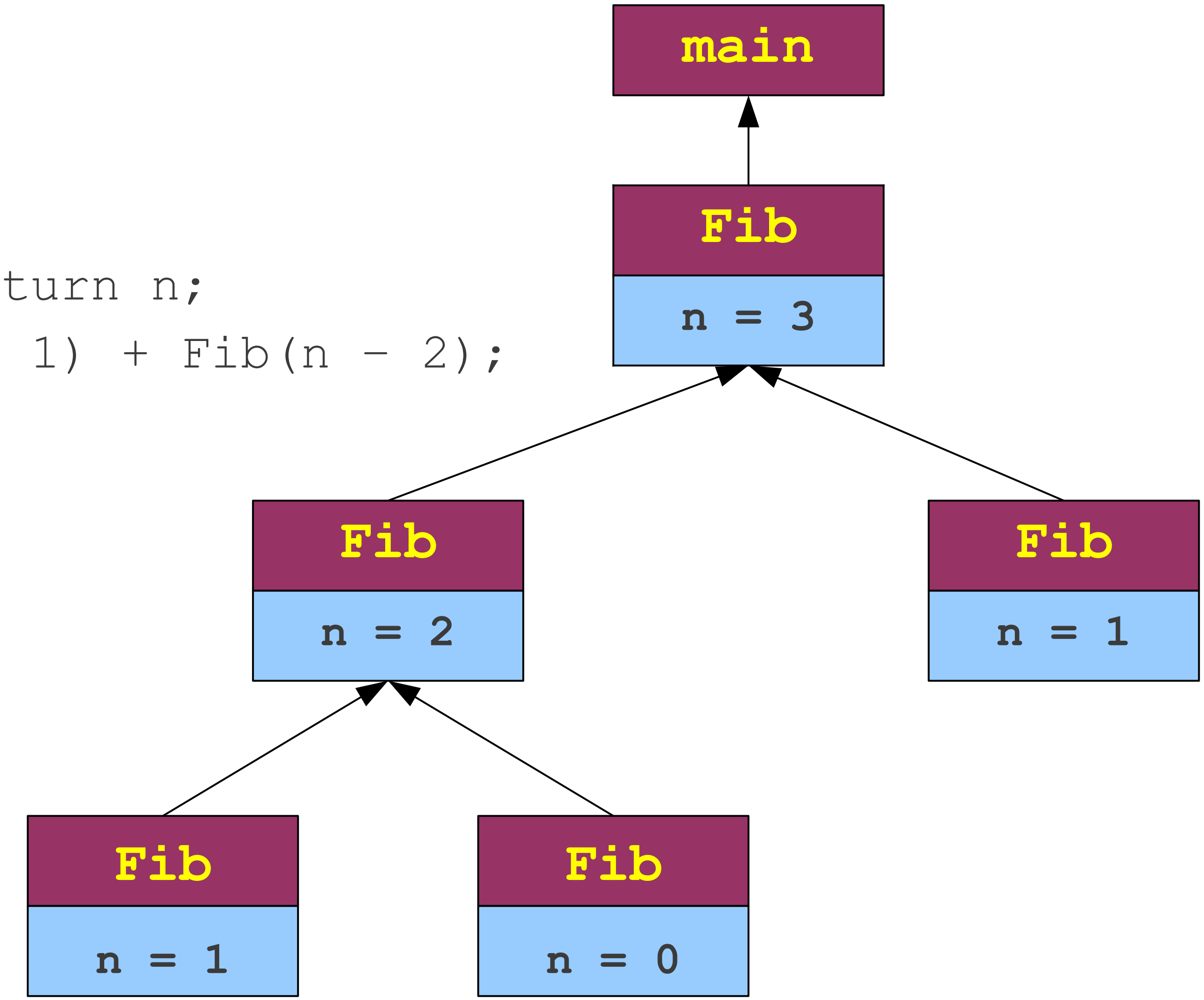
Activation Trees

```
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.”**
- 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());
}
```

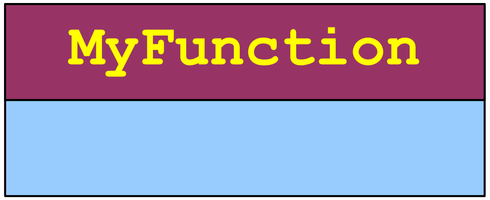
Closures

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

function MyFunction() {
    f = CreateCounter();
    print(f());
[redacted]
```

>

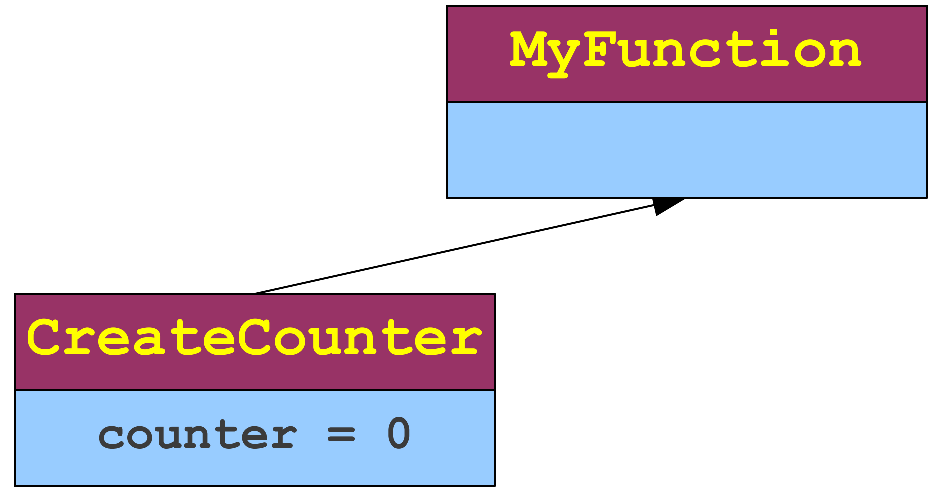
Closures

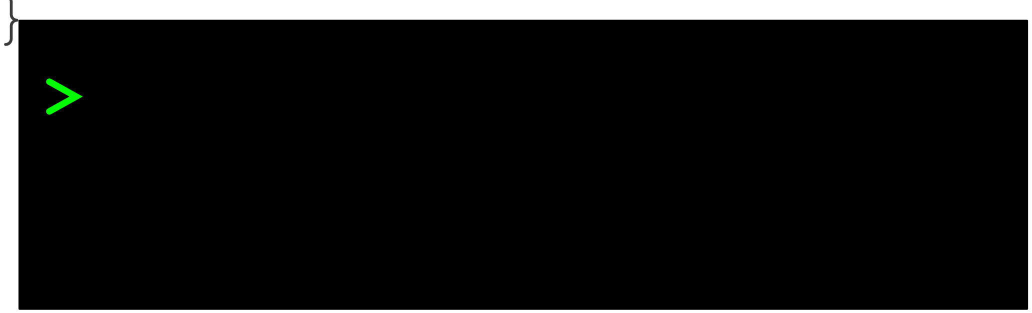
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}
function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```



Closures

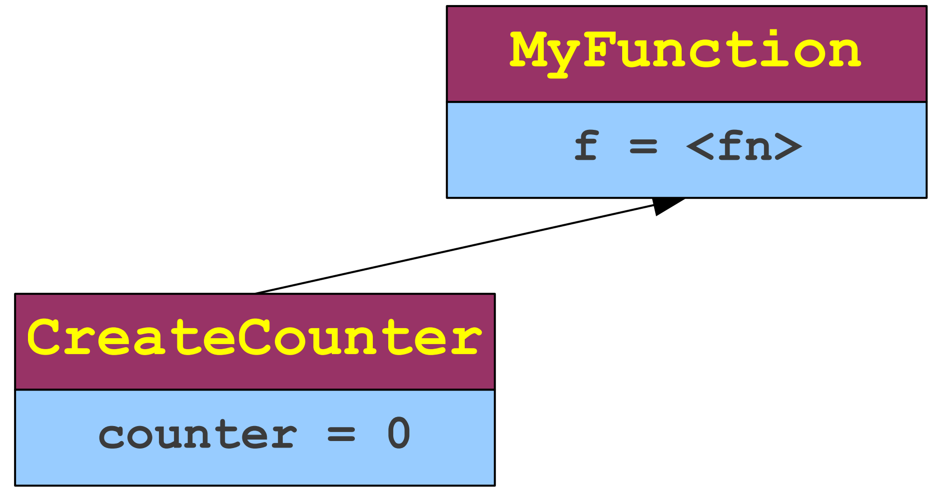
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

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());
}
```

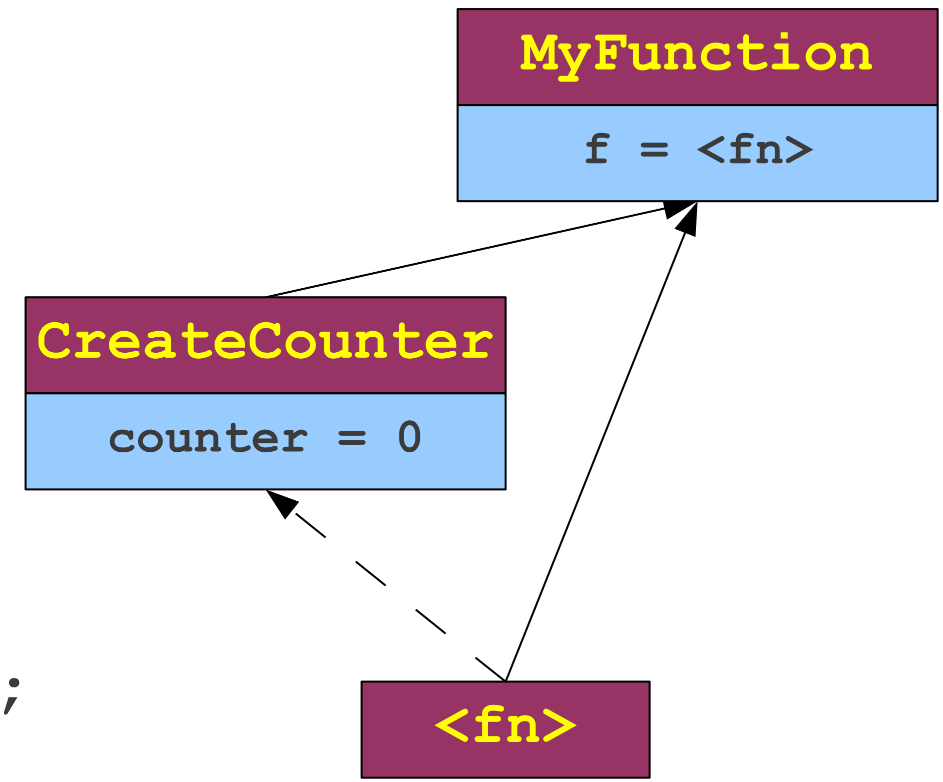



```
>
```

Closures

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

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

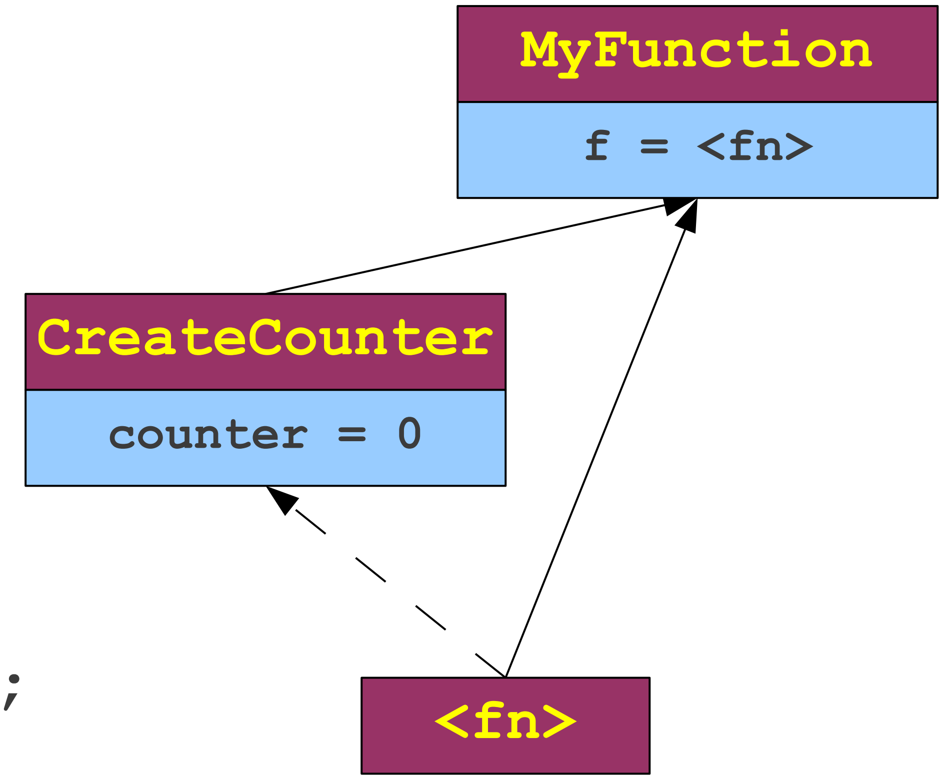
```
>
```



Closures

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

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

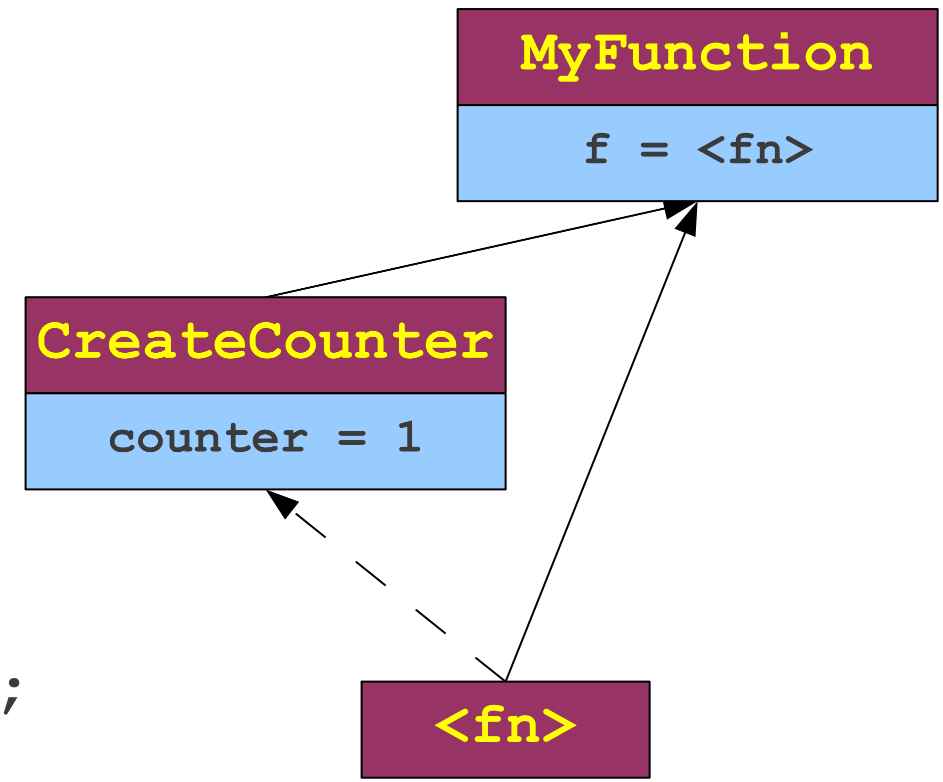
```
>
```



Closures

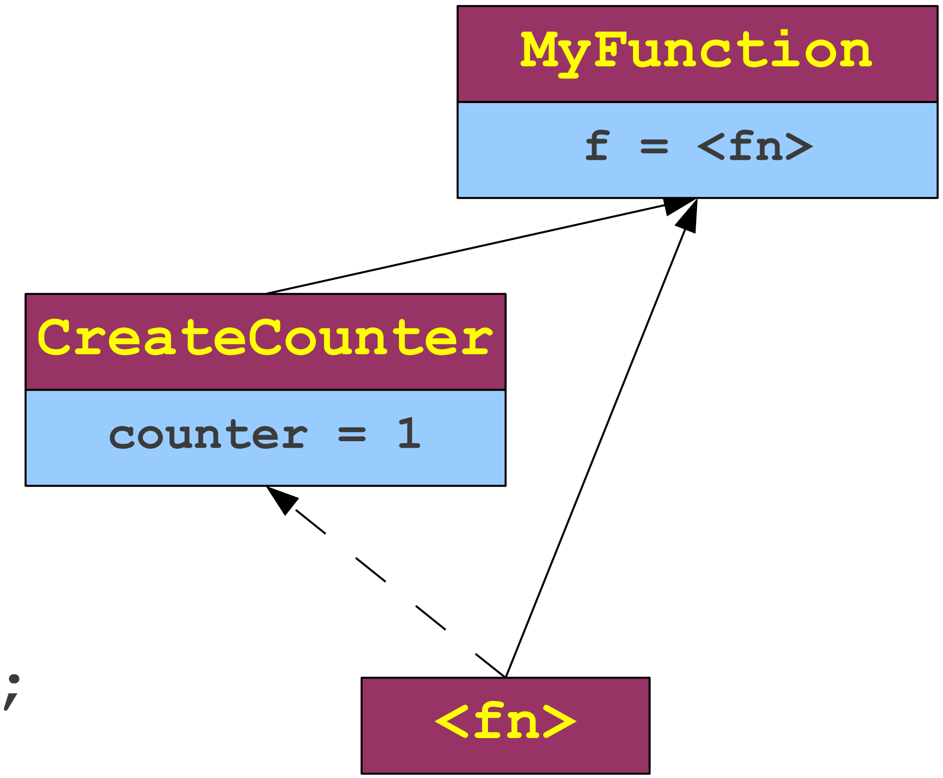
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

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());
}
```

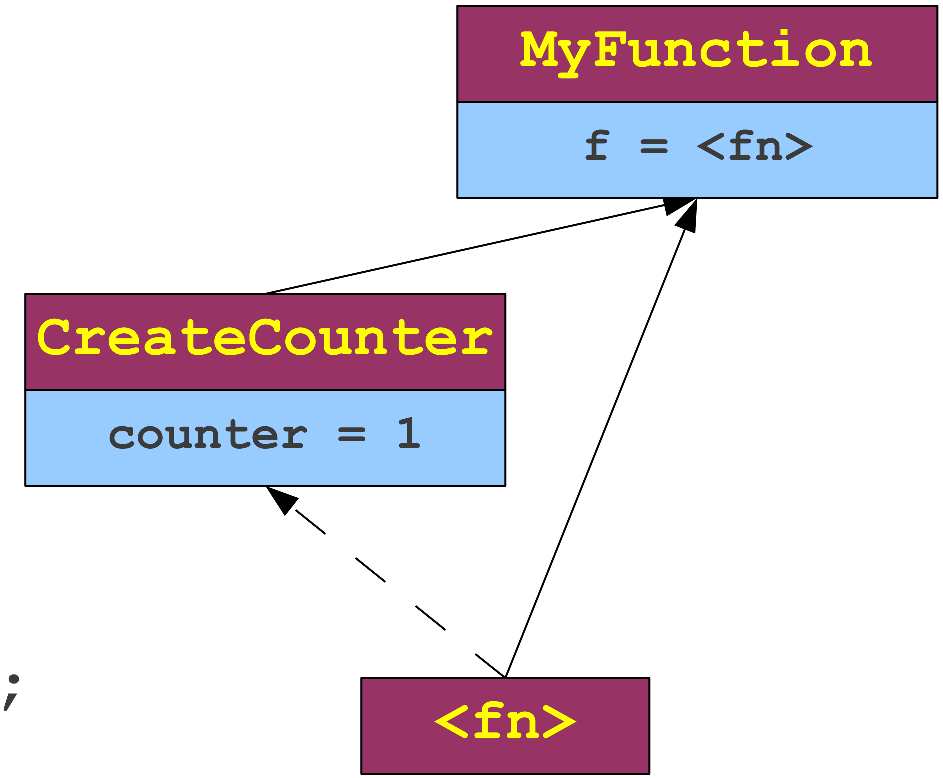



```
>
```

Closures

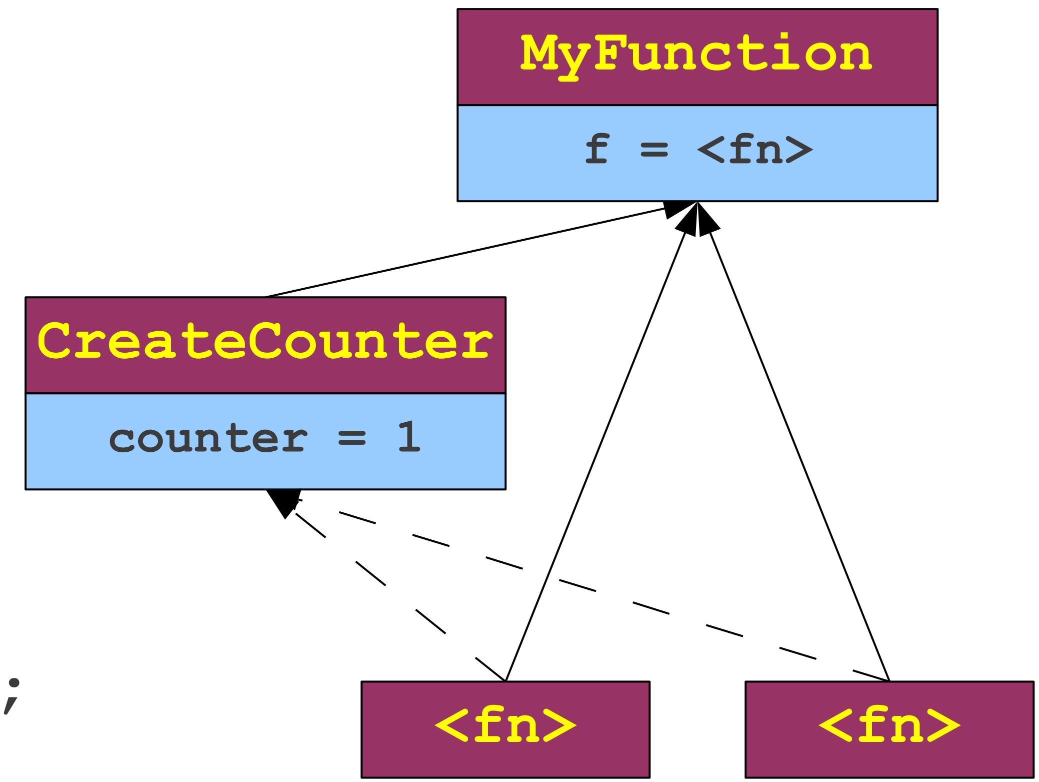
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

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());
}
```

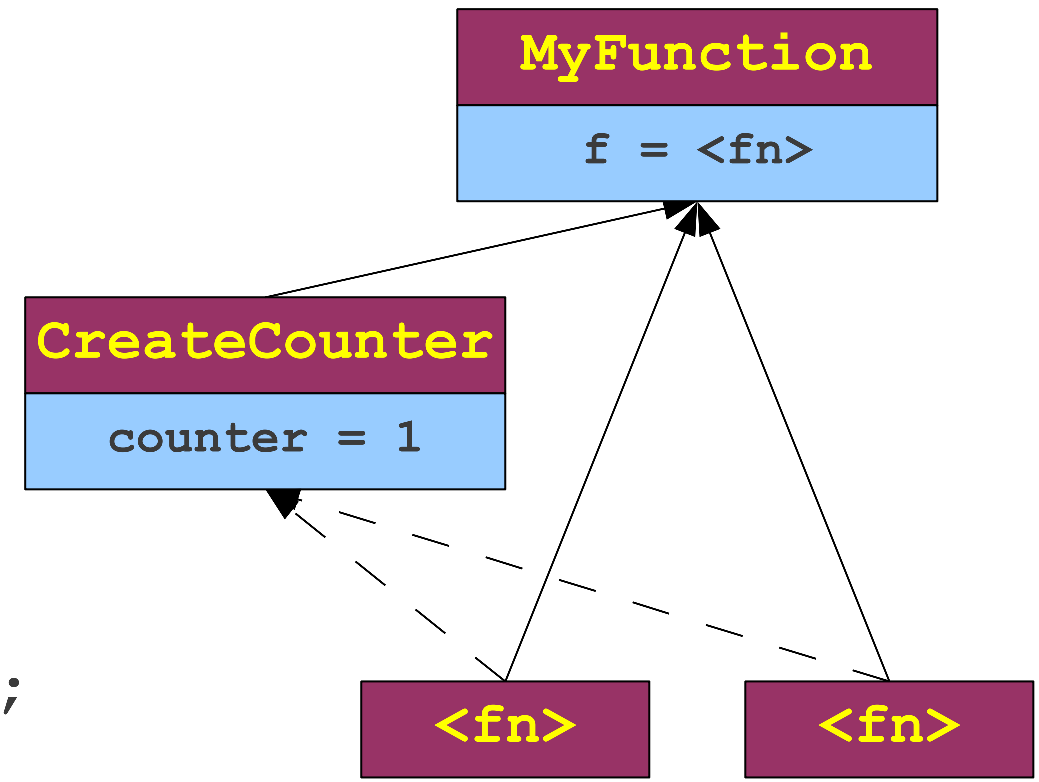



```
> 1
```

Closures

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

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

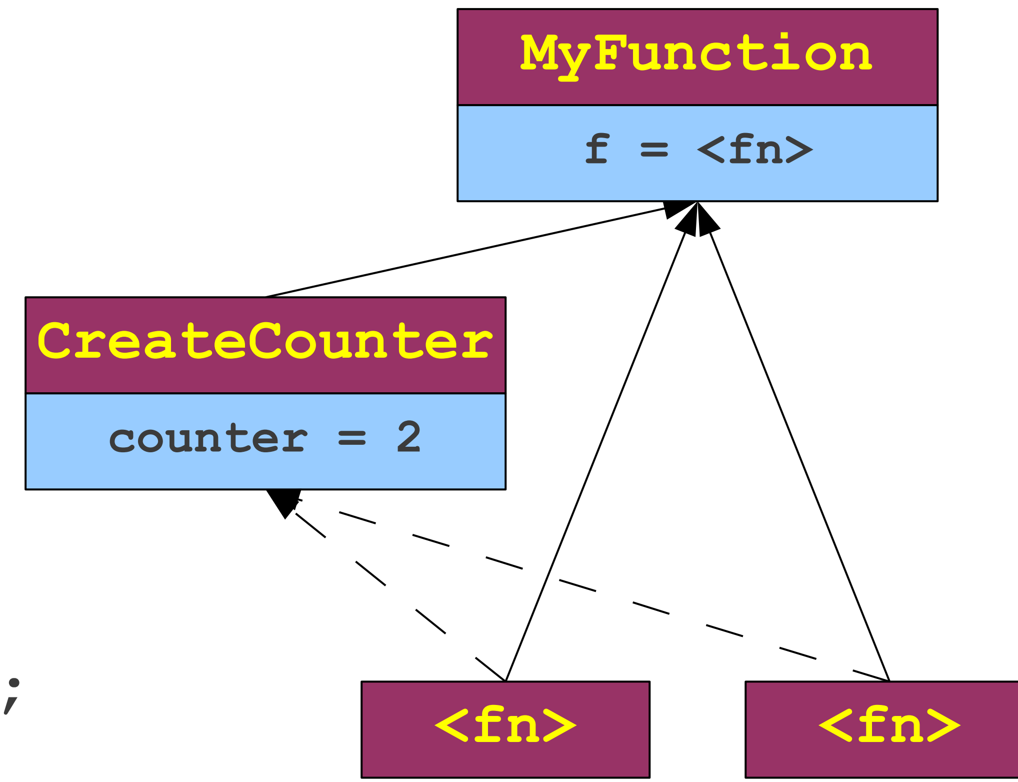
```
> 1
```



Closures

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

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

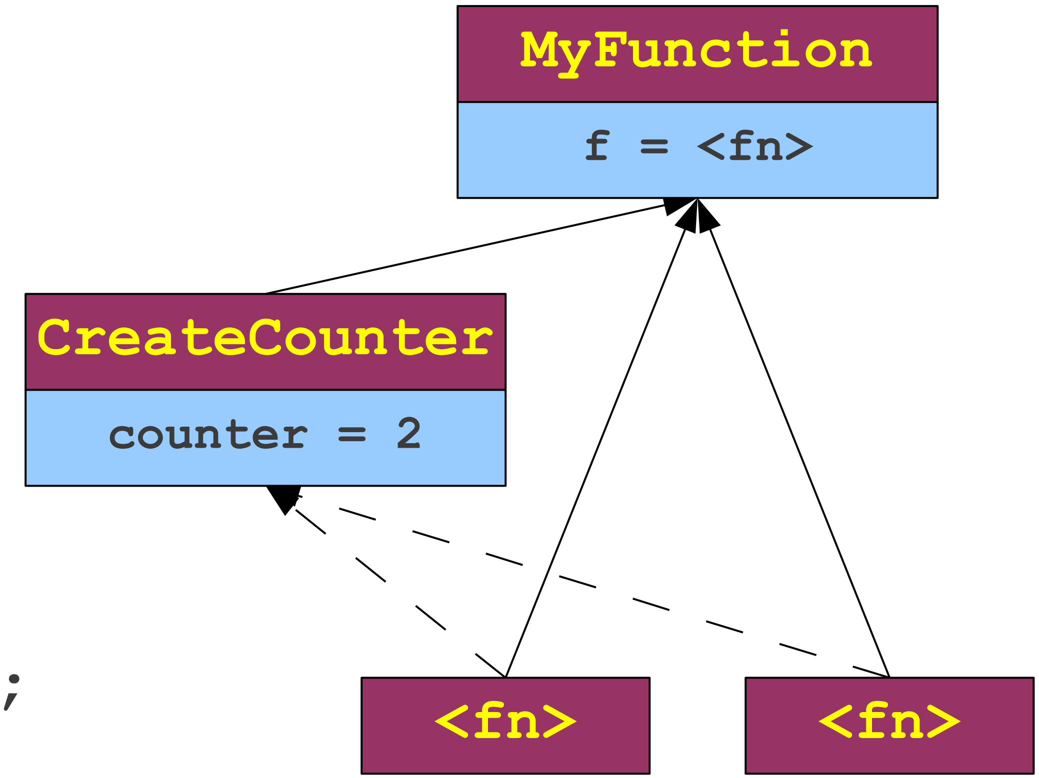
```
> 1
```



Closures

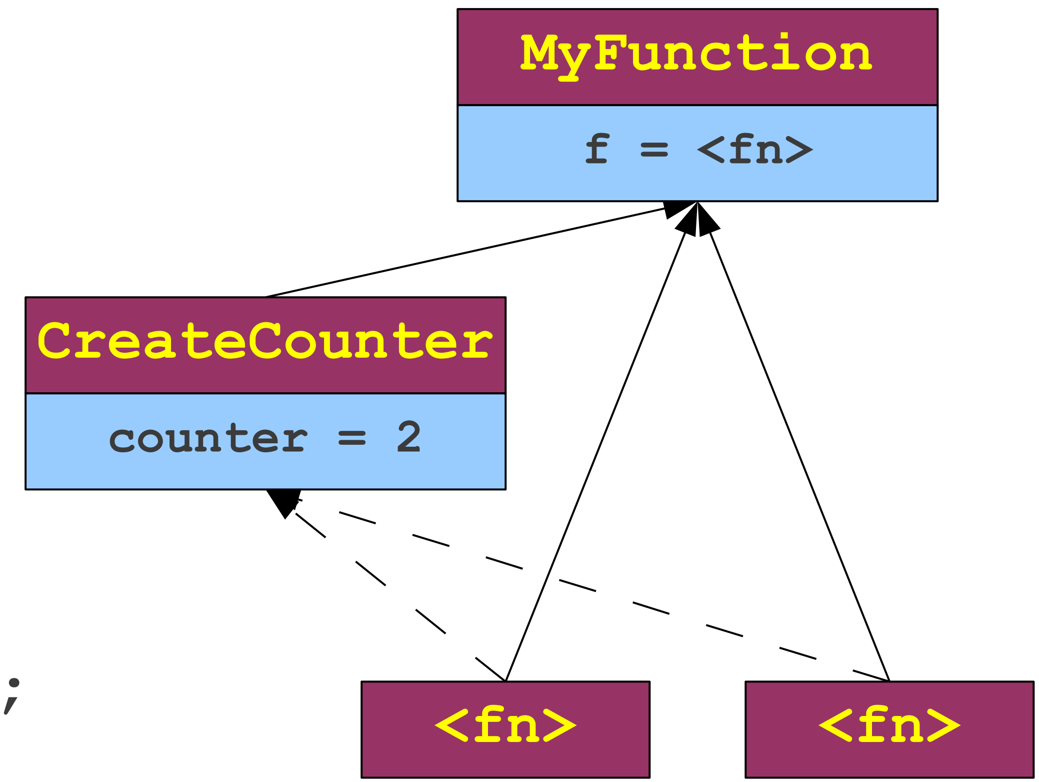
```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

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
```

myFunc

```
>
```

Coroutines

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

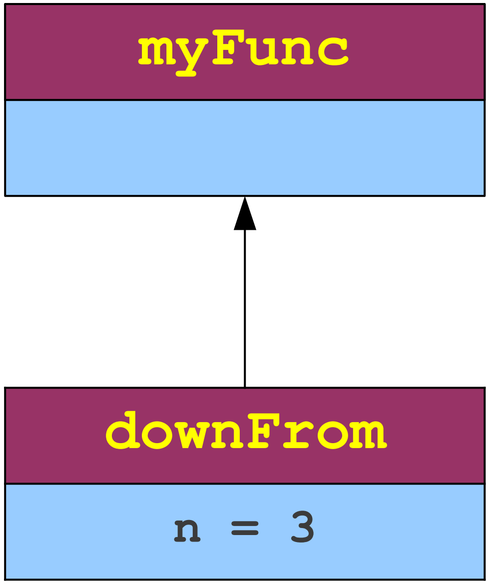
myFunc

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

```
>
```

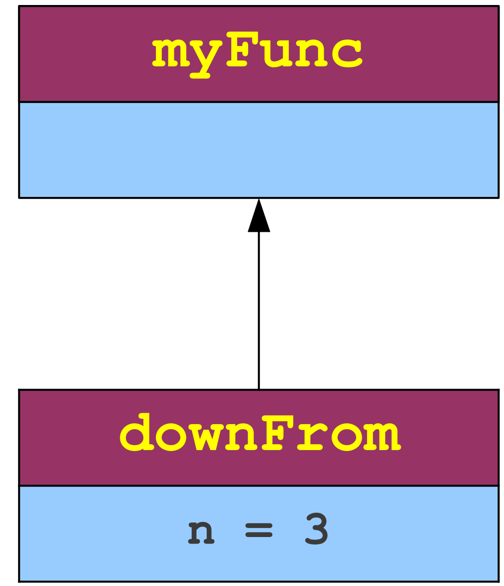
Coroutines

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


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



Coroutines

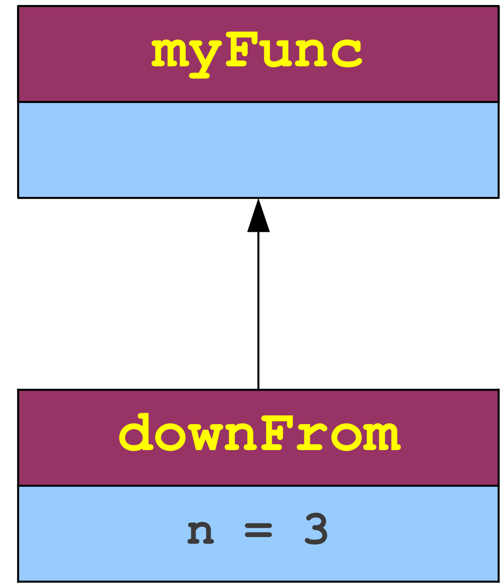
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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




```
>
```

Coroutines

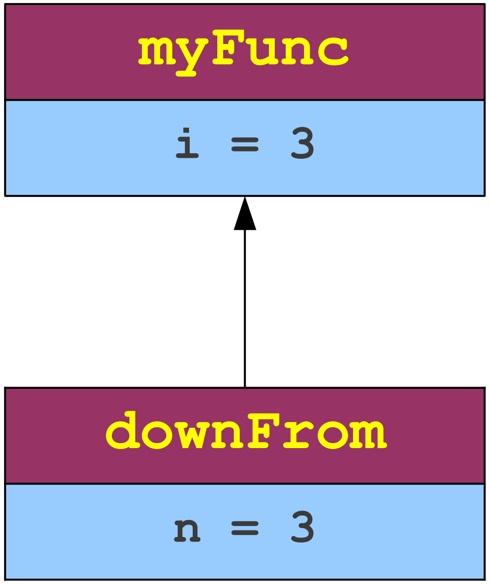
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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




```
>
```

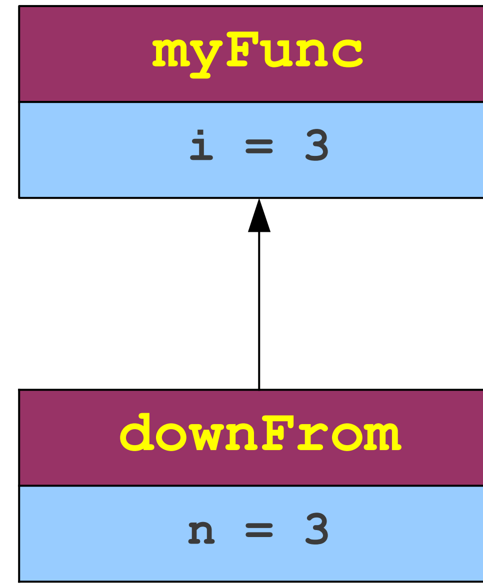
Coroutines

```
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
```

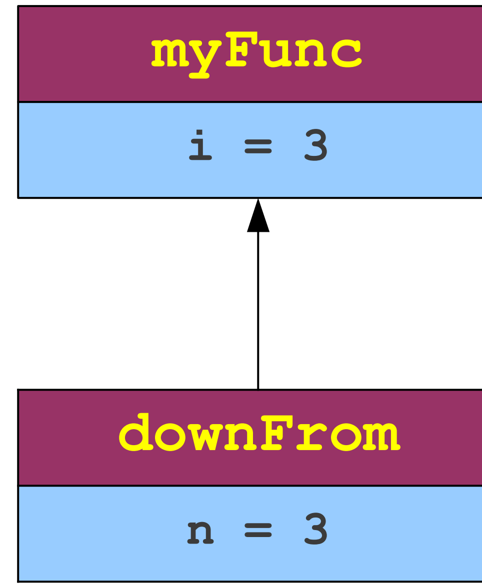



```
def myFunc():
    for i in downFrom(3)
        print i
```

```
>
```

Coroutines

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

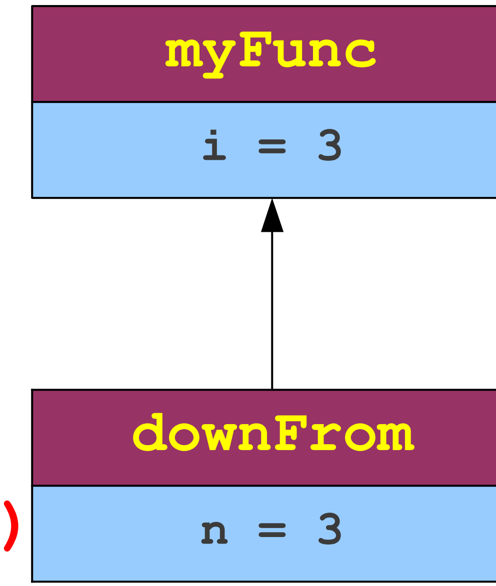



```
def myFunc():
    for i in downFrom(3)
        print i
```

```
> 3
```

Coroutines

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

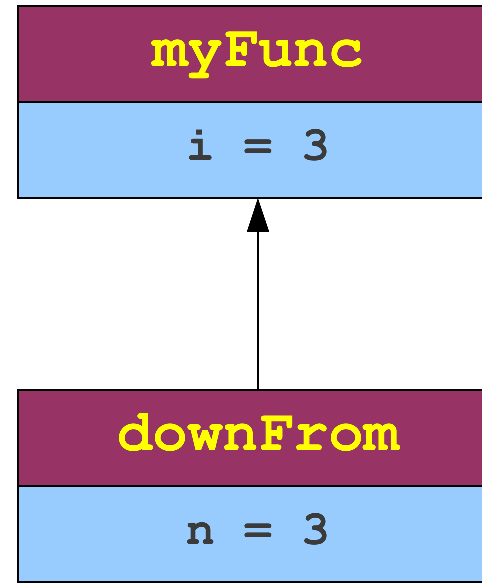



```
def myFunc():
    for i in downFrom(3)
        print i
```

```
> 3
```

Coroutines

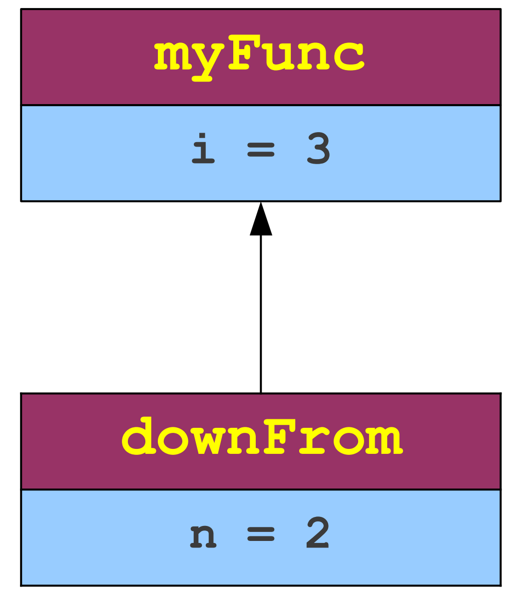
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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




```
> 3
```

Coroutines

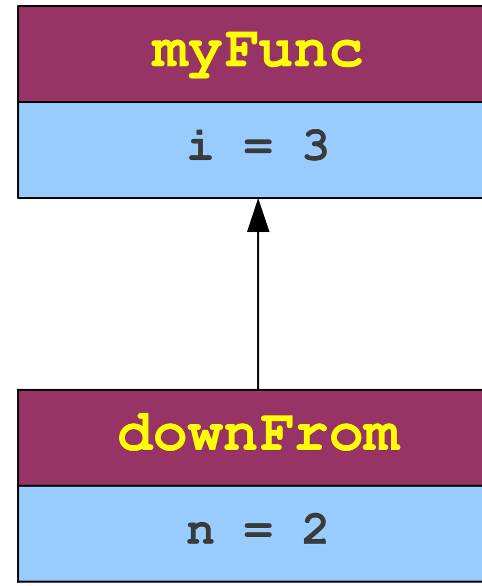
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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
```

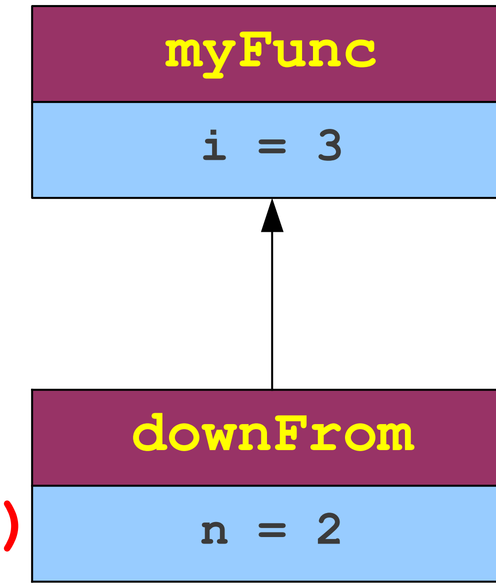



```
> 3
```

Coroutines

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

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

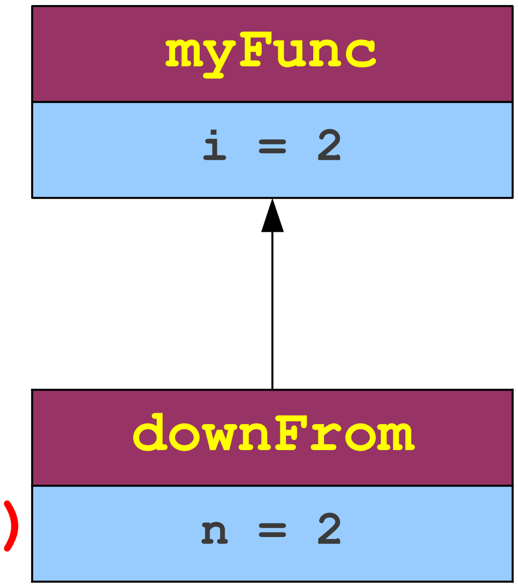



```
> 3
```

Coroutines

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

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

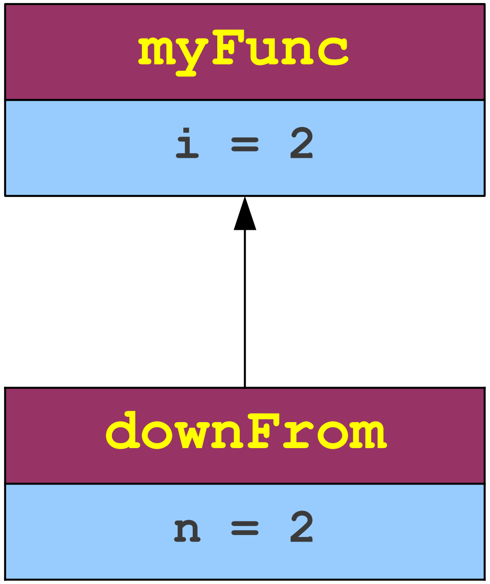



```
> 3
```

Coroutines

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

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

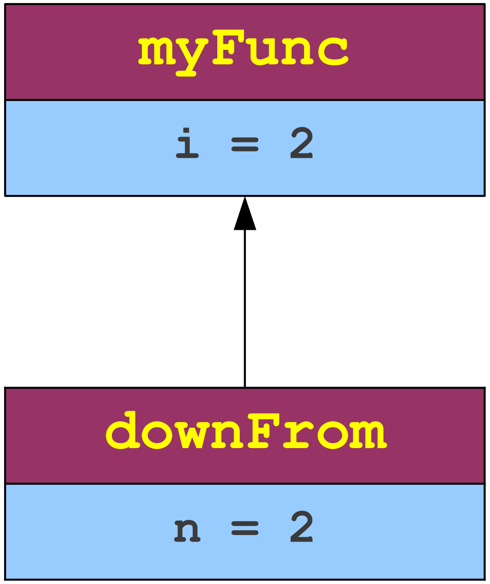



```
> 3
```

Coroutines

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

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

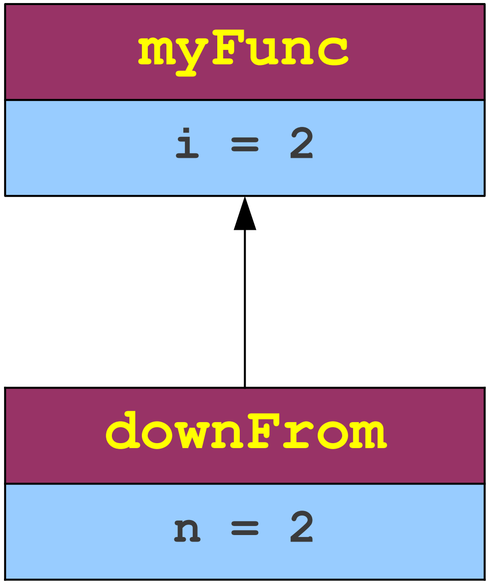



```
> 3
  2
```

Coroutines

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

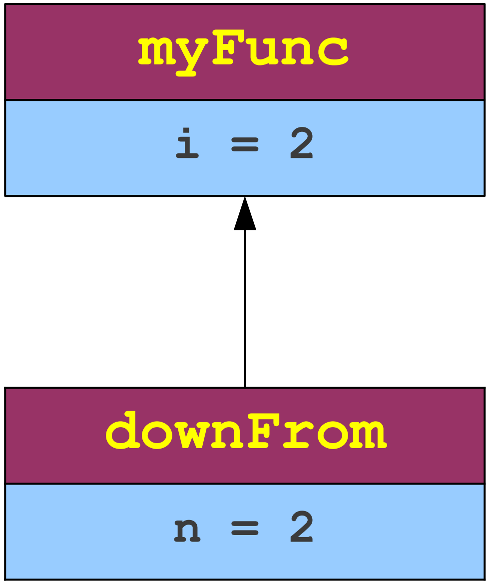
```
def myFunc():
    for i in downFrom(3)
        print i
```




```
> 3
  2
```

Coroutines

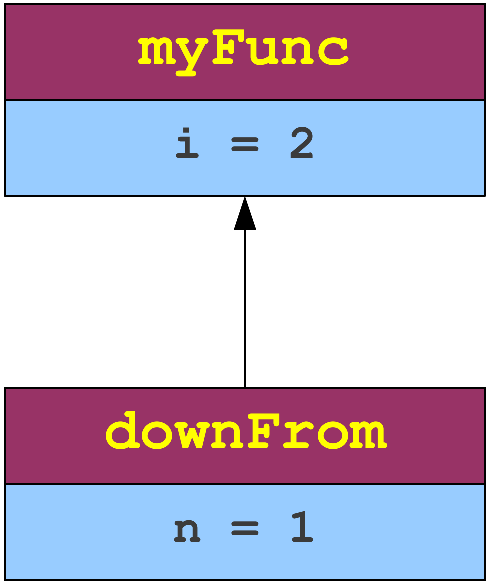
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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




```
> 3
  2
```

Coroutines

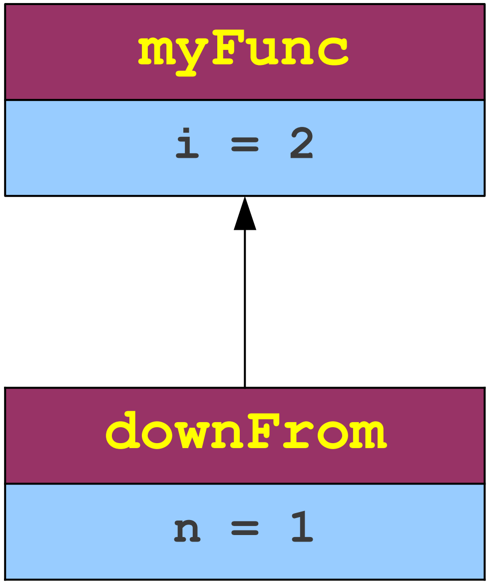
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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
```

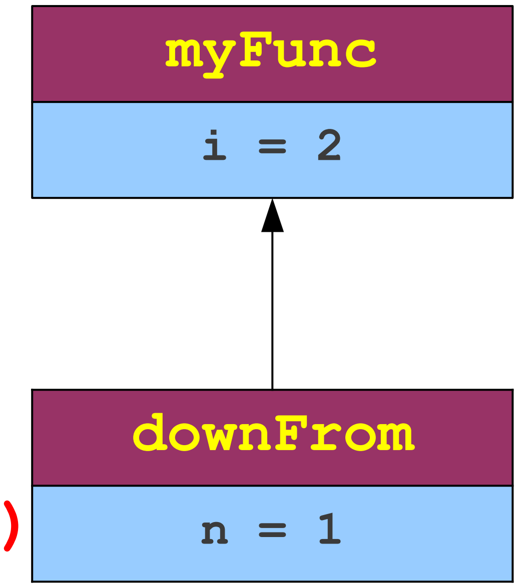



```
> 3
  2
```

Coroutines

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

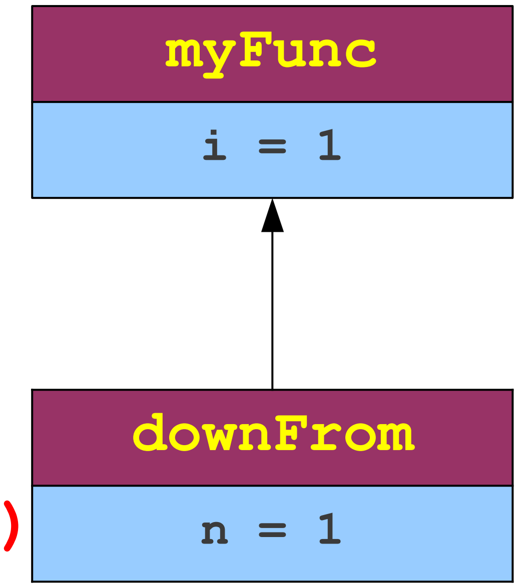
```
def myFunc():
    for i in downFrom(3)
        print i
```




```
> 3
  2
```

Coroutines

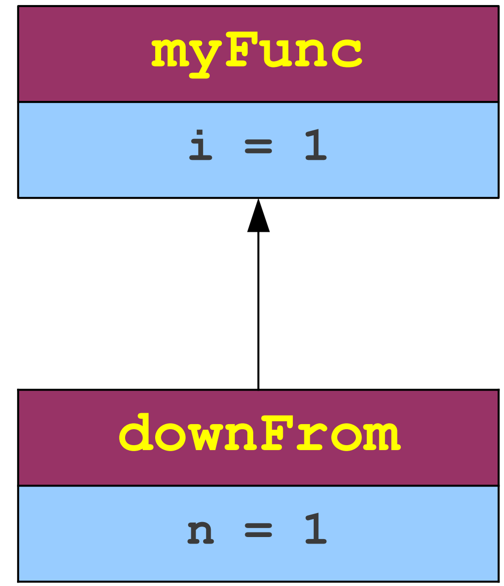
```
def downFrom(n):
    while n > 0:
        yield n
            n = n - 1

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




```
> 3
  2
```

Coroutines

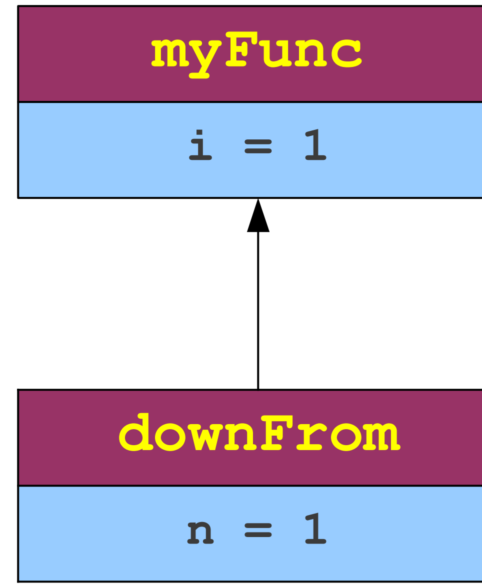
```
def downFrom(n):
    while n > 0:
        yield n
                    n = n - 1

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




```
> 3
  2
```

Coroutines

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



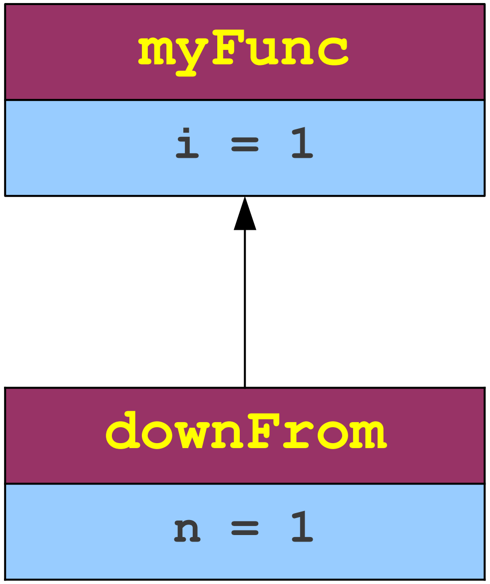

```
def myFunc():
    for i in downFrom(3)
        print i
```

```
> 3
  2
  1
```

Coroutines

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

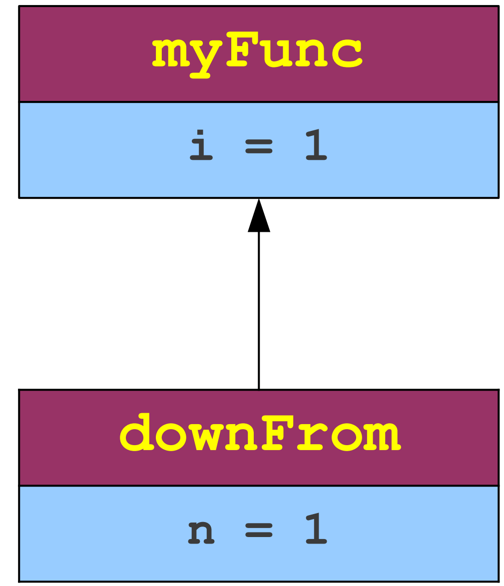
```
def myFunc():
    for i in downFrom(3)
        print i
```




```
> 3
  2
  1
```

Coroutines

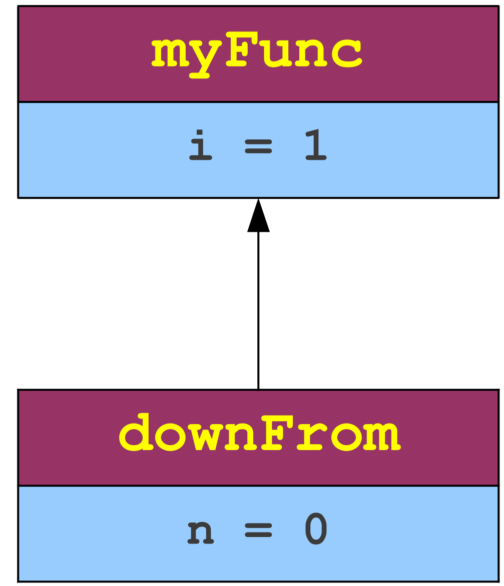
```
def downFrom(n):
    while n > 0:
        yield n
        n = n - 1


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
```

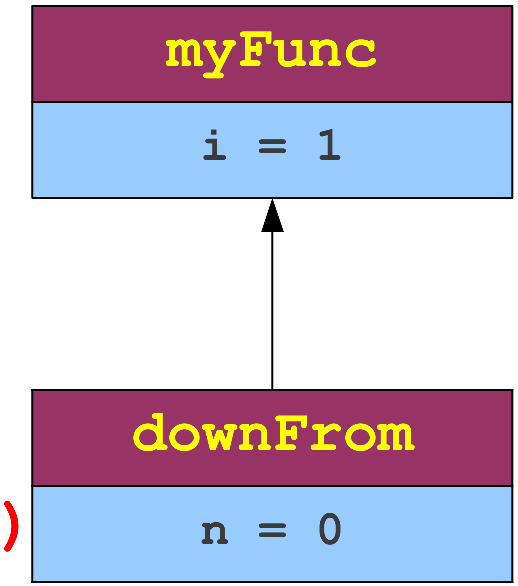



```
> 3
  2
  1
```

Coroutines

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

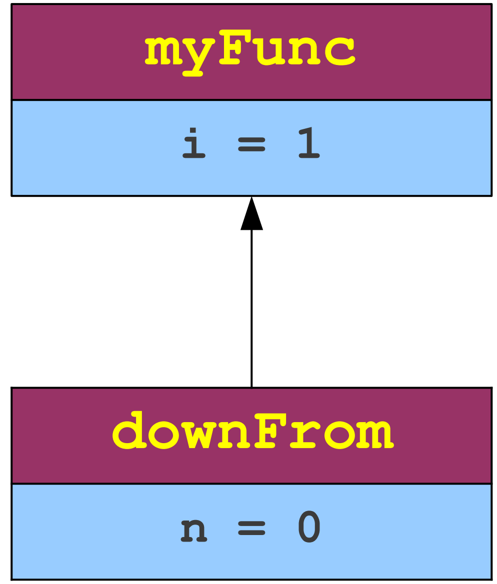
```
def myFunc():
    for i in downFrom(3)
        print i
```




```
> 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!**

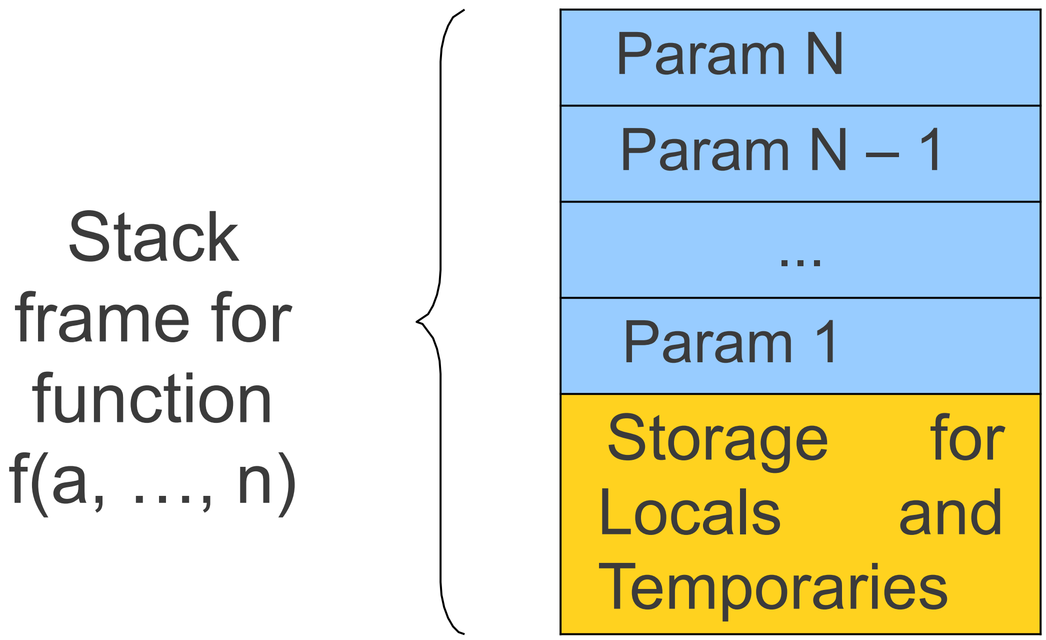
Functions in Decaf

- We use an explicit runtime stack.
- Each activation record needs to hold
  - All of its parameters.
  - All of its local variables.
  - All temporary variables introduced by the IR generator (more on that later).
- Where do these variables go?
- Who allocates space for them?

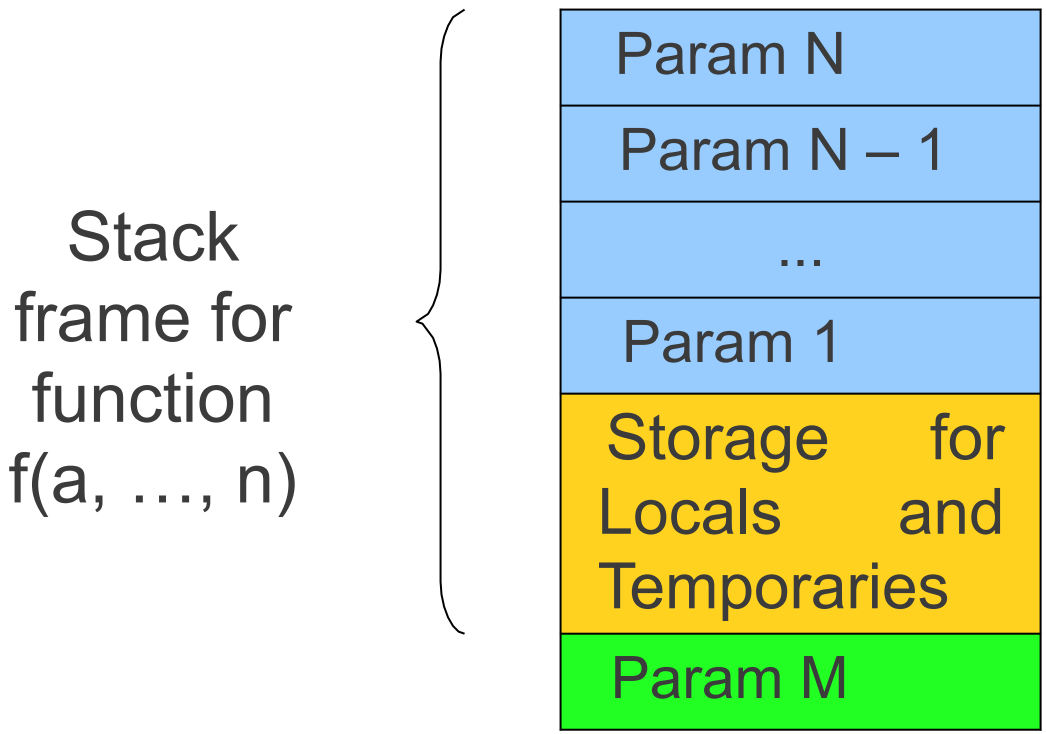
Decaf Stack Frames

- The **logical** layout of a Decaf stack frame is created by the IR generator.
  - Ignores details about machine-specific calling conventions.
  - We'll discuss today.
- The **physical** layout of a Decaf stack frame is created by the code generator.
  - Based on the logical layout set up by the IR generator.
  - Includes frame pointers, caller-saved registers, and other fun details like this.
  - We'll discuss when talking about code generation.

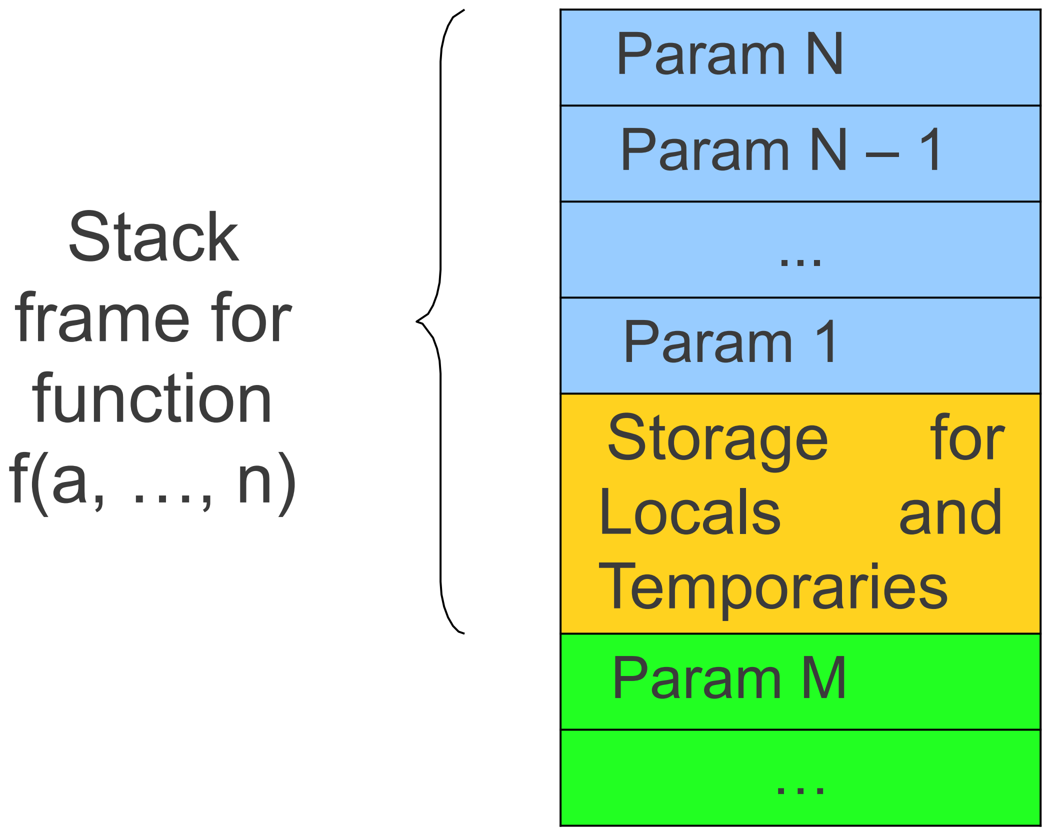
A Logical Decaf Stack Frame



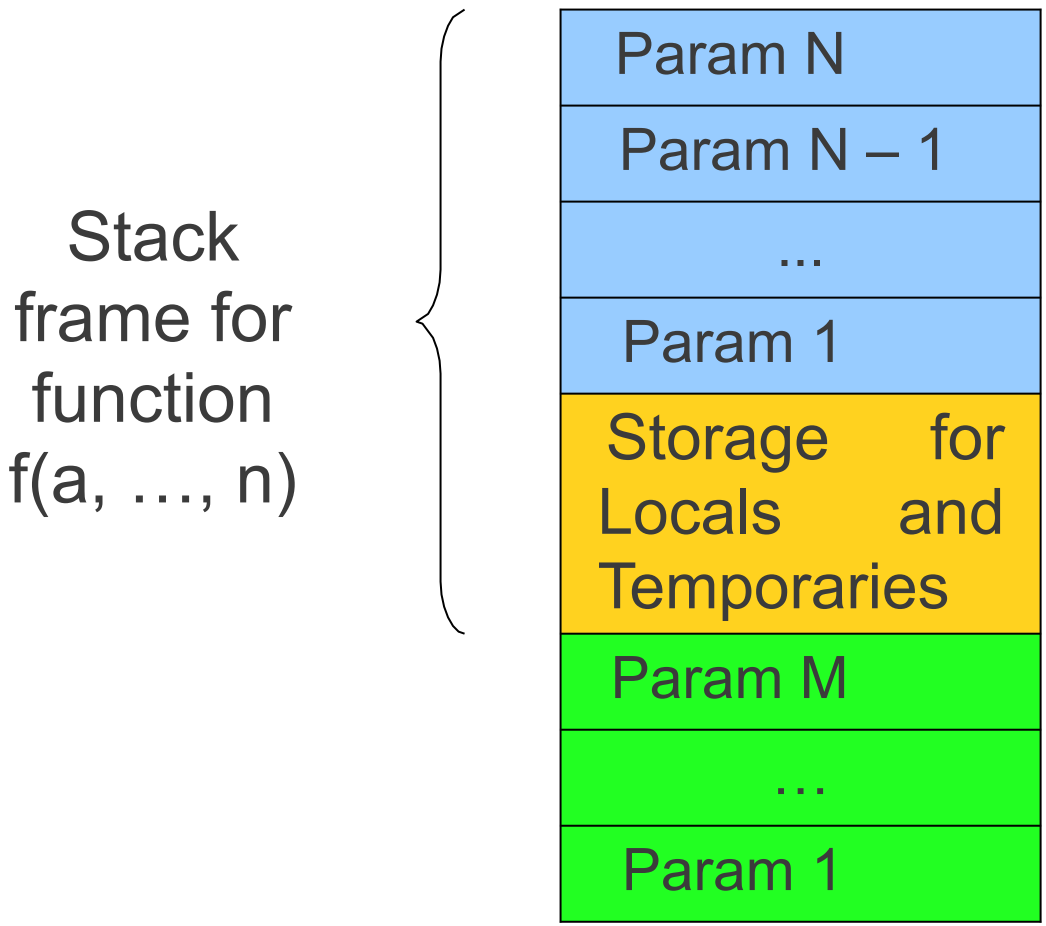
A Logical Decaf Stack Frame



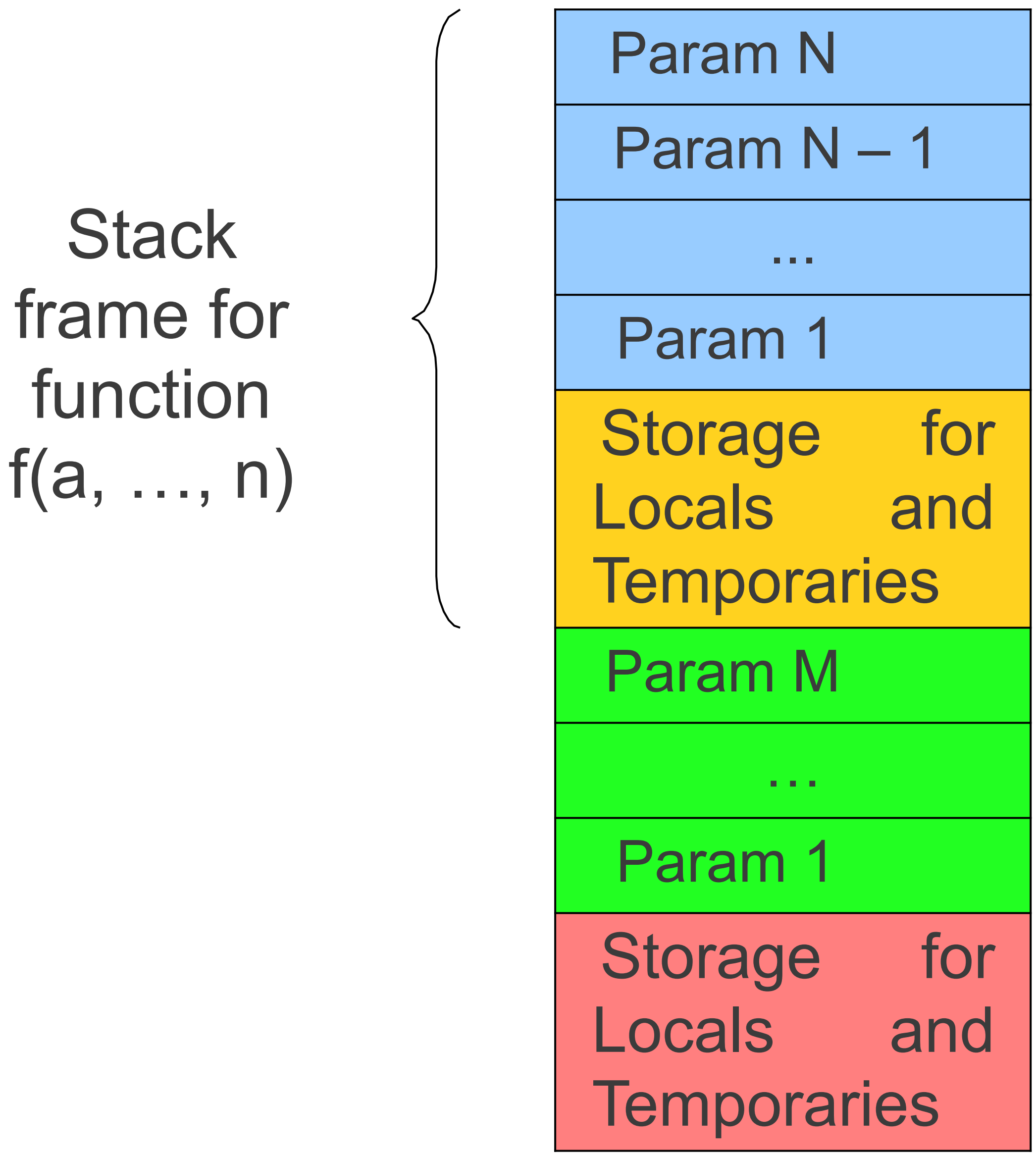
A Logical Decaf Stack Frame



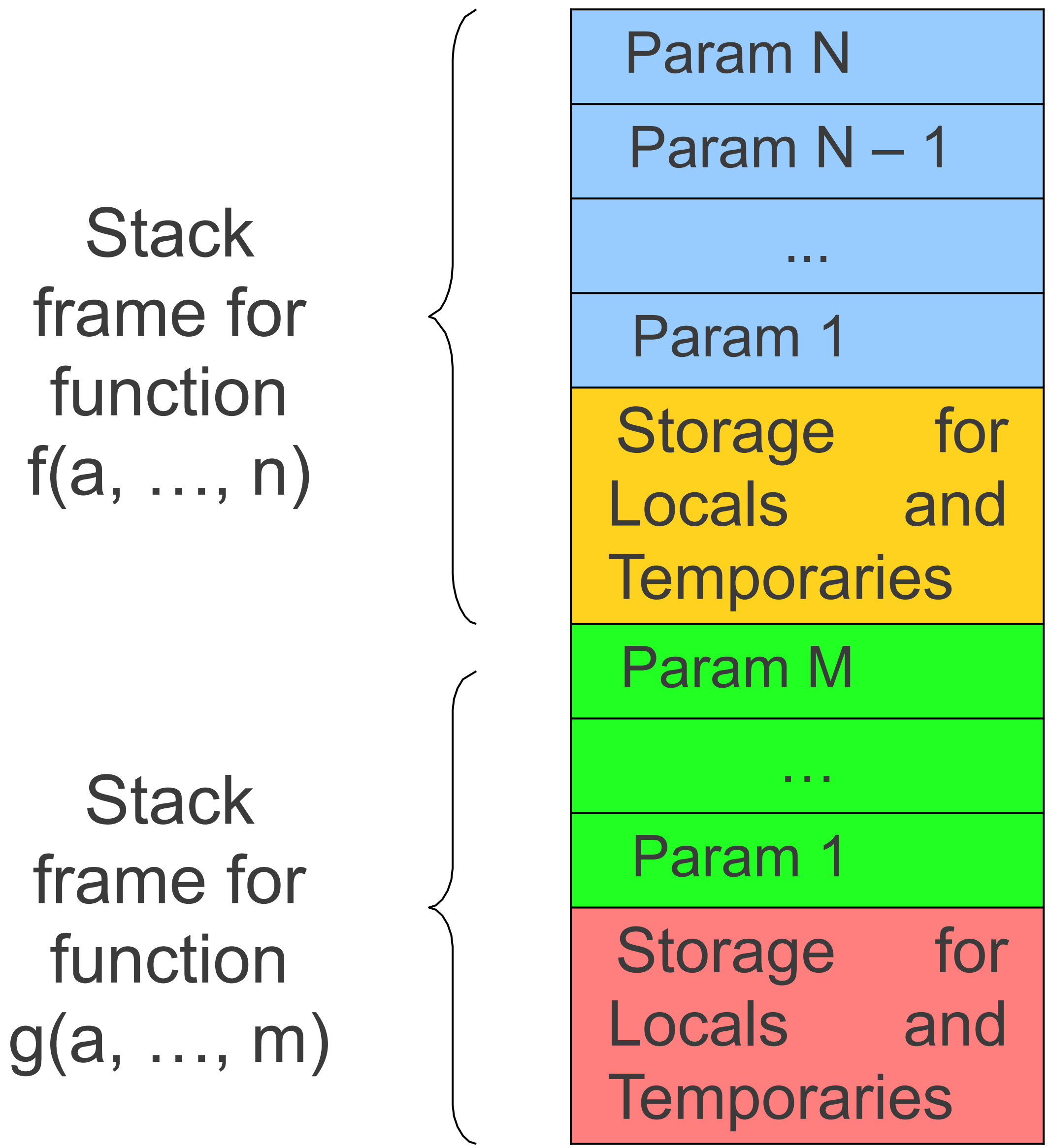
A Logical Decaf Stack Frame



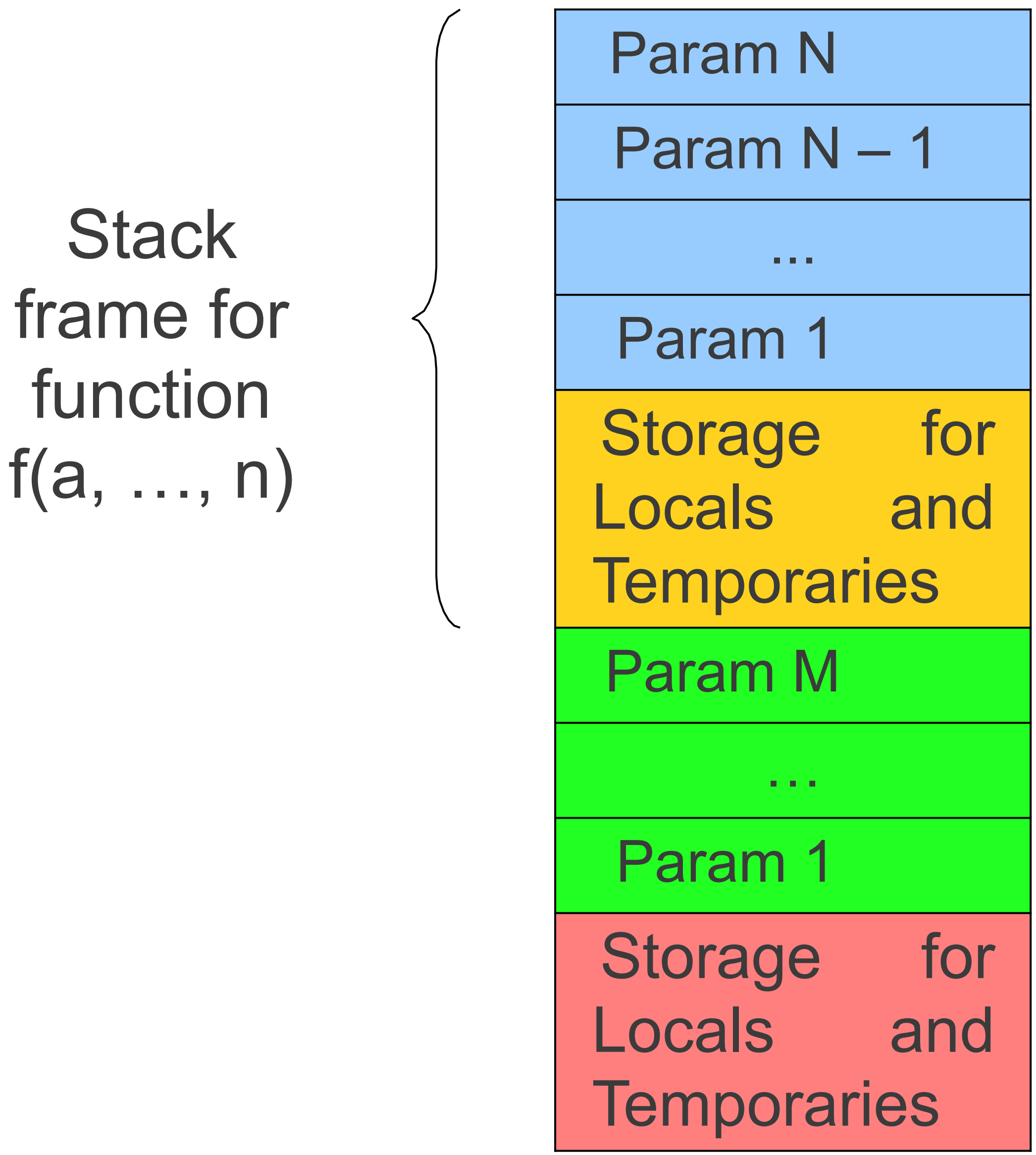
A Logical Decaf Stack Frame



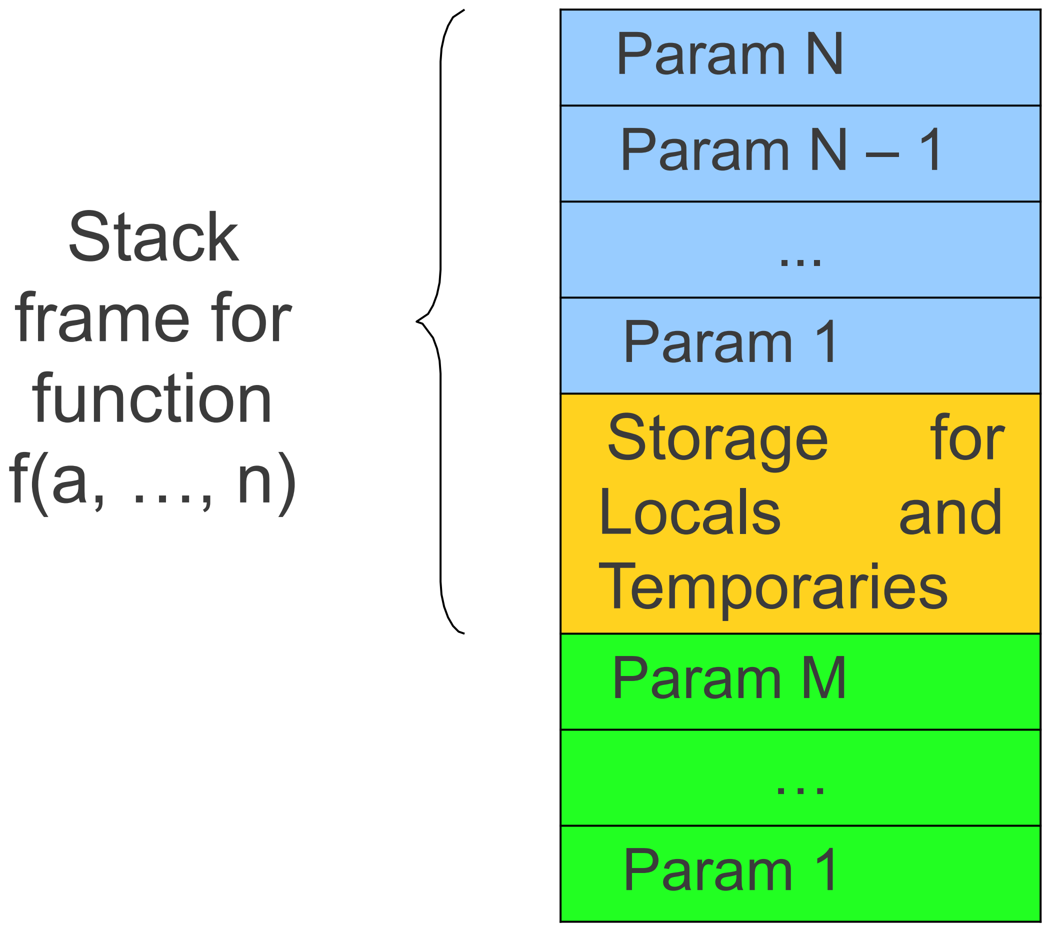
A Logical Decaf Stack Frame



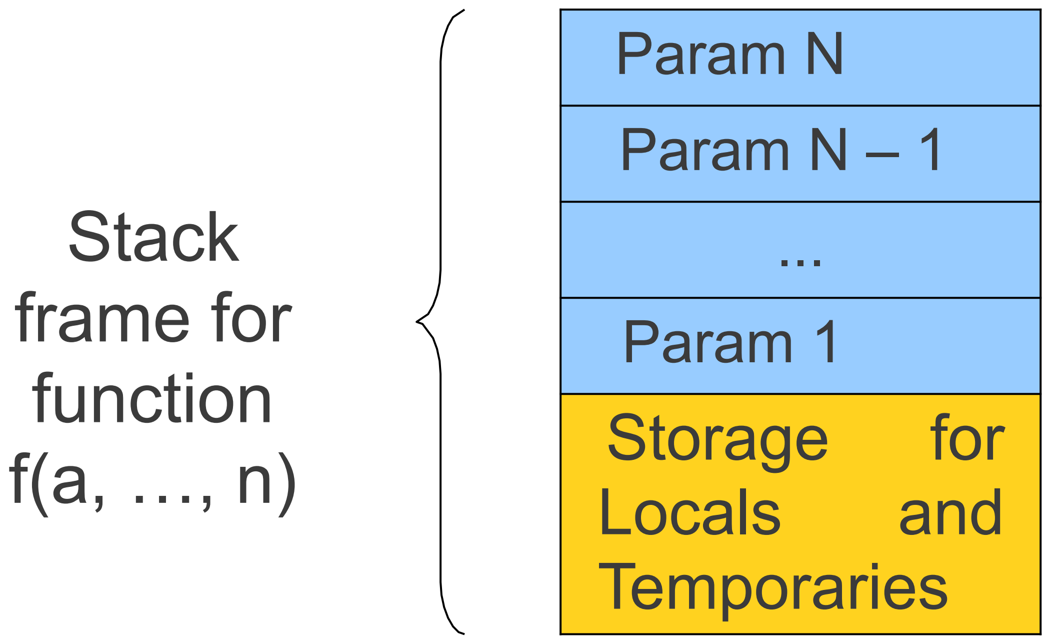
A Logical Decaf Stack Frame



A Logical Decaf Stack Frame



A Logical Decaf Stack Frame



Decaf IR Calling Convention

- Caller responsible for pushing and popping space for callee's arguments.

- Callee responsible for pushing and popping space for its own temporaries.
  - (Why?)

Parameter Passing Approaches

- Two common approaches.
- Call-by-value
  - Parameters are *copies* of the values specified as arguments.
- Call-by-reference:
  - Parameters are *pointers* to values specified as parameters.

Parameter Passing Approaches

- Call-by-result:
  - Initial value is undefined, but before control pass to caller it copies back.
  - E.g. Ada language
- Call-by-value/result:
  - Actual parameters copy into formal parameters at the end formal parameters copy back to caller variables.
  - E.g. Ada, Algol W., FORTRAN

Parameter Passing Approaches

- Call-by-name:
  - Each time the actual parameter re-evaluated.
  - E.g. Algol, Haskell, Scala
- Call-by-need:
  - Is a memorized call-by-name. zero or one time evaluation.
  - E.g. R, Haskell

Other Parameter Passing Ideas

- JavaScript: Functions can be called with any number of arguments.
  - Parameters are initialized to the corresponding argument, or **`undefined`** if not enough arguments were provided.
  - The entire parameters array can be retrieved through the **`arguments`** array.
- How might this be implemented?

Other Parameter Passing Ideas

- Python: **Keyword Arguments**
  - Functions can be written to accept any number of key/value pairs as arguments.
  - Values stored in a special argument (traditionally named `kwargs`)
  - `kwargs` can be manipulated (more or less) as a standard variable.
- How might this be implemented?

Example

```c
int n;

void printer(int k)
{
    n = n + 1;
    k = k + 4;
    printf("%d", n);
    return;
}

int main()
{
    n = 0;
    printer(n);
    printf("%d", n);
}
```

```
call by value:        1 1
call by value-result: 1 4
call by reference:    5 5
```

Example

```
int n;

void printer(int k)
{
    printf("%d", k);
    n++;
    printf("%d", k);
    return;
}

int main()
{
    n = 0;
    printer(n + 10);
}
```

```
call by value:    10 10
call by name:     10 11
```

Implementing Objects

Objects are Hard

- It is difficult to build an *expressive* and *efficient* object-oriented language.
- Certain concepts are difficult to implement efficiently:
  - Dynamic dispatch (virtual functions)
  - Interfaces
  - Multiple Inheritance
  - Dynamic type checking (i.e. **`instanceof`**)

Encoding C-Style **`structs`**

- A **`struct`** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

Encoding C-Style **`structs`**

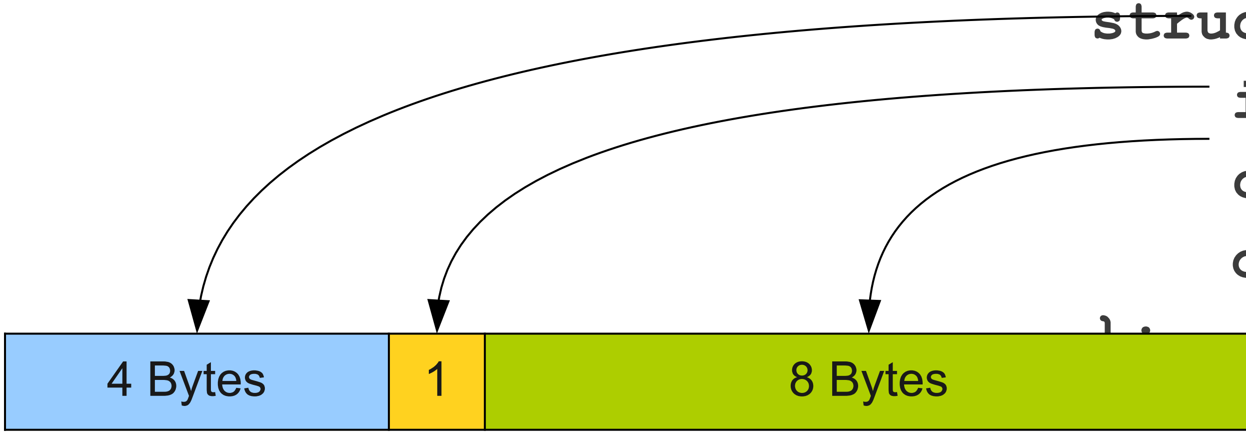
- A **`struct`** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {
    int myInt;
    char myChar;
    double myDouble;
};
```

Encoding C-Style `structs`

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {
    int myInt;
    char myChar;
    double myDouble;
};
```



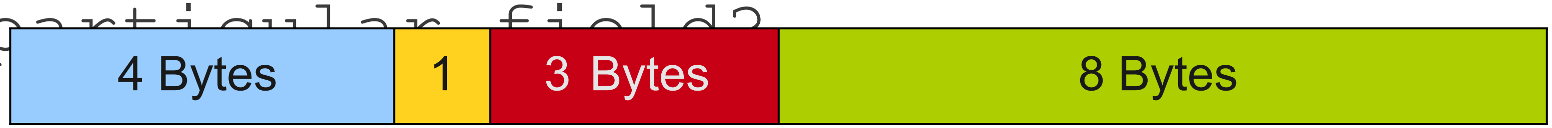
Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {
    int myInt;
    char myChar;
    double myDouble;
};
```

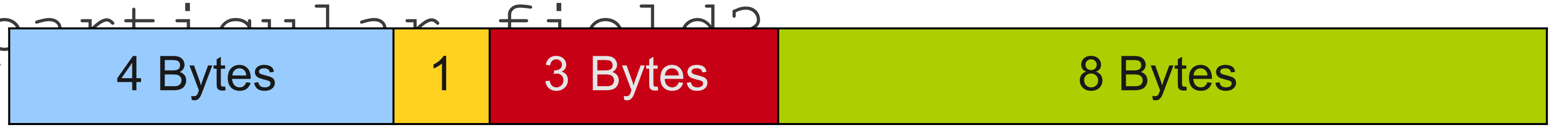
Accessing Fields

- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?



Accessing Fields

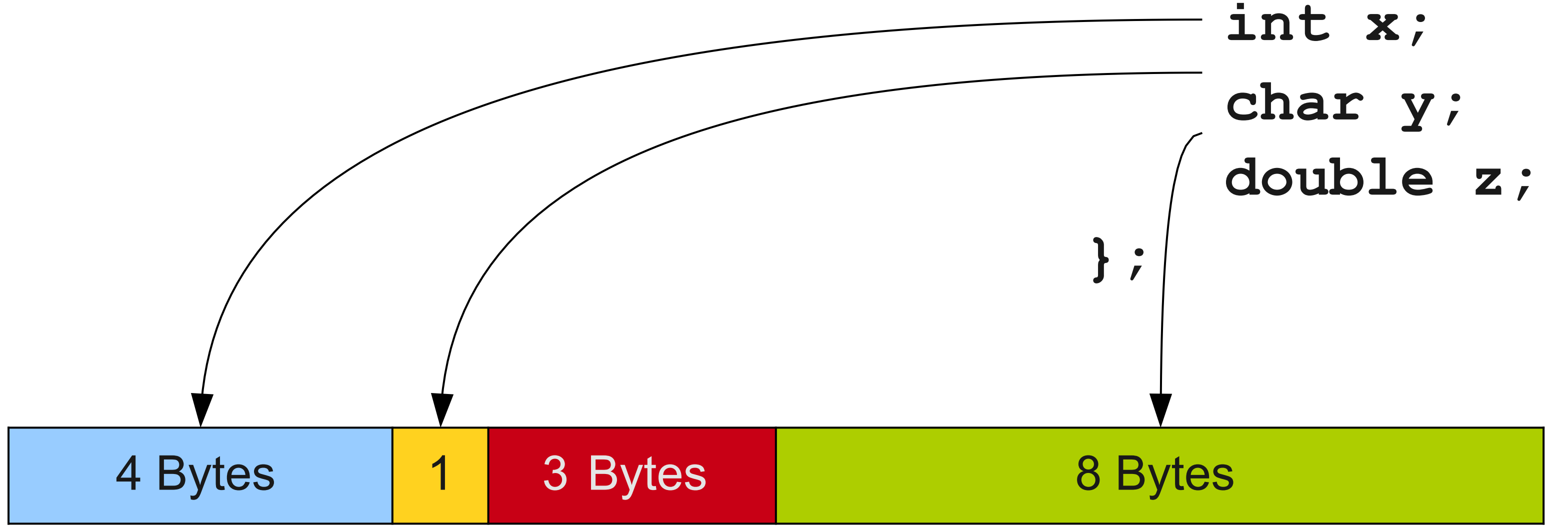
- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?



- Idea: Keep an internal table inside the compiler containing the offsets of each field.
- To look up a field, start at the base address of the object and advance forward by the appropriate offset.

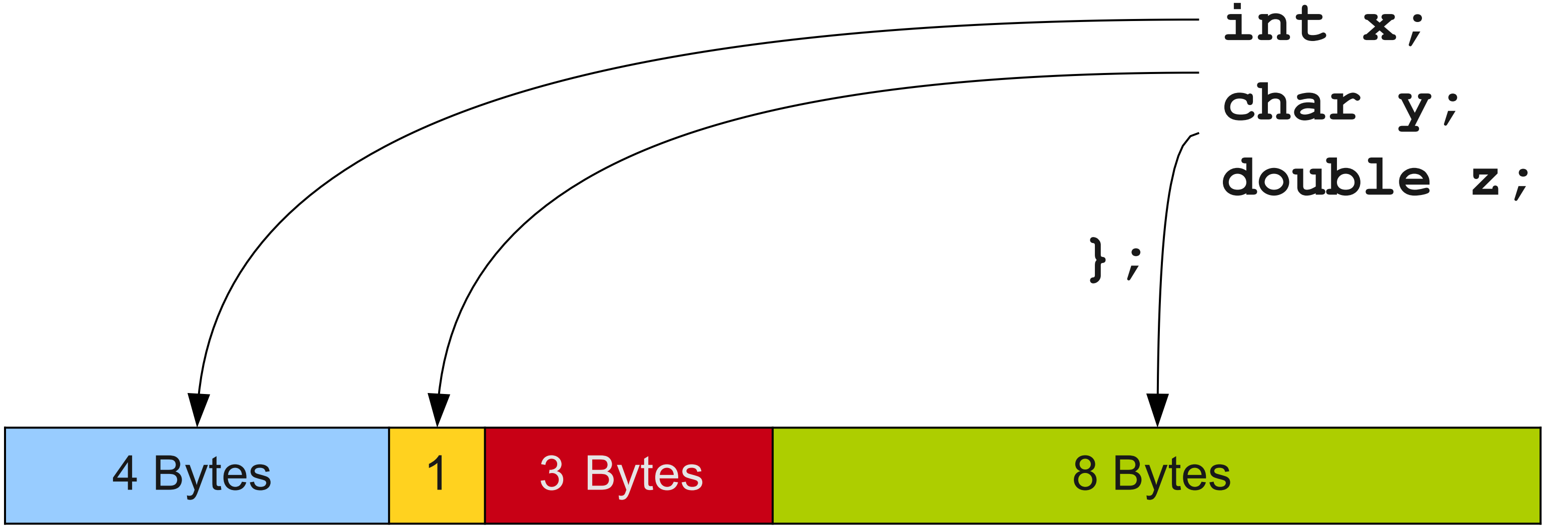
Field Lookup

```
struct MyStruct {
    int x;
    char y;
    double z;
};
```



Field Lookup

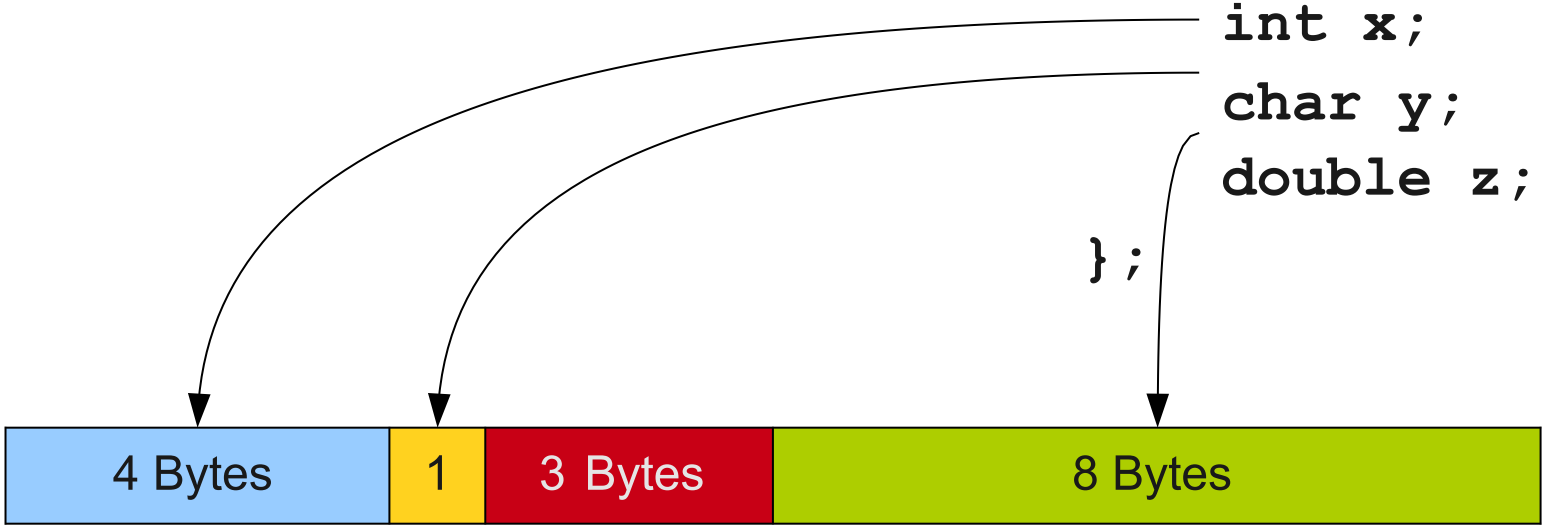
```
struct MyStruct {
    int x;
    char y;
    double z;
};
```




```
MyStruct* ms = new MyStruct;
ms->x = 137;
ms->y = 'A';
ms->z = 2.71
```

Field Lookup

```
struct MyStruct {
    int x;
    char y;
    double z;
};
```




```
MyStruct* ms = new MyStruct;
ms->x = 137;    store 137  0 bytes after ms
ms->y = 'A';    store 'A'  4 bytes after ms
ms->z = 2.71    store 2.71 8 bytes after ms
```

OOP without Methods

- Consider the following Decaf code:

```
class Base {
    int x;

    int y;

}
class Derived extends Base {

    int z;

}
```

- What will **`Derived`** look like in memory?

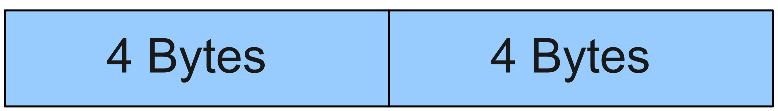
Memory Layouts with Inheritance

Memory Layouts with Inheritance

```
class Base {
    int x;
    int y;
};
```

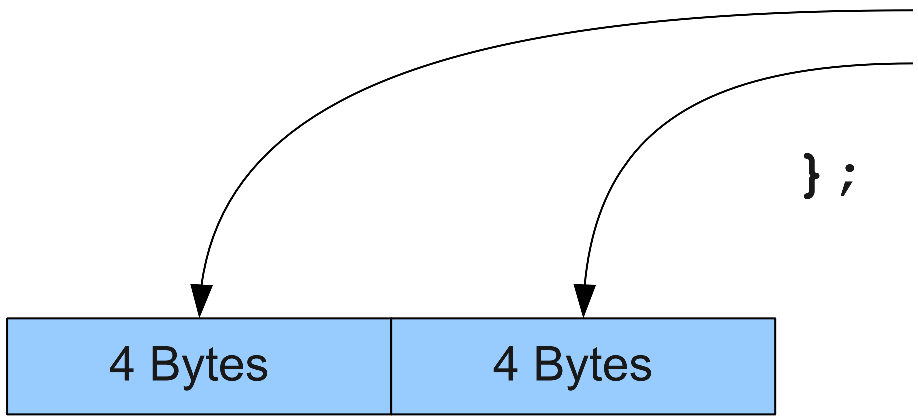
Memory Layouts with Inheritance

```
class Base {
    int x;
    int y;
};
```



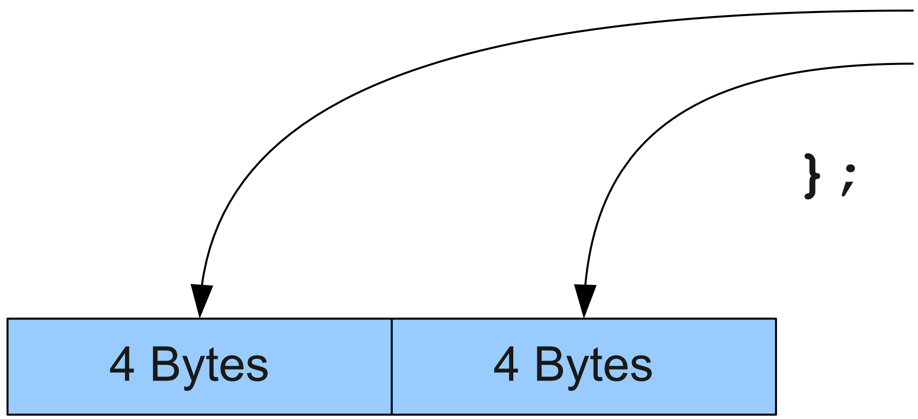
Memory Layouts with Inheritance

```
class Base {
    int x;
    int y;
};
```



Memory Layouts with Inheritance

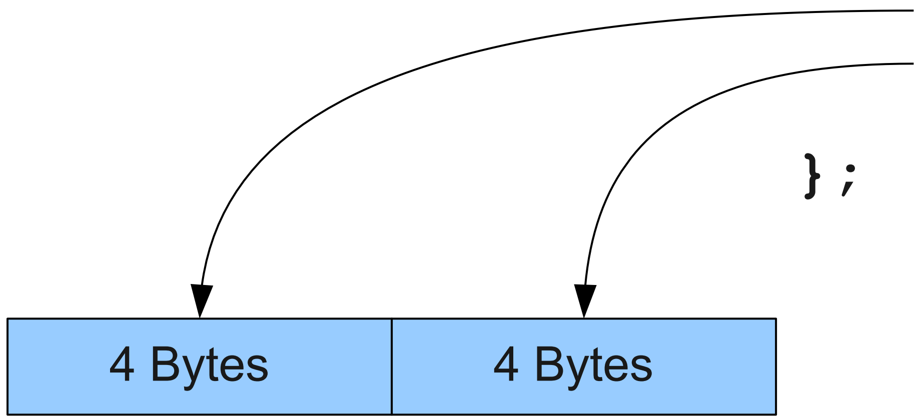
```
class Base {
    int x;
    int y;
};
```

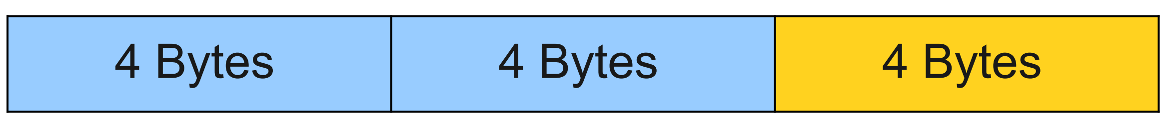



```
class Derived extends Base {
    int z;
};
```

Memory Layouts with Inheritance

```
class Base {
    int x;
    int y;
};
```

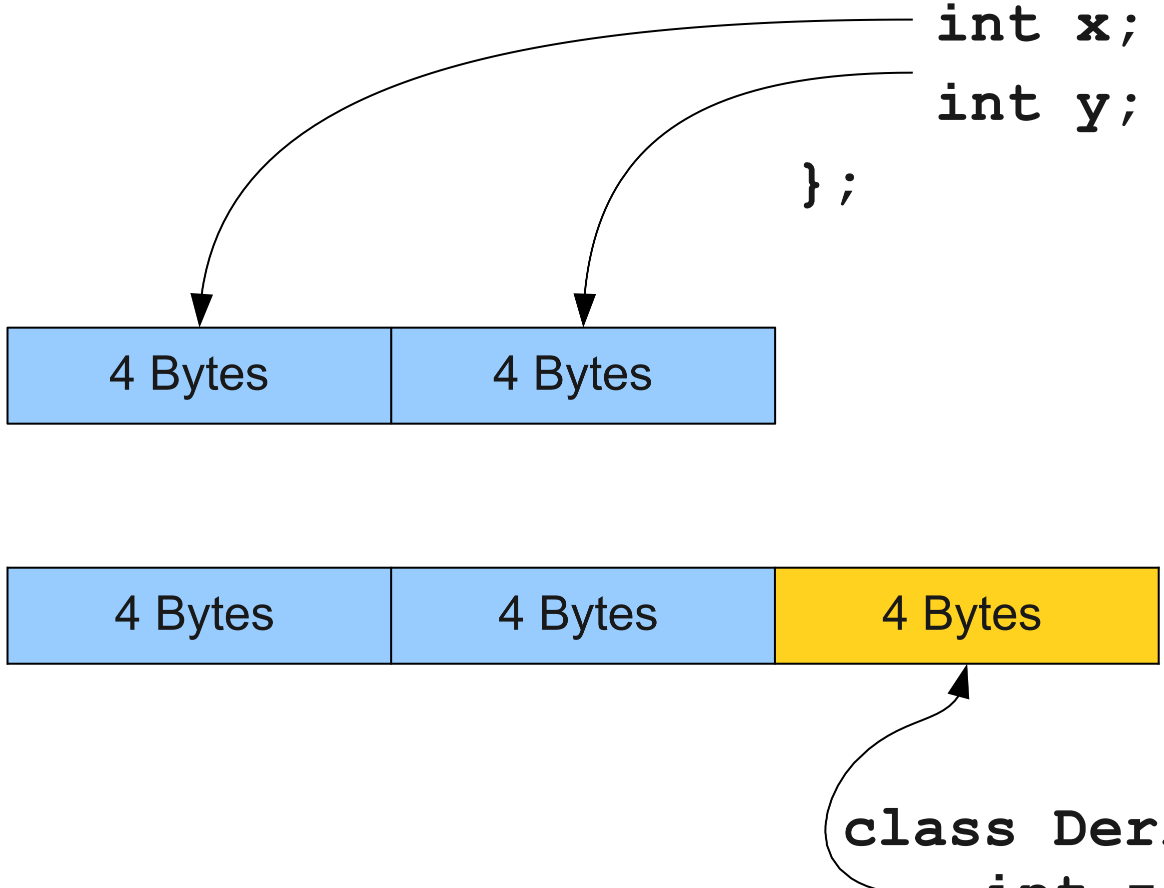





```
class Derived extends Base {
    int z;
};
```

Memory Layouts with Inheritance

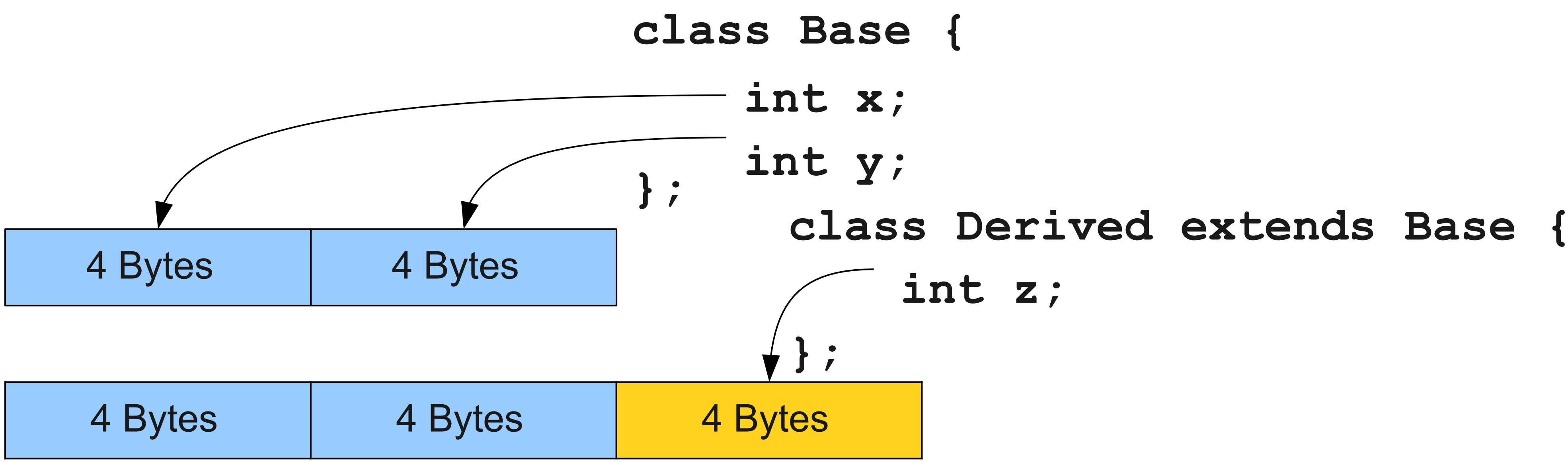
```
class Base {
    int x;
    int y;
};
```



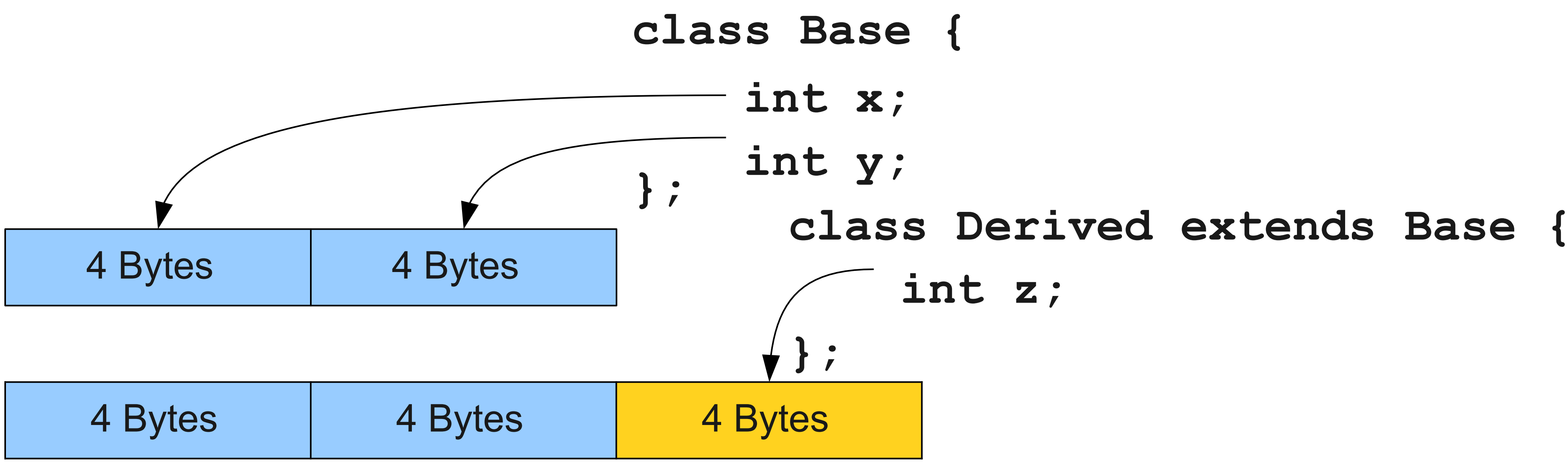

```
class Derived extends Base {
    int z;
};
```

Field Lookup With Inheritance

Field Lookup With Inheritance

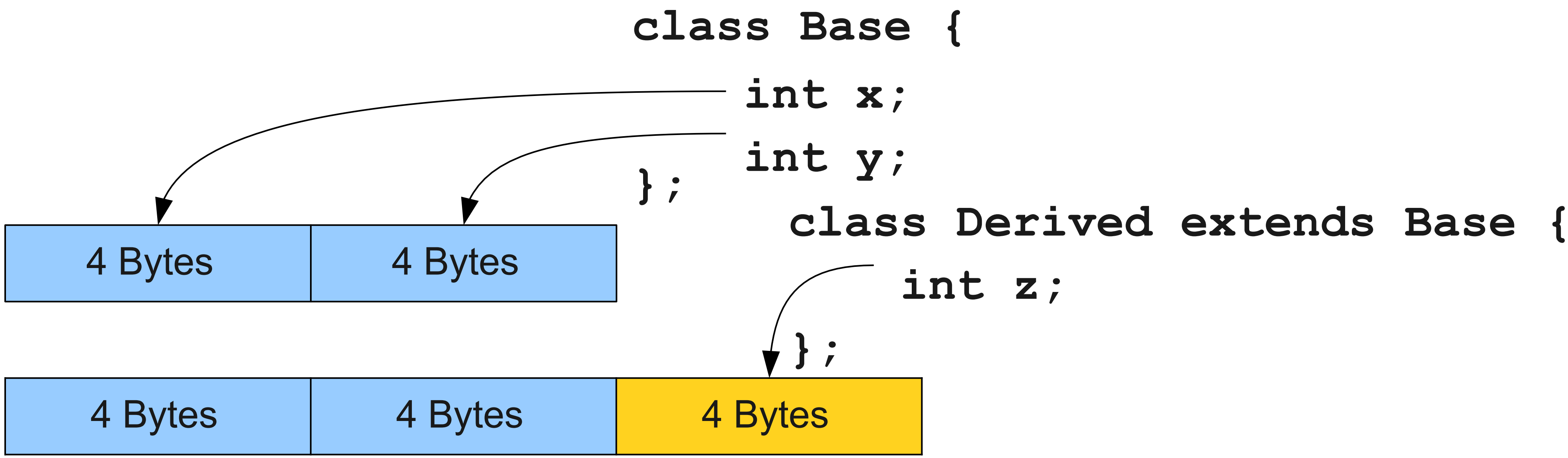


Field Lookup With Inheritance




```
Base ms = new Base;
ms.x = 137;
ms.y = 42;
```

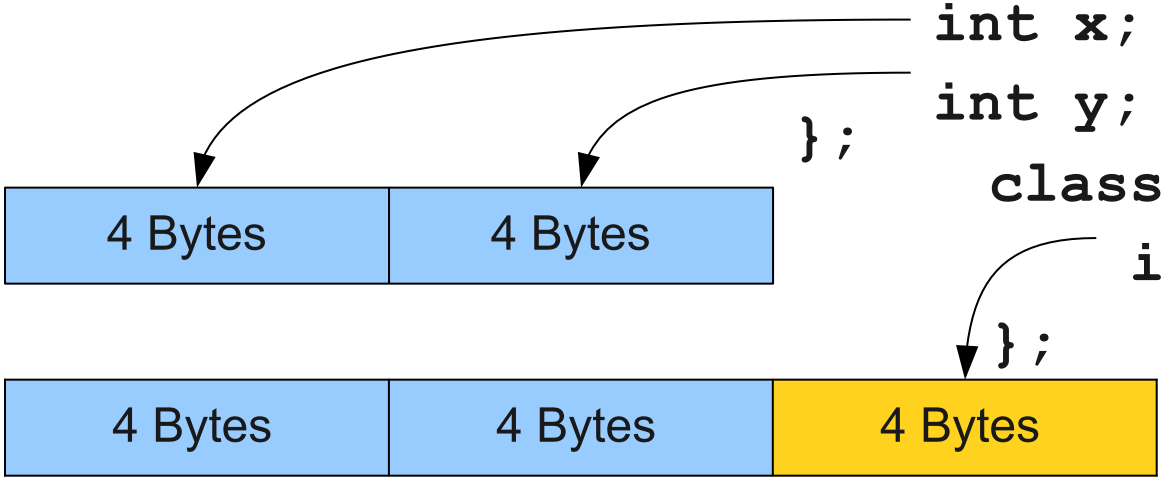
Field Lookup With Inheritance




```
Base ms = new Base;
ms.x = 137;     store 137 0 bytes after ms
ms.y = 42;      store 42  4 bytes after ms
```

Field Lookup With Inheritance

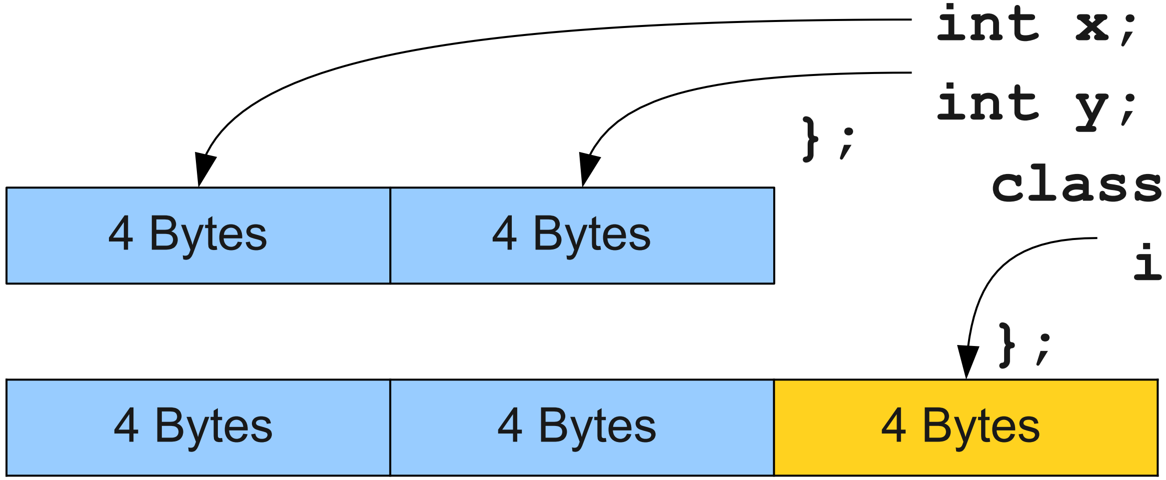
```
class Base {
    int x;
    int y;
};
class Derived extends Base {
    int z;
};
```




```
Base ms = new Derived;
ms.x = 137;
ms.y = 42;
```

Field Lookup With Inheritance

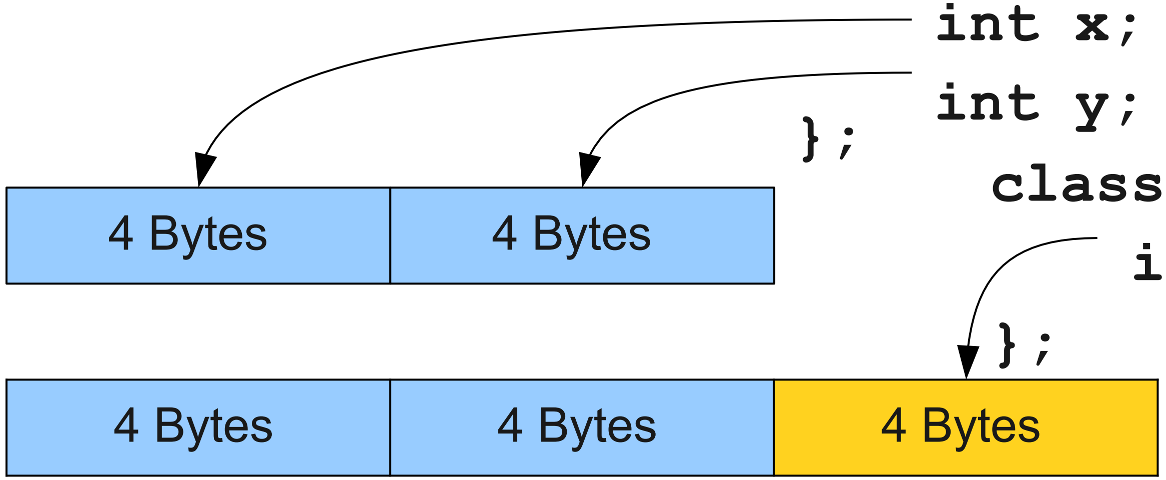
```
class Base {
    int x;
    int y;
};
    class Derived extends Base {
        int z;
    };
```




```
Base ms = new Derived;
ms.x = 137;
ms.y = 42;
```

Field Lookup With Inheritance

```
class Base {
    int x;
    int y;
};
    class Derived extends Base {
        int z;
    };
```



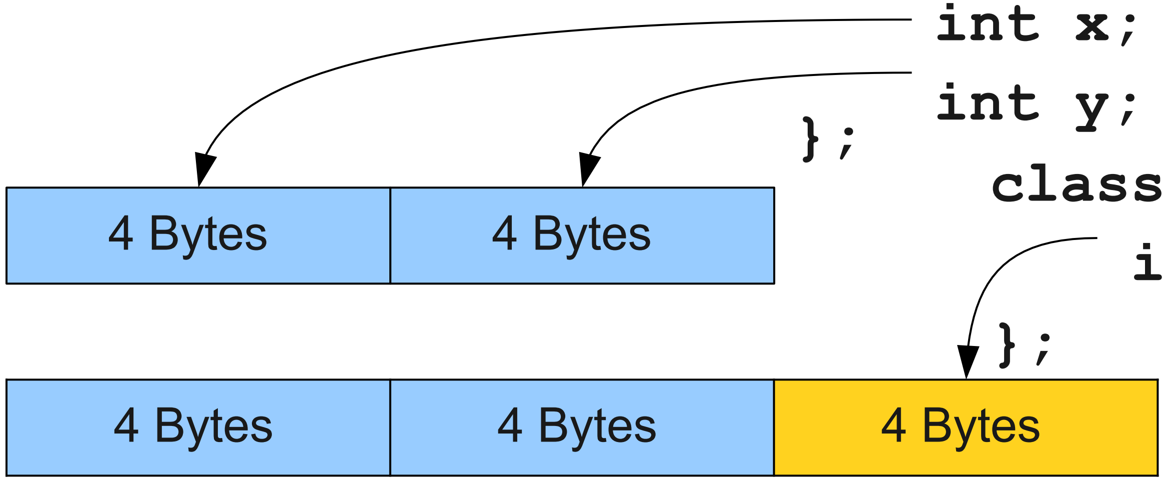

```
Base ms = new Derived;
ms.x = 137;
ms.y = 42;
```

store 137 0 bytes after ms

store 42 4 bytes after ms

Field Lookup With Inheritance

```
class Base {
    int x;
    int y;
};
    class Derived extends Base {
        int z;
    };
```




```
Base ms = new Base;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms

Base ms = new Derived;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms
```

Single Inheritance in Decaf

- The memory layout for a class D that extends B is given by the memory layout for B followed by the memory layout for the members of D.
  - Actually a bit more complex; we'll see why later.
- Rationale: A pointer of type B pointing at a D object still sees the B object at the beginning.
- Operations done on a D object through the B reference guaranteed to be safe; no need to check what B points at dynamically.

Summary of Function Calls

- The runtime stack is an optimization of the activation tree spaghetti stack.
- Most languages use a runtime stack, though certain language features prohibit this optimization.
- Activation records logically store a **control link** to the calling function and an **access link** to the function in which it was created.
- Decaf has the caller manage space for parameters and the callee manage space for its locals and temporaries.
- Call-by-value and call-by-name can be implemented using copying and pointers.
- More advanced parameter passing schemes exist!

Next Time

- **Implementing Objects**
  - Standard object layouts.
  - Objects with inheritance.
  - Implementing dynamic dispatch.
  - Implementing interfaces.
  - … and doing so efficiently!