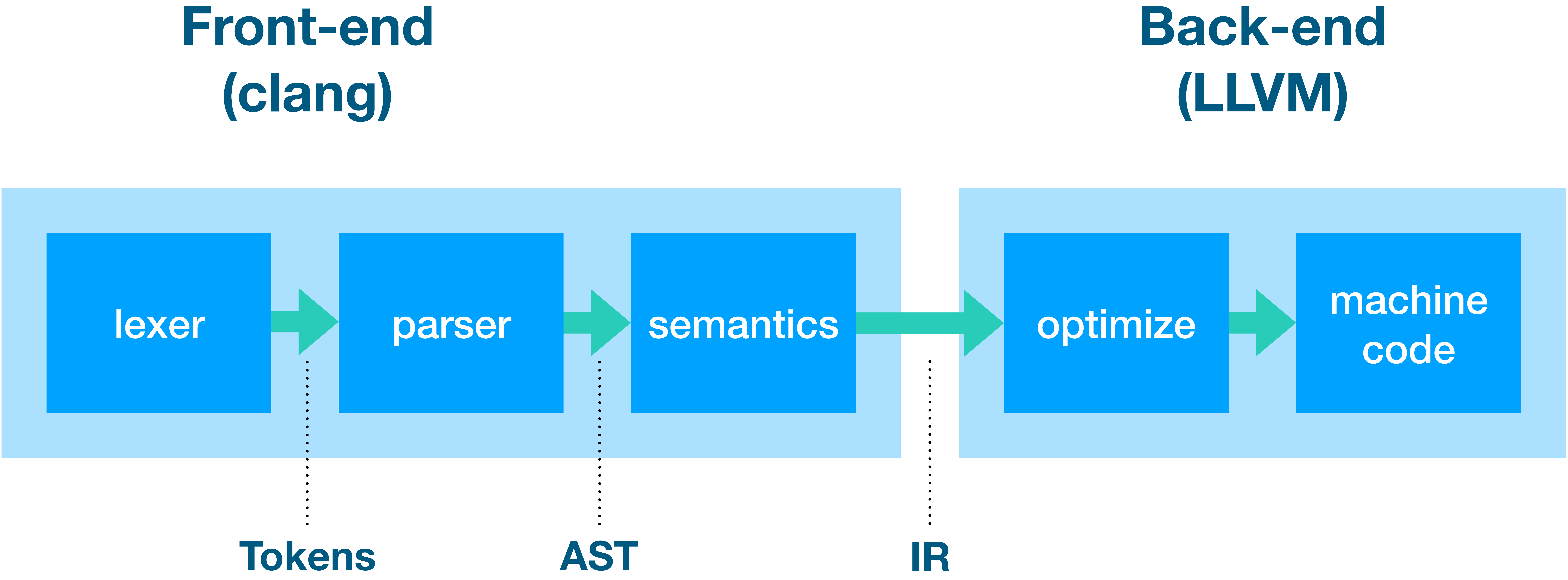
Dynamic Code Generation with LLVM

Jeff Cohen — ACCU 12/12/2018

What is LLVM?

Anatomy of a compiler



Front-end
(clang)

Back-end
(LLVM)

lexer

parser

semantics

optimize

machine code

Tokens

AST

IR

Intermediate Representation

- What the front-end generates and passes to LLVM.
- It is language-independent.
- It is *relatively* close to machine code, but independent of any specific machine architecture.
  - But it is **not** portable! Data sizes are explicit in IR, for example.
- It supports control-flow analysis, data-flow analysis, pointer aliasing analysis, and a whole bunch of other analyses in support of optimizations.

clang -O2 -S -emit-llvm fibo.c

```
int fibo(int arg) {
    return arg < 2 ? arg : fibo(arg-1) + fibo(arg-2);
}

; ModuleID = 'fibo.c'
source_filename = "fibo.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.14.0"

; Function Attrs: nounwind readnone ssp uwtable
define i32 @fibo(i32) local_unnamed_addr #0 {
  %2 = icmp slt i32 %0, 2
  br i1 %2, label %9, label %3

; <label>:3:                                  ; preds = %1
  %4 = add nsw i32 %0, -1
  %5 = tail call i32 @fibo(i32 %4)
  %6 = add nsw i32 %0, -2
  %7 = tail call i32 @fibo(i32 %6)
  %8 = add nsw i32 %7, %5
  ret i32 %8

; <label>:9:                                  ; preds = %1
  ret i32 %0
}
```

```
attributes #0 = { nounwind readnone ssp uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-
tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-
leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-
math"="false" "no-trapping-math"="false" "stack-
protector-buffer-size"="8" "target-cpu"="penryn" "target-
features"="+cx16,+fxsr,+mmx,+sahf,+sse,
+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"Apple LLVM version 10.0.0 (clang-1000.10.44.4)"}
```

LLVM Overview

- The top-most data structure is the context. It contains modules and types.
- A module contains functions, global variables, and external references.
- A function contains one or more basic blocks.
- A basic block contains a sequence of instructions that is entered only at the top and exits only at the bottom.
- An instruction consumes zero or more values and may produce a value itself.
- Values are typed! Instruction opcodes are overloaded on the types of their arguments.
- Types include the usual scalar data types, plus pointers, structs, and arrays.

Where can a value be used?

- Within a basic block, obviously only after the instruction that produces it.
- But can that value be used in a different basic block?
- Yes, but only if the execution of the producing instruction *dominates* the execution of the consuming instruction. In other words, it is impossible for the consuming instruction to execute without the producing instruction executing first.
- This is known as Static Single Assignment. (Think of functional programming languages with their immutable variables.)
- Values produced in divergent paths of execution can be merged into a single value via a phi instruction at the start of the basic block where control flow merges.
- Or you can use temp variables to pass values from one block to another.

Summary

- LLVM is powerful, but surprisingly easy to use.
- The On-Request Compiler (ORC) is highly modular and subdivided into layers, each of which are customizable or replaceable. Additional layers of your own creation can be inserted.
- There are many ways of using ORC. This calculator example shows but one way. The LLVM Kaleidoscope JIT tutorial demonstrates other features, such as lazy, on-demand compilation and remote compilation.

Links

- http://llvm.org
  - https://llvm.org/docs/GettingStarted.html
  - https://llvm.org/docs/LangRef.html
  - https://llvm.org/docs/tutorial/index.html
- https://llvm.org/devmtg/2016-11/Slides/Hames-ORC.pdf
  - 2016 ORCv1 Video: https://www.youtube.com/watch?v=hILdR8XRvdQ
  - 2018 Upcoming ORCv2: https://www.youtube.com/watch?v=MOQG5vkh9J8
- Source code: https://github.com/jeffc768/llvm-jit-example.git

Questions