LLVM

15-411/15-611 Compiler Design

Seth Copen Goldstein

February 26, 2026

Why LLVM?

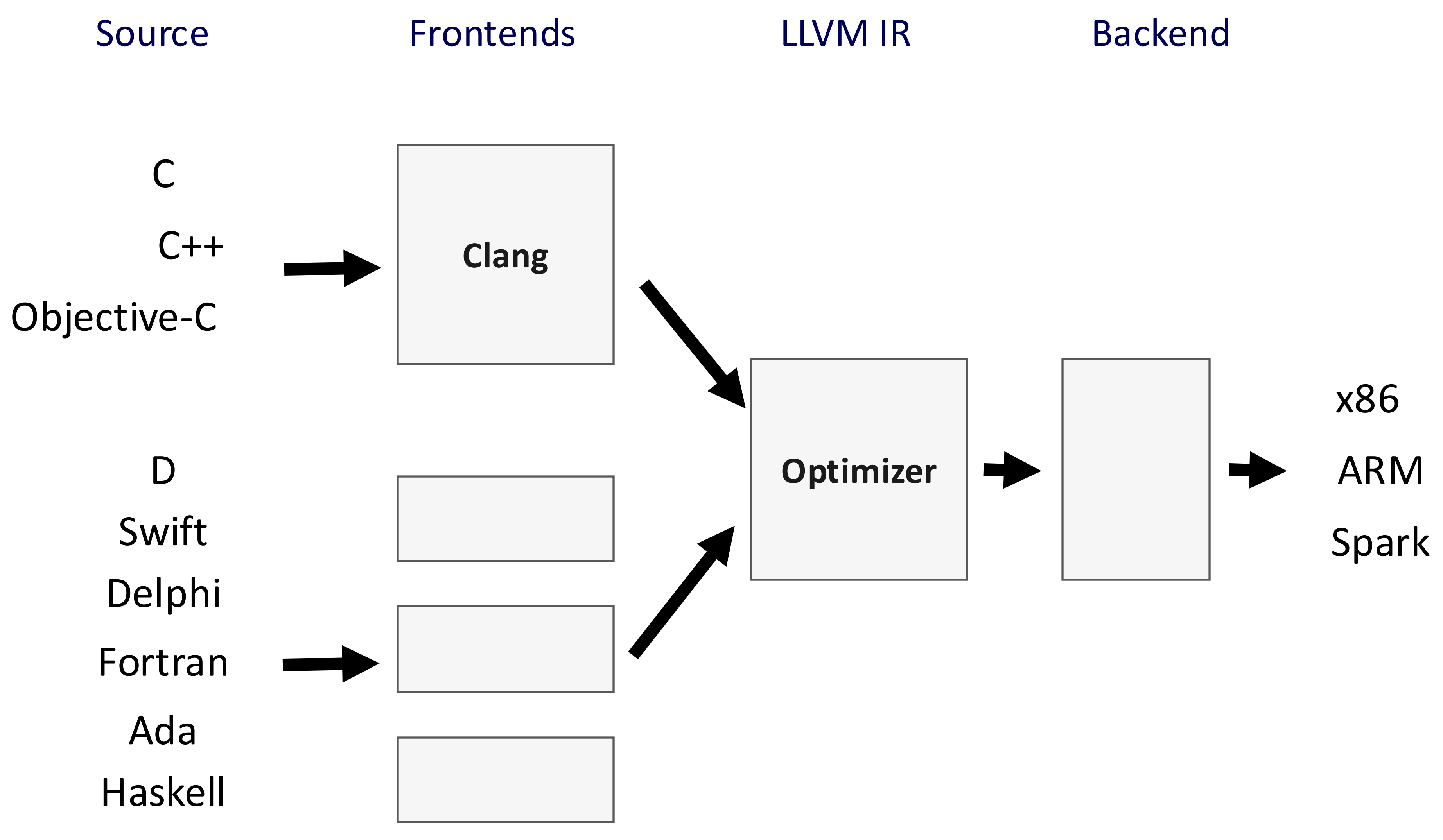
- Reusable compiler infrastructure.
- Separates frontend, optimization, and backend.
- Language-independent IR.
- Enables modular optimization passes.

A Brief History of LLVM

- Started in 2000 (Chris Lattner, UIUC).
- Originally stood for "Low Level Virtual Machine."
- Clang replaced GCC in many ecosystems.
  - Uses LLVM Core
- Now used by Rust, Swift, …

LLVM Architecture

- Frontend lowers source language to LLVM IR.
- Middle-end runs SSA-based optimizations.
- Backend lowers IR to machine code.
- IR is the contract between phases.



LLVM Compiler Framework

What is LLVM IR?

- Typed, SSA-based intermediate representation.
- Infinite virtual registers.
- Explicit control-flow graph.
- Textual (.ll) and bitcode (.bc) formats.

Structure of a .ll File

- A file defines a module.
- Contains
  - target info
  - globals
  - Declarations
  - definitions.

```
target triple = "x86_64-unknown-linux-gnu"

declare i32 @printf(ptr, ...)

define i32 @main() {
entry:
  ret i32 0
}
```

Example 1

Functions are parametrized with arguments and types.

```
int add (int x) {
  int y = 8128;
  return x+y;    }
```

```
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Local vars can be allocated on the stack or in temps.

Instructions have types: i32 is for 32bit integers.

No signed wrap: result of overflow undefined.

SSA: Single Assignment

- Each register assigned exactly once.
- New computation → new name.

```
define i32 @example(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  %double = add i32 %sum, %sum
  ret i32 %double
}
```

Declaring a Function

- Use 'declare' for external functions.
- No body provided.
- Types must match at call sites.

```
declare i32 @puts(ptr)
```

Defining a Function

- Use 'define' to provide implementation.
- Body consists of basic blocks.
- Must end in a terminator.

```
define i32 @add(i32 %x, i32 %y) {
entry:
  %r = add i32 %x, %y
  ret i32 %r
}
```

Basic Blocks

- Straight-line sequence of instructions.
- Begins with a label.
- Ends with exactly one terminator.

```
entry:
  %x = add i32 1, 2
  ret i32 %x
```

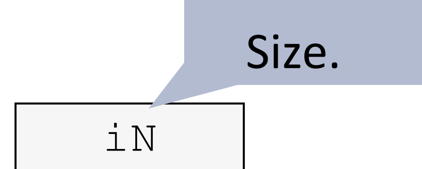
Terminator Instructions

- Every block must end with one terminator.
- Examples: **`ret`**, **`br`**, **`switch`**, **`unreachable`**.
- No instructions allowed after terminator.

LLVM Type System

- Strongly and explicitly typed.
- Primitive types: **`i1`**, **`i32`**, **`i64`**, **`float`**, **`double`**.
- Composite types: arrays, structs, functions.
- No implicit conversions.

Single Value Types



i1	a single-bit integer.
i32	a 32-bit integer.
i1942652	a really big integer of over 1 million bits.

Float Types

half	16-bit floating point value
float	32-bit floating point value
double	64-bit floating point value

Functions and Void

Void

No representation and no size.

```
void
```

Function Types

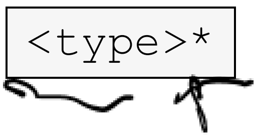
```
<returntype> (<parameter list>)
```

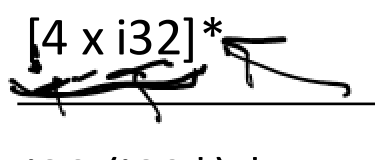
i32 (i32)	function taking an i32, returning an i32
float (i16, i32 *) *	Pointer to a function that takes an i16 and a pointer to i32, returning float.
i32 (i8*, ...)	A vararg function that takes at least one pointer to i8 (char in C), which returns an integer

Pointers and Vectors

Pointer Types

<type>*





[4 x i32]*	A pointer to array of four i32 values.
i32 (i32*) *	A pointer to a function that takes an i32*, returning an i32.

Vectors

< <# elements> x <elementtype> >

<4 x i32>	Vector of 4 32-bit integer values.
<8 x float>	Vector of 8 32-bit floating-point values.
<2 x i64>	Vector of 2 64-bit integer values.

Arrays and Structs

Arrays Types

```
[<# elements> x <elementtype>]
```

[40 x i32]	Array of 40 32-bit integer values.
[12 x [10 x float]]	12x10 array of single precision floating point values.
[2 x [3 x [4 x i16]]]	2x3x4 array of 16-bit integer values.

Struct Types

```
%T1 = type { <type list> }      ; Normal struct type
%T2 = type <{ <type list> }>    ; Packed struct type
```

{ i32, i32, i32 }	A triple of three i32 values
{ float, i32 (i32) *}	A pair, where the first elem. is a float and the second element is a pointer to a function that takes an i32, returning an i32.
<{ i8, i32 }>	A packed struct has no alignment or padding

Memory vs Registers

- Registers are SSA values.
- Memory accessed via alloca, load, store.

```
%p = alloca i32
store i32 5, ptr %p
%v = load i32, ptr %p
```

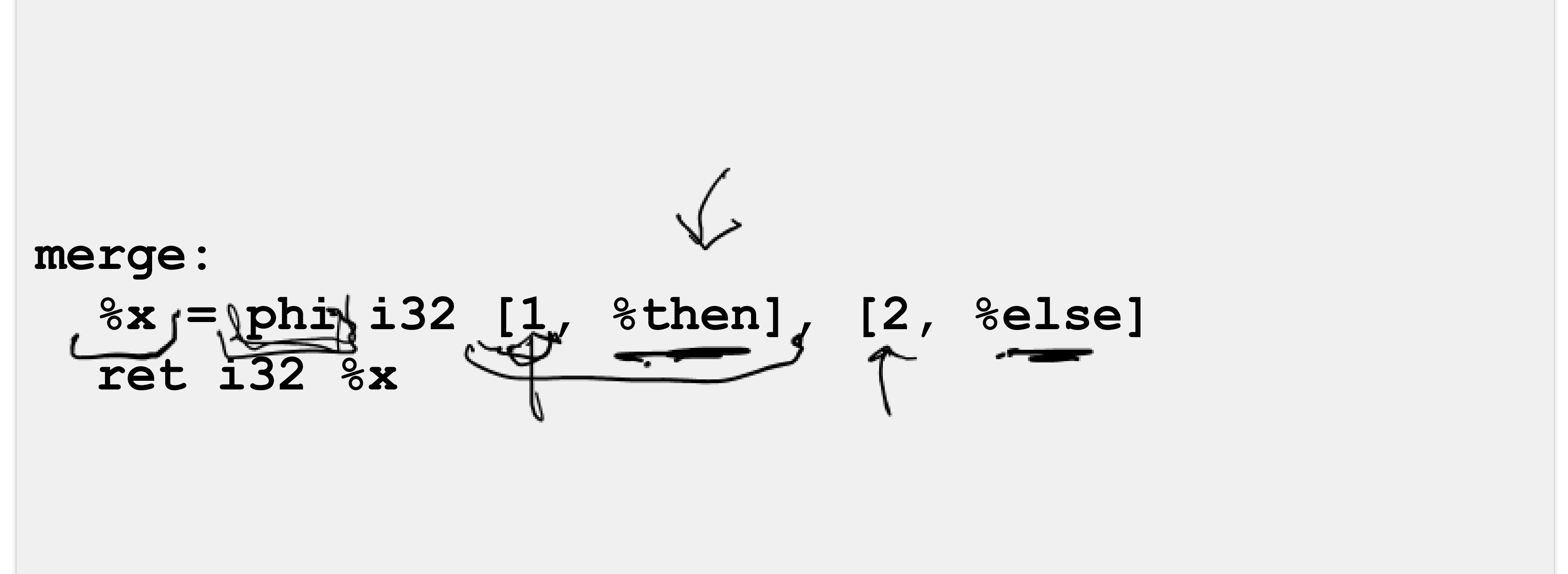
Control Flow in LLVM

- Explicit via branch instructions.
- No fall-through semantics.
- CFG structure is explicit.

```
%cmp = icmp sgt i32 %a, %b
br i1 %cmp, label %then, label %else
```

Phi Nodes

- Phi selects value based on predecessor.
- One incoming value per predecessor.
- Must appear at top of block.
- Types must match



If Example

```
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %btrue, label %bfalse

btrue:
  br label %end

bfalse:
  br label %end

end:
  ; If we came from %btrue, use %a. If from %bfalse, use %b.
  %retval = phi i32 [%a, %btrue], [%b, %bfalse]
  ret i32 %retval
```

Loops in LLVM

- Loops are back-edges in CFG.
- Loop variables require phi nodes.

```
loop:
  %i = phi i32 [%n, %entry], [%next, %loop]
  %next = sub i32 %i, 1
  %cond = icmp eq i32 %i, 0
  br i1 %cond, label %exit, label %loop
```

Compiling .ll Files

```
llvm-as file.ll -o file.bc
llc file.ll -filetype=obj
clang file.ll -o program
```

Omit **`-filetype`** if you want .s

Running Optimization Passes

```
opt -passes="instcombine,simplifycfg" file.ll -S -o out.ll
```

LLVM Analysis Passes

Basic-Block Vectorization
Profile Guided Block Placement
Break critical edges in CFG
Merge Duplicate Global
Simple constant propagation
Dead Code Elimination
Dead Argument Elimination
Dead Type Elimination
Dead Instruction Elimination
Dead Store Elimination
Deduce function attributes
Dead Global Elimination
Global Variable Optimizer
Global Value Numbering

Verifying the IR

```
opt -S passes=verify file.ll
```

Lookout for

- Missing terminator instruction.
- Type mismatch (i32 vs i64).
- Phi missing predecessor entry.
- Using undefined value.
- Incorrect block label.

Enjoy Spring Break!