中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compiler Design
编 译 器 构 造 实 验

Lab 8： Project-3

张献伟、吴坎、林泽佳

xianweiz.github.io

DCS292, 4/11/2023

Project 3: What?

- 文档描述：
  - Readme: https://github.com/arcsysu/SYsU-lang/tree/main/generator
  - Wiki: https://github.com/arcsysu/SYsU-lang/wiki/实验三代码生成
- 实现一个IR生成器
  - 输入：抽象语法树（由Project 2或Clang提供）
  - 输出：LLVM-IR（可以使用`lli`来运行）
- 总体流程
  - 引入Project2的parser（或使用clang）
  - 遍历得到的AST
  - 对各Function和Statement等生成IR代码
- 截止时间
  - **5/16/2023**

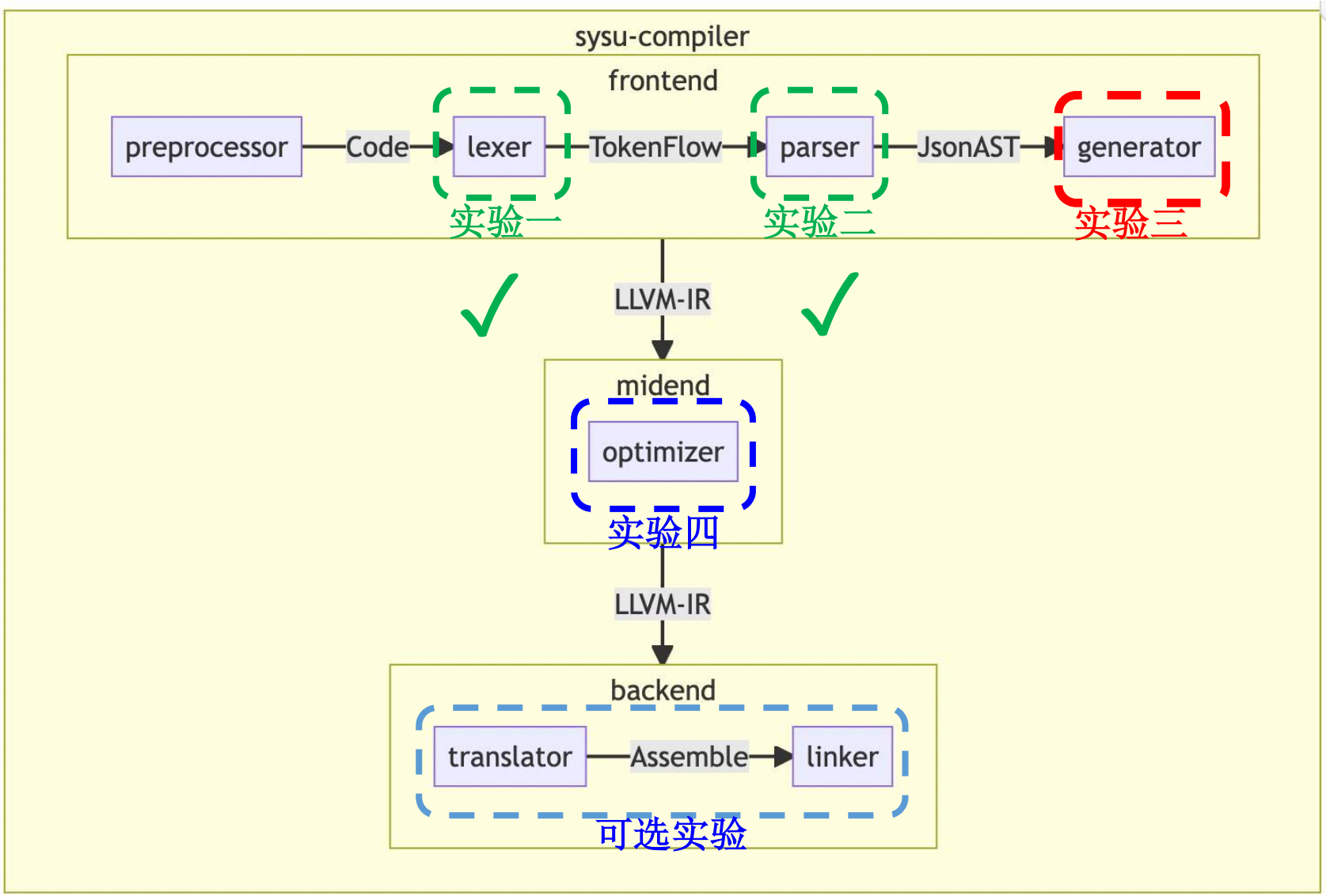
Project 3: How?

- 实现
  - $vim generator/generator.cc
- 编译
  - $cmake --build ~/sysu/build -t install
    - 输出：~/sysu/build/generator
- 运行

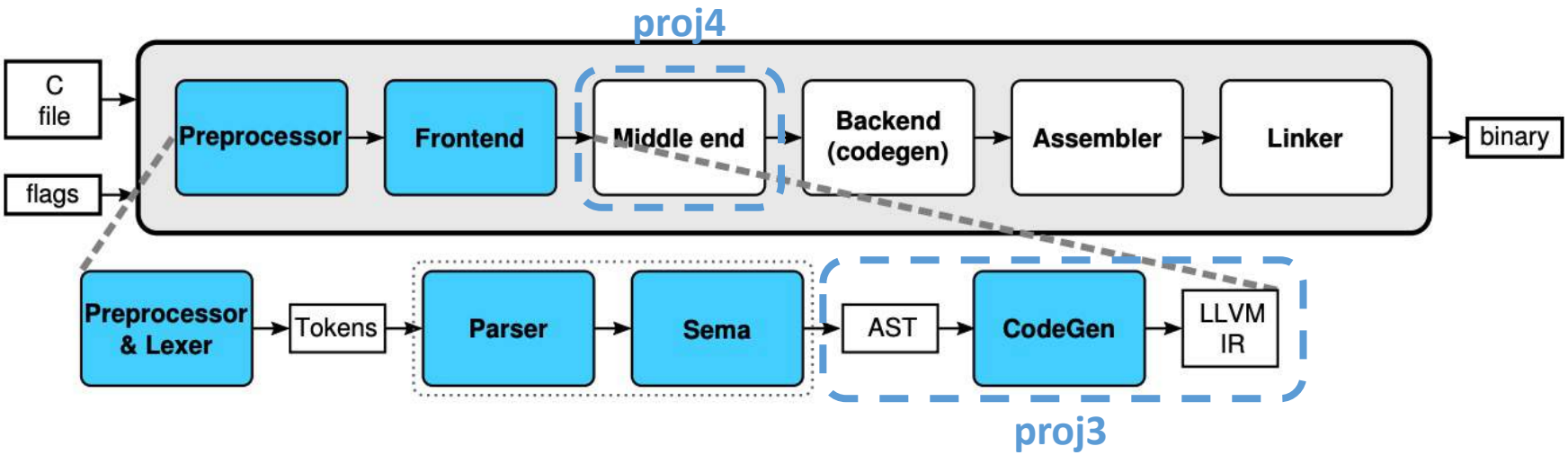
```
$( export PATH=~/sysu/bin:$PATH \
CPATH=~/sysu/include:$CPATH \
LIBRARY_PATH=~/sysu/lib:$LIBRARY_PATH \
LD_LIBRARY_PATH=~/sysu/lib:$LD_LIBRARY_PATH
&& clang -E tester/functional/000_main.sysu.c
| <THE_PARSER>
| sysu-generator )
```

  - Clang提供AST：<THE_PARSER> = clang -cc1 -ast-dump=json
  - Project2提供AST： <THE_ PARSER > = sysu-lexer | sysu-parser

Schedule[实验安排]



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

AST → IR: Example

$clang -Xclang -ast-dump -fsyntax-only ../tester/functional/000_main.sysu.c

```
TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>
                ... cutting out internal declarations of clang ...
`-FunctionDecl 0x2cf71448 <../tester/functional/000_main.sysu.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x2cf71560 <col:11, line:3:1>
    `-ReturnStmt 0x2cf71550 <line:2:5, col:12>
      `-IntegerLiteral 0x2cf71530 <col:12> 'int' 3
```

$clang -emit-llvm -S ../tester/functional/000_main.sysu.c

```
; ModuleID = '../tester/functional/000_main.sysu.c'
source_filename = "../tester/functional/000_main.sysu.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"

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 3
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt-fp-math"
="false" "disable-tail-calls"="false" "frame-pointer"="non-leaf" "less-precise-fpma
d"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"=
"false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-ma
th"="true" "stack-protector-buffer-size"="8" "target-cpu"="generic" "target-feature
s"="+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"}
```

AST → IR: Helloworld

Source

```
1  int main(){
2      return 3;
3  }
```

AST

```
TranslationUnitDecl 0xa8e6558 <<invalid sloc>> <invalid sloc>

-FunctionDecl 0xa942a10 <generator/000_main.sysu.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xa942b28 <col:11, line:3:1>
    `-ReturnStmt 0xa942b18 <line:2:5, col:12>
      `-IntegerLiteral 0xa942af8 <col:12> 'int' 3
```

IR

```
1  define dso_local i32 @main() {
2    %1 = alloca i32, align 4
3    store i32 0, ptr %1, align 4
4    ret i32 3
5  }
```

```
1  define dso_local i32 @main() {
2    ret i32 3
3  }
```

AST → IR: Local variable

Source

```
1  int main(){
2      int a = 3;
3      return a;
4  }
```

AST

```
TranslationUnitDecl 0xb7ae558 <<invalid sloc>> <invalid sloc>
        ... cutting out internal declarations of clang ...
`-FunctionDecl 0xb80abf8 <line:6:1, line:9:1> line:6:5 main 'int ()'
  `-CompoundStmt 0xb80ad98 <col:11, line:9:1>
    |-DeclStmt 0xb80ad38 <line:7:5, col:14>
    | `-VarDecl 0xb80acb0 <col:5, col:13> col:9 used a 'int' cinit
    |   `-IntegerLiteral 0xb80ad18 <col:13> 'int' 3
    `-ReturnStmt 0xb80ad88 <line:8:5, col:12>
      `-ImplicitCastExpr 0xb80ad70 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0xb80ad50 <col:12> 'int' lvalue Var 0xb80acb0 'a' 'int'
```

IR

```
1  define dso_local i32 @main() {
2    %1 = alloca i32, align 4
3    store i32 3, ptr %1, align 4
4    %2 = load i32, ptr %1, align 4
5    ret i32 %2
6  }
```

临时寄存器/变量：分配栈空间，地址存入%1，大小同i32类型，4B对齐

写内存：将0写入%1对应的内存中，4B对齐

读内存：将%1对应的内存中的数据读取到%2中

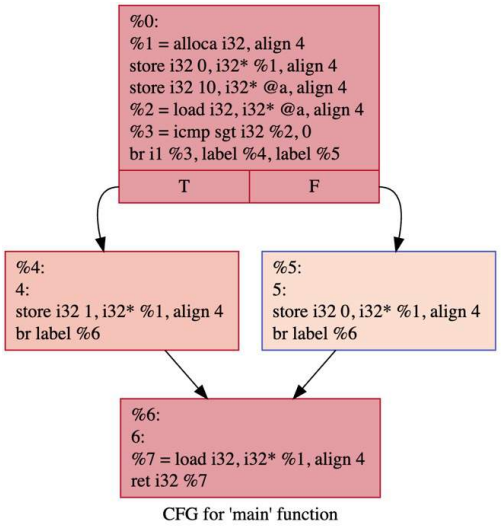
函数返回

AST → IR: Basic blocks

- $clang -emit-llvm -S ../tester/functional/027_if2.sysu.c
- Basic blocks[基本块]
  - Straight-line code sequence
  - No control flow divergence

```c
int a;
int main(){
    a = 10;
    if(a > 0){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 10, ptr @a, align 4
  %2 = load i32, ptr @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

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

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

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

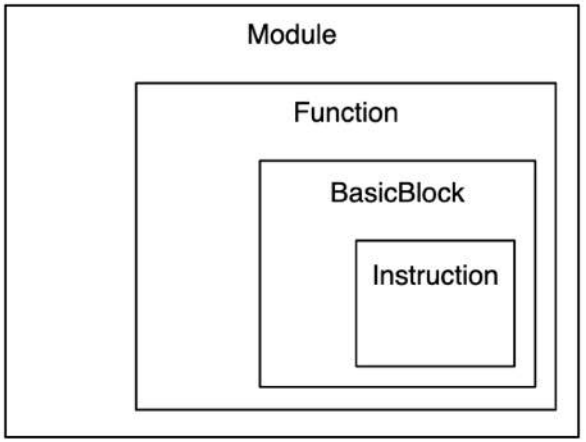


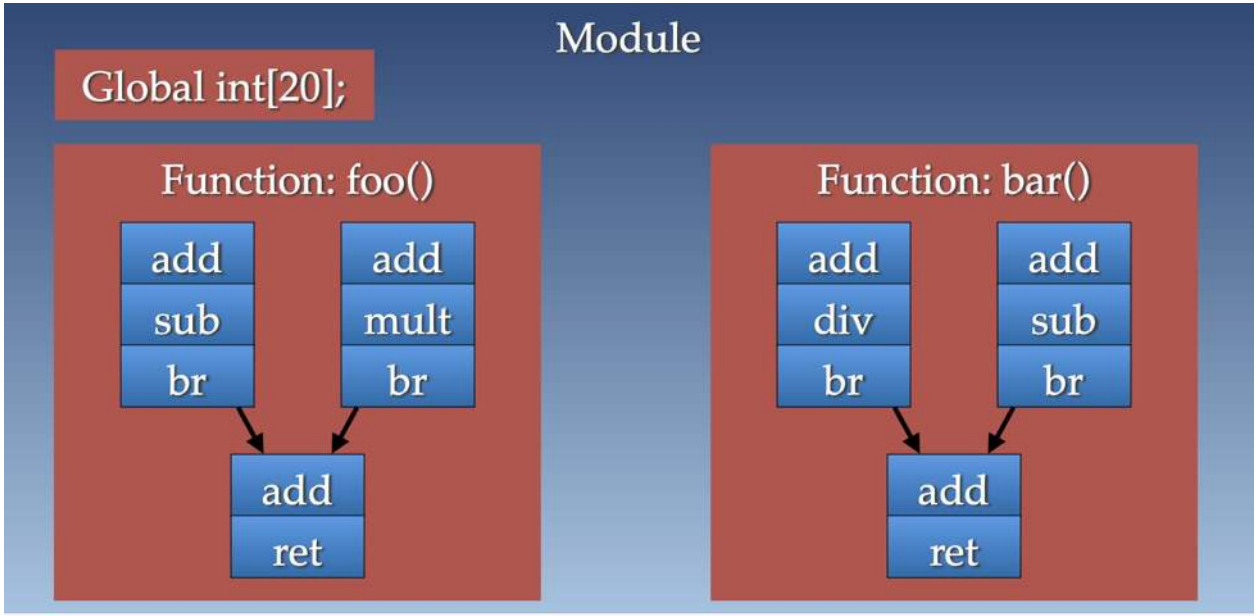
CFG for 'main' function

http://viz-js.com/

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 **Instruction**s





IR Overview (cont.)

```
1  ; ModuleID = 'generator/000_main.sysu.c'
2  source_filename = "generator/000_main.sysu.c"
3  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
4  target triple = "x86_64-unknown-linux-gnu"
5
6  @a = dso_local global i32 0, align 4
7
8  ; Function Attrs: noinline nounwind optnone uwtable
9  define dso_local i32 @main() #0 {…
25 }
26
27 attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
   "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8"
   "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
   "tune-cpu"="generic" }
28
29 !llvm.module.flags = !{!0, !1, !2, !3, !4}
30 !llvm.ident = !{!5}
31
32 !0 = !{i32 1, !"wchar_size", i32 4}
33 !1 = !{i32 7, !"PIC Level", i32 2}
34 !2 = !{i32 7, !"PIE Level", i32 2}
35 !3 = !{i32 7, !"uwtable", i32 2}
36 !4 = !{i32 7, !"frame-pointer", i32 2}
37 !5 = !{!"clang version 15.0.4"}
```

注释

源文件名

目标平台：数据布局[1]

目标平台：arch-vendor-os

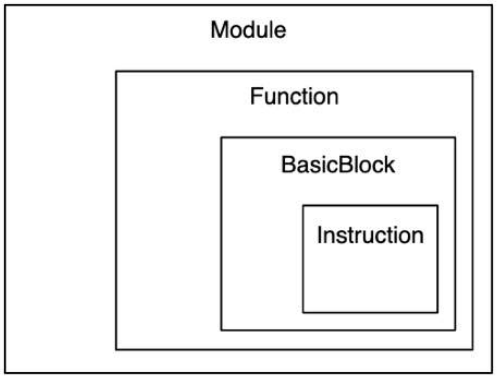
全局变量定义：@<变量名> = <可见域> <类型> 初值, 4B对齐

函数定义：define <返回类型> @<函数名>（参数）#属性[2]

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

IR Overview (cont.)

- 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]

- 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
  - 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

Workflow to Build a Function

LLVM内部数据结构，我们不直接操作它，只用将它作为参数传给需要的API

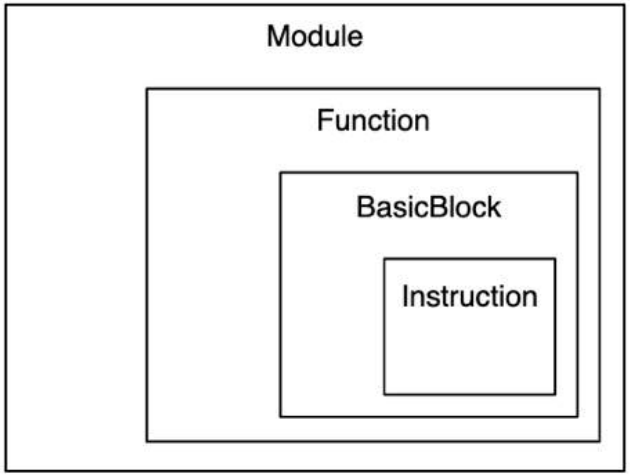
```
llvm::LLVMContext TheContext;
llvm::Module TheModule("helloworld", TheContext);

llvm::Function *buildFunctionDecl(Json json){
  auto function = llvm::Function::Create(/*...*/);
  auto BB = llvm::BasicBlock::Create(/*...*/);
  llvm::IRBuilder<> builder(BB);
  for(const auto &child: json["inner"]){
    buildStmt(&builder, child);
  }
}

void buildStmt(llvm::IRBuilder *builder, Json json){
  if(json["kind"] == "CompoundStmt"){
    // build compound statement
  } else if (json["kind"] == "ReturnStmt"){
    // build return statement
  } else if (json["kind"] == "SomeStmt"){
    // build some statement
  } else {
    // fallback
  }
}
```

创建Module

创建Function

创建Basic Block

使用IRBuilder来创建Instruction



Workflow to Build a Function

LLVM内部数据结构，我们不直接操作它，只用将它作为参数传给需要的API

```
llvm::LLVMContext TheContext;
llvm::Module TheModule("helloworld", TheContext);

llvm::Function *buildFunctionDecl(Json json){
  auto function = llvm::Function::Create(/*...*/);
  auto BB = llvm::BasicBlock::Create(/*...*/);
  llvm::IRBuilder<> builder(BB);
  for(const auto &child: json["inner"]){
    buildStmt(&builder, child);
  }
}
```

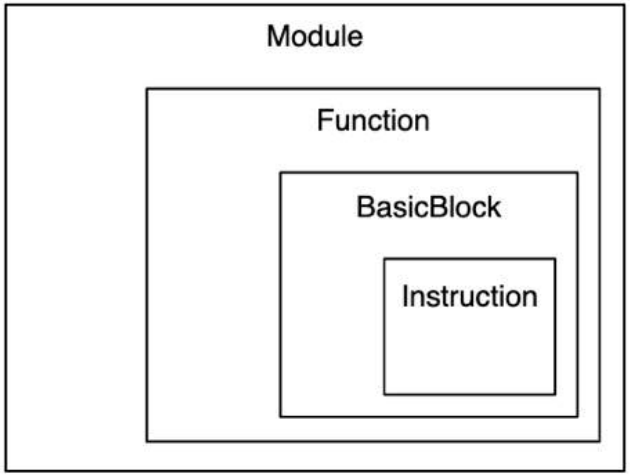
创建Module

创建Function

创建Basic Block

使用IRBuilder来创建Instruction

Module、Function和BasicBlock都是可以CRUD和遍历迭代的，并且拥有相应的父子关系




```
void buildStmt(llvm::IRBuilder *builder, Json json){
  if(json["kind"] == "CompoundStmt"){
    // build compound statement
  } else if (json["kind"] == "ReturnStmt"){
    // build return statement
  } else if (json["kind"] == "SomeStmt"){
    // build some statement
  } else {
    // fallback
  }
}
```

generator.cc

```
void buildTranslationUnitDecl(const llvm::json::Object *O) {
  if (O == nullptr)
    return;
  if (auto kind = O->get("kind")->getAsString()) {
    assert(*kind == "TranslationUnitDecl");
  } else {
    assert(0);
  }
  if (auto inner = O->getArray("inner"))
    for (const auto &it : *inner)
      if (auto P = it.getAsObject())
        if (auto kind = P->get("kind")->getAsString()) {
          if (*kind == "FunctionDecl")
            buildFunctionDecl(P);
        }
}
} // namespace

int main() {
  auto llvmin = llvm::MemoryBuffer::getFileOrSTDIN("-");
  auto json = llvm::json::parse(llvmin.get()->getBuffer());
  buildTranslationUnitDecl(json->getAsObject());
  TheModule.print(llvm::outs(), nullptr);
}
```

根节点

遍历内部节点

具体IR生成

从文件或stdin获取AST文本

解析为JSON格式

遍历AST，生成IR

输出IR

generator.cc (cont.)

```
llvm::LLVMContext TheContext;
llvm::Module TheModule("-", TheContext);

llvm::Function *buildFunctionDecl(const llvm::json::Object *O) {
  // First, check for an existing function from a previous declaration.
  auto TheName = O->get("name")->getAsString()->str();
  llvm::Function *TheFunction = TheModule.getFunction(TheName);

  if (!TheFunction)
    TheFunction = llvm::Function::Create(
        llvm::FunctionType::get(llvm::Type::getInt32Ty(TheContext), {}, false),
        llvm::Function::ExternalLinkage, TheName, &TheModule);

  if (!TheFunction)
    return nullptr;

  // Create a new basic block to start insertion into.
  auto BB = llvm::BasicBlock::Create(TheContext, "entry", TheFunction);
  llvm::IRBuilder<> Builder(BB);

  if (auto RetVal = llvm::ConstantInt::get(
          TheContext, /* i32 3(decimal) */ llvm::APInt(32, "3", 10))) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    llvm::verifyFunction(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

用于保存全局的状态，在多线程执行的时候，可以每个线程一个LLVMContext，避免竞争

LLVM IR程序的顶层结构

创建一个函数，并指派给Module

参数：类型

参数：链接方式、函数名、该函数待插入的模块
“ExternalLinkage”表示该函数可能定义于当前模块之外，且/或可以被当前模块之外的函数调用。

int main()

为创建的Function添加Basic Block

使用IRBuilder插入指令到BB

return 3

返回值指令语句

https://releases.llvm.org/11.0.1/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html
https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

https://github.com/arcsysu/SYsU-lang/blob/main/generator/generator.cc

Example: 027_if2.sysu.c

What we will learn from this example

- Global variable
- Variable assignment
- Build binary operation
- Build branch

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

```
TranslationUnitDecl 0xb712558 <<invalid sloc>> <invalid sloc>
|---------...cutting out internal declarations of clang...
|-VarDecl 0xb76ea10 <generator/000_main.sysu.c:1:1, col:5> col:5 used a 'int'
`-FunctionDecl 0xb76eb10 <line:2:1, line:10:1> line:2:5 main 'int ()'
  `-CompoundStmt 0xb76ed48 <col:11, line:10:1>
    |-BinaryOperator 0xb76ebf0 <line:3:2, col:6> 'int' '='
    | |-DeclRefExpr 0xb76ebb0 <col:2> 'int' lvalue Var 0xb76ea10 'a' 'int'
    | `-IntegerLiteral 0xb76ebd0 <col:6> 'int' 10
    `-IfStmt 0xb76ed18 <line:4:2, line:9:2> has_else
      |-BinaryOperator 0xb76ec68 <line:4:5, col:9> 'int' '>'
      | |-ImplicitCastExpr 0xb76ec50 <col:5> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0xb76ec10 <col:5> 'int' lvalue Var 0xb76ea10 'a' 'int'
      | `-IntegerLiteral 0xb76ec30 <col:9> 'int' 0
      |-CompoundStmt 0xb76ecb8 <col:11, line:6:2>
      | `-ReturnStmt 0xb76eca8 <line:5:3, col:10>
      |   `-IntegerLiteral 0xb76ec88 <col:10> 'int' 1
      `-CompoundStmt 0xb76ed00 <line:7:6, line:9:2>
        `-ReturnStmt 0xb76ecf0 <line:8:3, col:10>
          `-IntegerLiteral 0xb76ecd0 <col:10> 'int' 0
```

Example: Global Variable

Global variable
Variable assignment
Build binary operation
Build branch

Create Global Variable

- Just “new” it!
- The returned pointer is the *in-memory* representation of the global variable itself
- If named, could be looked up in module

```
// 创建全局变量，@a = dso_local global i32 0, align 4
auto globVarA =
    new llvm::GlobalVariable(/*持有该变量声明的模块*/ TheModule,
                             /*变量类型*/       builder.getInt32Ty(),
                             /*isConstant*/     false,
                             /*链接类型*/       llvm::GlobalValue::CommonLinkage,
                             /*initializer*/    llvm::Constant::getNullValue(builder.getInt32Ty()),
                             /*变量名*/         "a");
// 通过名字查找全局变量，实际上globVarA == anotherA
auto anotherA = TheModule.getGlobalVariable("a");
```

https://releases.llvm.org/11.0.1/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html
https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

Example: Variable Assignment & Expr

Global variable
Variable assignment
Build binary operation
Build branch

Create Assignment and operation expression

- The “instruction” is also the “virtual register”
- LLVM IR is strongly typed, identified by `llvm::Type`
- Constant values are represented by the `llvm::Constant` class
  - 👨‍⚕️ `builder.CreateLoad(10, globVarA);`
  - 👩 `builder.CreateLoad(builder.getInt32(10), globVarA);`

```
// 通过名字查找全局变量
auto globVarA = TheModule.getGlobalVariable("a");
// store i32 10, ptr @a, align 4
builder.CreateStore(builder.getInt32(10), globVarA);
// %1 = load i32, ptr @a, align 4
auto localA = builder.CreateLoad(globVarA->getValueType(), globVarA);
// %2 = icmp sgt i32 %1, 0
auto aGreaterThanZero = builder.CreateICmpSGT(localA, builder.getInt32(0));
```

Example: Branching

Create Branch

- Create new basic block
- Create conditional branch
- Change IRBuilder's insert point

Global variable
Variable assignment
Build binary operation
Build branch

```
// Assume we already have created the "aGreaterThanZero" instruction
auto ifBB = llvm::BasicBlock::Create(TheContext, "", function);
auto elseBB = llvm::BasicBlock::Create(TheContext, "", function);

// br i1 %2, label %3, label %4
builder.CreateCondBr(aGreaterThanZero, ifBB, elseBB);

// Insert in the "if" basic block
// 3:              ; preds = %entry
//   ret i32 1
builder.SetInsertPoint(ifBB);
builder.CreateRet(builder.getInt32(1));

// Insert in the "else" basic block
// 4:              ; preds = %entry
//   ret i32 0
builder.SetInsertPoint(elseBB);
builder.CreateRet(builder.getInt32(0));
```

Example: Visit Json Recursively

```
llvm::Value *buildStmt(llvm::IRBuilder<> *builder, Json json){
  if(json["kind"] == "BinaryOperator"){
    auto lhs = buildStmt(builder, json["inner"][0]);
    auto rhs = buildStmt(builder, json["inner"][1]);
    return builder.createSomeOperation(lhs, rhs);
  } else if(json["kind"] == "DeclRefExpr"){
    auto value = /*symbolTable.find(json["name"]);*/
    return builder.createLoad(/*typeOfValue*/, value);
  } else if(json["kind"] == "IntegerLiteral"){
    return builder->getInt32(json["value"]);
  } else if ...
    ..... some more implementations
}
```

Summary

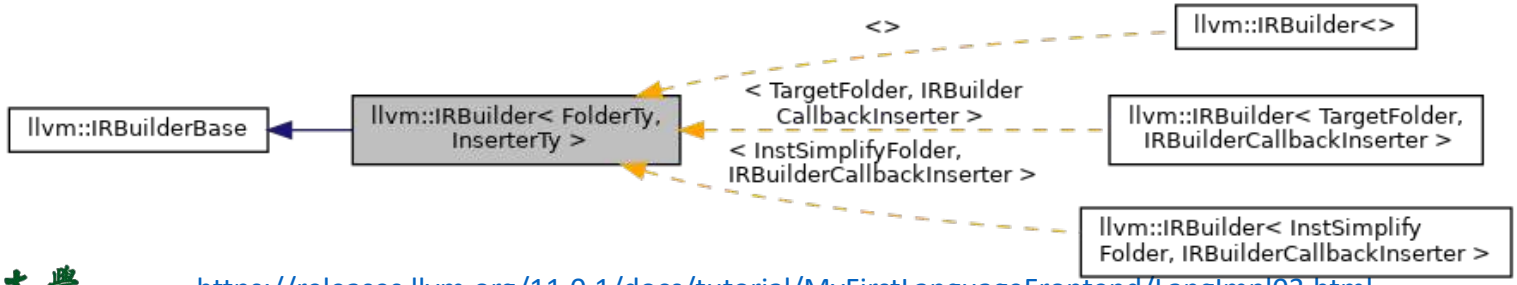
- **Hierarchy of IR**: Module, Function, Basic Block, Instruction
- **Infrustracture of LLVM**: Context, Module, IRBuilder
- **IRBuilder introduction**: IRBuilder::Create***()

祝大家实验顺利

Backup …

Variables in codegen[相关变量]

- **TheContext**: an opaque object that owns a lot of core LLVM data structures, such as the type and constant value tables
- **TheModule**: an LLVM construct that contains functions and global variables
  - In many ways, it is the top-level structure that the LLVM IR uses to contain code
- **Builder**: a helper object that makes it easy to generate LLVM instructions
  - Instances of the IRBuilder class template keep track of the current place to insert instructions and has methods to create new instructions



https://releases.llvm.org/11.0.1/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html
https://releases.llvm.org/8.0.0/docs/tutorial/LangImpl03.html

Visualize IR[可视化]

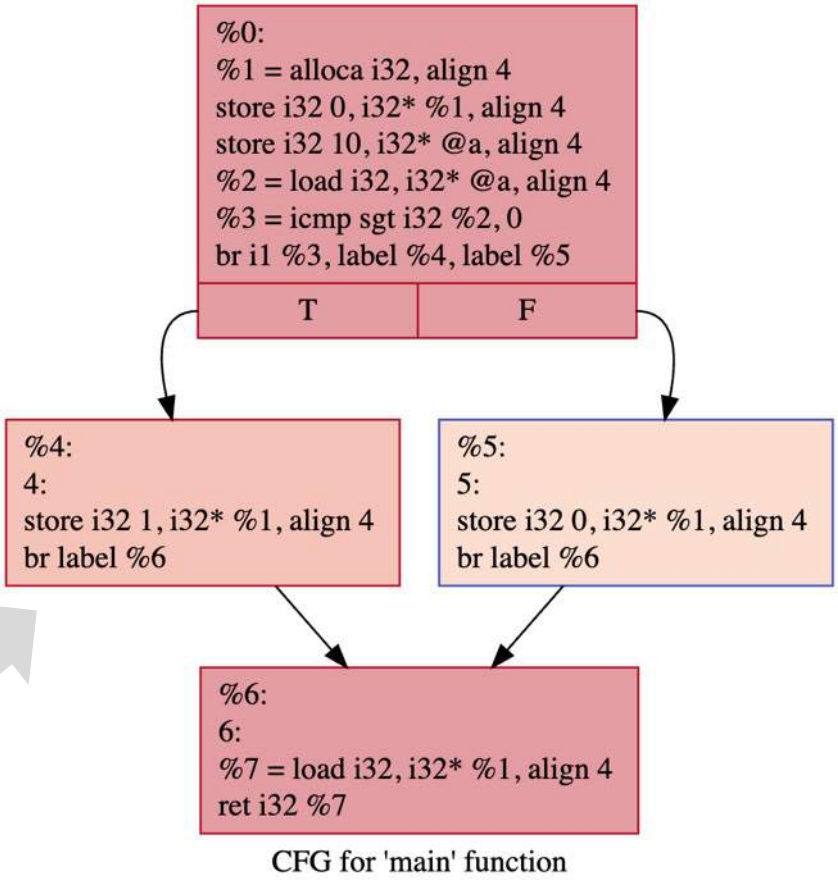
- $clang -emit-llvm -S ../tester/functional/027_if2.sysu.c

```
@a = dso_local global i32 0, align 4

define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, 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:
  store i32 1, i32* %1, align 4
  br label %6

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

6:
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}
```

$opt -dot-cfg 027_if2.sysu.ll [→ .main.dot]

```
digraph "CFG for 'main' function" {
        label="CFG for 'main' function";

        Node0x2a784a90 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%0:\l  %1 = alloca i32, align 4\l  store i32 0, i32* %1, align 4\l
  store i32 10, i32* @a, align 4\l  %2 = load i32, i32* @a, align 4\l  %3 = icmp sg
t i32 %2, 0\l  br i1 %3, label %4, label %5\l|{<s0>T|<s1>F}}"];
        Node0x2a784a90:s0 -> Node0x2a784c70;
        Node0x2a784a90:s1 -> Node0x2a784cc0;
        Node0x2a784c70 [shape=record,color="#b70d28ff", style=filled, fillcolor="#e
8765c70",label="{%4:\l4:                                                \l  store i
32 1, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784c70 -> Node0x2a784e50;
        Node0x2a784cc0 [shape=record,color="#3d50c3ff", style=filled, fillcolor="#f
7b39670",label="{%5:\l5:                                                \l  store i
32 0, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784cc0 -> Node0x2a784e50;
        Node0x2a784e50 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%6:\l6:                                                \l  %7 = lo
ad i32, i32* %1, align 4\l  ret i32 %7\l}"];
}
```

http://viz-js.com/



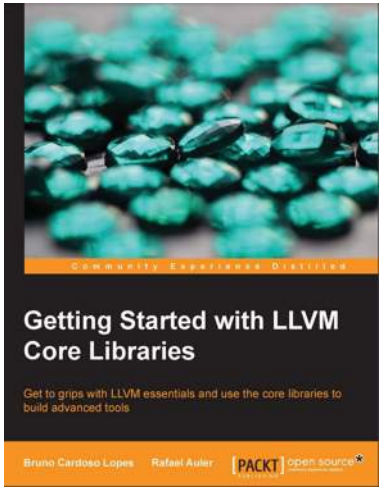
CFG for 'main' function

More …

- $(export PATH=~/sysu/bin:$PATH \ CPATH=~/sysu/include:$CPATH \ LIBRARY_PATH=~/sysu/lib:$LIBRARY_PATH \ LD_LIBRARY_PATH=~/sysu/lib:$LD_LIBRARY_PATH && clang -E tester/functional/000_main.sysu.c | <THE_PARSER> | sysu-generator)
  - S0: get AST
    - $clang -cc1 -ast-dump=json ../tester/functional/000_main.sysu.c > ast.json
  - S1: gen IR
    - $cat ast.json | ~/sysu/build/generator/sysu-generator
- Execute the IR file[1]: $lli *.ll*
  - Result: $echo $?
- Further compile the IR file: $clang *.ll* [-o ./a.out]
  - $(export PATH=~/sysu/bin:$PATH CPATH=~/sysu/include:$CPATH LIBRARY_PATH=~/sysu/lib:$LIBRARY_PATH LD_LIBRARY_PATH=~/sysu/lib:$LD_LIBRARY_PATH && clang -lsysy -lsysu *.ll* [-o ./a.out])
- 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*

[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

参考资料



LLVM Tutorial: Table of Contents

Kaleidoscope: Implementing a Language with LLVM

My First Language Frontend with LLVM Tutorial

This is the "Kaleidoscope" Language tutorial, showing how to implement a si

- 1. Kaleidoscope: Kaleidoscope Introduction and the Lexer
- 2. Kaleidoscope: Implementing a Parser and AST
- 3. Kaleidoscope: Code generation to LLVM IR
- 4. Kaleidoscope: Adding JIT and Optimizer Support
- 5. Kaleidoscope: Extending the Language: Control Flow
- 6. Kaleidoscope: Extending the Language: User-defined Operators
- 7. Kaleidoscope: Extending the Language: Mutable Variables
- 8. Kaleidoscope: Compiling to Object Code
- 9. Kaleidoscope: Adding Debug Information
- 10. Kaleidoscope: Conclusion and other useful LLVM tidbits

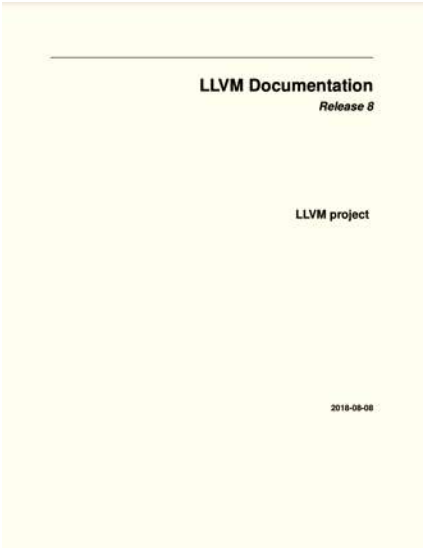
https://llvm.org/docs/tutorial/

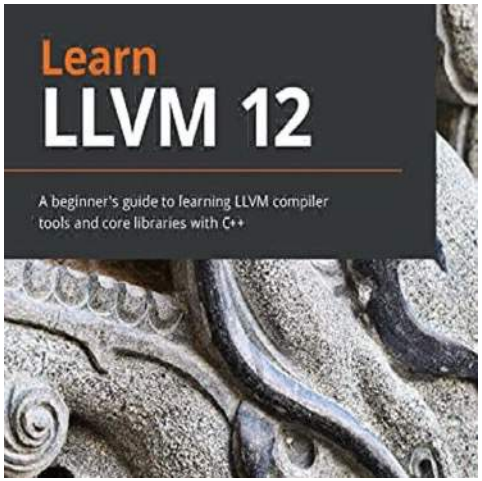
LLVM COMPILER INFRASTRUCTURE

LLVM Home | Documentation »

https://llvm.org/docs/

https://faculty.sist.shanghaitech.edu.cn/faculty/songfu/course/spring2018/CS131/llvm.pdf





https://github.com/xiaoweiChen/Learn-LLVM-12

https://bcain-llvm.readthedocs.io/_/downloads/llvm/en/latest/pdf/