



电子科技大学

University of Electronic Science and Technology of China

基于 LLVM 的类 C 编译器及其实现

学 院：	计算机学院
专 业：	计算机科学与技术
姓 名：	XXX
学 号：	XXX
课程名称：	程序设计语言与编译
任课老师：	XXX

一 . 背景介绍.....	3
二 . 基本框架.....	3
三 . 程序部署与效果展示.....	4
环境搭建.....	4
操作系统运行效果.....	5
编译器交叉编译.....	6
四 . 课程内容.....	8
class 1 实现词法分析器	8
class 2 使用 javacc 生成词法分析器.....	9
class 2 plus 使用 javacc 多文件预处理	10
class 3 使用语法分析器分析一个小程序（无代码）	10
class 4 实现 c- 的词法与语法分析器	11
class 5 预备知识 LLVM	12
class 5 LLVM IR 的实现	13
入侵式链表.....	13
IR 的构成	14
class 6 语义分析 和 生成中间代码 LLVM IR.....	15
Scope 类.....	16
IRBuilder 类.....	16
Visitor 类	17
常量折叠（ ConstFolding ）	17
class 7 backend, 将 LLVM IR 翻译成 ASM	17
寄存器与内存分配.....	18
函数栈帧.....	19

一. 背景介绍

电子科技大学 编译原理挑战课，为比赛做的基础训练

该比赛为 [全国大学生计算机系统能力大赛](#) 的系统编译方向，由华为主办

由于比赛可以用 java (~~可惜不能用 python~~)，所以就选择了 java

讲解视频 我也上传到了 [B 站](#) 空间，欢迎大家观看。

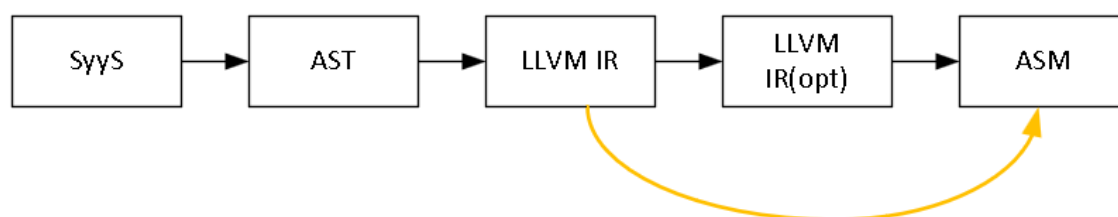
项目地址：[Eric-is-good/c_compiler](#): 电子科技大学编译原理挑战课，使用 java 写一个类 c 编译器，最终用 java 重写 LLVM 前后端，使编译后的代码运行在我的操作系统上。
([github.com](#))

经过商量，老师找了个更适合我的方向，就不参加比赛了。

具体是我写过一个 [操作系统](#) (x86 I32)，于是我打算给我的操作系统写一个编译器，即高级语言 sy 翻译为 **类 x86 I32 汇编**，因为我的操作系统 寄存器使用 和 函数栈帧使用 和传统 x86 有区别。sy 语言是比赛需要设计的语言，他的定义我们放在 sy 文件夹下，我们实现了这个语言除了浮点数和数组以外的所有功能（因为我的操作系统不支持浮点数）。

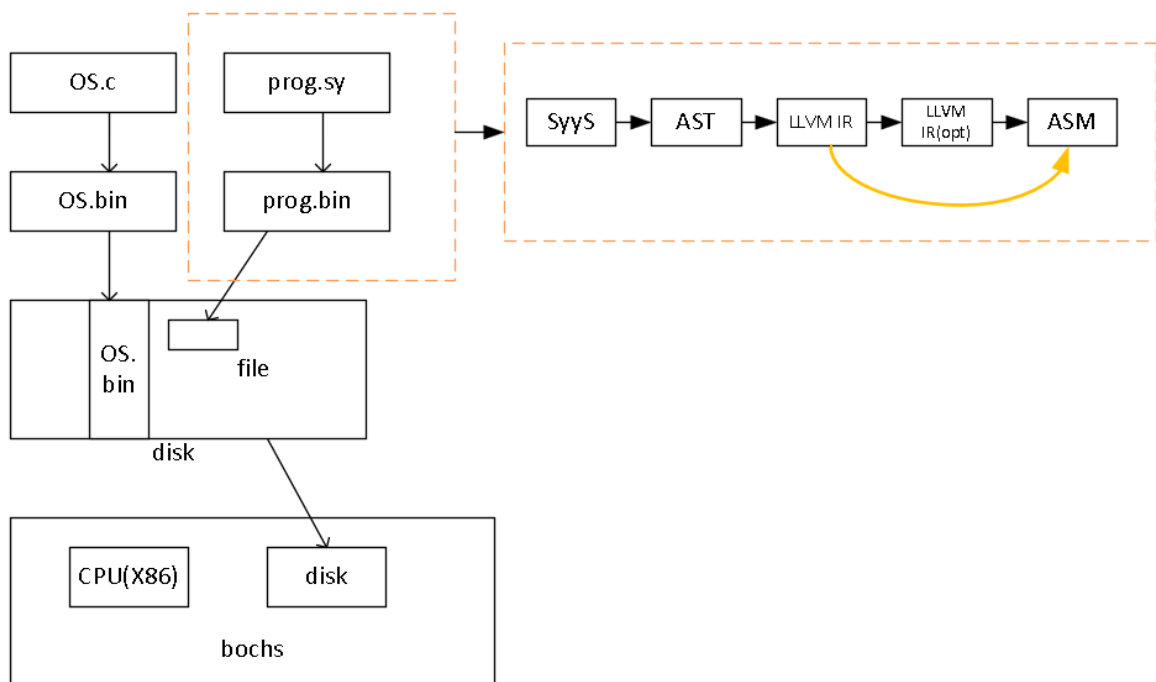
二. 基本框架

我们的编译器架构如下



我们跳过了 LLVM 的优化，在生成 LLVM IR 之后就直接翻译成 ASM，LLVM IR 的优化有现成的工具 `llvm-opt`，我们要做的就是前端后端，这应该是实现一门自定义语言最具有性价比的方式。

这是我们结合了我的操作系统的总体架构



我们的程序使用 `sy` 语言，结合这里实现的编译器，完成程序在操作系统上安装运行。

三. 程序部署与效果展示

环境搭建

在上一节的总体架构图中，我们为了实现交叉编译，环境分为两个，一个是运行操作系统的 `bochs` 环境，一个是运行编译器的 `Java 17` 环境。

bochs 环境：我们使用 `docker` 技术，使用[我配置好的镜像](#)。在安装 `docker` 软件后，只需一行命令安装环境。

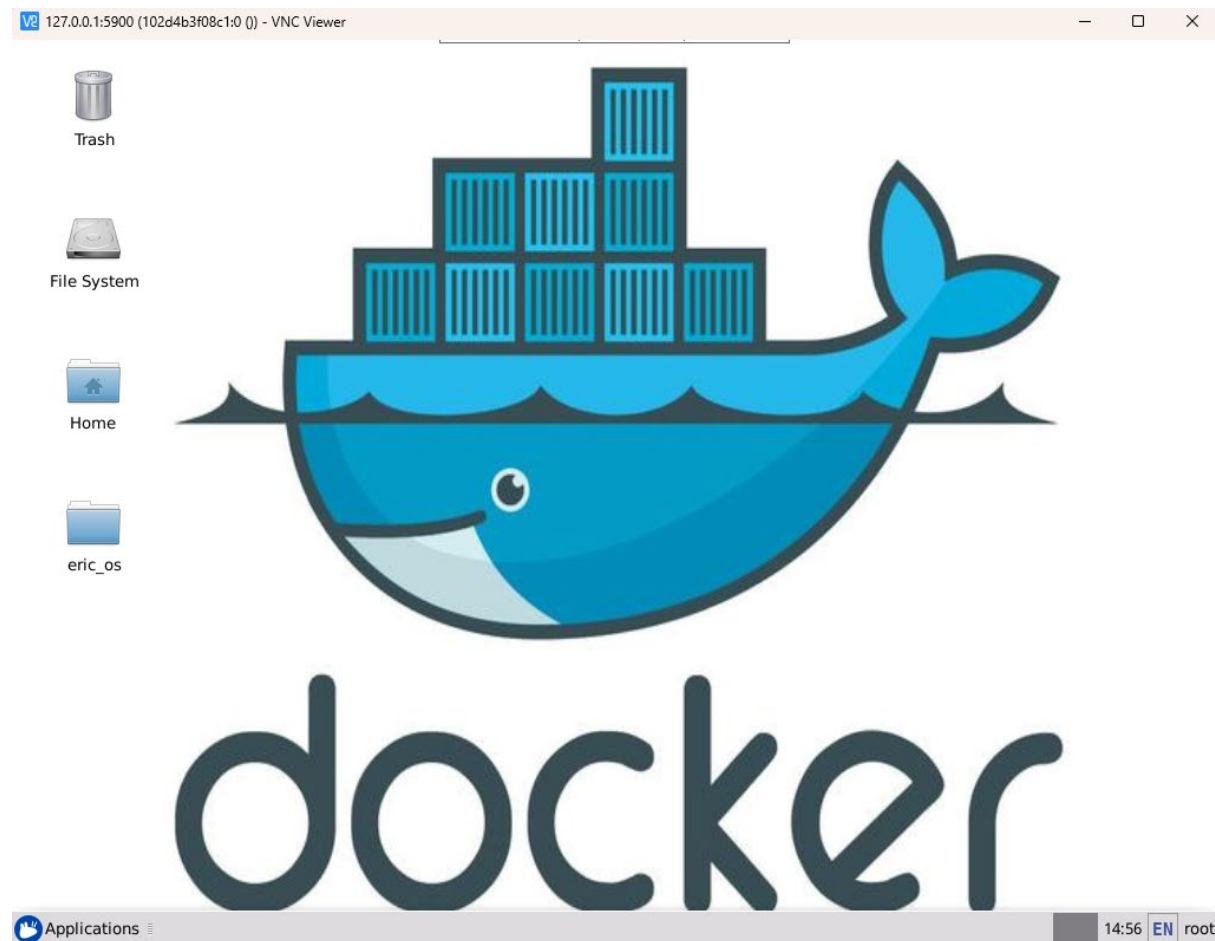
```
docker pull ericju/ubuntu-desktop-bochs:1.1 （安装环境）
```

`docker run -d -p 22:22 -p 5900:5900 ericju/ubuntu-desktop-bochs:1.1`（运行环境，默认密码为 123456）

然后我们就可以使用 `ssh`，`xftp` 和 `VNC` 可视化桌面了。其中，`VNC` 连接地址为 `ip:5900`

获取 [VNC viewer](#)

当我们使用 `VNC` 连接上 `bochs` 环境后，可以看到这样的效果：



Java 17 环境：首先我们要安装 `java 17`，再下载 `antlr` 包（代码 `c_compiler\third_party` 中提供）

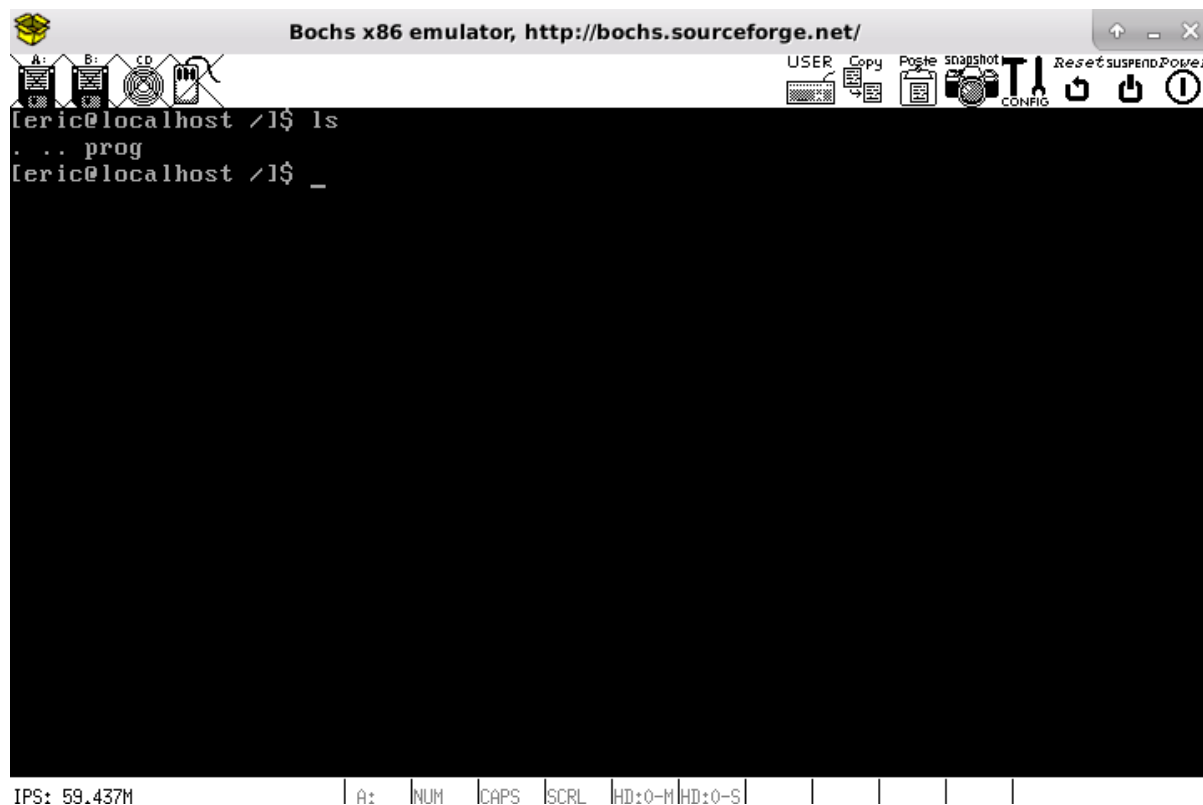
操作系统运行效果

操作系统代码地址：[Eric-is-good/eric_os](#): 记录自己从 0 到 1 写一个操作系统 ([github.com](#))

运行方法：

1. 进入 `eric_os`, `make all` 编译 (如果是第一次运行需要去掉 `makefile` 里面的 `clean`)。
2. `bash run.sh` 进入 `bochs`, 回车后开始仿真, 输入 `c` 并回车取消断点即可。

效果如下



```
Bochs x86 emulator, http://bochs.sourceforge.net/
USER Copy Paste snapshot CONFIG Reset suspend Power
eric@localhost /1$ ls
. . . prog
eric@localhost /1$ _
```

IPS: 59.437M | A: | NUM | CAPS | SCRL | HD:0-M | HD:0-S | | | | |

编译器交叉编译

1. 运行 `full_compiler/src` 下的 `Hello.java`, 会在 `test_syys` 下写入中间码和汇编代码 (这三个文件要提前创建好), 把 `sy` 文件翻译成对应中间码和汇编代码。
2. 将汇编代码拷贝到 `eric_os/command` 的 `prog.S` 里面, 运行 `bash compile.sh`, 得到以下结果。

```
root@102d4b3f08c1:~/Desktop/eric_os/command# bash compile.sh
16+1 records in
16+1 records out
8392 bytes (8.4 kB, 8.2 KiB) copied, 0.0002007 s, 41.8 MB/s
```

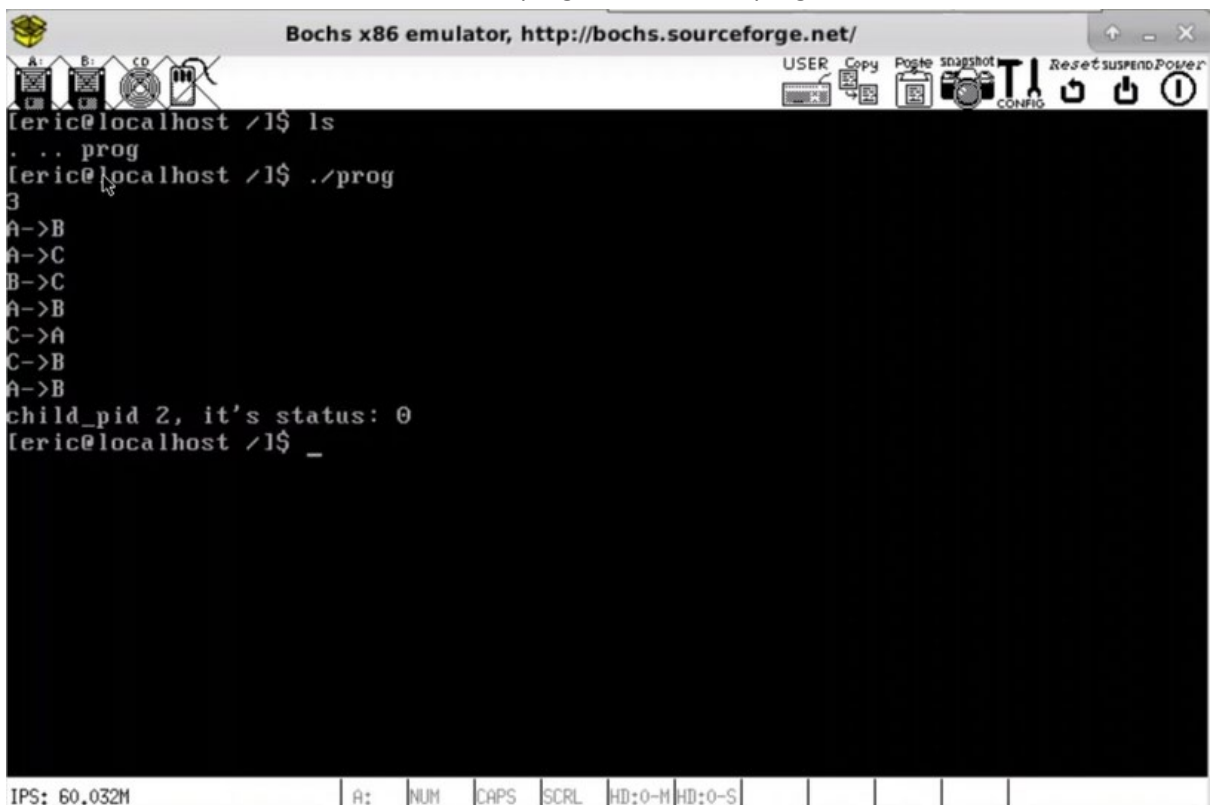
3. 记录编译后的文件大小, 在操作系统启动前读入, 修改 `eric_os/kernel/main.c` 文件对应位置, 然后重新编译操作系统 `make all`, 这次他将读入 `prog.S` 文件对应的二进制码。

```

/*****          写入应用程序          *****/
uint32_t file_size = 8392;
uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
struct disk* sda = &channels[0].devices[0];
void* prog_buf = sys_malloc(file_size);
ide_read(sda, 300, prog_buf, sec_cnt);
int32_t fd = sys_open("/prog", O_CREAT|O_RDWR);
if (fd != -1) {
    if(sys_write(fd, prog_buf, file_size) == -1) {
        printk("file write error!\n");
        while(1);
    }
}
}

```

4. 启动操作系统，运行 ls，发现多了一个 prog 文件，使用 ./prog 来运行文件即可。



```

Bochs x86 emulator, http://bochs.sourceforge.net/
[eric@localhost ~]$ ls
. .. prog
[eric@localhost ~]$ ./prog
3
A->B
A->C
B->C
A->B
C->A
C->B
A->B
child_pid 2, it's status: 0
[eric@localhost ~]$ _

```

到这里，我们完成了编译器与操作系统的结合。

四 . 课程内容

根据我的学习进度，分课程的记录学习进度。在 `class` 文件夹里面，有每一课的学习内容。

我们借鉴了 [这个代码](#)，并以此为模板学习。感谢大佬们的开源。

比赛官方也有开源代码学习，我们也可以看看他们的。

如果你想直接进入 `llvm` 相关内容，建议从 `class 4` 开始看。

class 1 实现词法分析器

单词	编码	单词	编码
标识符	01	/	13
常数	02	<	14
int	03	<=	15
if	04	>	16
else	05	>=	17
while	06	!=	18
for	07	==	19
read	08	=	20
write	09	(21
+	10)	22
-	11	,	23
*	12	;	24

- 非法字符
- 非法数字（多个小数点，数字+非数字）

第二次再输出分词

之所以要两次，是因为没有前缀后缀，非法数字无法识别，我又懒得搞匹配后又退回，多令牌机制也不想碰（叹气）

class 2 plus 使用 javacc 多文件预处理

多文件编译时，使用递归处理 `include` 文件。

果然，菜鸡（指我）都喜欢递归。

我们的思路是深度优先，因为程序就是一棵树（以主函数为根节点），我们先使用递归进入下一层，最后再添加本结点内容，这样可以使被调用程序代码在调用代码之前被添加，我们最后把所有文件合成一个大字符串。

```
// 伪代码
find_include(path){
    while(true){
        word = next_word(); // 打开文件, 读入本文件下一个词语
        if word == <EOF>:
            break;
        if word == #include "path_xx":
            find_include(path_xx); // 递归
        if word == programe: // 如果读到代码
            total_content.add(word);
    }
}
```

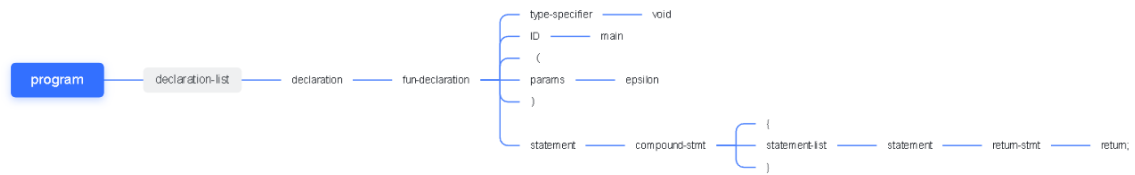
class 3 使用语法分析器分析一个小程序（无代码）

c- 语法书在 class_3 里面，老师要我们用飞书画思维导图，也就是那个语法树（飞书打钱）

```
void main(){
    return;
}
```

得到这样的语法树

思维导图



class 4 实现 c- 的词法与语法分析器

我们使用 [antlr 4](#)，我感觉可能要抛弃 [javacc](#) ？

关于 [解决左递归](#)

我们实现的 c- 语法解析器实现效果（按照 c- 语法书）

```
int main(int args[]){
    int a;
    a = 6;
    a = a+10;
    if(a>9){
        return a;
    }else{
        int b;
        b = a * a;
    }
}
```

[图片地址](#)

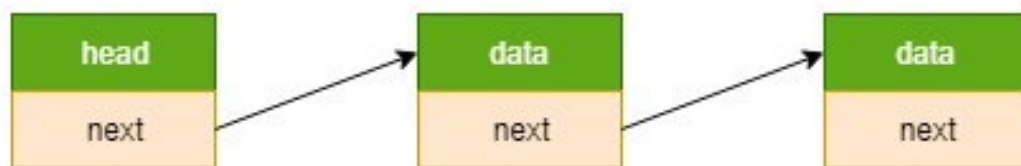
如下

class 5 LLVM IR 的实现

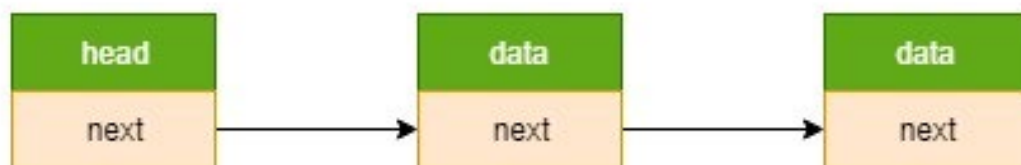
入侵式链表

在 `class_5\src\utils` 里面，实现的就是这个具有通用性的链表。

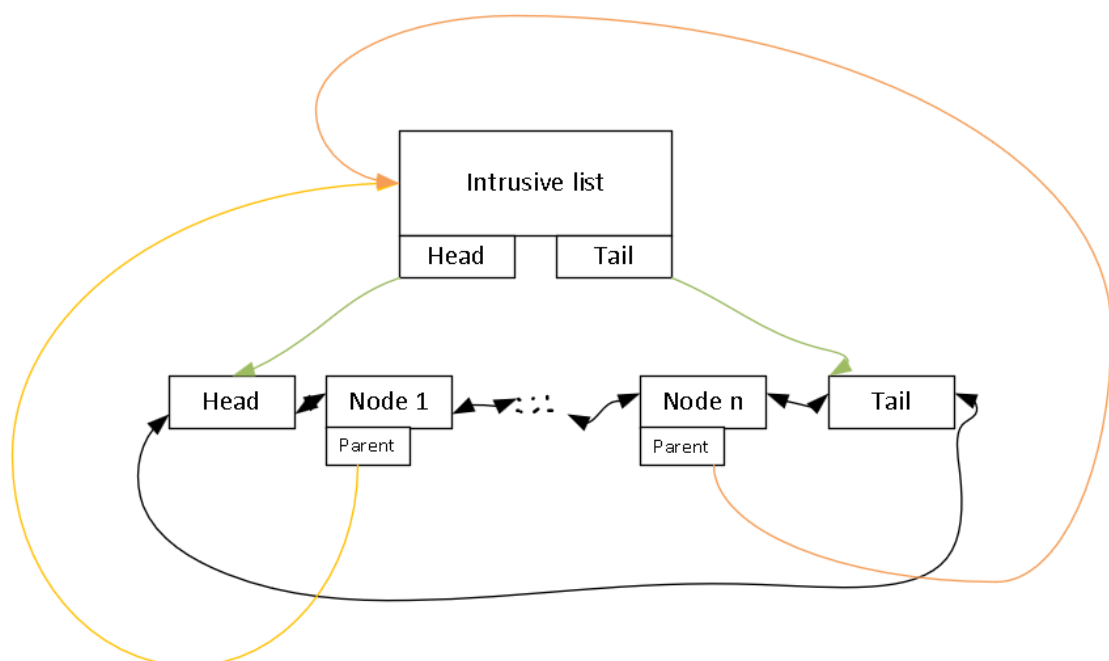
传统链表



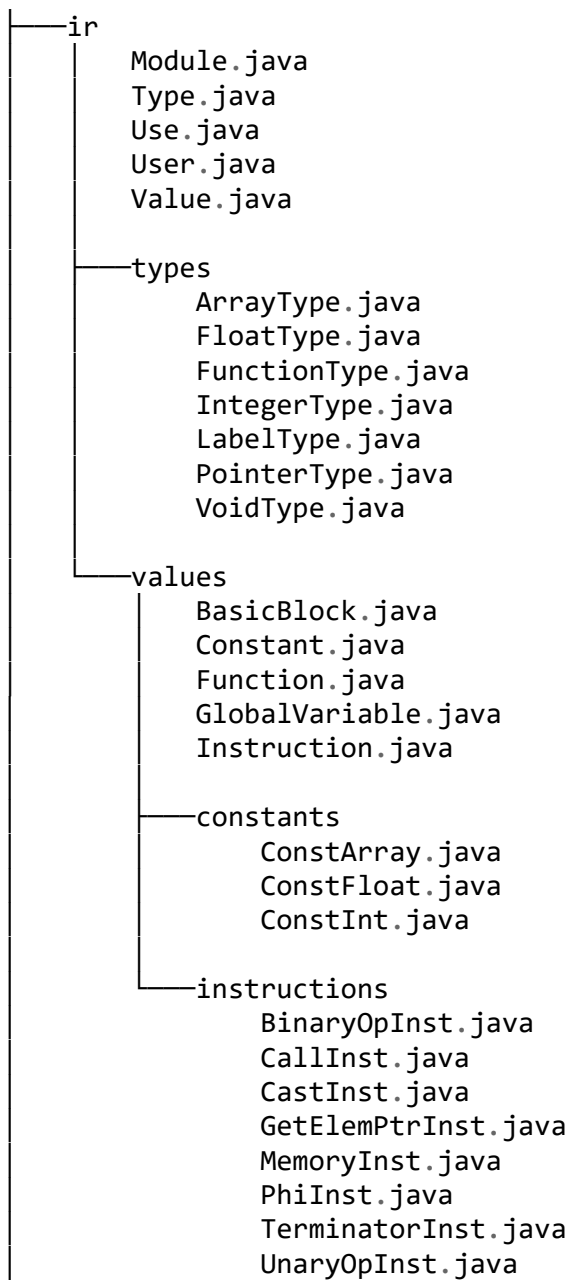
入侵式链表



我们的实现方式



IR 的构成



- Value 类，Value 是一个非常基础的基类，一个继承于 Value 的子类表示它的结果可以被其他地方使用。在我们的实现中 Use 作为 Value 和 User 的桥梁。
- User 类，一个继承于 User 的类表示它会使用一个或多个 Value 对象。
- Type 类，所有 type 继承于此，不同的 type 有其自己独特的属性。在 types 文件夹里面。
- 基于 Value 类和 User 类，我们实现了从指令集类到代码再到函数的类，在 values 里面。通过继承迭代器，Function 迭代 BasicBlock，BasicBlock 迭代 Instruction。

- Instruction 抽象类，tag 表示指令类型。所有 instruction 继承于它，在 instructions 文件夹里面。
- Module 类，Module 可以理解为一个完整的编译单元。一般来说，这个编译单元就是一个源码文件，如一个后缀为 cpp 的源文件。
- Function 类，这个类顾名思义就是对应于一个函数单元。Function 可以描述两种情况，分别是函数定义和函数声明。
- BasicBlock 类，这个类表示一个基本代码块，“基本代码块”就是一段没有控制流逻辑的基本流程，相当于程序流程图中的基本过程（矩形表示）。
- Instruction 类，指令类就是 LLVM 中定义的基本操作，比如加减乘除这种算数指令、函数调用指令、跳转指令、返回指令等等。指令中每个变量是个 Value。

class 6 语义分析和生成中间代码 LLVM IR

比赛需要实现一个语言 SysY2022，文档在 class_6/SysY2022 下。

从现在开始，我们使用 java 17 来匹配比赛要求。

1. parser 和 lexer 直接 antlr 生成，官方有指导书，我们就直接用了别人的 g4 文件。在 class_6/src/frontend 下。重点在于 visitor 的代码。visit 一遍后，生成 in-memory IR。
2. llvm ir 中，在 class_6/src/ir 下，在上节课实现。
3. 将 in-memory IR 输出，我们有一个 IREmitter 类，在 class_6/src/frontend 下。

这便是我们生成中间代码的主框架

```
// 1.sy -> 1.ll
public static void main(String[] args) throws Exception{
    CharStream inputFile = CharStreams.fromFileName("test_syys/1.sy");

    /* Lexical analysis */
    SysYLexer lexer = new SysYLexer(inputFile);
    CommonTokenStream tokenStream = new CommonTokenStream(lexer);

    /* Parsing */
    SysYParser parser = new SysYParser(tokenStream);
    ParseTree ast = parser.compUnit();

    /* Intermediate code generation */
    // Initialized all the container and tools.
    Module module = new Module(); // last class
    Visitor visitor = new Visitor(module);
    // Traversal the ast to build the IR.
    visitor.visit(ast);
}
```

```

    /* Emit the IR text to an output file for testing. */
    IREmitter emitter = new IREmitter("test_syys/1.11");
    emitter.emit(module, true);
}

```

而本章的重点在于 frontend 里面的 IRBuilder Scope 和 Visitor 这三个代码。

```

├── frontend
│   ├── IRBuilder.java
│   ├── IREmitter.java
│   ├── Scope.java
│   ├── SysYBaseVisitor.java
│   ├── SysYLexer.java
│   ├── SysYParser.java
│   ├── SysYVisitor.java
│   └── Visitor.java    // 重点

```

Scope 类

用于判断变量和函数作用范围。

```

private final ArrayList<HashMap<String, Value>> tables = new ArrayList<>
();

```

符号表以哈希表实现，每次进入一个 block 会通过 scopeIn 和 scopeOut 来添加删除符号表来调整当前范围，通过 addDecl 向当前符号表添加符号。duplicateDecl 函数判重。

IRBuilder 类

调用以返回指令，以 add 为例

```

public BinaryOpInst buildAdd(Value lOp, Value rOp) {
    ...
    getCurBB().insertAtEnd(instAdd);    // 添加指令到当前块
    return instAdd;
}

```

在 visitAddExp 调用


```

/**
 * addExp : mulExp (('+' / '-' ) mulExp)*
 */
@Override
public Void visitAddExp(SysYParser.AddExpContext ctx) {
    ...
    switch (ctx.getChild(2 * i - 1).getText())
    {
        case "+": -> lOp = builder.buildAdd(lOp, rOp);    //
        case "-": -> lOp = builder.buildSub(lOp, rOp);
        default -> {}
    }
    ...
}

```

返回生成指令

Visitor 类

建议阅读顺序：从 `visitNumber` 函数开始，对照 `g4` 文件向上规约，一边规约，一边看代码。

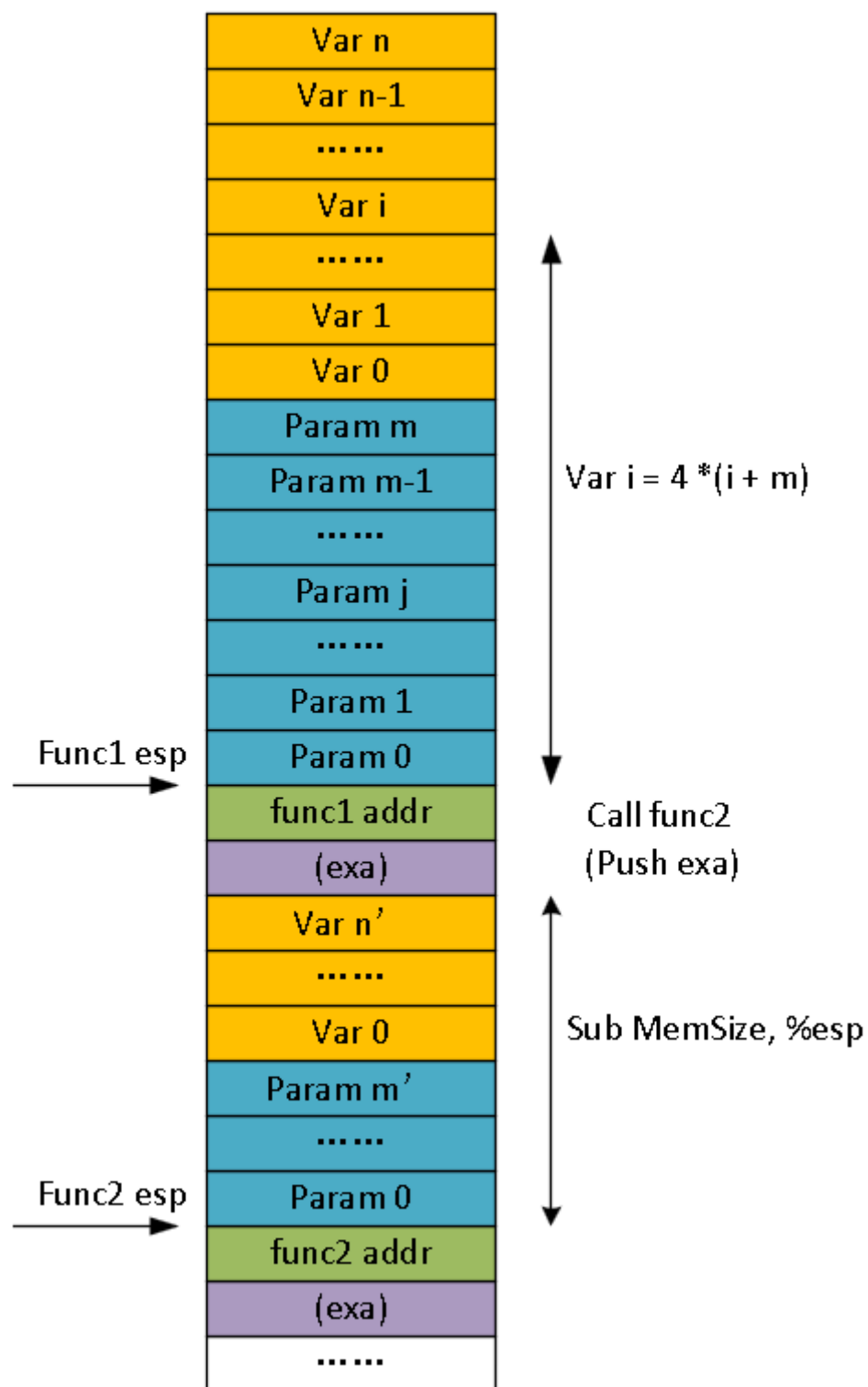
但注意：`visit` 默认是从上到下深度优先访问树的。我们可以在 `visitXXX` 里面改变他的访问顺序。

常量折叠 (`ConstFolding`)

将常量放进常量表，减少赋值时的运算。

`class 7 backend`，将 LLVM IR 翻译成 ASM

后端分为两个部分，一个是寄存器与内存分配策略 (`Mem To Reg`)，另一个是函数栈帧的策略。



寄存器与内存分配

```
store i32 %0, i32* %4
```

我们只使用 寄存器 `eax` 和 `ecx`（仅在除法时）。

我们为每一个 llvm ir 里面的 %i 分配一个内存，即图中的 var i，地址为 $esp + 4 * (i + m)$

所有指令的中转使用 eax，例如上面这一句翻译为

```
movl 0(%esp), %eax
movl %eax, 4(%esp)
```

函数栈帧

第一步是传参。

上图展示了传参策略，以 func 1 调用 func 2 为示例。

```
func1(){
    .....
    func2(param 0,param 1,....,param n1)
    func3(param 0,param 1,....,param n2)
    .....
}
```

其中， $m = \max(n_1, n_2, \dots, n_k)$ ，即所有在 func1 被调用函数的最大传参数。

llvm ir 代码为

```
call i32 @func2(i32 %0, i32 %1, ....., i32 %n1)
```

我们仅使用堆栈传参，将参数从低到高存到 param 区，param 区则按照最大传参量设计，这样，我们就完成了使用堆栈传参。

执行 call 汇编时，会存储 func 1 的地址。（绿色区域）

如果 func 2 有返回值，我们需要 push exa，然后用 exa 来接收返回值。（紫色区域）

第二步是栈帧的开辟。

MemSize 的大小为 func 2 的 var 大小与 param 大小之和。var 大小为 $\max(\%i)$ ，即 func 2 的 llvm ir 中最大的 %i 中的 i 大小。param 大小为 $\max(n_1, n_2, \dots, n_k)$ ，即所有在 func 2 被调用函数的最大传参数，与上文的 m 一致。

```
subl 4 * m_func2, %esp
```

如何访问 func 1 传给 func 2 的函数参数呢？

很简单， $\text{esp} + \text{MemSize} + (4) + 4 + i * 4$ 即为第 i 个参数的位置。

第一个 4 为 `exa` 的占用大小，加了括号表示当且仅当 `func 2` 有返回值时才会 `push exa`。

第二个 4 为 `func 1` 的地址占用空间，在 `call` 时会自动存储。

总而言之，即图中的 `func2 esp + 蓝色区域与橙色区域 + (紫色区域) + 绿色区域 + func 1 的部分蓝色区域` 抵达 `param j`。

最后，结束调用返回到 `func 1` 时，需要将 `esp` 归位到 `func 1` 的 `esp`，即

```
addl 4 * m_func2, %esp
```