# Introduction to LLVM IR and Passes

Collected by rainoftime
pyaoaa@zju.edu.cn

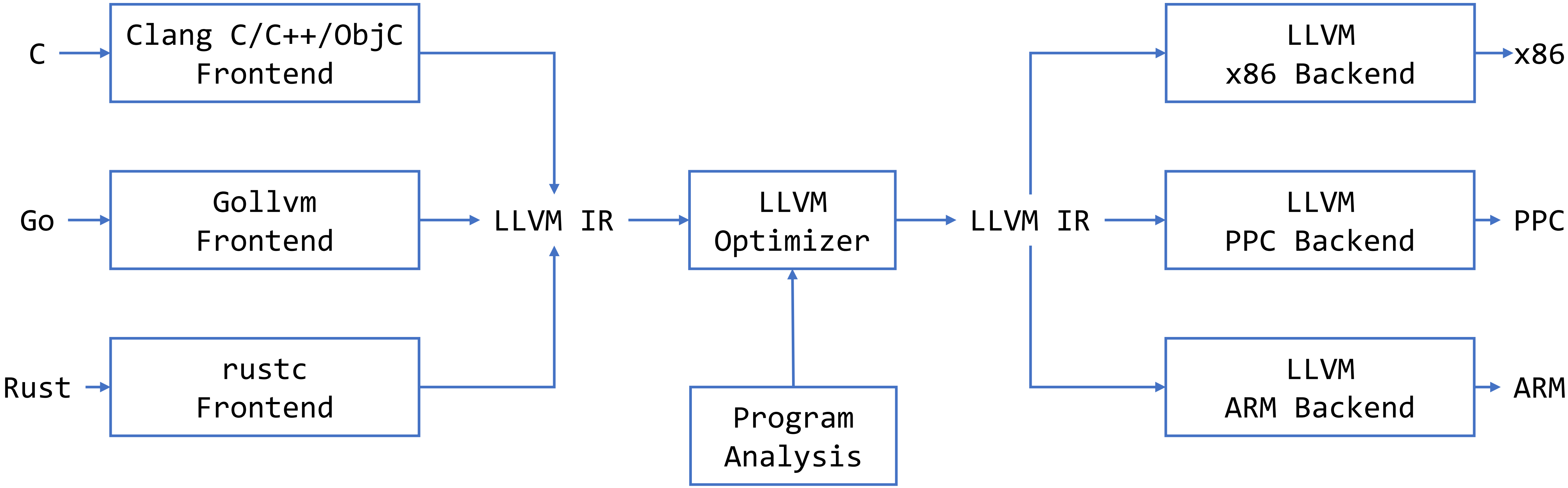
# Outline

- Introduction to LLVM IR
- Writing LLVM Passes

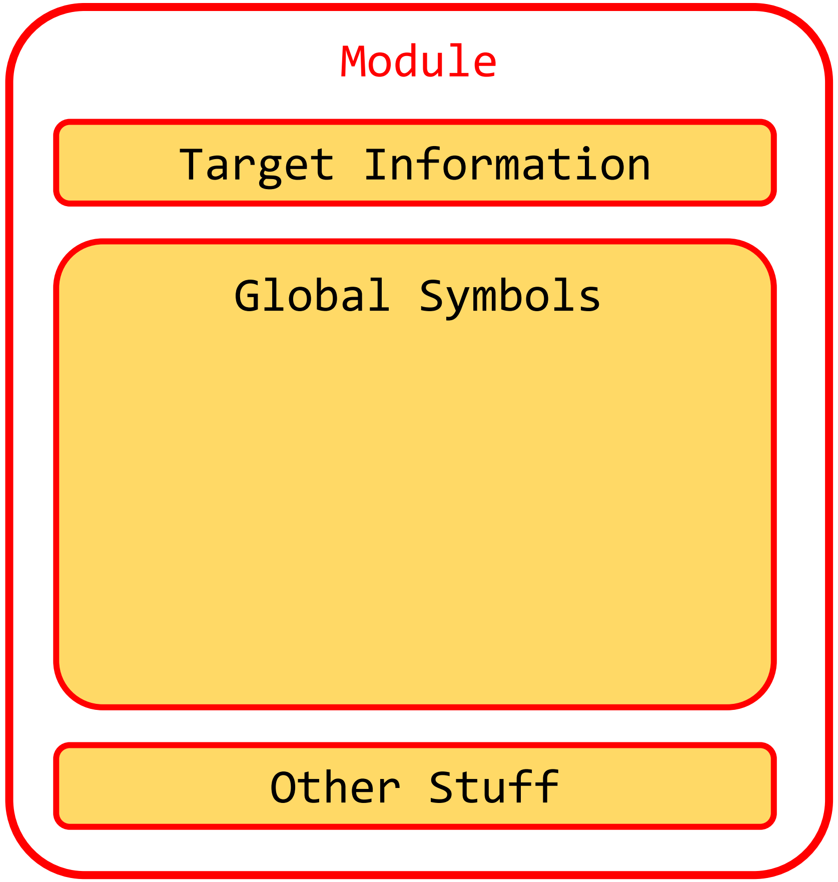
# What is LLVM IR

- The **LLVM I**ntermediate **R**epresentation:
  - **is a low level programming language**
    - **RISC-like instruction set**
  - **providing type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly**
    - **high-level languages can map to IR cleanly**
  - **is used throughout all phases of the LLVM compilation strategy**
    - **enabling efficient code optimization**

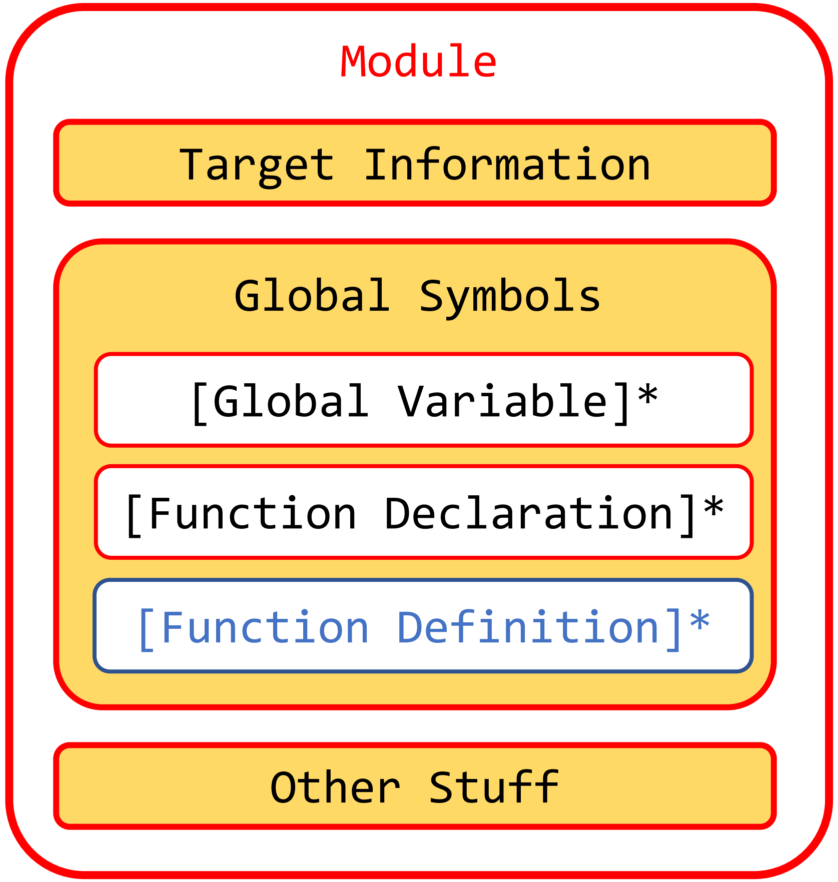
# IR & the Compilation Process



# Simplified IR Layout



# Simplified IR Layout



Module

Target Information

Global Symbols

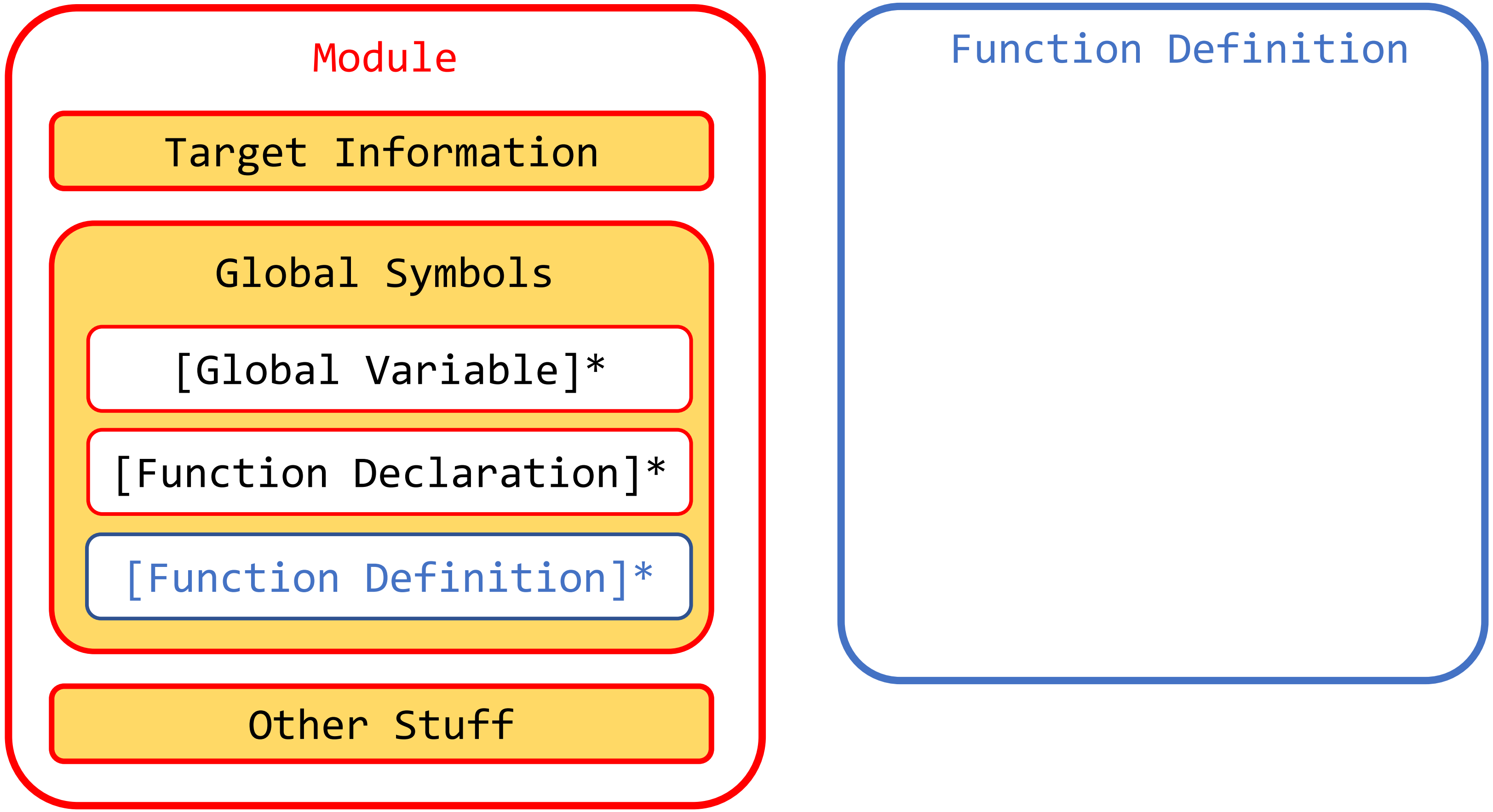
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

# Simplified IR Layout



Module

Target Information

Global Symbols

[Global Variable]*

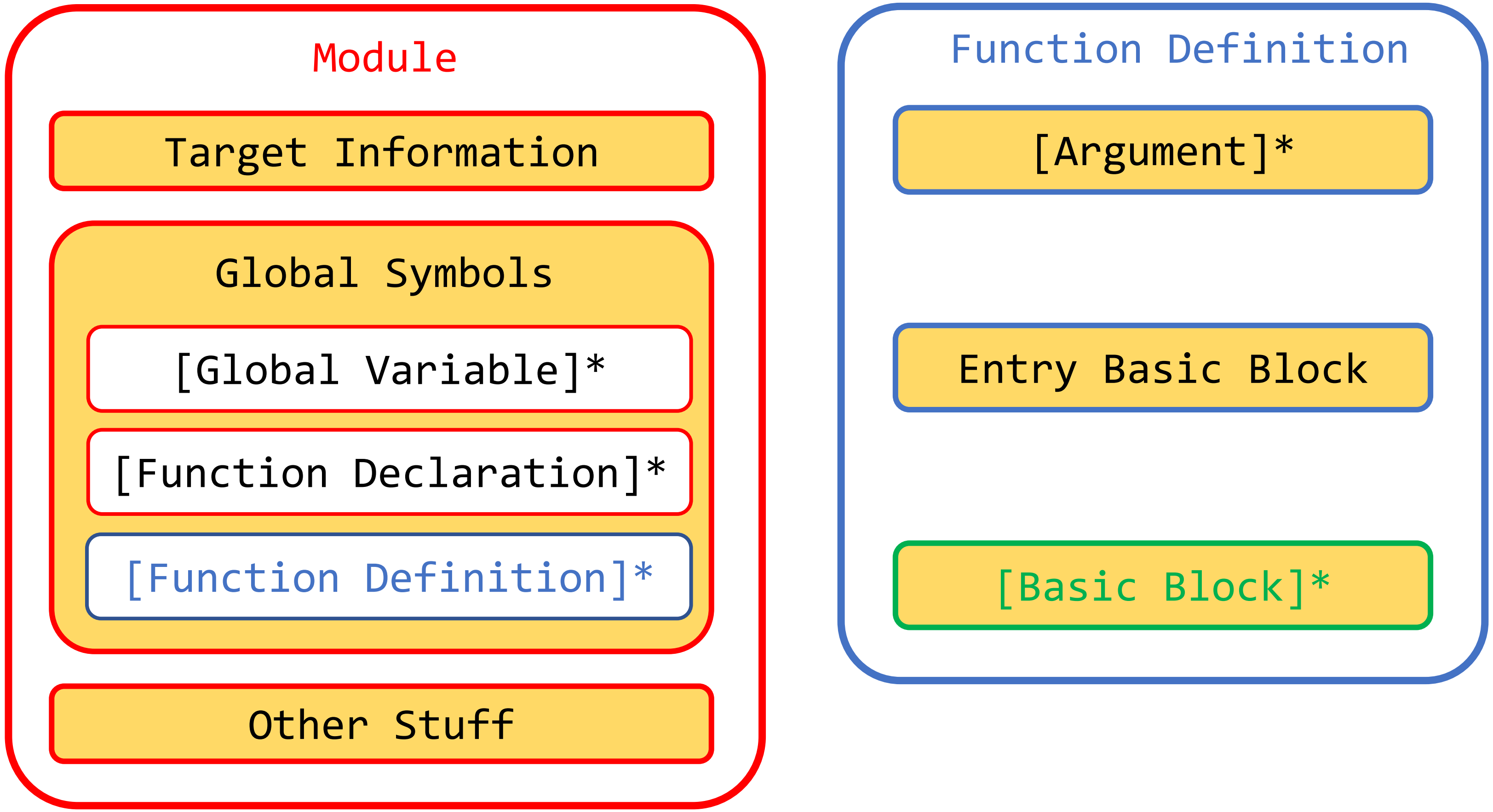
[Function Declaration]*

[Function Definition]*

Other Stuff

Function Definition

# Simplified IR Layout



Module

Target Information

Global Symbols

[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

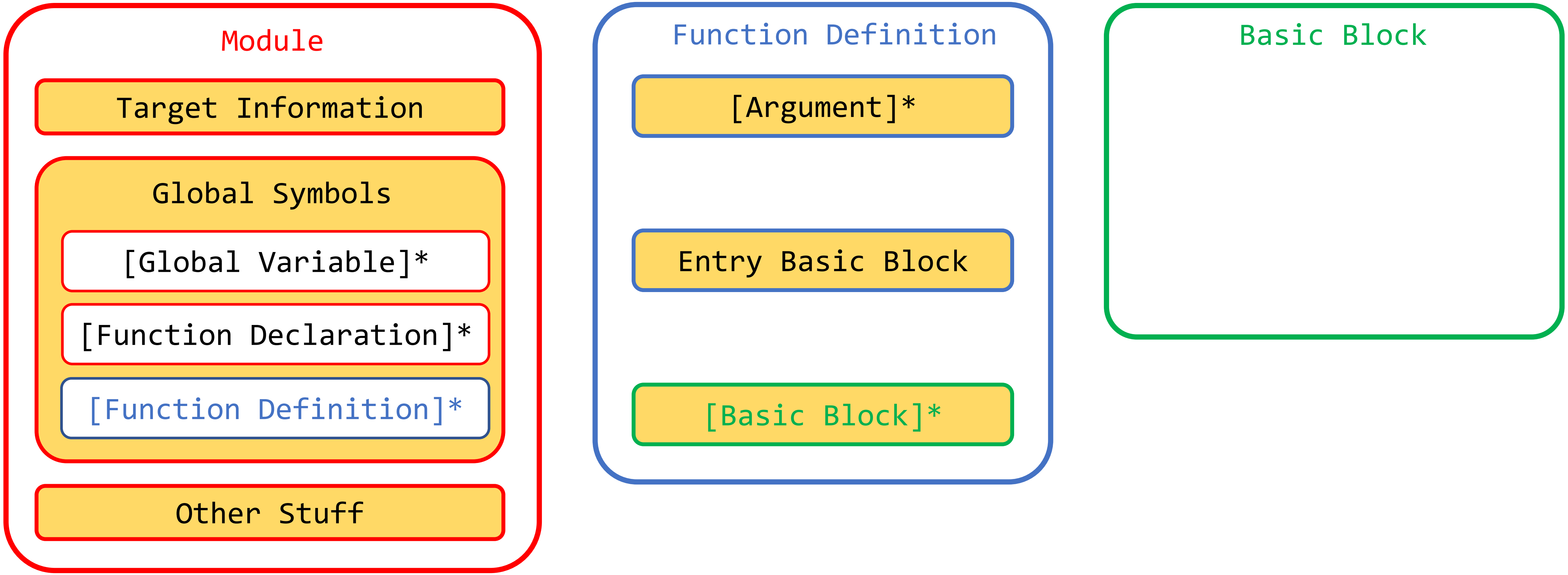
Function Definition

[Argument]*

Entry Basic Block

[Basic Block]*

# Simplified IR Layout



Module

Target Information

Global Symbols

[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

Function Definition

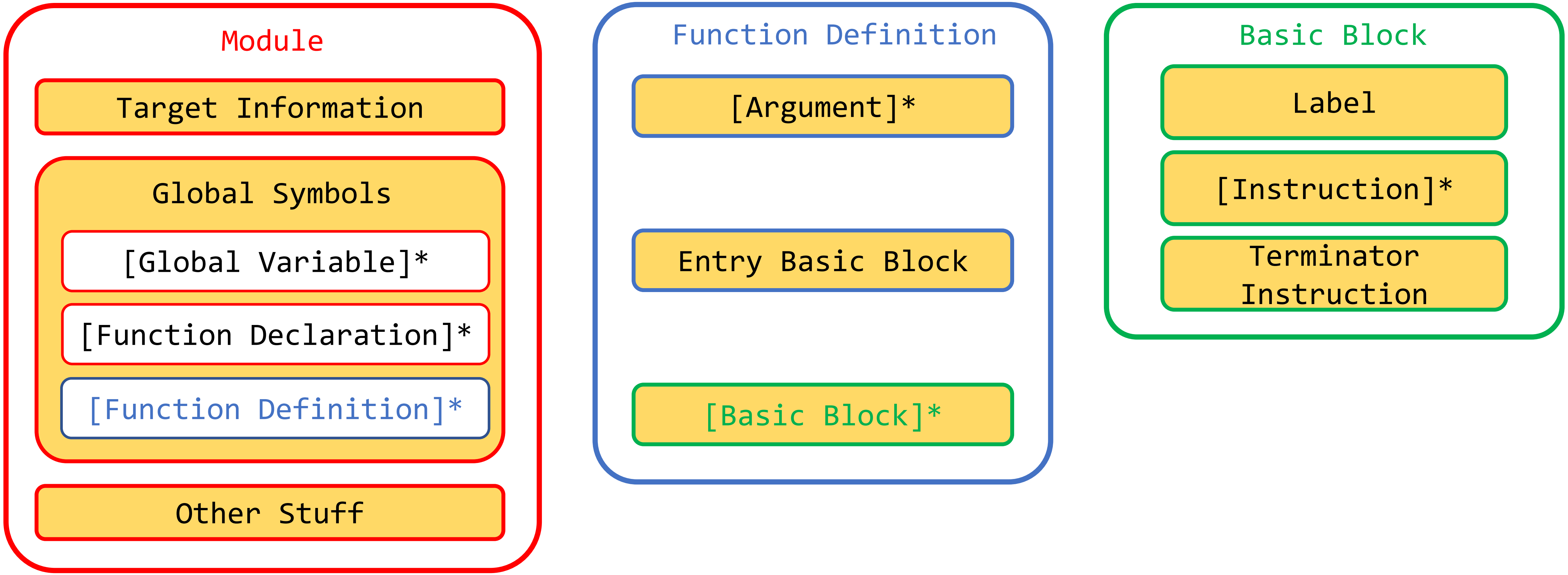
[Argument]*

Entry Basic Block

[Basic Block]*

Basic Block

# Simplified IR Layout



Module

Target Information

Global Symbols

[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

Function Definition

[Argument]*

Entry Basic Block

[Basic Block]*

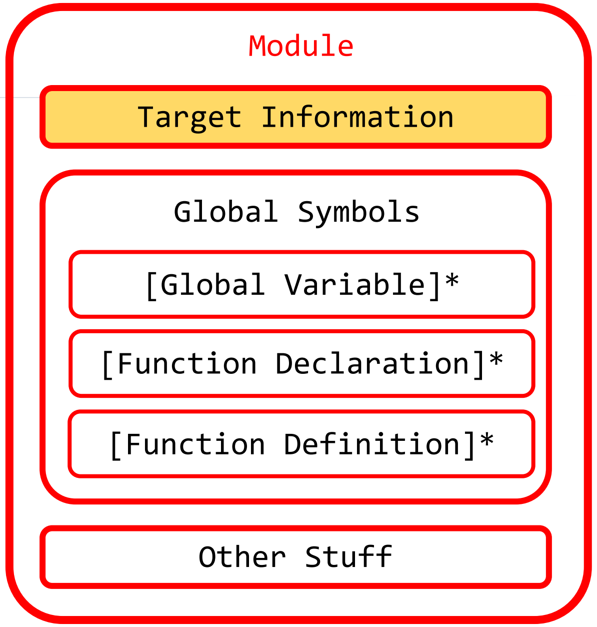
Basic Block

Label

[Instruction]*

Terminator Instruction

# Target Information

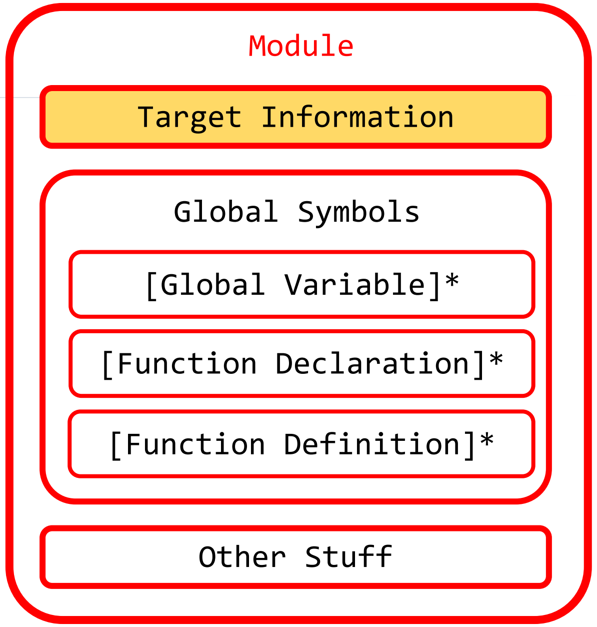


A module may specify a target specific data layout string that specifies how data is to be laid out in memory:

Little endian

```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
```

# Target Information



Module

Target Information

Global Symbols

[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

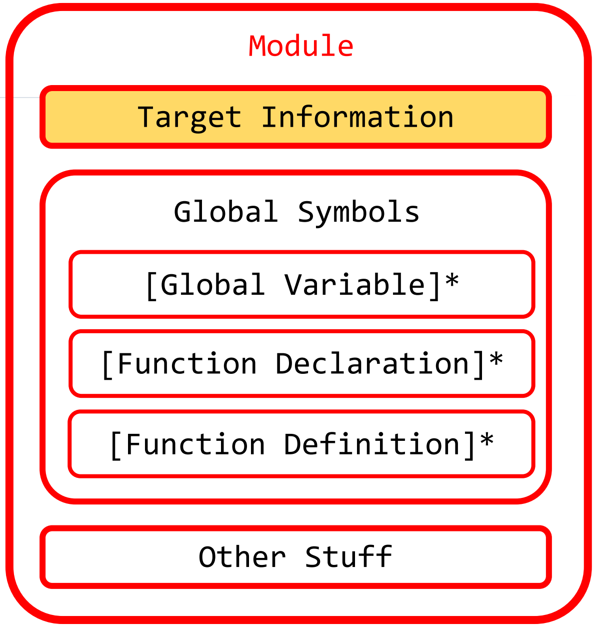
A module may specify a target specific data layout string that specifies how data is to be laid out in memory:

Pointer
size & alignment

Little endian

```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
```

# Target Information



Module

Target Information

Global Symbols

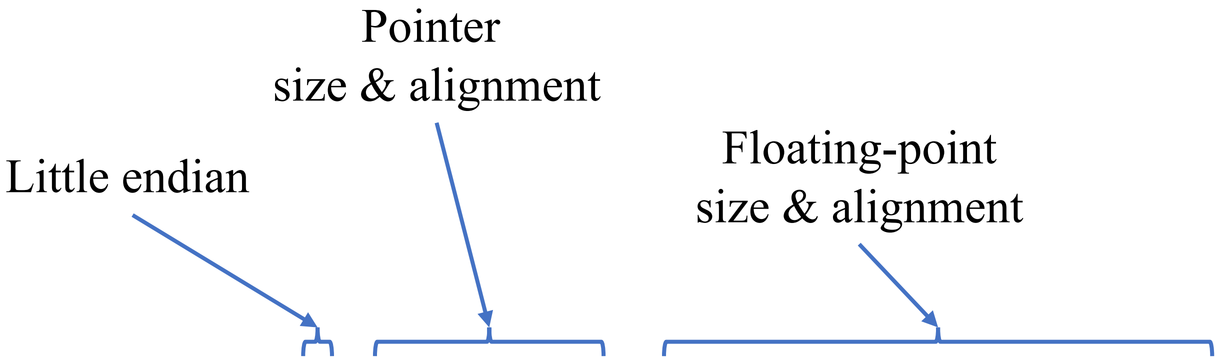
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

A module may specify a target specific data layout string that specifies how data is to be laid out in memory:



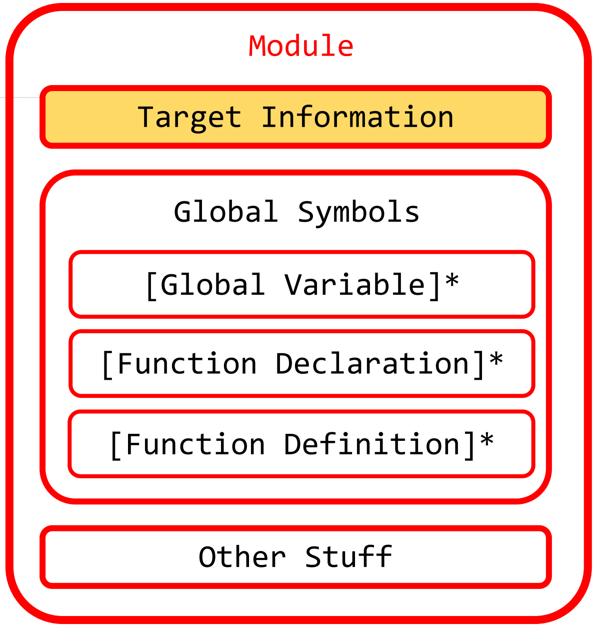
Pointer
size & alignment

Little endian

Floating-point
size & alignment

```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
```

# Target Information



Module

Target Information

Global Symbols

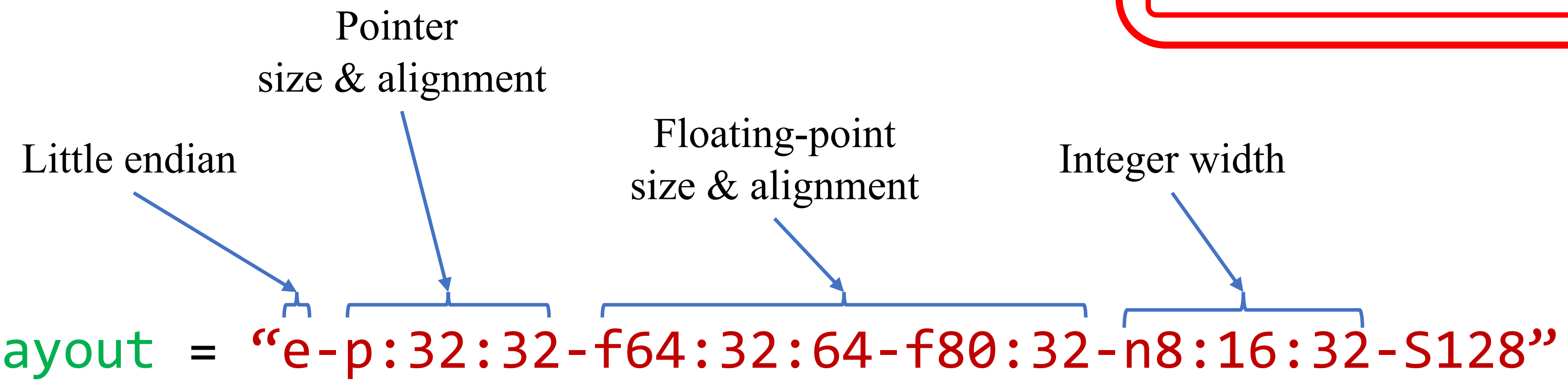
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

A module may specify a target specific data layout string that specifies how data is to be laid out in memory:



Pointer size & alignment

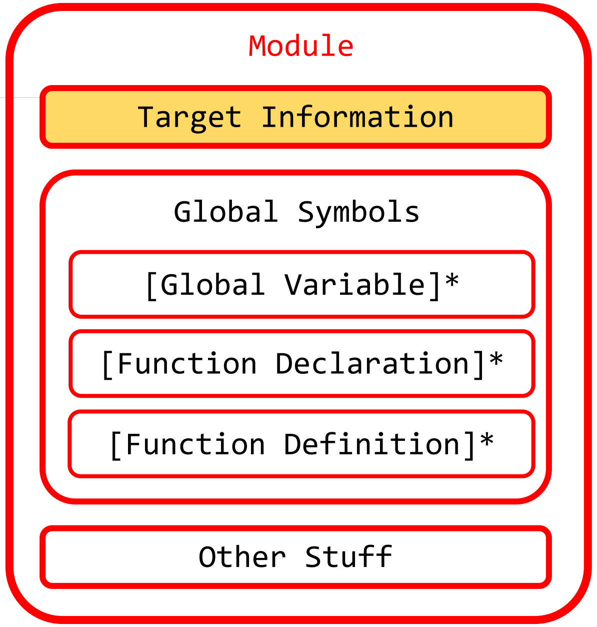
Little endian

Floating-point size & alignment

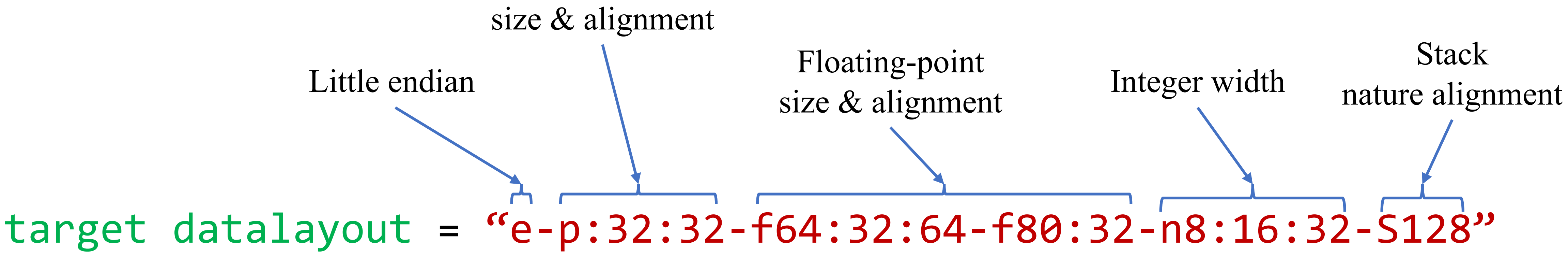
Integer width

```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
```

# Target Information

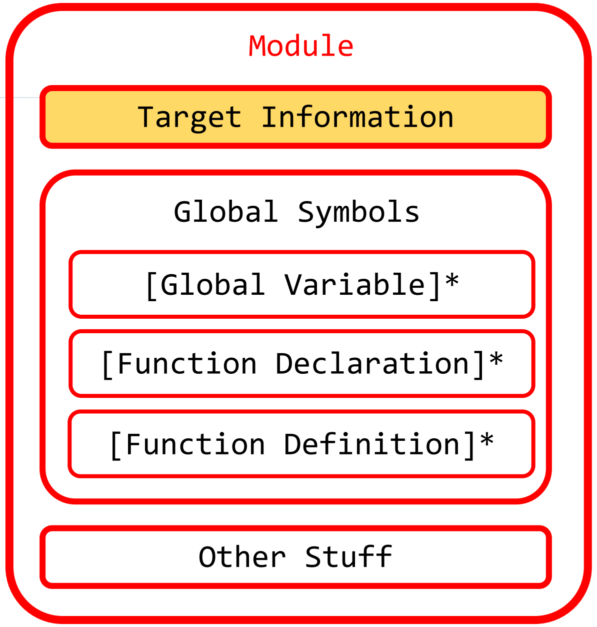


A module may specify a target specific data layout string that specifies how data is to be laid out in memory:



```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
```

# Target Information



Module

Target Information

Global Symbols

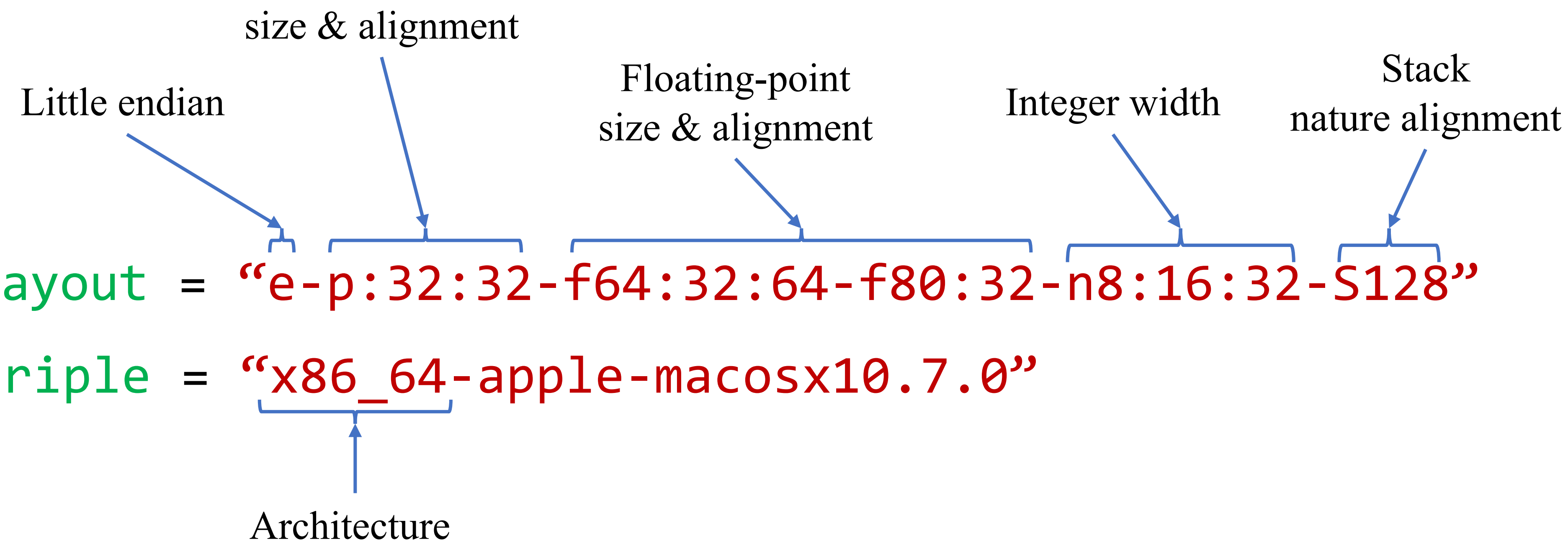
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

A module may also specify a target triple string that describes the target host:



Pointer size & alignment

Little endian

Floating-point size & alignment

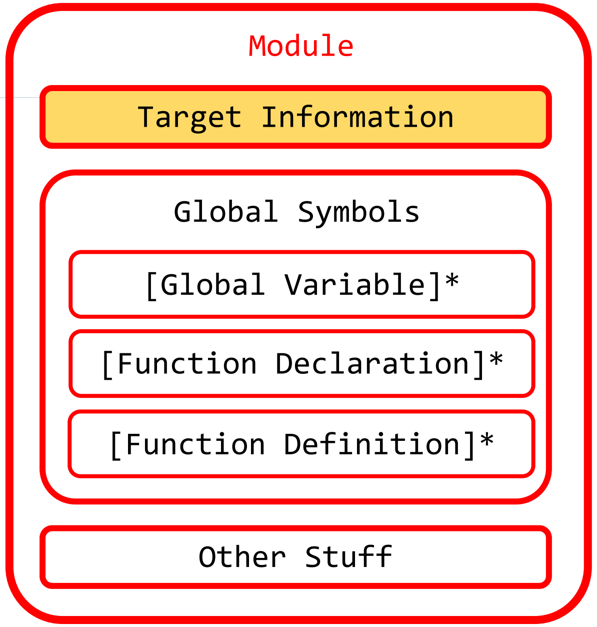
Integer width

Stack nature alignment

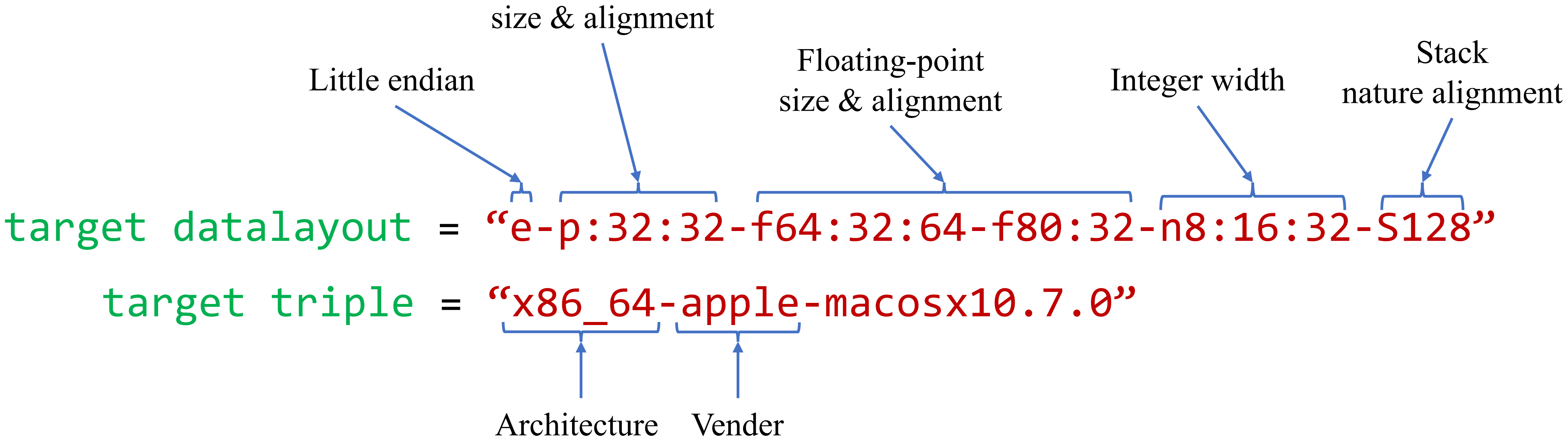
```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
     target triple = “x86_64-apple-macosx10.7.0”
```

Architecture

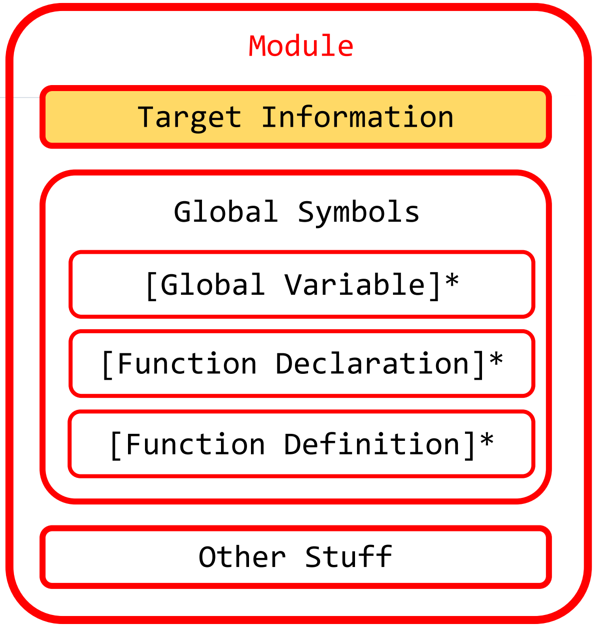
# Target Information



A module may also specify a target triple string that describes the target host:


```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
     target triple = “x86_64-apple-macosx10.7.0”
```

# Target Information



Module

Target Information

Global Symbols

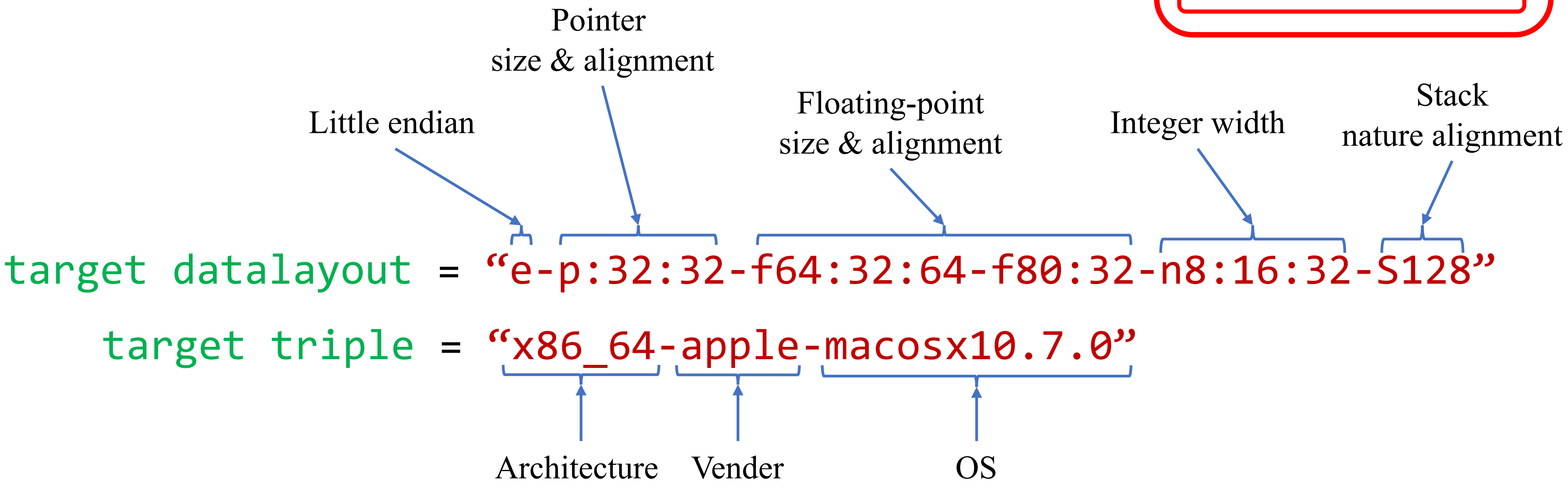
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

A module may also specify a target triple string that describes the target host:



```
target datalayout = “e-p:32:32-f64:32:64-f80:32-n8:16:32-S128”
     target triple = “x86_64-apple-macosx10.7.0”
```

Little endian

Pointer size & alignment

Floating-point size & alignment

Integer width

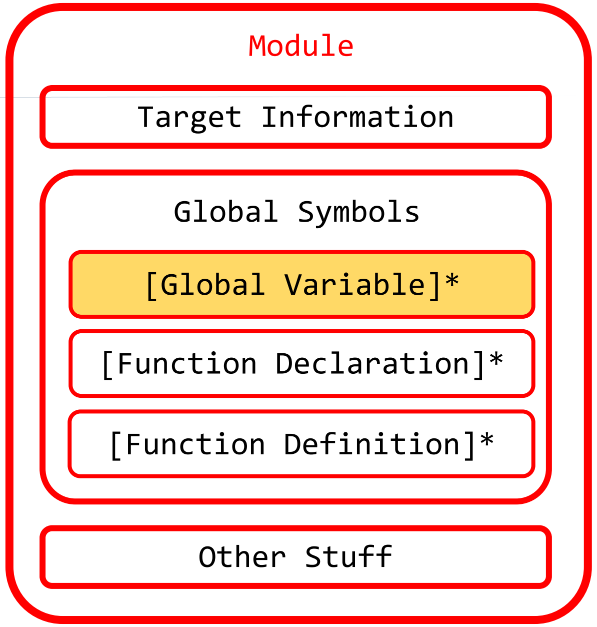
Stack nature alignment

Architecture

Vender

OS

# Global Variables



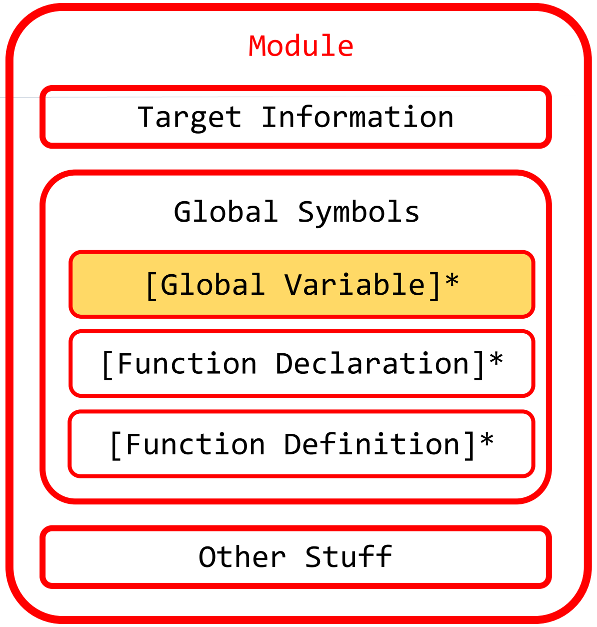
Global variables define regions of memory allocated at compilation time instead of run-time.

# Global Variables

Global variables define regions of memory allocated at compilation time instead of run-time.

- Name prefixed with “@”

```
@gv =
```



Module

Target Information

Global Symbols

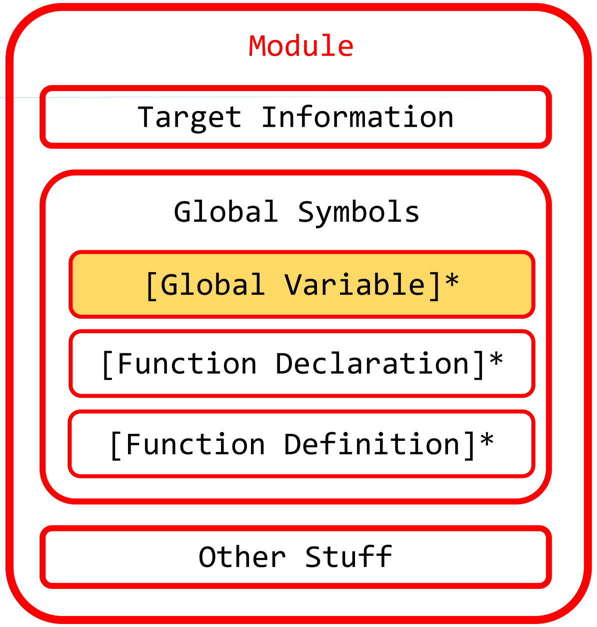
[Global Variable]*

[Function Declaration]*

[Function Definition]*

Other Stuff

# Global Variables

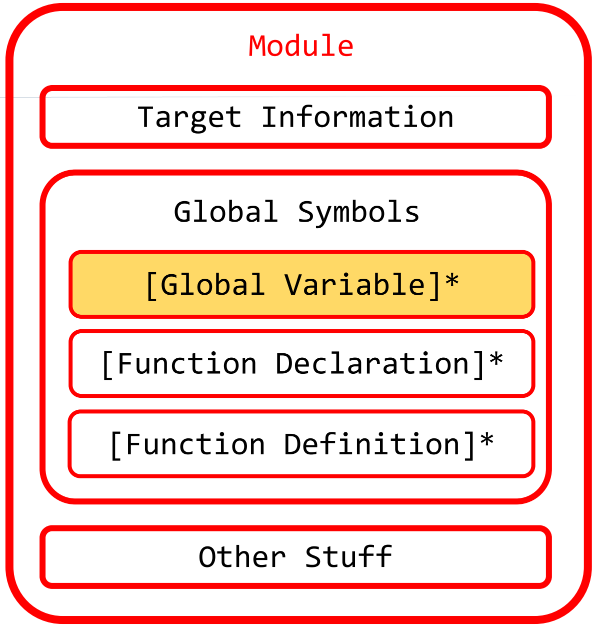


Global variables define regions of memory allocated at compilation time instead of run-time.

- Name prefixed with “@”
- Have the keyword global
    - xor constant

```
@gv =
@gv = global
@gv = constant
```

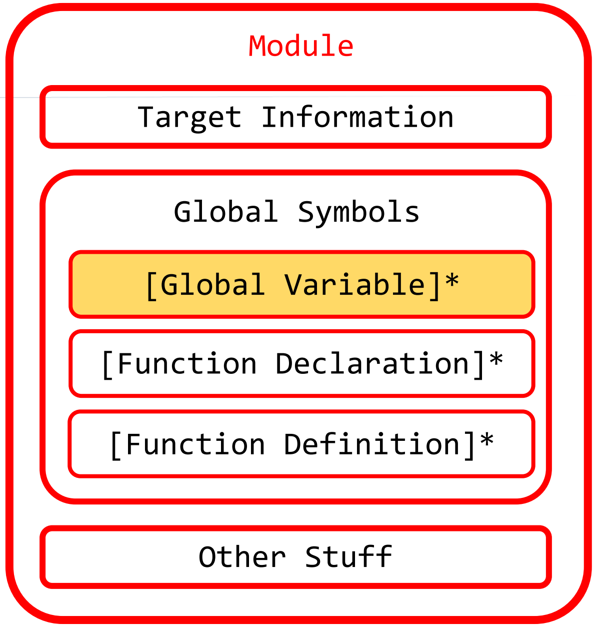
# Global Variables



Global variables define regions of memory allocated at compilation time instead of run-time.

- Name prefixed with "@" `@gv =`
- Have the keyword global `@gv = global`
  - xor constant `@gv = constant`
- Must have a type `@gv = global i32`

# Global Variables

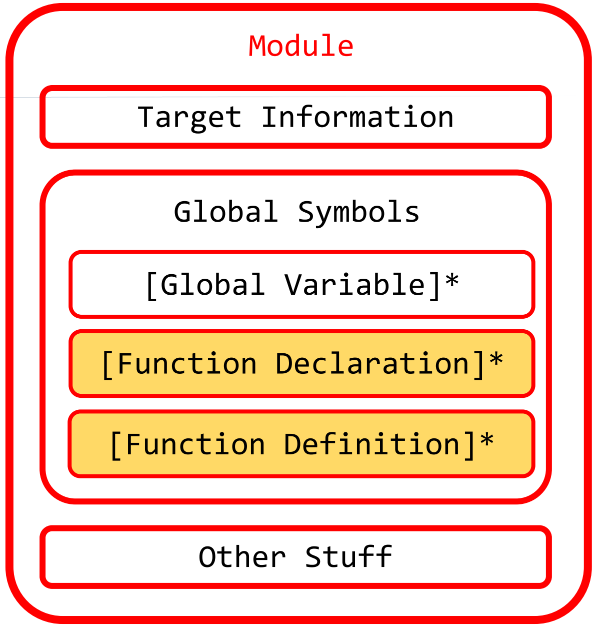


Global variables define regions of memory allocated at compilation time instead of run-time.

- Name prefixed with “@”
- Have the keyword global
  - xor constant
- Must have a type
- Must be initialized

```
@gv =
@gv = global
@gv = constant
@gv = global i32
@gv = global i32 0xdeadbeef
```

# Functions



Module

Target Information

Global Symbols

[Global Variable]*

[Function Declaration]*

[Function Definition]*

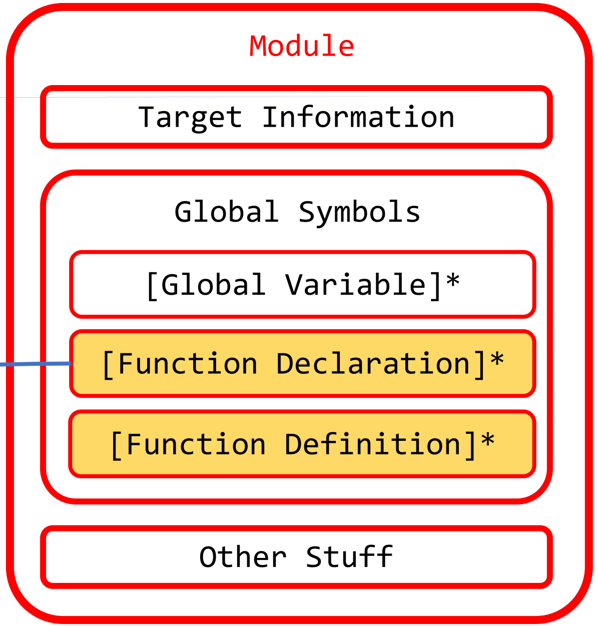
Other Stuff

## An example to compute factorial

```
int factorial(int val);

int main(){
    return factorial(5) * 6
        == factorial(6);
}
```

```
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

# Functions



## An example to compute factorial

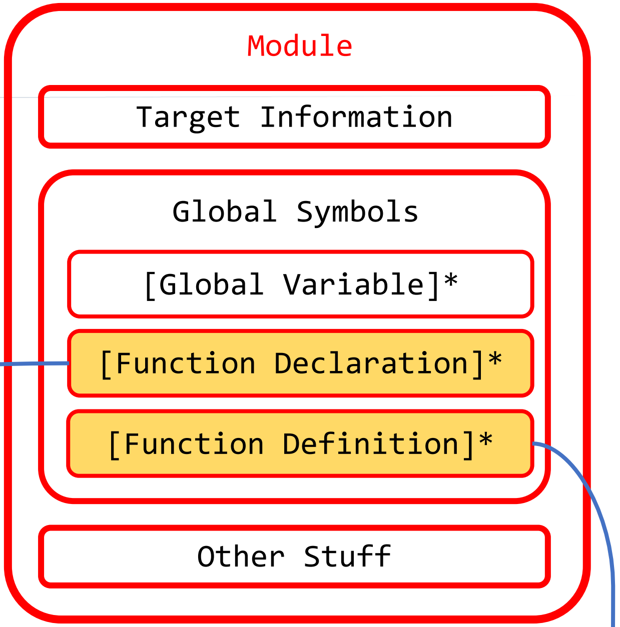
```c
int factorial(int val);

int main(){
    return factorial(5) * 6
        == factorial(6);
}
```

```llvm
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

# Functions

## An example to compute factorial

```c
int factorial(int val);

int main(){
    return factorial(5) * 6
        == factorial(6);
}
```



Module

Target Information

Global Symbols

[Global Variable]*

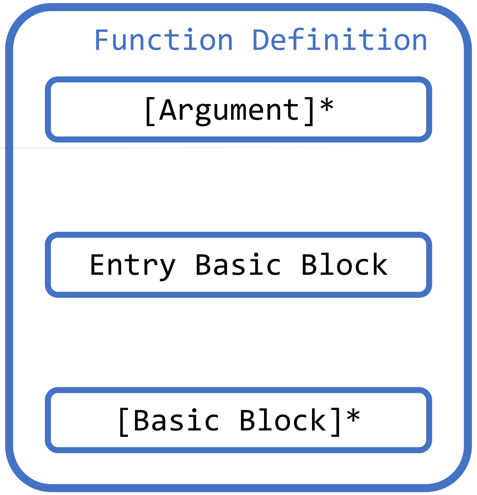
[Function Declaration]*

[Function Definition]*

Other Stuff

```llvm
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

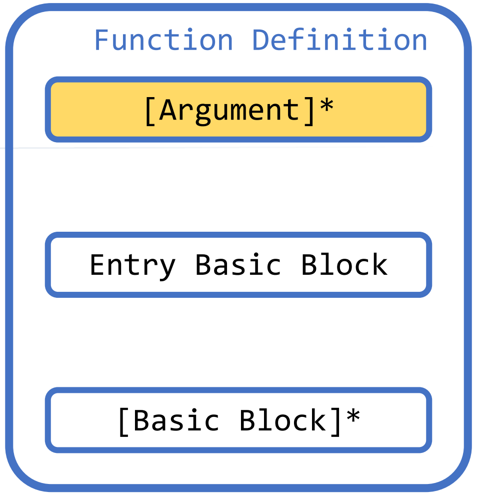
# Functions: Recursive Factorial



```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

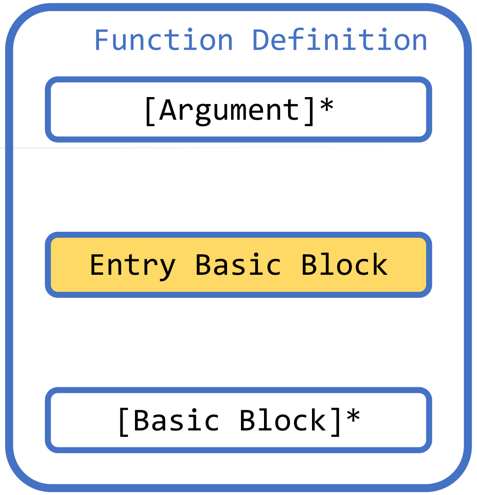
# Functions: Recursive Factorial



```c
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```llvm
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

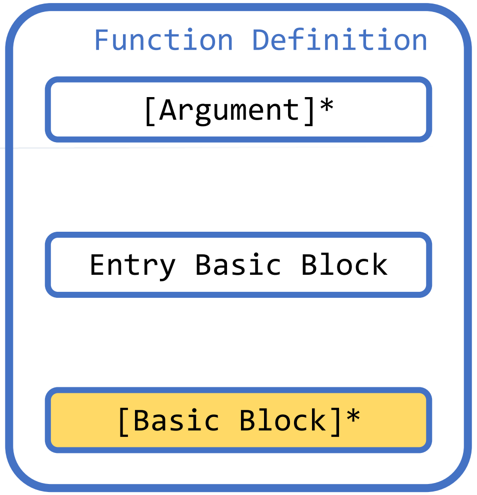
# Functions: Recursive Factorial



```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Functions: Recursive Factorial



Function Definition

[Argument]*

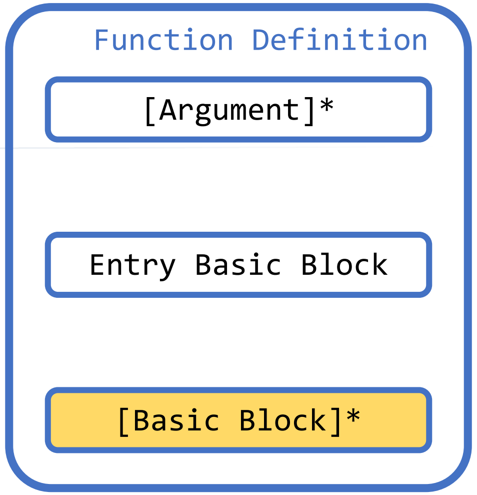
Entry Basic Block

[Basic Block]*

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Functions: Recursive Factorial



Function Definition

[Argument]*

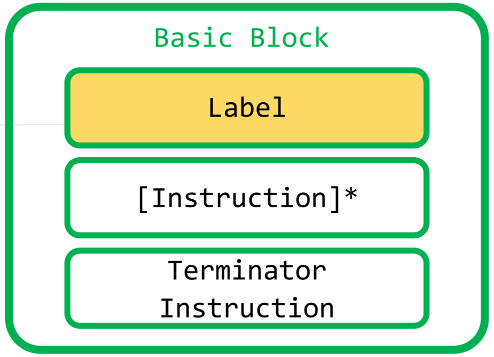
Entry Basic Block

[Basic Block]*

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks



Basic Block

Label

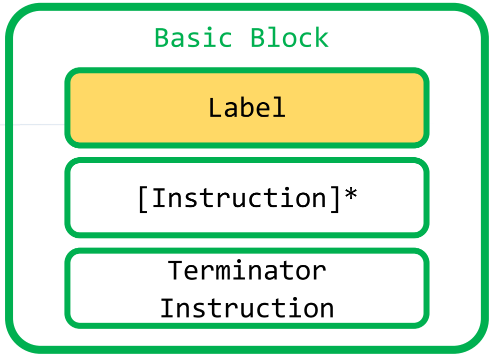
[Instruction]*

Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Labels

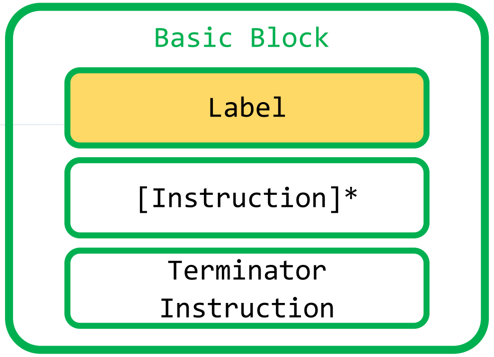


```c
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every Basic Block has a label

```llvm
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Labels

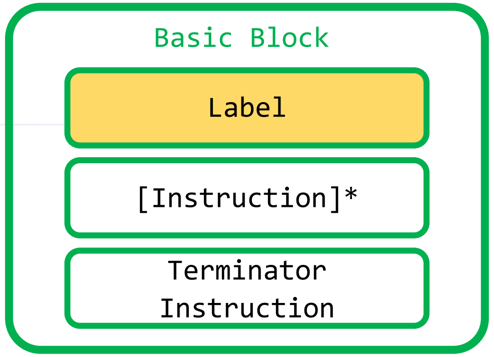


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block has a label
- If it is not explicit

```
declare i32 @factorial(i32)

define i32 @main(){
     %0 = call i32 @factorial(i32 5)
     %1 = mul i32 %0, 6
     %2 = call i32 @factorial(i32 6)
     %3 = icmp eq i32 %1, %2
     %retval = zext i1 %3 to i32
     ret i32 %retval
}
```

# Basic Blocks: Labels

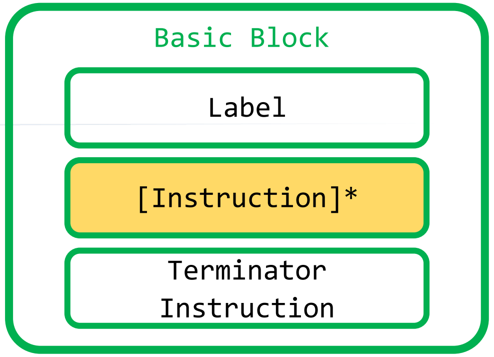


```c
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block has a label
- If it is not explicit
  - the compiler labels it with a number
  - starting from `0`

```llvm
declare i32 @factorial(i32)

define i32 @main(){
0:   ; implicit
     %0 = call i32 @factorial(i32 5)
     %1 = mul i32 %0, 6
     %2 = call i32 @factorial(i32 6)
     %3 = icmp eq i32 %1, %2
     %retval = zext i1 %3 to i32
     ret i32 %retval
}
```

# Basic Blocks: Instructions

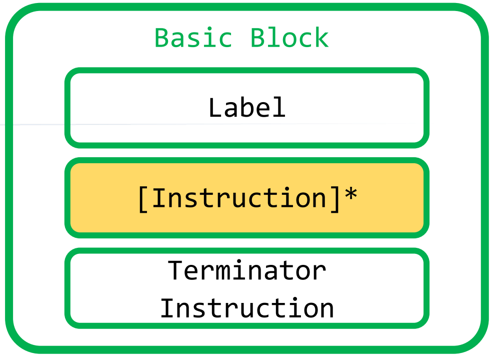


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions


Basic Block

Label

[Instruction]*

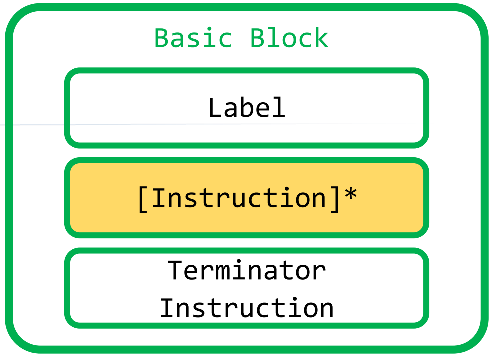
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
    - arithmetic computation,

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions

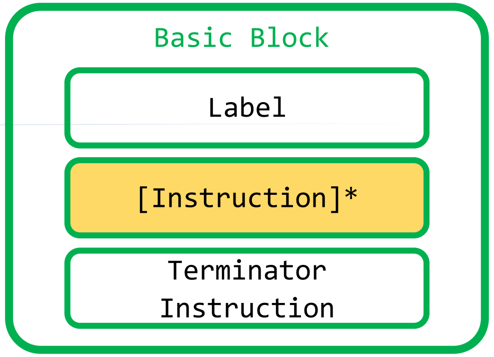


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
    - arithmetic computation,
    - comparison, etc.

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions



Basic Block

Label

[Instruction]*

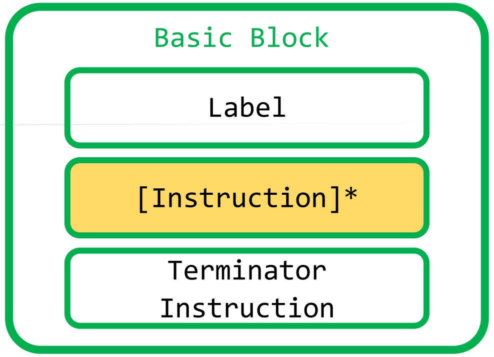
Terminator Instruction

```c
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
  - arithmetic computation,
  - comparison, etc.
- Low-level language → “local” variables

```llvm
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions


Basic Block

Label

[Instruction]*

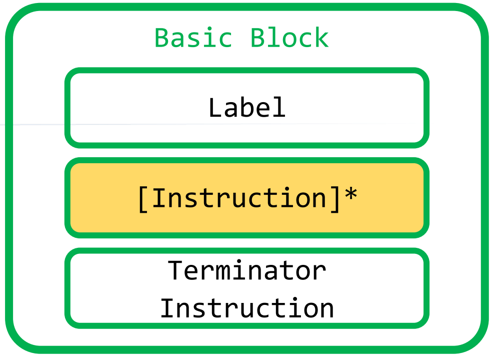
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
  - arithmetic computation,
  - comparison, etc.
- Low-level language → “local” variables
  - Virtual registers

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions



Basic Block

Label

[Instruction]*

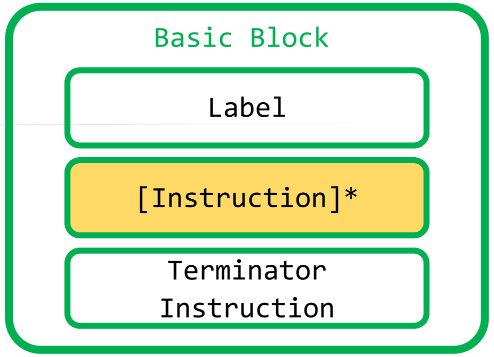
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
  - arithmetic computation,
  - comparison, etc.
- Low-level language → “local” variables
  - Virtual registers
  - Two flavors of names
    - %<name>

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions

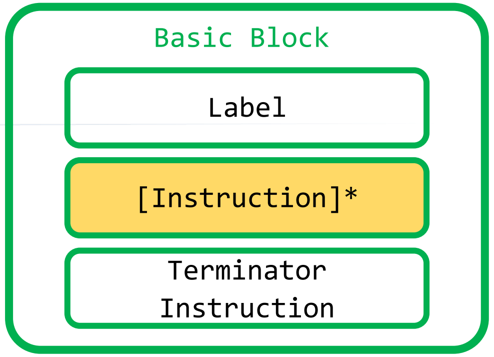


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

- Instructions fulfill basic operations.
  - arithmetic computation,
  - comparison, etc.
- Low-level language → “local” variables
  - Virtual registers
  - Two flavors of names
    - %<name>
    - %<number>

# Basic Blocks: Instructions

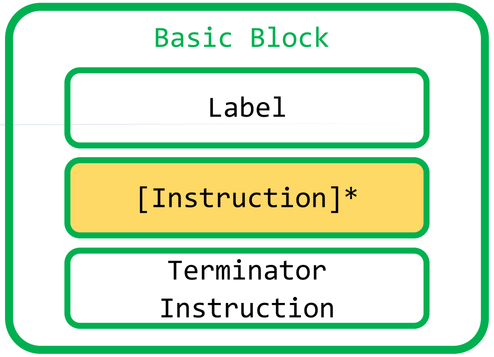


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Instructions fulfill basic operations.
  - arithmetic computation,
  - comparison, etc.
- Low-level language → "local" variables
  - Virtual registers
  - Two flavors of names
    - %<name>
    - %<number>
  - "LLVM IR has infinite registers"

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions

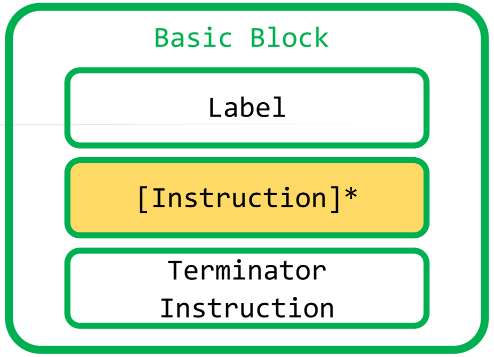


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- LLVM-IR is an SSA-based representation

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions

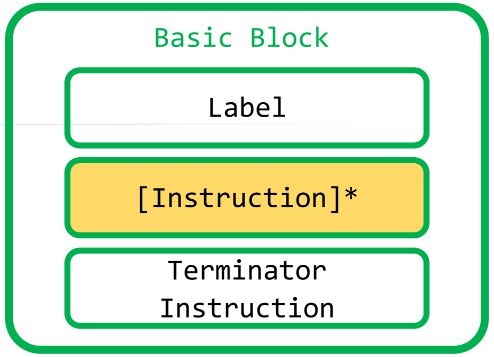


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

- LLVM-IR is an SSA-based representation
  - Static Single Assignment

# Basic Blocks: Instructions

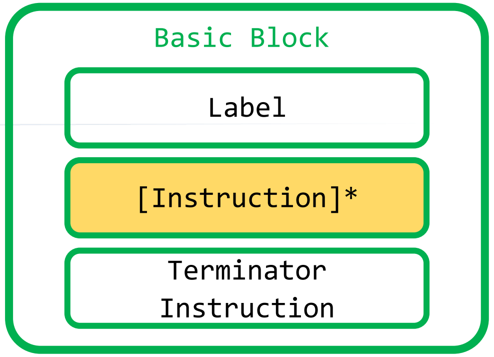


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- LLVM-IR is an SSA-based representation
  - Static Single Assignment
  - every variable is assigned exactly once
  - ... is defined before it is used

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions


Basic Block

Label

[Instruction]*

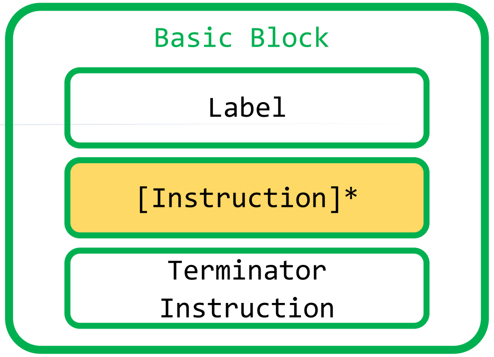
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- LLVM-IR is an SSA-based representation
  - Static Single Assignment
  - every variable is assigned exactly once
  - ... is defined before it is used

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Instructions

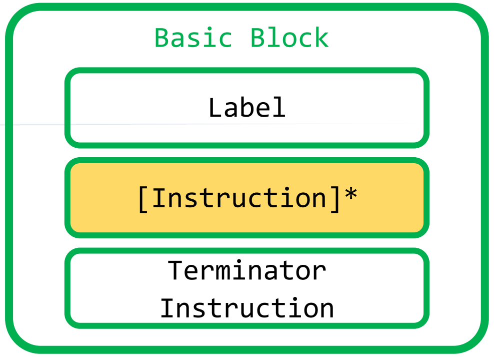


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

- LLVM-IR is an SSA-based representation
    - **S**tatic **S**ingle **A**ssignment
    - every variable is assigned exactly once
    - ... is defined before it is used

# Basic Blocks: Instructions


Basic Block

Label

[Instruction]*

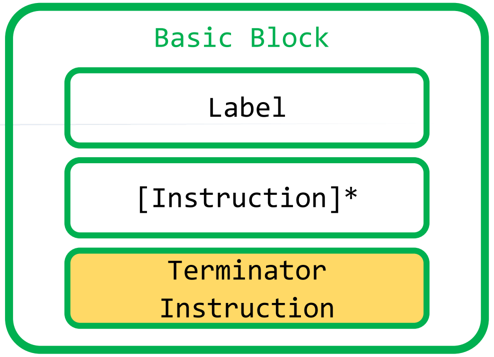
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- LLVM-IR is an SSA-based representation
  - Static Single Assignment
  - every variable is assigned exactly once
  - ... is defined before it is used

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Terminator Instructions



Basic Block

Label

[Instruction]*

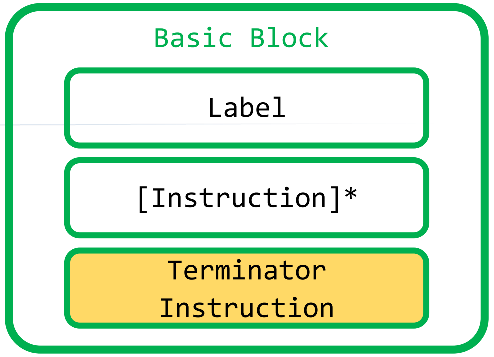
Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block ends with a terminator instruction

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Terminator Instructions

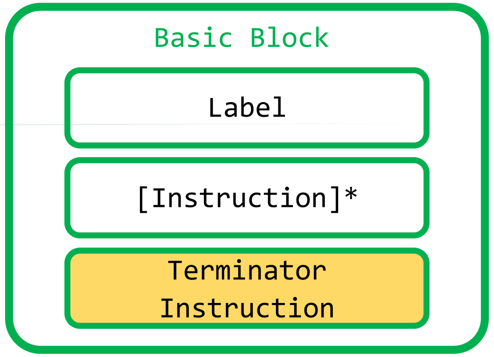


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block ends with a terminator instruction
  - `br`: branch
  - `ret`: return
  - `switch`: switch
  - exception handling instructions
  - …

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

# Basic Blocks: Terminator Instructions



Basic Block

Label

[Instruction]*

Terminator Instruction

```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block ends with a terminator instruction
  - br: branch
  - ret: return
  - switch: switch
  - exception handling instructions
  - …
- indicating the next bb to be executed

```
define i32 @factorial(i32 %val){
entry:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, %entry ]
    ret i32 retval
}
```

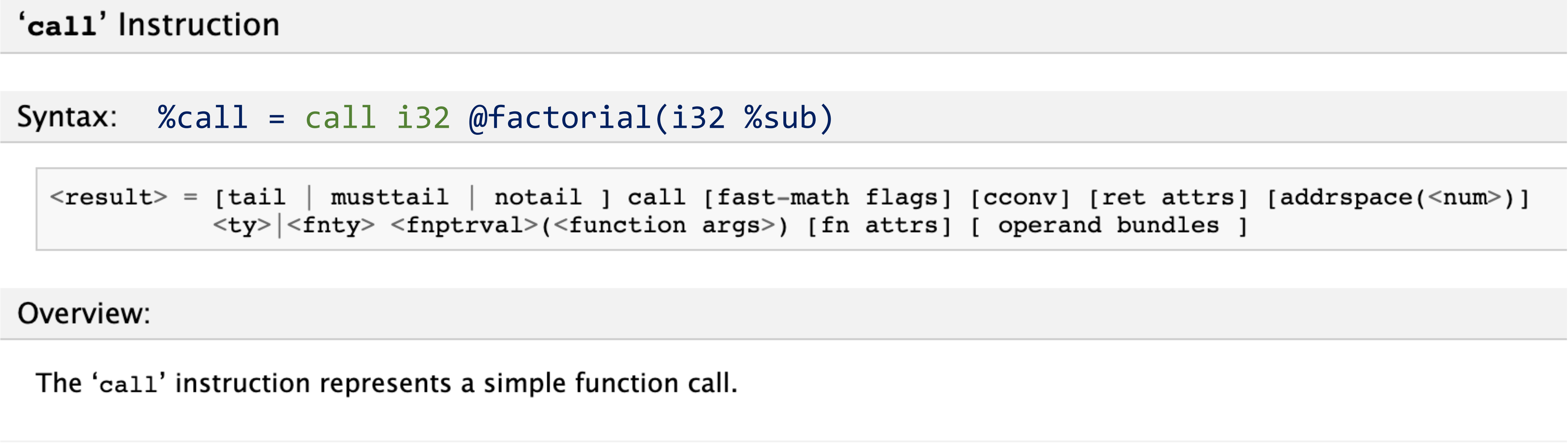
# The LangRef is Helpful

- A tip of the iceberg

# The LangRef is Helpful

- A tip of the iceberg
- Instructions often have **many** variants
  - **What else could a `call` instruction need?**

# The LangRef is Helpful



## ‘`call`’ Instruction

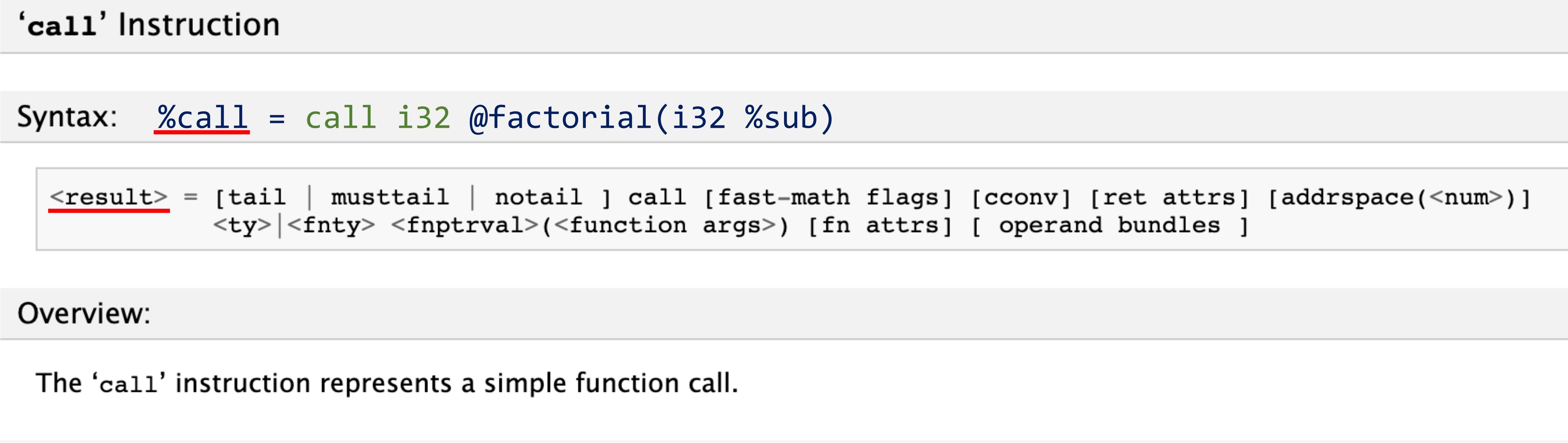
### Syntax: `%call = call i32 @factorial(i32 %sub)`

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           <ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

### Overview:

The ‘`call`’ instruction represents a simple function call.

# The LangRef is Helpful



## ‘call’ Instruction

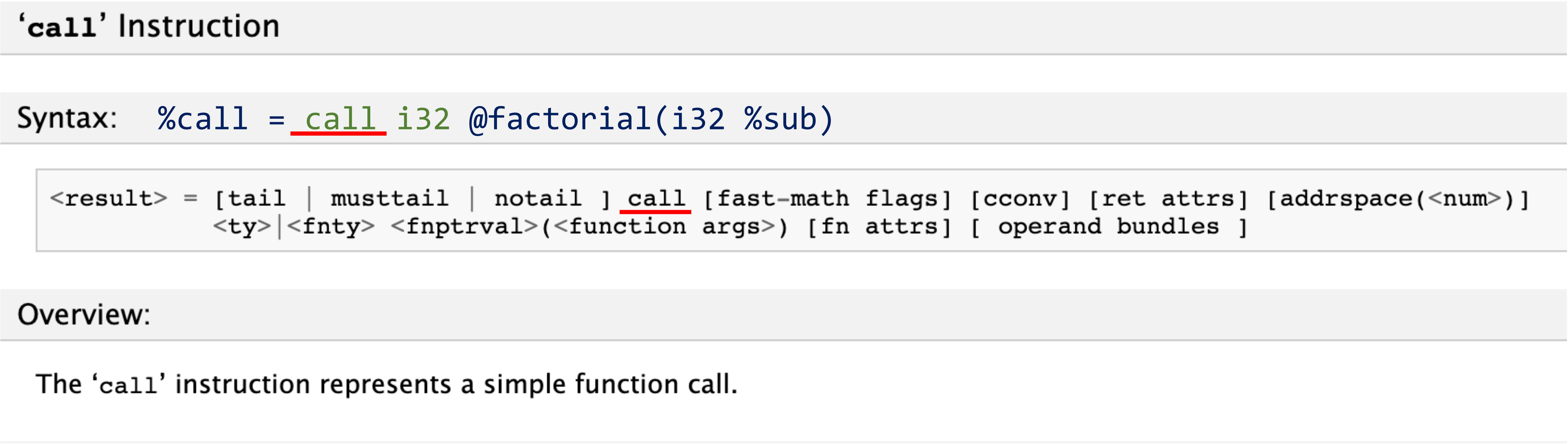
Syntax: `%call = call i32 @factorial(i32 %sub)`

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           <ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

## Overview:

The ‘call’ instruction represents a simple function call.

# The LangRef is Helpful



## ‘`call`’ Instruction

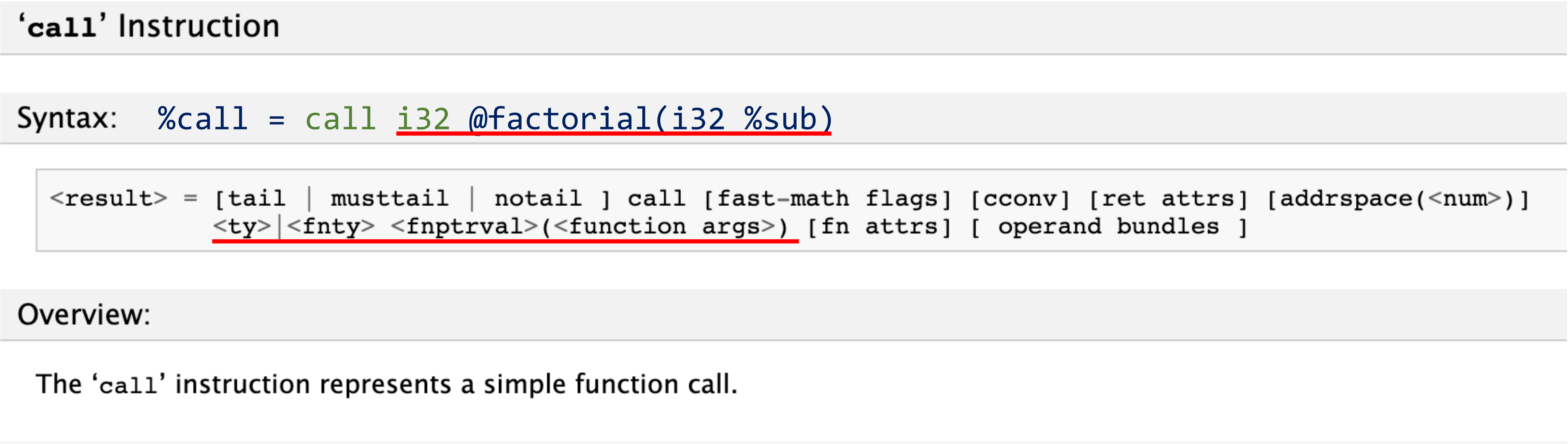
Syntax: `%call = call i32 @factorial(i32 %sub)`

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           <ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

Overview:

The ‘`call`’ instruction represents a simple function call.

# The LangRef is Helpful



### ‘call’ Instruction

Syntax: %call = call i32 @factorial(i32 %sub)

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           <ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

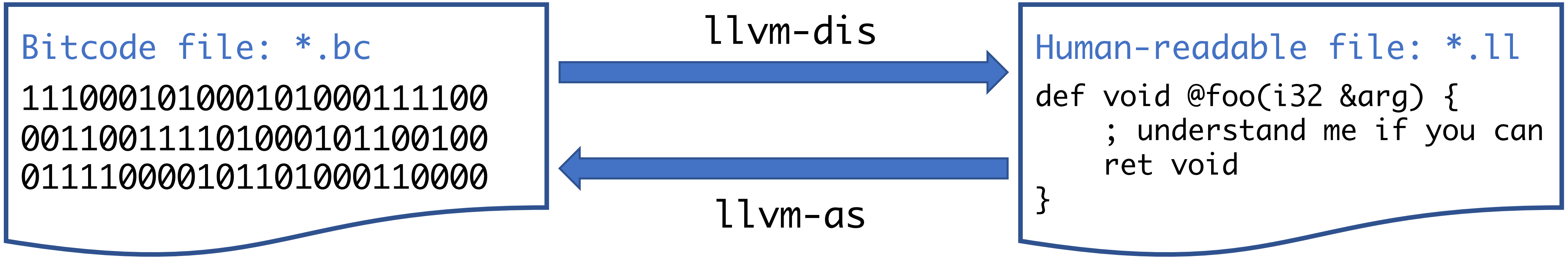
Overview:

The ‘call’ instruction represents a simple function call.

# Summary

- LLVM-IR is an SSA-based representation
  - **providing the capability of representing high-level languages**
  - **enabling code optimization**
- LLVM Language Reference Manual
  - **well-documented**
  - **useful for self-teaching**
  - **just google it and enjoy the journey**

# IR Representation



Bitcode file: *.bc

```
111000101000101000111100
001100111101000101100100
011110000101101000110000
```

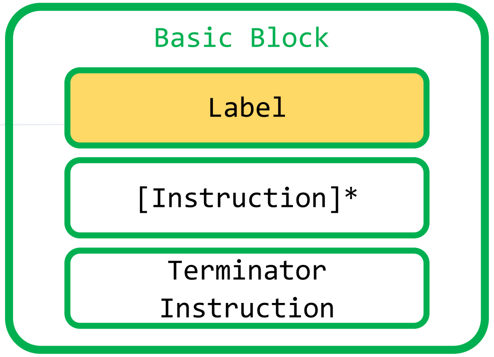
llvm-dis

llvm-as

Human-readable file: *.ll

```
def void @foo(i32 &arg) {
    ; understand me if you can
    ret void
}
```

# Basic Blocks: Labels

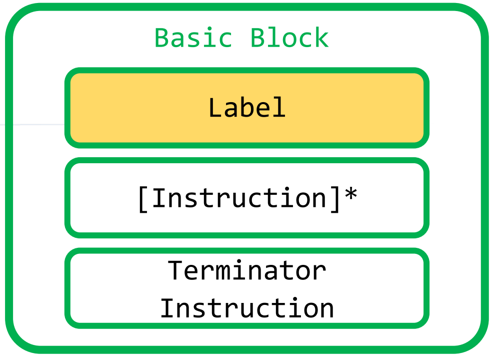


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block has a label
- If it is not explicit

```
define i32 @factorial(i32 %val){
~~entry:~~
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, ~~%entry~~ ]
    ret i32 retval
}
```

# Basic Blocks: Labels

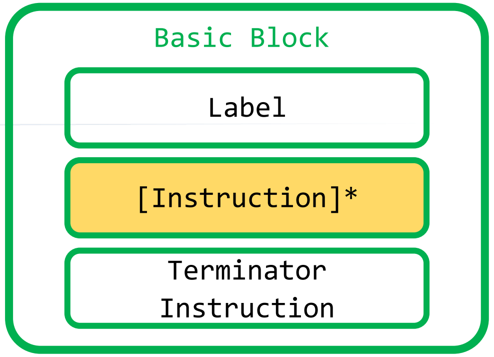


```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

- Every basic block has a label
- If it is not explicit
    - the compiler labels it with numbers
    - starting from `0`

```
define i32 @factorial(i32 %val){
~~entry:~~ 0:
    %cmp = icmp eq i32 %val, 0
    br i1 %cmp, label %return, label %if.end

if.end:
    %sub = add i32 %val, -1
    %call = call i32 @factorial(i32 %sub)
    %mul = mul i32 %call, %val
    br label %return

return:
    %retval = phi i32 [ %mul, %if.end ], [ 1, ~~%entry~~ %0 ]
    ret i32 retval
}
```

# Basic Blocks: Instructions



```
// val is non-negative
int factorial(int val){
    if (val == 0)
        return 1;
    return val * factorial(val-1);
}
```

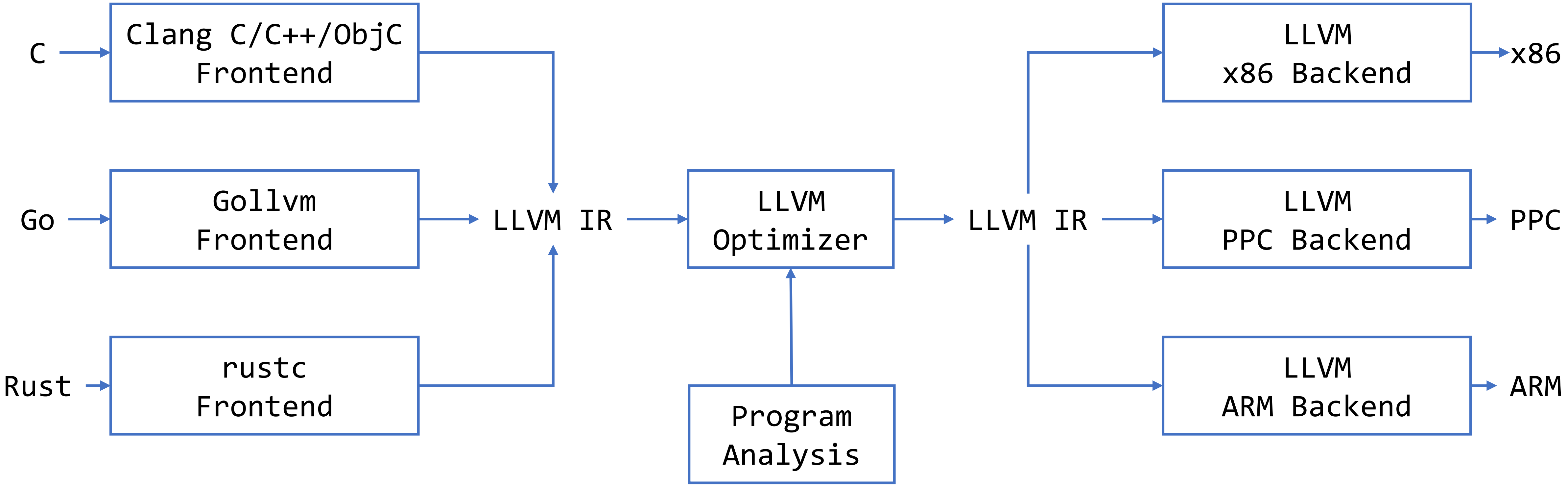
- Instructions fulfill basic operations.
    - arithmetic computation,
    - comparison, etc.
- Low-level language → “local” variables
    - Virtual registers
    - Two flavors of names
        - %<name>
        - %<number>

```
define i32 @factorial(i32 %val){
entry:
    %0 = icmp eq i32 %val, 0
    br i1 %0, label %return, label %if.end

if.end:
    %1 = add i32 %val, -1
    %2 = call i32 @factorial(i32 %1)
    %3 = mul i32 %1, %val
    br label %return

return:
    %4 = phi i32 [ %3, %if.end ], [ 1, %entry ]
    ret i32 4
}
```

# Outline

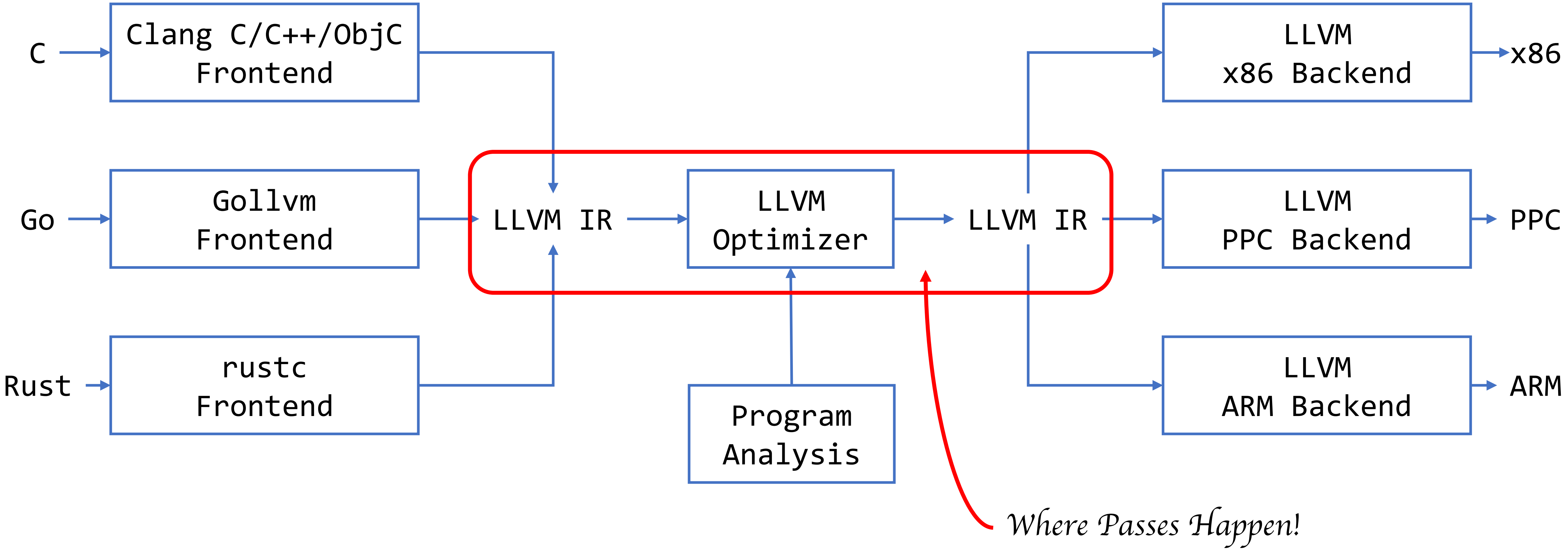
- Introduction to LLVM IR
- Writing LLVM Passes

# What is a Pass?



# What is a Pass?

- Passes perform the transformations and optimizations on LLVM-IR.

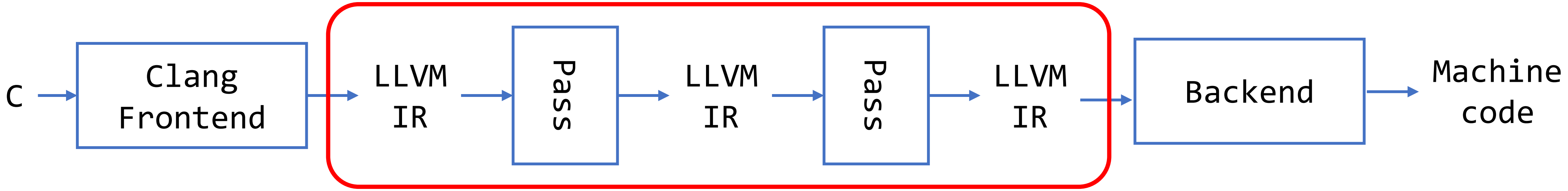


# What is a Pass?

- Depending on how a pass works, it inherits from the classes of:
    - **ModulePass: using the entire program as a unit**
    - **CallGraphSCCPass: traversing the program bottom-up the call graph**
        - **callees before callers**
    - **FunctionPass: executing on each function independent of all others in a program**
    - **LoopPass: executing on each loop independent of all others in a function**
        - **in loop nested order: outer most loop is processed last**
    - **RegionPass: executing on each single entry single exit region in the function**
        - **in nested order**

# What is a Pass?

- Passes perform the transformations and optimizations on LLVM-IR.



# How to Implement a Pass?

- Assuming LLVM has been configured and built:
    1. **Write pass code (in C++)**
    2. **Set up a build script**
    3. **Run the pass**

# How to Implement a Pass?

- Assuming LLVM has been configured and built:
  1. **Write pass code (in C++)**
  2. **Set up a build script**
  3. **Run the pass**

# Start from Hello_world

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

# Write Pass Code (in C++)

```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

`$(LLVM_HOME)/lib/Transforms/Hello`

# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

# Write Pass Code (in C++)

```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

Analysis Code

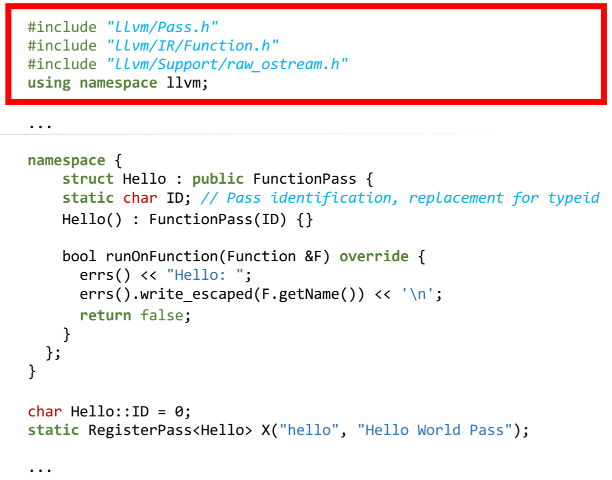
# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

Analysis Code

Registration Code

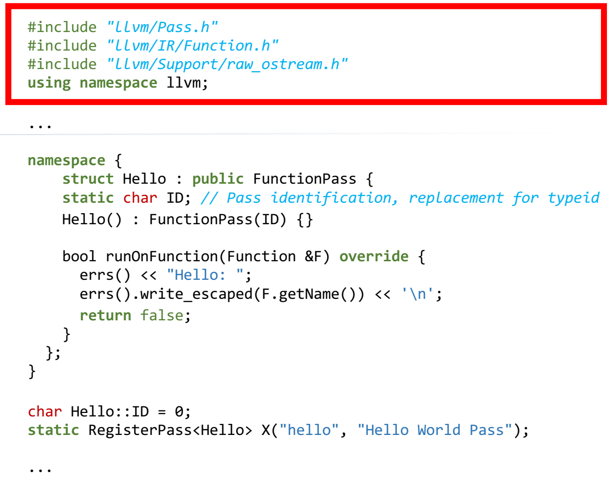
# Write Pass Code (in C++)



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
#include "llvm/Pass.h”                    // We’re writing a Pass

#include "llvm/IR/Function.h”             // It operates on Functions

#include "llvm/Support/raw_ostream.h”     // We’ll do some printing

using namespace llvm;
```

# Write Pass Code (in C++)



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
#include "llvm/Pass.h”                      // We’re writing a Pass

#include "llvm/IR/Function.h”               // It operates on Functions

#include "llvm/Support/raw_ostream.h”       // We’ll do some printing

using namespace llvm;                       // APIs from include files live in
                                            //   the llvm namespace
```

# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
namespace {                                          // Hello is only visible to the current file
    struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}
```

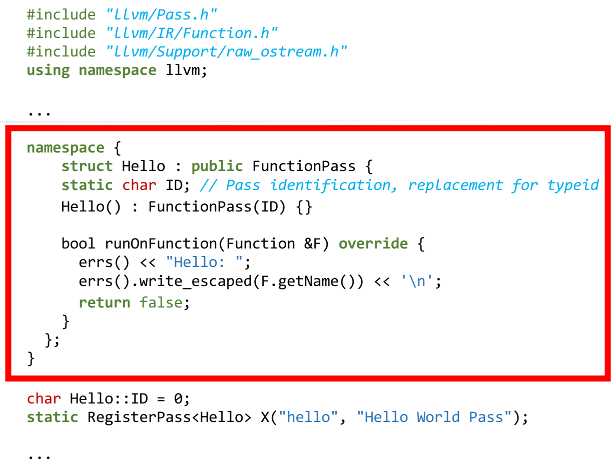
# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...
namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
namespace {
    struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}
```

```
// Hello is only visible to the current file
// Hello inherits from FunctionPass
```

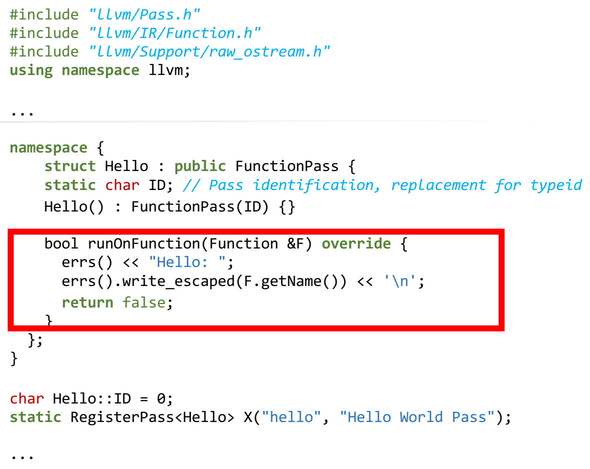
# Write Pass Code (in C++)



```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```cpp
namespace {                                          // Hello is only visible to the current file
    struct Hello : public FunctionPass {             // Hello inherits from FunctionPass
    static char ID;
    Hello() : FunctionPass(ID) {}                    // Declare the pass ID used by LLVM to identify

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}
```

# Write Pass Code (in C++)



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

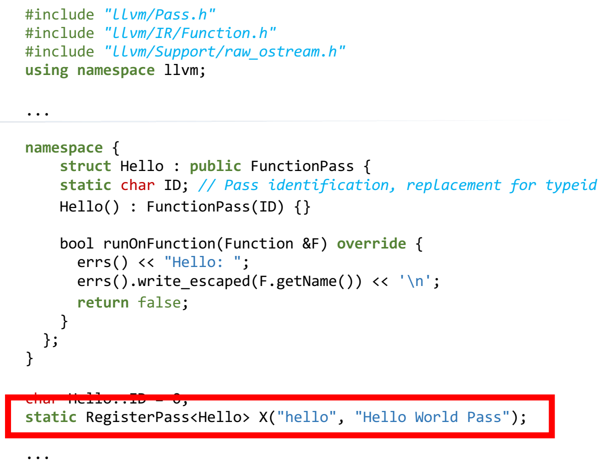
```
bool runOnFunction(Function &F) override {     // override the virtual method of FunctionPass

    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n’;

    return false;
}
```

# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
bool runOnFunction(Function &F) override {     // override the virtual method of FunctionPass

    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n’; // output the function name

    return false;
}
```

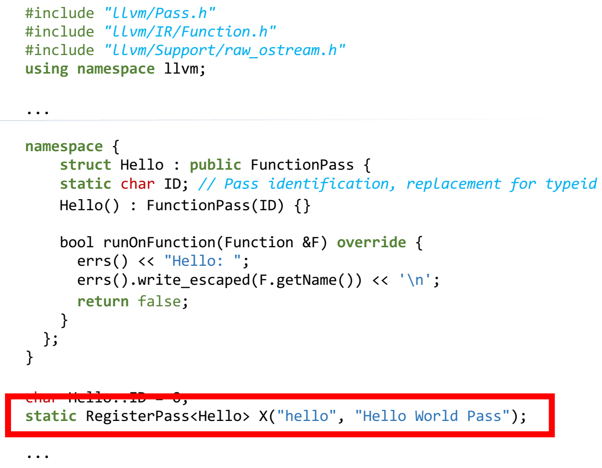
# Write Pass Code (in C++)



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

```
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

# Write Pass Code (in C++)



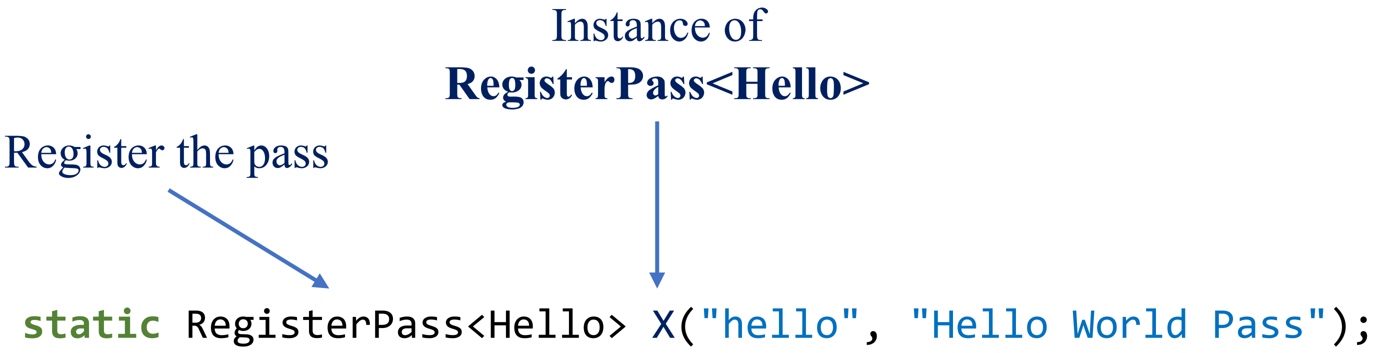
```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

Register the pass

```
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

# Write Pass Code (in C++)

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```

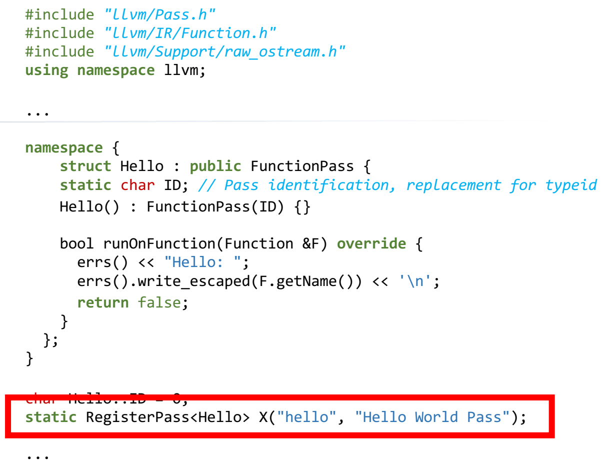


Instance of **RegisterPass<Hello>**

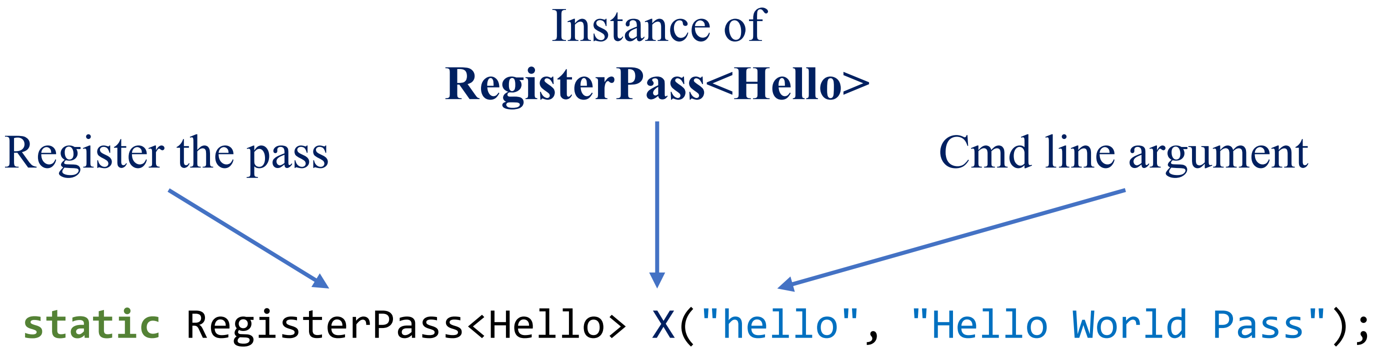
Register the pass

```
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

# Write Pass Code (in C++)



```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```



Instance of **RegisterPass<Hello>**

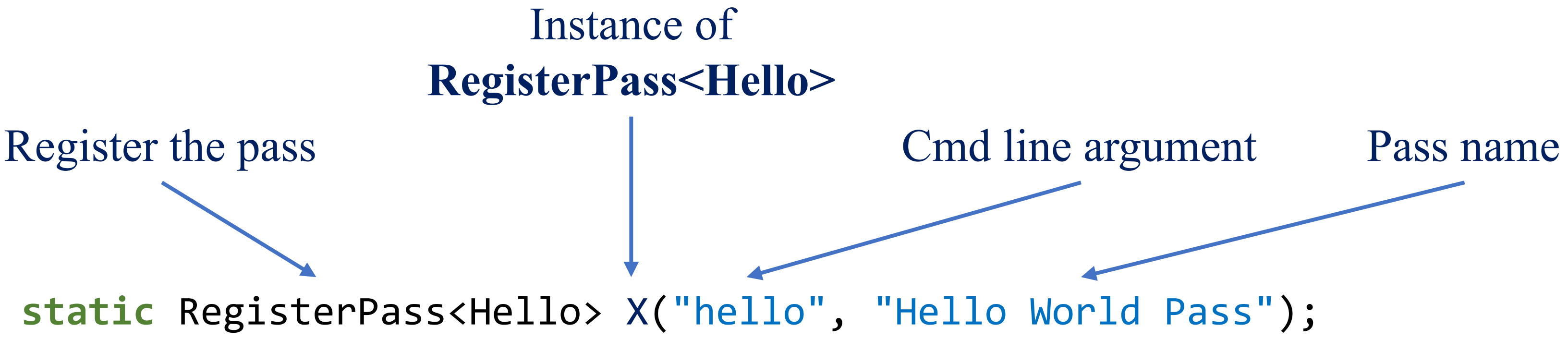
Register the pass

Cmd line argument

```cpp
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

# Write Pass Code (in C++)

```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass");

...
```



Instance of **RegisterPass<Hello>**

Register the pass

Cmd line argument

Pass name

```cpp
static RegisterPass<Hello> X("hello", "Hello World Pass");
```

# How to Write a Pass?

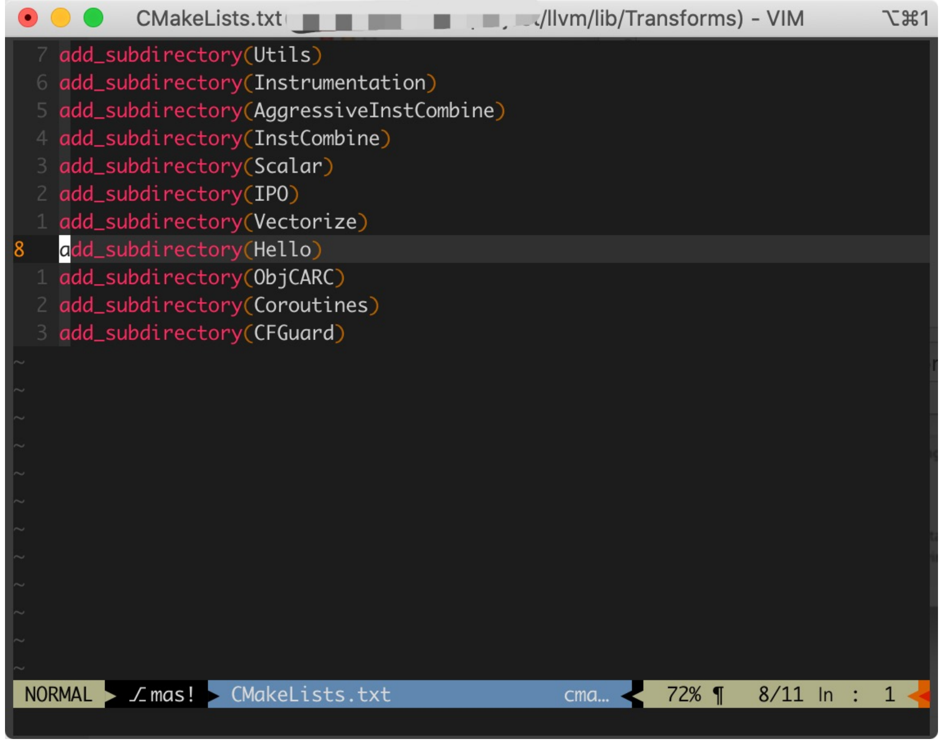
- Assuming LLVM has been configured and built:
  1. **Write pass code (in C++)**
  2. **Set up a build script**
  3. **Run the pass**

# Set up a Build Script

- $(LLVM_HOME)/lib/Transforms/CMakeLists.txt
  - **add_subdirectory(Hello)**

# Set up a Build Script

- $(LLVM_HOME)/lib/Transforms/CMakeLists.txt
  - add_subdirectory(Hello)



```
add_subdirectory(Utils)
add_subdirectory(Instrumentation)
add_subdirectory(AggressiveInstCombine)
add_subdirectory(InstCombine)
add_subdirectory(Scalar)
add_subdirectory(IPO)
add_subdirectory(Vectorize)
add_subdirectory(Hello)
add_subdirectory(ObjCARC)
add_subdirectory(Coroutines)
add_subdirectory(CFGuard)
```

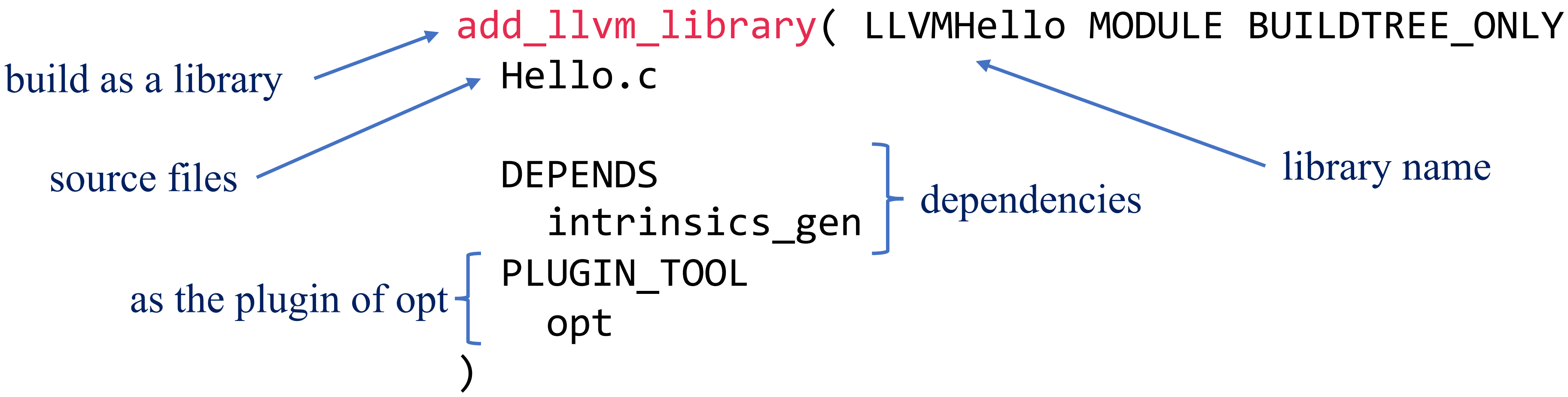
# Set up a Build Script

- $(LLVM_HOME)/lib/Transforms/CMakeLists.txt
  - **add_subdirectory(Hello)**
- $(LLVM_HOME)/lib/Transforms/Hello/CMakeLists.txt

```
add_llvm_library( LLVMHello MODULE BUILDTREE_ONLY
  Hello.c

  DEPENDS
    intrinsics_gen
  PLUGIN_TOOL
    opt
)
```

# Set up a Build Script

- $(LLVM_HOME)/lib/Transforms/CMakeLists.txt
  - **add_subdirectory(Hello)**
- $(LLVM_HOME)/lib/Transforms/Hello/CMakeLists.txt



```
add_llvm_library( LLVMHello MODULE BUILDTREE_ONLY
  Hello.c

  DEPENDS
    intrinsics_gen
  PLUGIN_TOOL
    opt
)
```

build as a library

source files

as the plugin of opt

dependencies

library name

# How to Write a Pass?

- Assuming LLVM has been configured and built:
  1. **Write pass code (in C++)**
  2. **Set up a build script**
  3. **Run the pass**

# Run a Pass with Opt

- $ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null

# Run a Pass with Opt

```
$ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null
```

load pass as a shared library

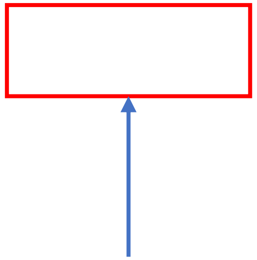
# Run a Pass with Opt

```
$ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null
```

Pass file path

# Run a Pass with Opt

```
$ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null
```



Command line argument

# Run a Pass with Opt

- `$ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null`

Target IR file

# Run a Pass with Opt

- $ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null

```
int factorial(int val);

int main(){
    return factorial(5) * 6
        == factorial(6);
}
```

```
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

# Run a Pass with Opt

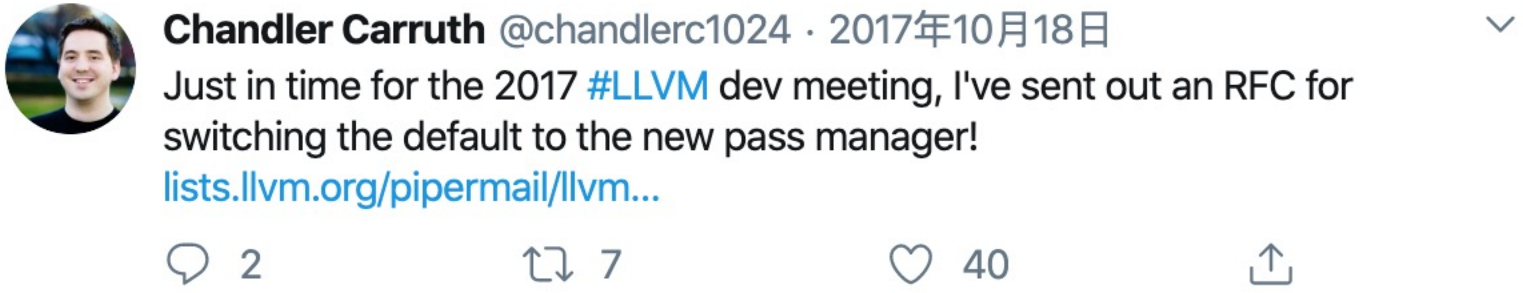
- $ opt -load lib/LLVMHello.so -hello < factorial.ll > /dev/null

```
int factorial(int val);

int main(){
    return factorial(5) * 6
        == factorial(6);
}
```

```
declare i32 @factorial(i32)

define i32 @main(){
    %0 = call i32 @factorial(i32 5)
    %1 = mul i32 %0, 6
    %2 = call i32 @factorial(i32 6)
    %3 = icmp eq i32 %1, %2
    %retval = zext i1 %3 to i32
    ret i32 %retval
}
```

```
Hello: factorial
Hello: main
```

# Legacy vs. New



**Chandler Carruth** @chandlerc1024 · 2017年10月18日

Just in time for the 2017 #LLVM dev meeting, I've sent out an RFC for switching the default to the new pass manager! lists.llvm.org/pipermail/llvm...

# Legacy vs. New

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World
Pass");

...
```

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
  struct Hello : public PassInfoMixin<Hello> {
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return PreservedAnalyses::all();
    }
  };
}

extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "Hello", "v0.1",
    [](PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [](StringRef PassName, FunctionPassManager &FPM,
           ArrayRef<PassBuilder::PipelineElement>) {
          if (PassName == "Hello"){
            FPM.addPass(Hello());
            return true;
          }
          return false;
        });}};}
```

# Legacy vs. New

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

...

namespace {
    struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World
Pass");

...
```

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

namespace {
  struct Hello : public PassInfoMixin<Hello> {
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return PreservedAnalyses::all();
    }
  };
}

extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "Hello", "v0.1",
    [](PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [](StringRef PassName, FunctionPassManager &FPM,
          ArrayRef<PassBuilder::PipelineElement>) {
          if (PassName == "Hello"){
            FPM.addPass(Hello());
            return true;
          }
          return false;
        });}};}
```

# Legacy vs. New

- Legacy: new passes should be the subclasses of the defined ones

```
namespace {
    struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }
  };
}
```

# Legacy vs. New

- Legacy: new passes should be the subclasses of the defined ones

```
namespace {
  struct Hello : public FunctionPass {
  static char ID;
  Hello() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
   errs() << "Hello: ";
   errs().write_escaped(F.getName()) << '\n';
   return false;
  }
 };
}
```

# Legacy vs. New

- Legacy: new passes should be the subclasses of the defined ones
- New: all classes providing methods running on IR are passes

# Legacy vs. New

- Legacy: new passes should be the subclasses of the defined ones
- New: all classes providing methods running on IR are passes

```
namespace {
  struct Hello : public PassInfoMixin<Hello> {
    PreservedAnalyses run(Function &F,FunctionAnalysisManager &FAM) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return PreservedAnalyses::all();
    }
  };
}
```

# Summary

- With LLVM passes, we could implement analysis to transform or optimize the IR code
- There are two styles of pass managers

## Thank you all for your attention