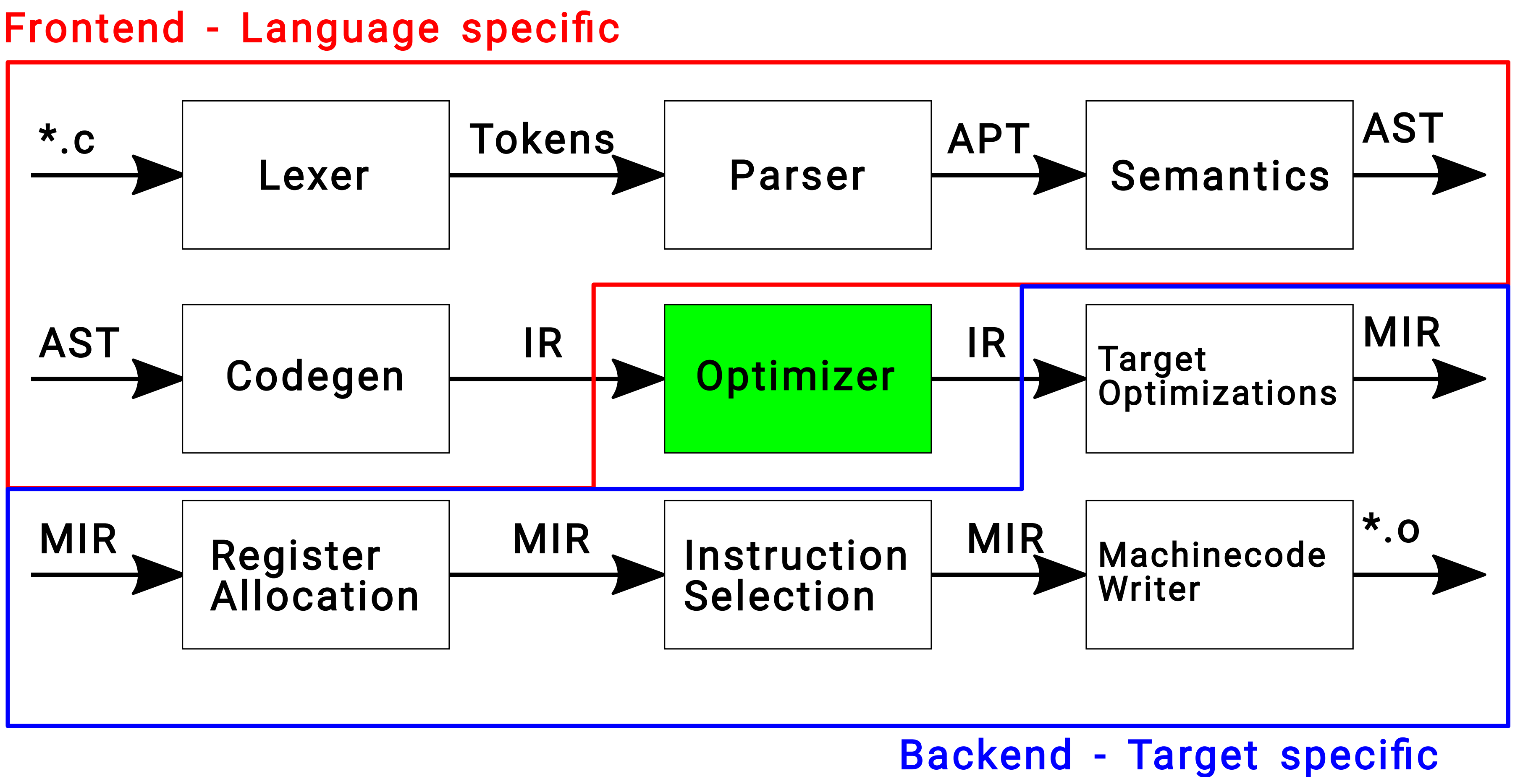
An introduction to LLVM

Content

- What’s LLVM?
- LLVM IR Language
  - Types
  - Structure
  - Instructions
- Simple example Compiler

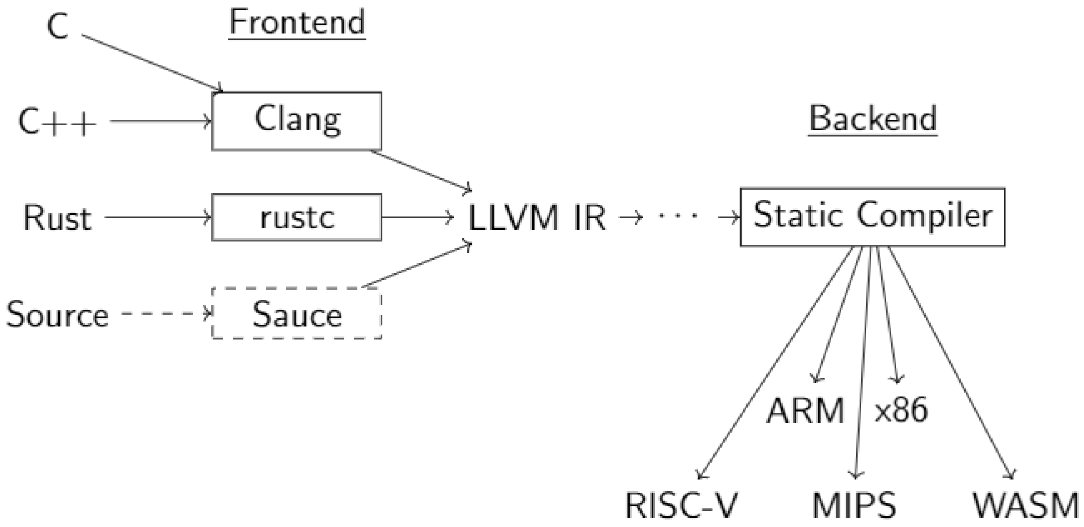
What’s LLVM?



Frontend - Language specific
*.c
Lexer
Tokens
Parser
APT
Semantics
AST
AST
Codegen
IR
Optimizer
IR
Target Optimizations
MIR
MIR
Register Allocation
MIR
Instruction Selection
MIR
Machinecode Writer
*.o
Backend - Target specific

What’s LLVM?

- Optimizer & Backends
- C abstraction level
- Modular & Composable
- C++ and C/FFI API



LLVM IR Language

- Target language for Frontends
- Assembly-like
- Strong static typing
- (Largely) Backend target independent
- Representations:
  - Data structure
  - Textual
  - Bit code

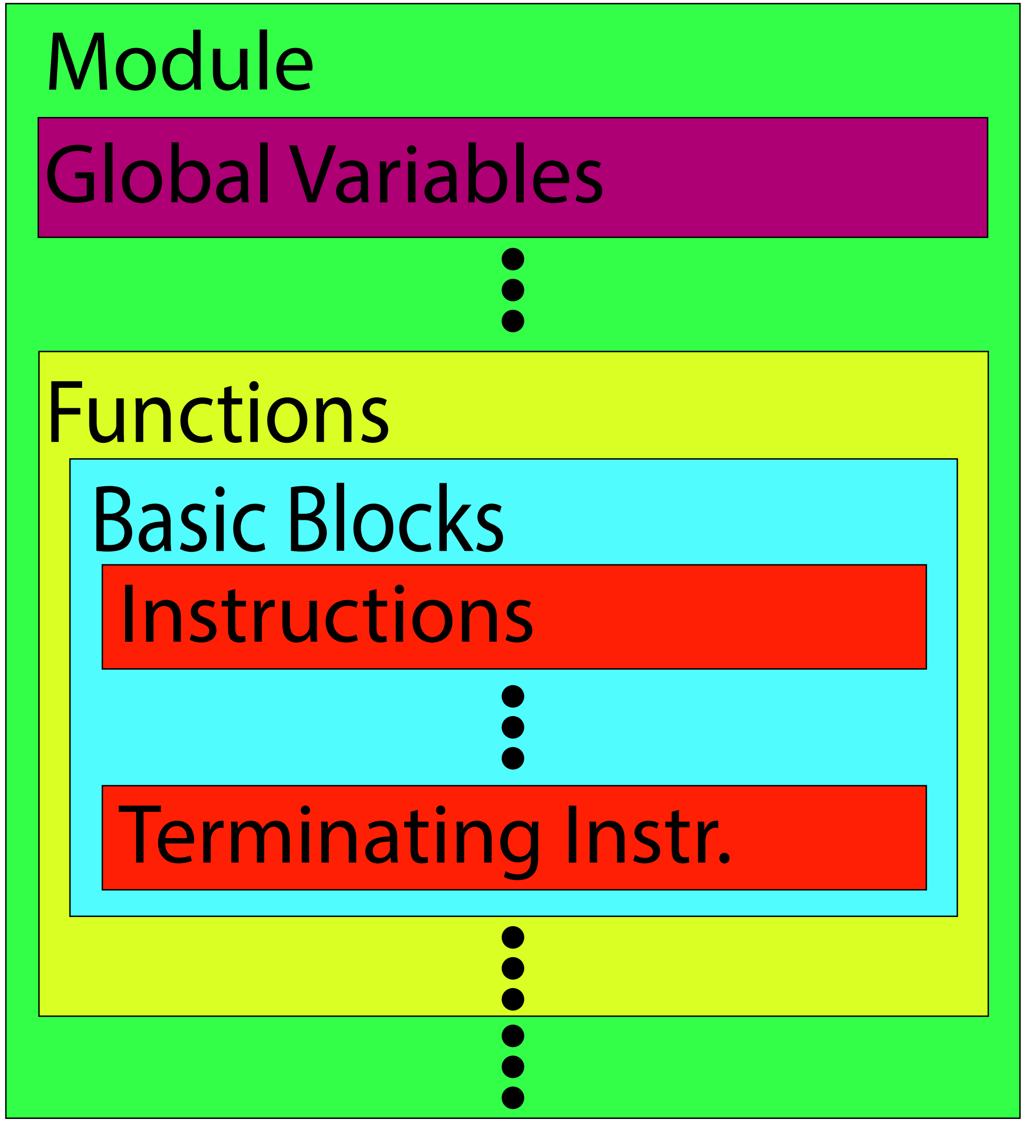
LLVM IR Language - Types

void	Void datatype, only valid as return type
iN eg. i32	Integer with given **Bit**-width – **Signless!**
float, double, half, fp128…	IEEE 754 Floating point
<type> [addrspace(N)]* eg. i32*	Pointer type with address space
[N x <type>] eg. [10 x i8]	Array types
{ <type> {, <type>} } eg. { i32, i32 }	Structure type, sequential in memory, fields unnamed; access via index
<type> ([<type> {, <type>}]) eg. i32 (i8*,float)	Function type, not first class

LLVM IR Language – Identified Structure Types

- Definition: `%Pair = type { i32, i32 }`
- Usage: `%Pair*`
- Allows Recursion: `%Node = type { i32, %Node* }`
- May be opaque: `%decl = type opaque`
- Equality based on identifier, not structure

LLVM IR Language - Hierarchy



LLVM IR Language - Module

```
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-p270:32:32-
p271:32:32-p272:64:64-i64:64-f80:128-
n8:16:32:64-S128"

%Pair = type { i32, i32 }

declare i32 @puts(i8*)

@foo = global i32 0

define i32 @square(i32) {
    %2 = mul nsw i32 %0, %0
    ret i32 %2
}
```

- Compilation Unit
- Target Triple & Datalayout
- Identified Structure types
- Function declarations & definitions
- Global variables
- Metadata (incl. Debug info)
- Symbol names starting with @

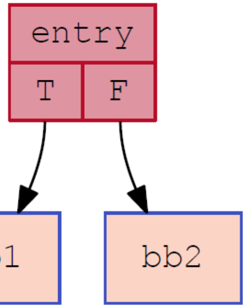
LLVM IR Language - Functions

- Declaration: `declare i32 @puts(i8*)`
- Definition:

```
define i32 @double(i32 %arg) {
    ...
}
```

LLVM IR Language – Basic Blocks

```
define i32 @random(i1 %cond, i32 %value) {
entry:
    br i1 %cond, label %bb1, label %bb2
bb1:
    ret i32 %value
bb2:
    %double = mul i32 %value, 2
    ret i32 %double
}
```



CFG for 'random' function

- Starts with label
- List of instructions
- Ends with Terminator instr.
  - ret, br, unreachable…
- Predecessors
- Successors
- First block: Entry block

LLVM IR Language – Instructions

```
%z = add i32 %x, %y
```

- Produces 0 or 1 Values
- 0 to N Operands:
  - Instruction Results, Basic Block or Function parameter:
    - Prefixed with `%`
  - Constants:
    - i1 Constants: `true, false`
    - iN Constants: `35`
    - Pointer Constant: `null`
    - Floating Point Constant: `1.3`
  - Globals:
    - Prefixed with `@`
    - Pointer type

LLVM IR Language – Values

- Immutable, cannot be reassigned (SSA!)
- Named or Unnamed
  - Unnamed get monotonically increasing number
    - Numbering must be correct in textual syntax nevertheless!
  - Optionally unspecified in syntax:
    - Function parameters
    - Entry Block

```
define i32 @quadruple(i32) {
    %result = add i32 %0, %0
    %2 = add i32 %result, %result
    ret i32 %2
}
```

- First parameter not specified -> `%0`
- Entry block not specified -> `%1`
- Result of first add named
- Result of second add unnamed
  -> has to be `%2`

LLVM IR Language – alloca

```
%ptr = alloca <type>
eg. %ptr = alloca i32
```

- Allocates storage on the stack
- Returns value of type <type>*
- Memory uninitialized
- Deleted upon function return
- Should be placed in the entry block

LLVM IR Language – `load`, `store`

```
%value = load <type>, <type>* <ptr>
```
eg. `%value = load i32, i32* %ptr`

- Gets last stored value from the referenced memory

```
store <type> <op>, <type>* <ptr>
```
eg. `store i32 0, i32* %ptr`

- Stores value into the referenced memory

LLVM IR Language – Working with memory example

```
define i32 @foo(i1 %cond) {
    %1 = alloca i32
    br i1 %cond, label %bb0, label %bb1

bb0:
    store i32 0, i32* %1
    br label %continue

bb1:
    store i32 99, i32* %1
    br label %continue

continue:
    %2 = load i32, i32* %1
    ret i32 %2
}
```

LLVM IR Language – Global variables

- Definition: `@<name> = global <type> <init>`
  - Requires constant initialization
- Declaration: `@<name> = external global <type>`
- Usage has pointer type

```
@foo = global i32 0

define void @bar() {
    store i32 5, i32* @foo
    ret void
}
```

LLVM IR Language – getelementptr

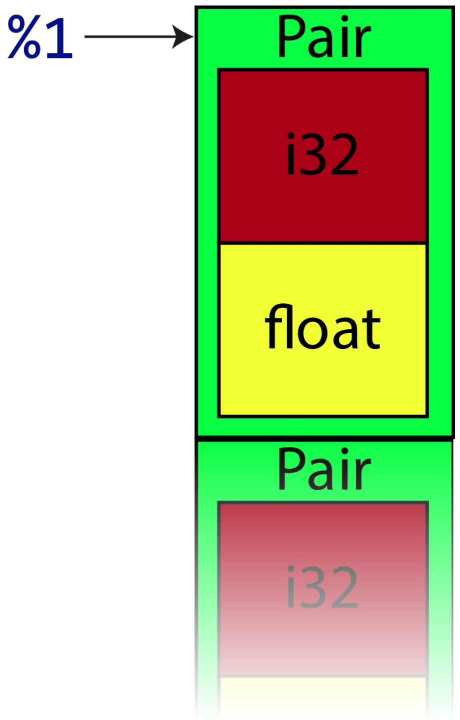
```
%<ptr> = getelementptr <type>, <type>* %<ptr>, { <type> <idx> }
```

- Used to apply pointer offsets
- Used for indexing structures and arrays
- Index constant for structures

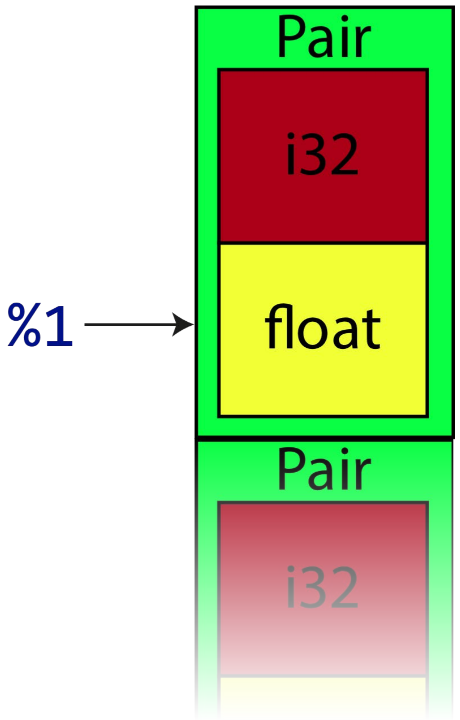
```
%Pair = type { i32, float }

define float @second(%Pair* %value) {
    %1 = getelementptr %Pair, %Pair* %value, i32 0, i32 1
    %2 = load float, float* %1
    ret float %2
}
```

Offset via: `%Pair*`
Using: `i32 0`
Result type: `%Pair*`



Indexing into: `%Pair`
Using: `i32 1`
Result type: `float*`



- First index pointer offset
- Subsequent indices move within the element type (array or struct)
  - Struct indices must be constant
- Returns pointer to resulting element
- **Does not load! Just pointer arithmetic**

LLVM IR Language – Integer arithmetic

`%res = add <type> <op1>, <op2>`	Addition
`%res = sub <type> <op1>, <op2>`	Subtraction
`%res = mul <type> <op1>, <op2>`	Multiply
`%res = sdiv <type> <op1>, <op2>`	Signed divide
`%res = udiv <type> <op1>, <op2>`	Unsigned divide
`%res = srem <type> <op1>, <op2>`	Signed remainder
`%res = urem <type> <op1>, <op2>`	Unsigned remainder
`%res = trunc <type> <op> to <type>`	Truncate integer value
`%res = zext <type> <op> to <type>`	Zero extend integer value
`%res = sext <type> <op> to <type>`	Sign extend integer value

LLVM IR Language – Integer arithmetic

- Two's complement arithmetic
- UB on Overflow configurable:

`%res = add i32 %x, %y`	Overflow with wraparound semantics
`%res = add nsw i32 %x, %y`	Signed Overflow is UB
`%res = add nuw i32 %x, %y`	Unsigned Overflow is UB
`%res = add nsw nuw i32 %x, %y`	Signed Overflow & Unsigned Overflow is UB

LLVM IR Language – Floating point arithmetic

`%res = fadd <type> <op1>, <op2>`	Addition
`%res = fsub <type> <op1>, <op2>`	Subtraction
`%res = fmul <type> <op1>, <op2>`	Multiply
`%res = fdiv <type> <op1>, <op2>`	Division
`%res = frem <type> <op1>, <op2>`	Remainder
`%res = fptrunc <type> <op> to <type>`	Truncate floating point
`%res = fpext <type> <op> to <type>`	Extend floating point
`%res = fptoui <type> <op> to <type>`	Floating point to unsigned int
`%res = fptosi <type> <op> to <type>`	Floating point to signed int
`%res = uitofp <type> <op> to <type>`	Unsigned int to floating point
`%res = sitofp <type> <op> to <type>`	Signed int to floating point

LLVM IR Language - Comparison

```
%b = icmp <pred> %x, %y
%b = fcmp <pred> %x, %y
```

- Produce `i1` results

Integer comparison predicates

eq	Equal
ne	Not equal
(u|s)gt	Greater
(u|s)ge	Greater-equal
(u|s)lt	Less
(u|s)le	Less-equal

Floating point comparison predicates

(o|u)eq	Equal
(o|u)ne	Not equal
(o|u)gt	Greater
(o|u)ge	Greater-equal
(o|u)lt	Less
(o|u)le	Less-equal

- (u) unsigned integer
- (s) signed integer

- (u) unordered floating point comp, supports NaN values
- (o) ordered floating point cmp, no NaN support

LLVM IR Language – `call`

```
[%res =] call <type> <fptr>([<type> <arg>{,<type> <arg>}])
```

- Calls function pointer with given arguments
- Produces 0 or 1 results

LLVM IR Language – Terminator instr.

- At end of every Basic Block
- Deems successors and predecessors

`ret <type> <value>`	Return with value
`ret void`	Return from void function
`br label <dest>`	Unconditional branch
`br i1 <cond>, label <true>, label <false>`	Conditional branch
`unreachable`	Unreachable code
`switch <type> <value>, label <default>` `{ <type> <val>, label <dest> }`	Switch/jump table

DEMO

Thank you for your attention!