中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第19讲：中间代码(1)

张献伟

xianweiz.github.io

DCS290, 4/25/2023

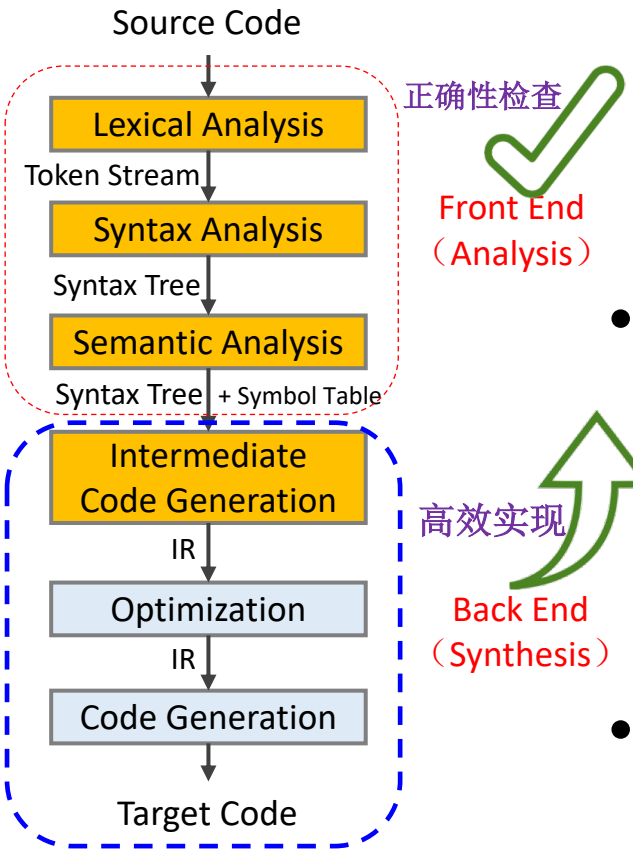
Quiz Questions

- Q1: how do perform semantic analysis using CFG?
  CFG + attributes/symbol + rules/production → SDD → rules embedded into the production body (action) → SDT.
- Q2: suppose it is L-SDD, can C.*c*, B.*b* and A.*a* be synthesized?
  A.*a* and C.*c* must be inherited. B.*b* can be either.
- Q3: convert the L-SDD into SDT.
  A → B {C.*c* = B.*b* + A.*a*} C

Production	Semantic Rule
A → BC	C.*c* = B.*b* + A.*a*

- Q4: for the above SDT, how to make it suitable for LR?
  Add marker and empty rule to move the action to the end of production rule, just likewise S-SDD.
- Q5: why symbol table is important for semantic analysis?
  To track symbols' info like name, type, value, scope, etc, for semantic analysis and afterwards code generation.

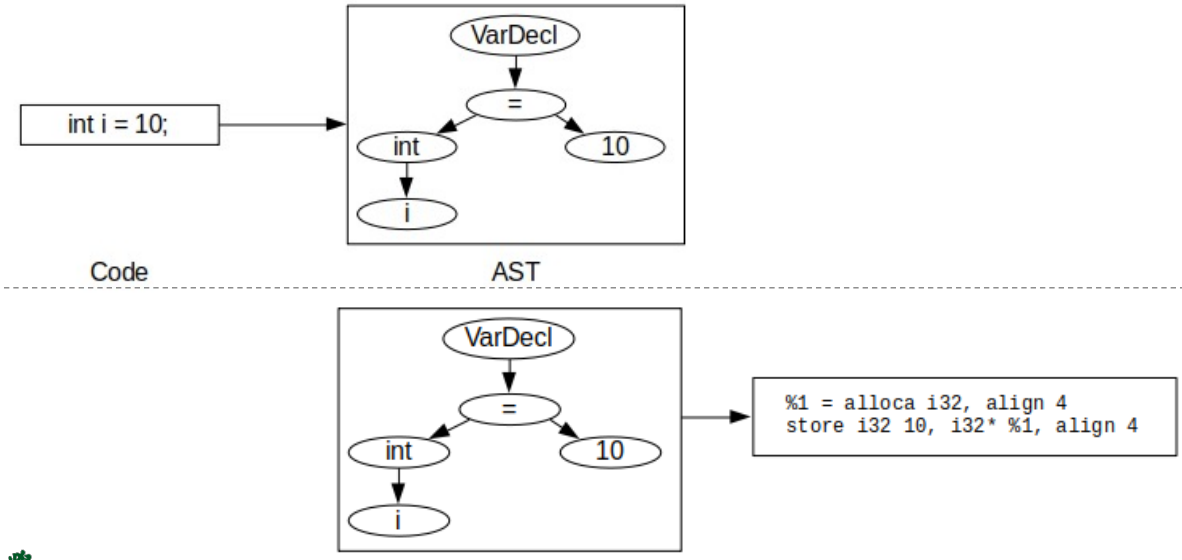
Compilation Phases[编译阶段]



- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
  - Is the program lexically well-formed?
    - E.g., x#y = 1
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
  - Is the input program syntactically well-formed?
    - E.g., for(i = 1)
- **Semantic**: AST → AST + symbol table
  - SDD, SDT, typing, scoping, …
  - Does the input program has a well-defined meaning?
    - E.g., int x; y = x(1)

Generating Code: AST to IR[IR生成]

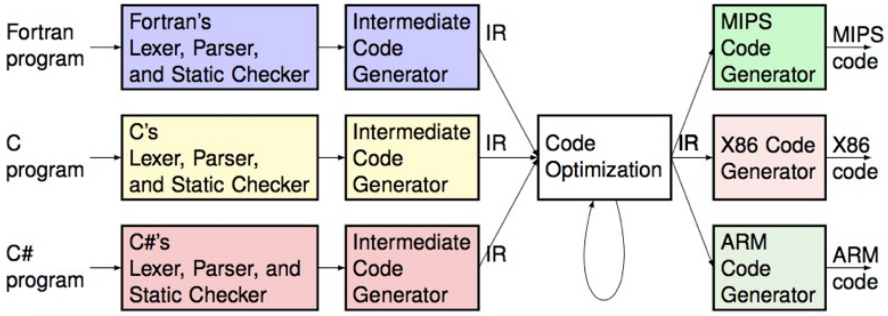
- By now, we have
  - An AST, annotated with scope and type information
- Next, to generate **intermediate representation** (IR)
  - Traversing the AST after the parse[单独遍历]
    - Writing a codeGen() method for the appropriate kinds of AST nodes
  - Syntax-directed translation[语法制导]
    - Generating code while parsing



http://freecompilercamp.org/clang-basics/

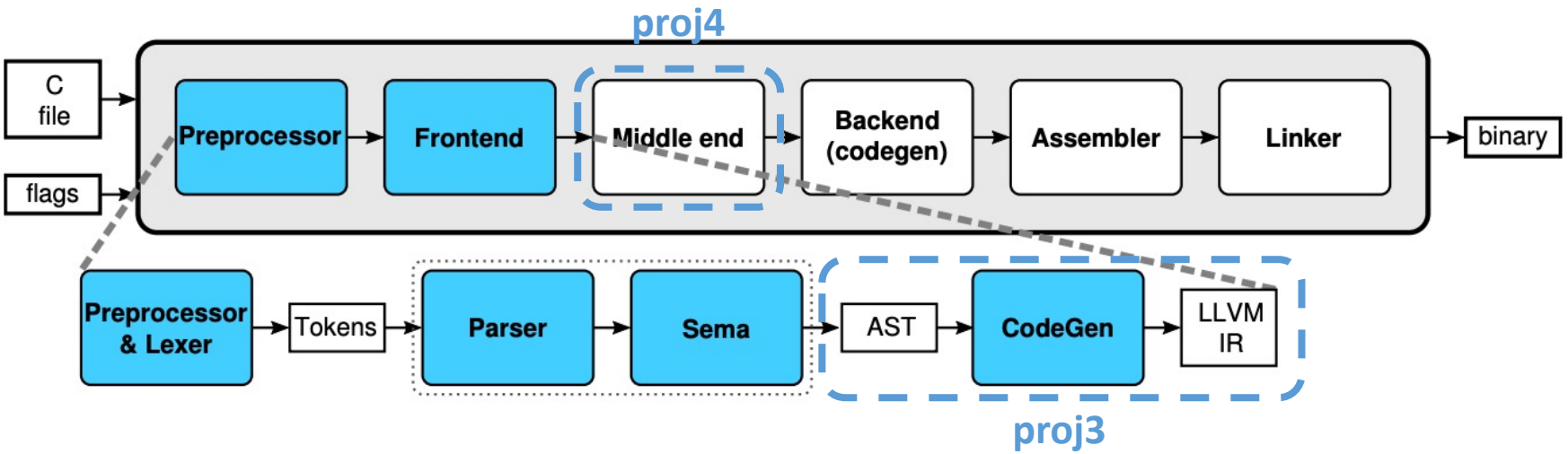
Modern Compilers[现代编译器]

- Compilation flow[三段式编译]
  - First, translate the source program to some form of intermediate representation (IR, 中间表示)
  - Then convert from there into machine code[机器代码]
- IR provides advantages[IR的优势]
  - Increased abstraction, cleaner separation, and retargeting, etc



m languages
n machines
O(*mn*) vs. O(*m*+*n*)

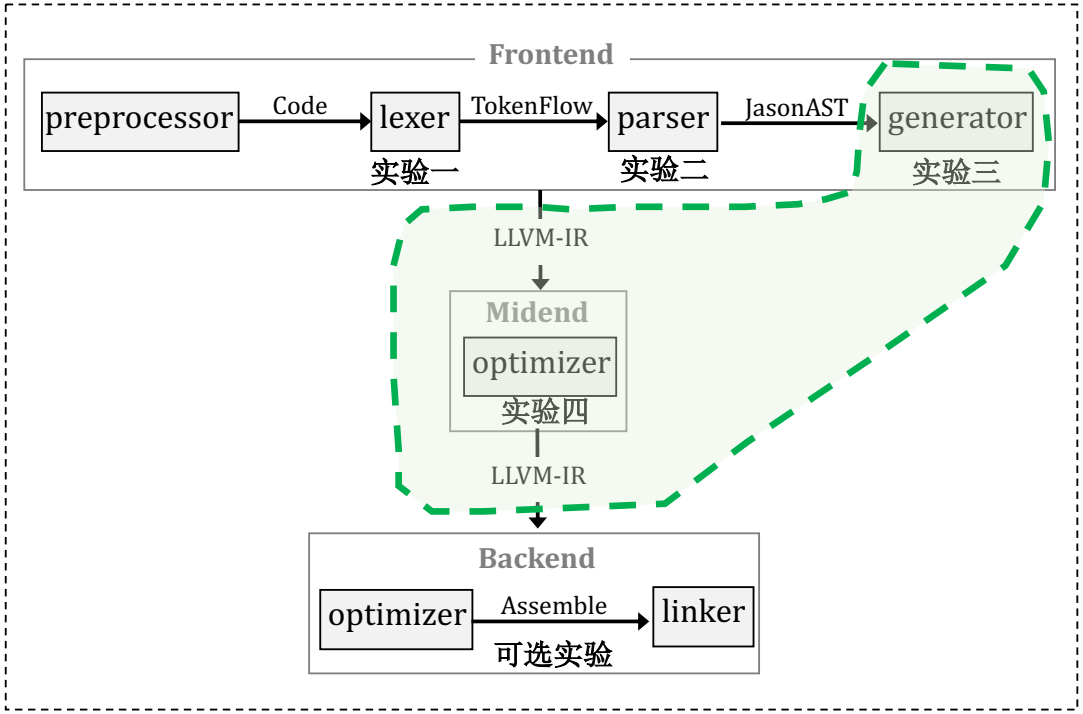
CodeGen[中间代码生成]

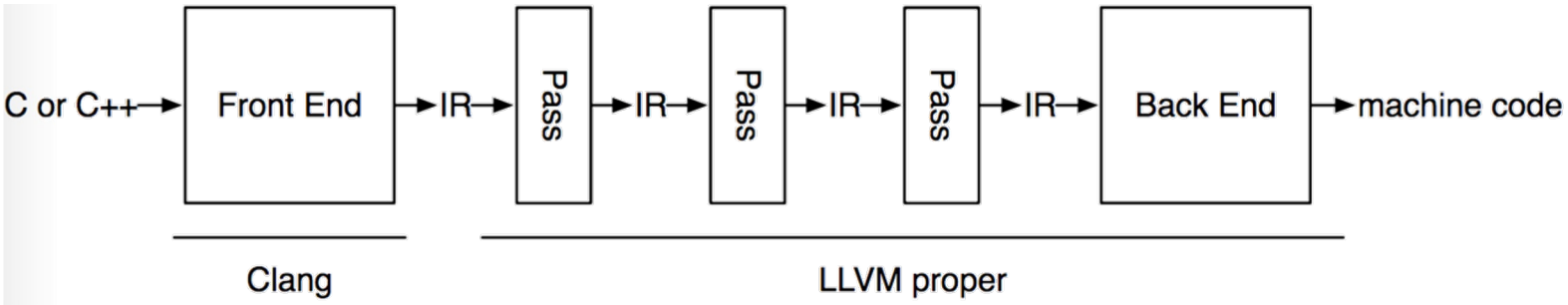


- Not to be confused with LLVM CodeGen! (which generates machine code)
- Uses AST visitors, IRBuilder, and TargetInfo
  - AST visitors
    - RecursiveASTVisitor for visiting the full AST
    - StmtVisitor for visiting Stmt and Expr
    - TypeVisitor for Type hierarchy

https://llvm.org/devmtg/2019-10/slides/ClangTutorial-Stulova-vanHaastregt.pdf
https://clang.llvm.org/docs/RAVFrontendAction.html

CodeGen (cont.)





src → AST: Example

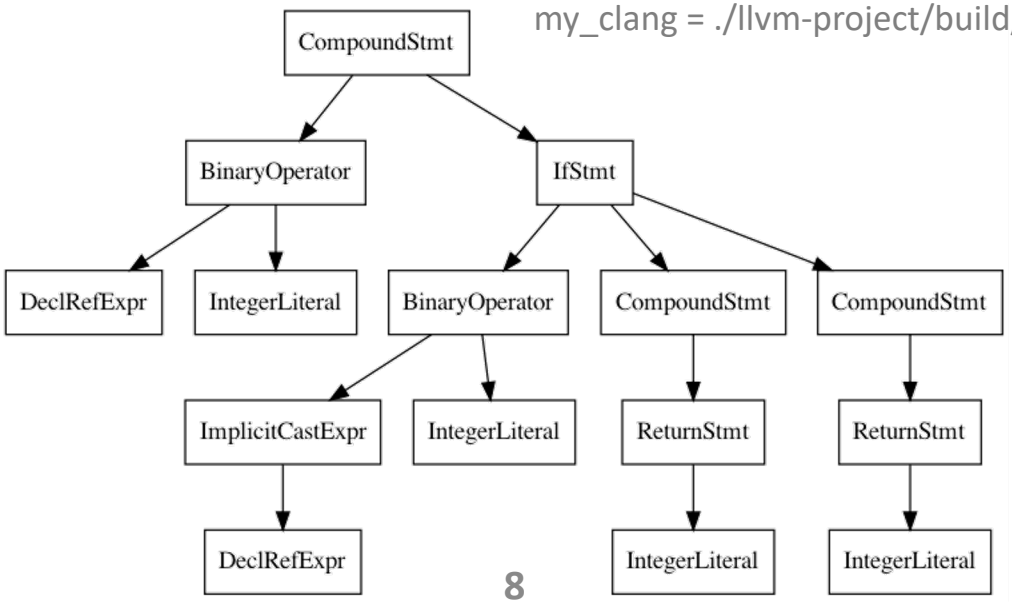
ir_test.c

```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

$<*my_clang*> -cc1 -ast-view ir_test.c
$dot -Tpng -o ir_test.png ir_test.dot

my_clang = ./llvm-project/build/bin/clang

AST



AST → IR: Example

$<*my_clang*> -Xclang -ast-dump -fsyntax-only ir_test.c

```
TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>
... cutting out internal declarations of clang ...
|-VarDecl 0x13800f670 <ir_test.c:1:1, col:5> col:5 used a 'int'
`-FunctionDecl 0x13800f778 <line:3:1, line:11:1> line:3:5 main 'int ()'
  `-CompoundStmt 0x13800f9a8 <col:12, line:11:1>
```

$<*my_clang*> -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx11.0.0"

@a = global i32 0, align 4

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  store i32 3, ptr @a, align 4
  %0 = load i32, ptr @a, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 1, ptr %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  store i32 0, ptr %retval, align 4
  br label %return

return:                                           ; preds = %if.else, %if.then
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "s
otprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+z

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"clang version 17.0.0 (https://github.com/llvm/llvm-project.git f759275c1c8ed91f19a6b8db228115c7f75d460b)"}
```

AST → IR: Example (cont.)

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx11.0.0"

@a = global i32 0, align 4

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  store i32 3, ptr @a, align 4
  %0 = load i32, ptr @a, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 1, ptr %retval, align 4
  br label %return

if.else:
  store i32 0, ptr %retval, align 4
  br label %return

return:
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "s
otprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+z

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"clang version 17.0.0 (https://github.com/llvm/llvm-project.git f759275c1c8ed91f19a6b8db228115c7f75d460b)"}
```

注释

源文件名

目标平台：数据布局[1], {endian, mangling, data align, native reg size, stack align}

目标平台：arch-vendor-os

全局变量定义：@<变量名> = <可见域> <类型> 初值, 4B对齐

函数定义：define <返回类型> @<函数名>（参数）#属性[2]

```
// allocate memory for a 32b value
// store 0 in the memory slot pointed by the register
// store 3 in the memory slot of 'a'
// do comparison
// branch

 ; preds = %entry     // if (a > 0) {
// store 1 into 'retval'
// jump to return

 ; preds = %entry     // } else {
// store 0 into 'retval'
// jump to return

 ; preds = %if.else, %if.then
// load 'retval'
// return
```

```
1  int a;
2
3  int main() {
4    a = 3;
5
6    if (a > 0) {
7      return 1;
8    } else {
9      return 0;
10   }
11 }
```

函数属性

模块级别元数据信息[3]

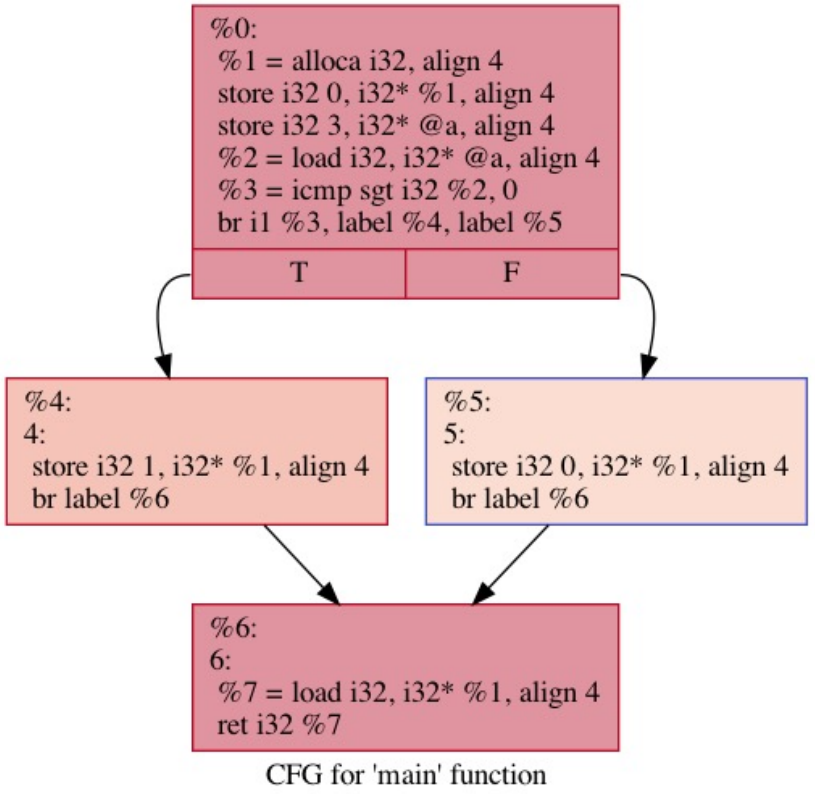
Clang版本信息

[1] https://llvm.org/docs/LangRef.html#data-layout
[2] https://llvm.org/docs/LangRef.html#function-attributes
[3] LLVM之IR 篇（1）：零基础快速入门 LLVM IR

AST → IR: Example (cont.)

$clang -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128
target triple = "aarch64-unknown-linux-gnu"

@a = dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 3, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:                                                ; preds = %0
  store i32 1, i32* %1, align 4
  br label %6

5:                                                ; preds = %0
  store i32 0, i32* %1, align 4
  br label %6

6:                                                ; preds = %5, %4
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt
" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="fa
features"="+neon" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"Debian clang version 11.0.1-2"}
```



CFG for 'main' function

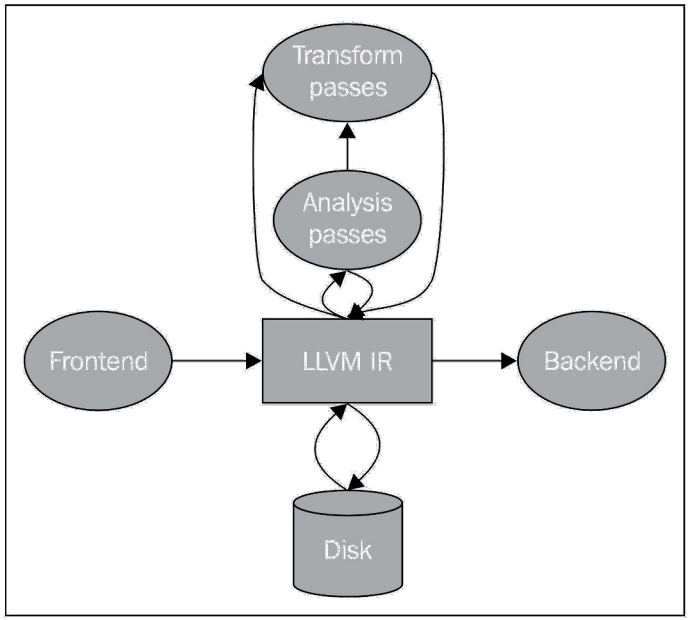
$dot -Tpng -o ir_test.png .ir_test.dot

$opt -dot-cfg ir_test.ll

IR Forms

- Three different forms (these three forms are equivalent)
  - In-memory compiler IR [在内存中的编译中间语言]
  - On-disk bitcode file [.bc, 在硬盘上存储的二进制中间语言]
  - Human readable plain text file [.ll, 人类可读的代码语言]

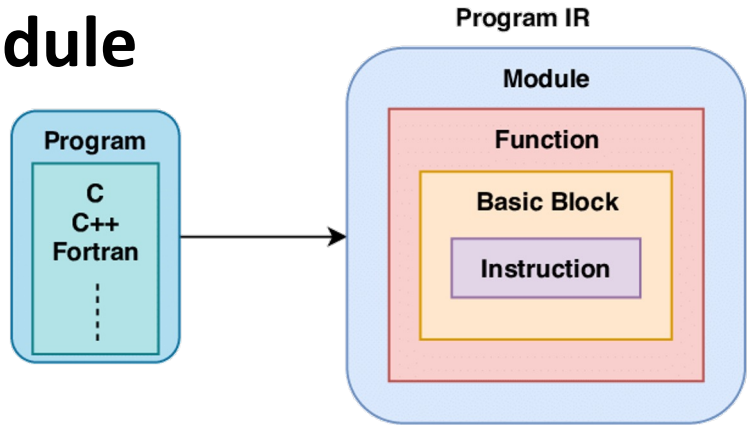
- Translate to bitcode file[2]: $llvm-as *.ll [-o *.bc]
  - Reverse: $llvm-dis *.bc -o *.ll
  - Further compile the bitcode[3]:
    - $llc -march=x86 *.bc -o out.x86
- Execute the IR file[1]: $lli *.ll
  - Result: $echo $?

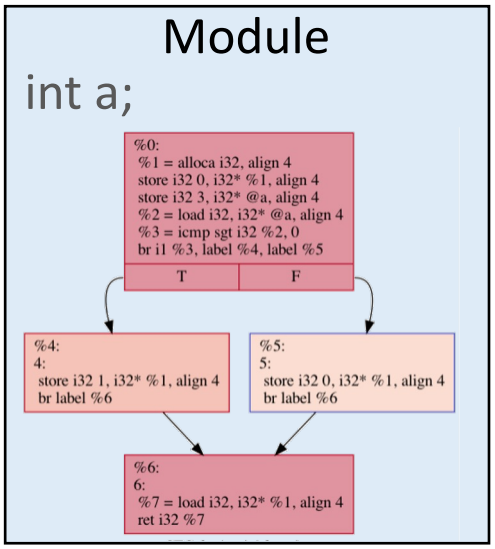


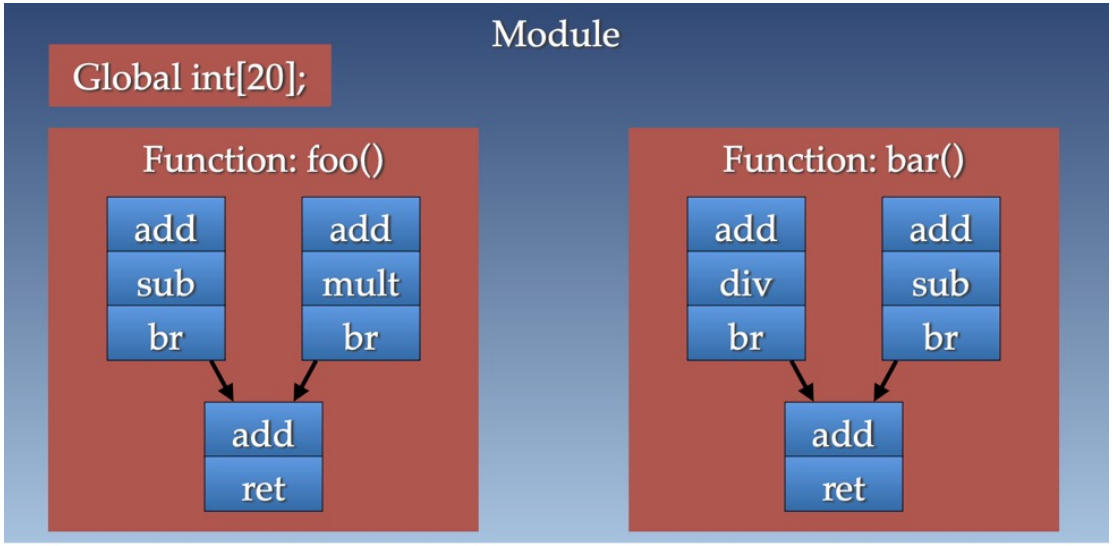
[1] https://www.llvm.org/docs/CommandGuide/lli.html
[2] https://www.llvm.org/docs/CommandGuide/llvm-as.html
[3] https://www.llvm.org/docs/CommandGuide/llc.html

IR Overview

- Each assembly/bitcode file is a **Module**
- Each Module is comprised of
  - Global variables
  - A set of **Function**s which consists of
    - A set of **Basic Blocks**
      - Which is further comprised of a set of **Instructions**







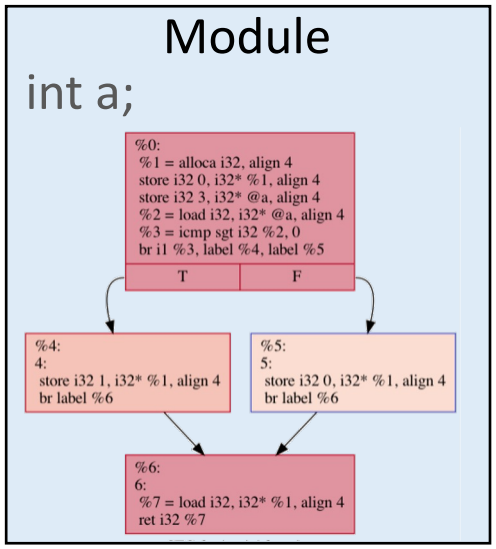
https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4_copy.pdf

IR Overview (cont.)

- LLVM IR resembles **three-address code** (TAC)
  - Two source operands, one separate destination operand
  - **Static Single Assignment** (SSA) form, making life easier for optimization writers[静态单赋值]
    - >. SSA means we define variables before use and assign to variables only once
- LLVM IR is machine independent[机器无关]
  - An unlimited set of virtual registers (labelled %0, %1, %2, …)
    - >. It's the backend's job to map from virtual to physical registers
  - Rather than allocating specific sizes of datatypes, we retain types
    - >. Again, the backend will take this type info and map it to platform's datatype

LLVM Steps

- Context: llvm:LLVMContext TheContext
- Module: llvm::Module TheModule("-", TheContext);
  - Function: auto function = llvm::Function::Create(…, TheModule)
    - auto block = llvm::BasicBlock::Create(TheContext, "entry", function)
      - llvm::IRBuilder<> builder(block);
        - builder.CreateRet(tmp);




```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

```
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/IR/Type.h>
#include <llvm/IR/Verifier.h>
#include <llvm/Support/JSON.h>
#include <llvm/Support/MemoryBuffer.h>
#include <llvm/Support/raw_ostream.h>
```

Three-Address Code[三地址码]

- High-level assembly where each operation has **at most three** operands. Generic form is X = Y op Z[最多3个操作数]
  - where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values
- Characteristics[特性]
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps[控制流->跳转]
  - Machine independent
    - Operations are generic (not tailored to any specific machine)
    - Function calls represented as generic call nodes
    - Uses symbolic names rather than register names (actual locations of symbols are yet to be determined)
- Design goal: for easier machine-independent optimization

Example

- For example, x * y + x * y is translated to

  t1 = x * y ; t1, t2, t3 are temporary variables

  t2 = x * y

  t3 = t1 + t2

  - Can be generated through a depth-first traversal of AST
  - Internal nodes in AST are translated to temporary variables

- Notice: repetition of x * y[重复]
  - Can be later eliminated through a compiler optimization called common subexpression elimination (CSE)[通用子表达式消除]

    t1 = x * y

    t3 = t1 + t1

  - Using 3-address code rather than AST makes it:
    - Easier to spot opportunities (just find matching RHSs)
    - Easier to manipulate IR (AST is much more cumbersome)