



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 11: Project-3

张献伟、吴坎

xianweiz.github.io

DCS292, 4/28/2022



中山大學
SUN YAT-SEN UNIVERSITY



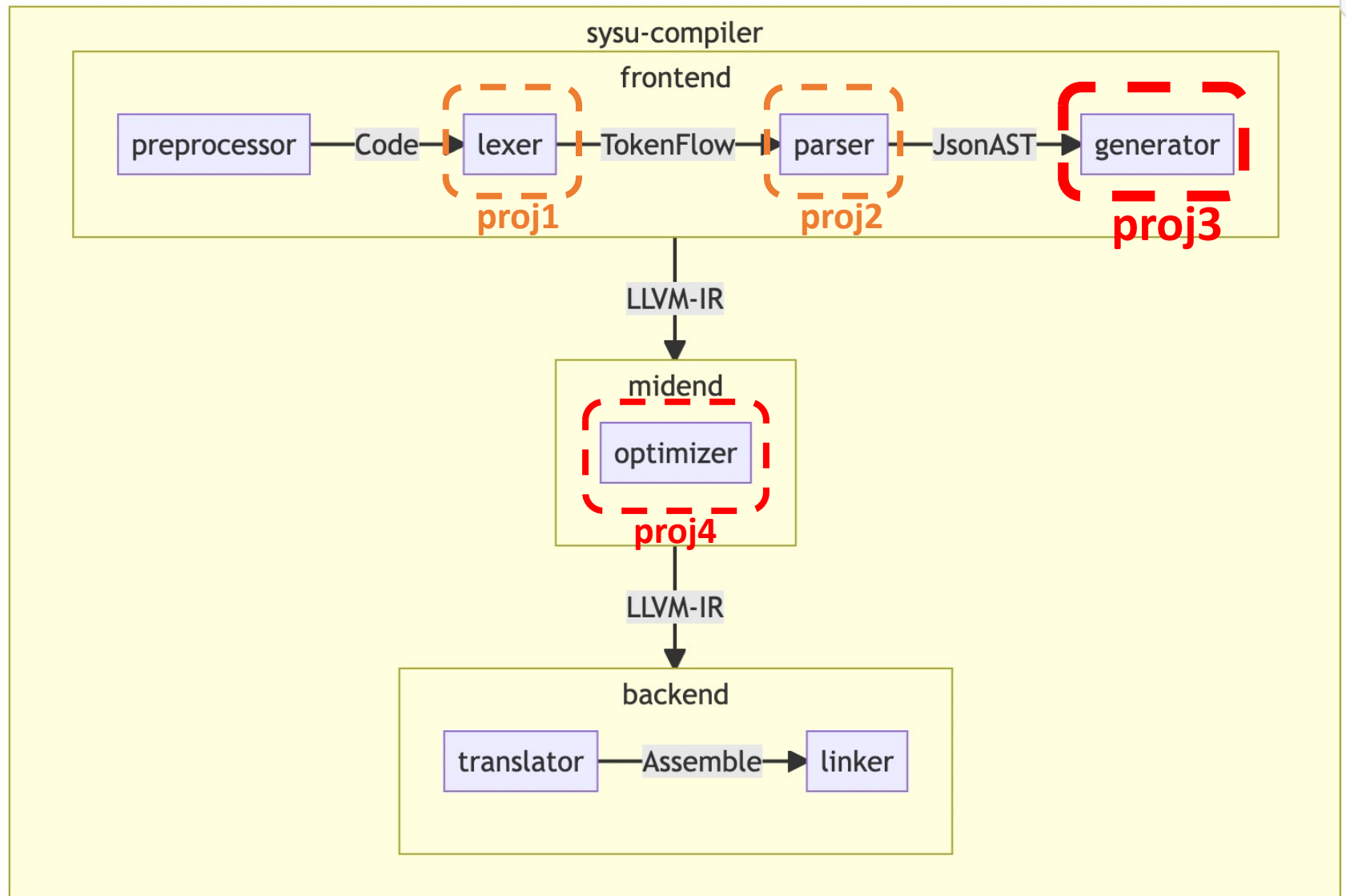
Project 3: What?

- 文档描述: <https://github.com/arcsysu/SYsU-lang/tree/main/generator>
- 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
- 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
 - **5/26/2022**

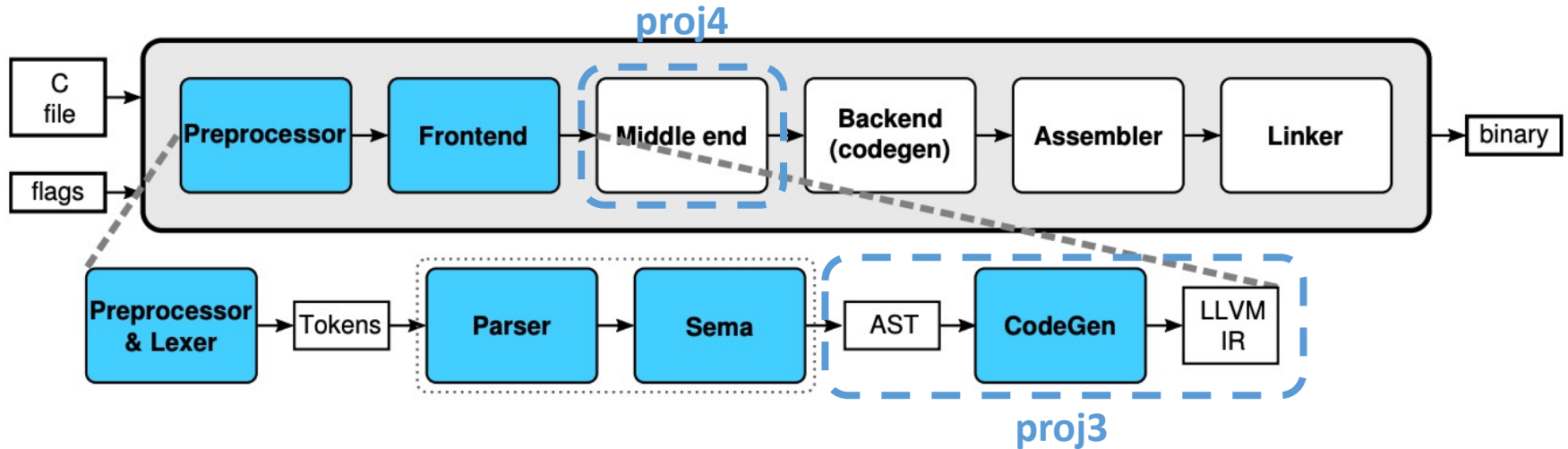
Project 3: How?

- 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-parser`

Where we are NOW?



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

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
```

- [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

```
$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
```



- [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

```
$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"}
```

[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

```
$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'  注释
source_filename = "../tester/functional/000_main.sysu.c"  源文件名
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128"  目标平台: 数据布局[1]
target triple = "aarch64-unknown-linux-gnu"  目标平台: arch-vendor-os

; Function Attrs: noinline nounwind
define dso_local i32 @main() #0 {  函数定义: define <返回类型> @<函数名> (参数) #属性[2]
    %1 = alloca i32, align 4  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
    store i32 0, i32* %1, align 4  数据写入内存: 将0写入%1对应的内存中, 4B对齐
    ret i32 3  函数返回
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt-fp-math"
="false" "disable-tail-calls"="false" "frame-pointer"="non-leaf" "less-precise-fpmad"="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-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="generic" "target-feature s="+neon" "unsafe-fp-math"="false" "use-soft-float"="false" }  函数属性

!llvm.module.flags = !{!0}  模块级别元数据信息[3]
!llvm.ident = !{!1}  Clang版本信息

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

- [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.)

Source

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



TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>

... cutting out internal declarations of clang ...

AST

```
-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
```



IR

```
define i32 @main() {  
    %1 = alloca i32  
    store i32 0, i32* %1  
    ret i32 3  
}
```



```
define i32 @main() {  
    ret i32 3  
}
```

AST → IR: Example (cont.)

Source

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



AST

```
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
```



IR

```
define i32 @main() {  
    %1 = alloca i32  
    store i32 0, i32* %1  
    ret i32 3  
}
```



```
define i32 @main() {  
    ret i32 3  
}
```

LLVM IR[中间代码]

- 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 text assembly language 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, %3...)
 - 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 the size of the datatype
 - We write LLVM IR in Static Single Assignment (SSA) form, making life easier for optimization writers[静态单赋值]
 - SSA just means we define variables before use and assign to variables only once

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);
}
```

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) 遍历内部节点
            if (auto P = it.getAsObject())
                if (auto kind = P->get("kind")->getAsString()) {
                    if (*kind == "FunctionDecl")
                        buildFunctionDecl(P); 具体IR生成
                }
}

} // namespace

int main() {
    auto llvmin = llvm::MemoryBuffer::getFileOrSTDIN("-"); 从文件或stdin获取AST文本
    auto json = llvm::json::parse(llvmin.get()->getBuffer()); 解析为JSON格式
    buildTranslationUnitDecl(json->getAsObject()); 遍历AST, 生成IR
    TheModule.print(llvm::outs(), nullptr); 输出IR
}
```

根节点

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;
}
```

<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>



generator.cc (cont.)

```
llvm::LLVMContext TheContext; 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::Module TheModule("-", TheContext); LLVM IR程序的顶层结构
```

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

创建一个函数，并指派给Module

参数：类型

int main()

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。

```
    if (!TheFunction)  
        return nullptr;
```

```
    // Create a new basic block to start insertion into.
```

```
    auto BB = llvm::BasicBlock::Create(TheContext, "entry", TheFunction); 为创建的Function添加Basic Block
```

```
    llvm::IRBuilder<> Builder(BB); 使用IRBuilder插入指令到BB
```

```
    if (auto RetVal = llvm::ConstantInt::get(  
        TheContext, /* i32 3(decimal) */ llvm::APInt(32, "3", 10))) {
```

return 3

```
        // Finish off the function.
```

```
        Builder.CreateRet(RetVal); 返回值指令语句
```

```
    // Validate the generated code, checking for consistency.
```

```
    llvm::verifyFunction(*TheFunction);
```

```
    return TheFunction;
```

```
}
```

```
    // Error reading body, remove function.
```

```
    TheFunction->eraseFromParent();
```

```
    return nullptr;
```

```
}
```

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

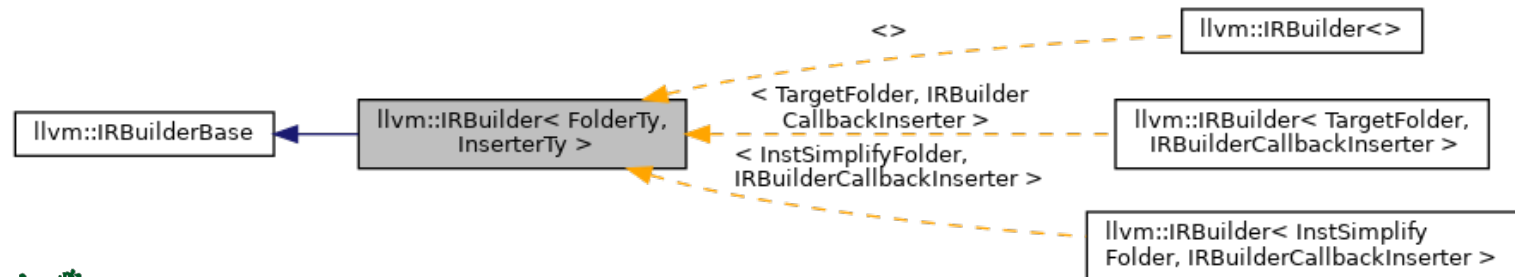
<https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html>

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



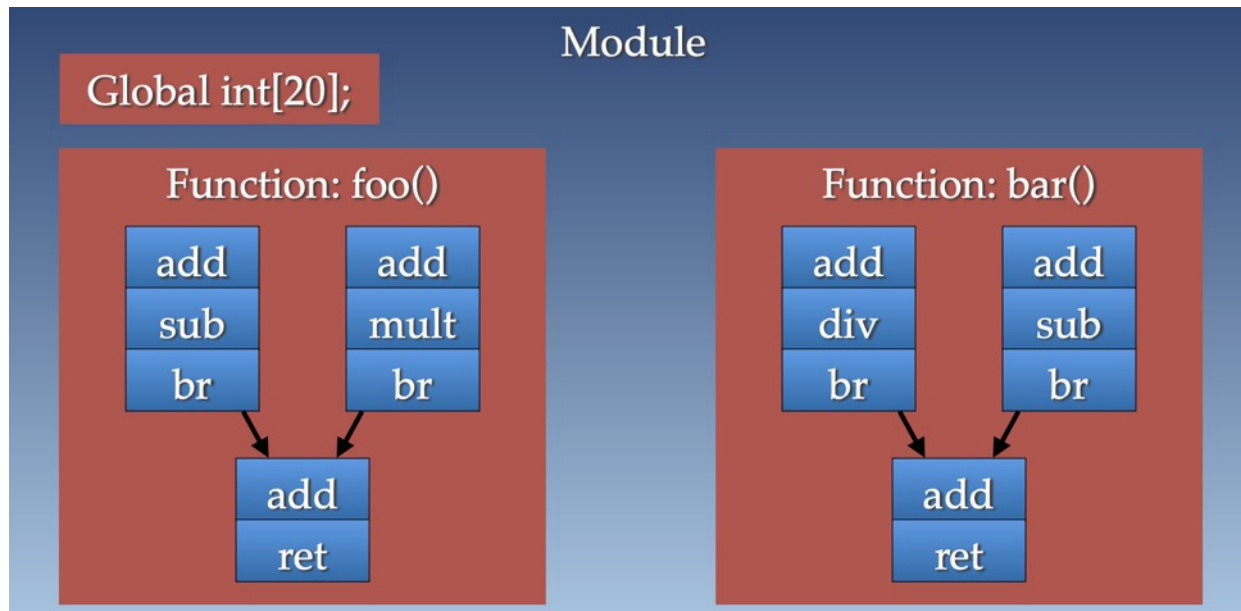
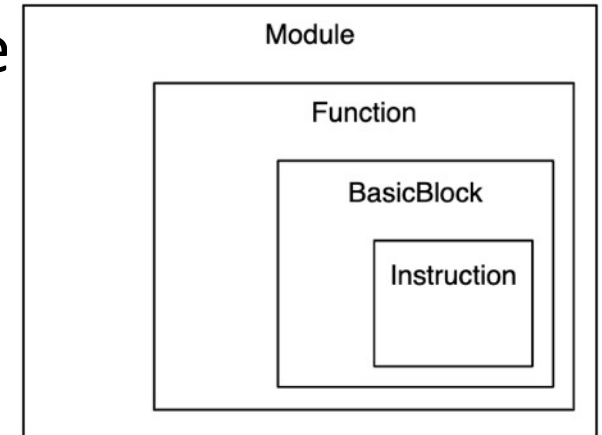
Variables in codegen[相关变量]

- **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](https://releases.llvm.org/11.0.1/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html) class template keep track of the current place to insert instructions and has methods to create new instructions



IR Overview

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



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
```

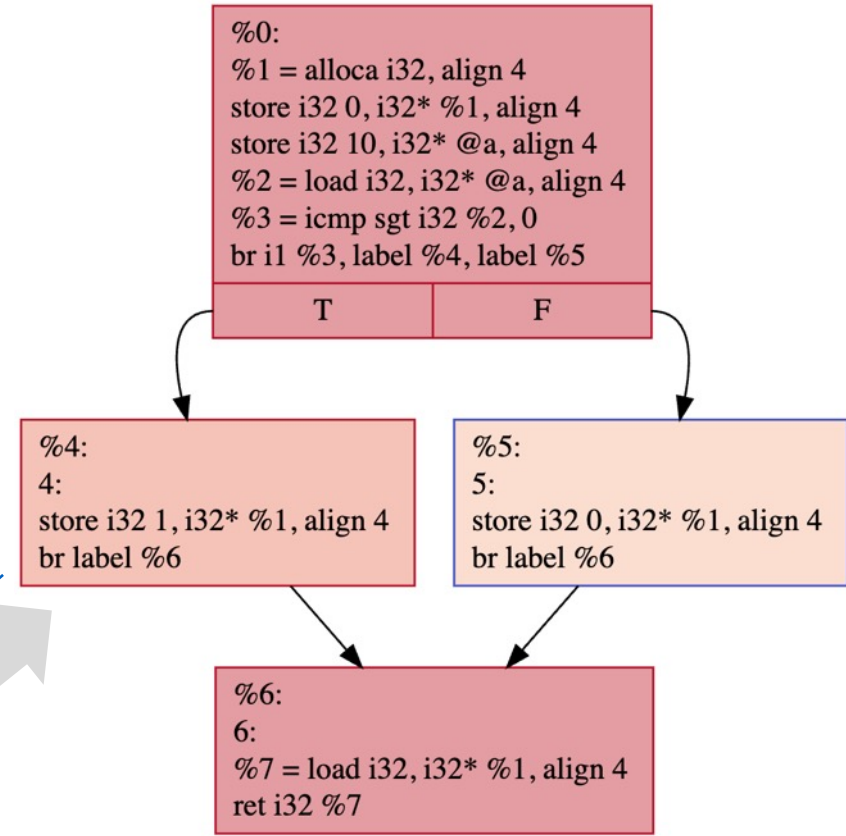
```
5:  
  store i32 0, i32* %1, align 4  
  br label %6
```

```
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="#b70d28ff",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 sgt 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="#e8765c70",label="{%4:\\l4: store i32 1, i32* %1, align 4\\l br label %6\\l}"}];  
  Node0x2a784c70 -> Node0x2a784e50;  
  Node0x2a784cc0 [shape=record,color="#3d50c3ff", style=filled, fillcolor="#f7b39670",label="{%5:\\l5: store i32 0, i32* %1, align 4\\l br label %6\\l}"}];  
  Node0x2a784cc0 -> Node0x2a784e50;  
  Node0x2a784e50 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b70d28ff",label="{%6:\\l6: %7 = load i32, i32* %1, align 4\\l ret i32 %7\\l}"}];  
  ad i32, i32* %1, align 4\\l ret i32 %7\\l}"}];  
}
```

<http://viz-js.com/>



CFG for 'main' function

More ...

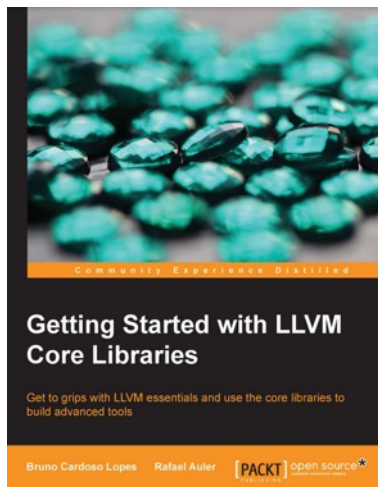
- `$(export PATH=~/.sysu/bin:$PATH \ CPATH=~/.sysu/include:$CPATH \
LIBRARY_PATH=~/.sysu/lib:$LIBRARY_PATH \
LD_LIBRARY_PATH=~/.sysu/lib:$LD_LIBRARY_PATH && clang -E
tester/functional/000_main.sysu.c | <THE_PARSER> | sysu-generator)`
 - S0: get AST
 - `$clang -cc1 -ast-dump=json ../tester/functional/000_main.sysu.c > 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

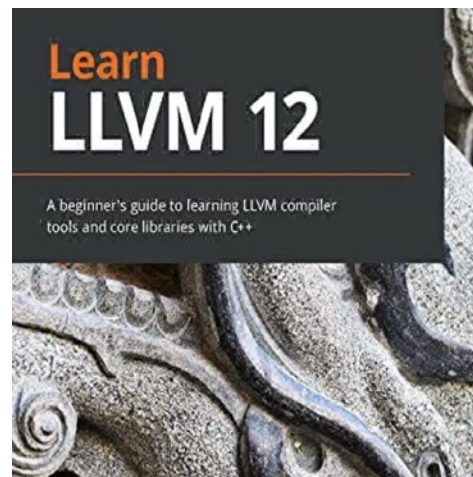
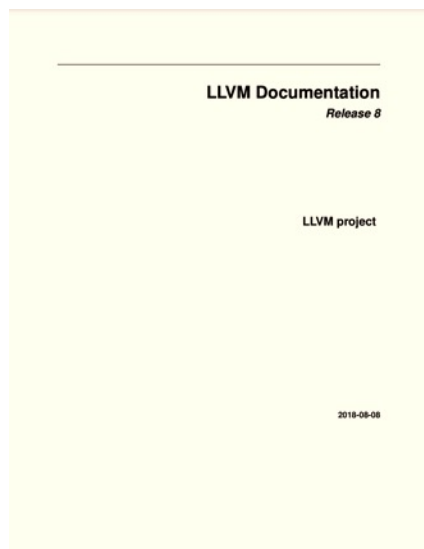
<https://llvm.org/docs/tutorial/>



[LLVM 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/en/latest/pdf/>