

《编译原理》课程实验报告

小组人数：1人

组员姓名：颜尔迅

组员学号：3190106216

目录：

一、序言

- 1.1 概述
- 1.2 开发环境
- 1.3 代码说明
 - 1.3.1 源码目录结构
 - 1.3.2 代码运行
 - 1.3.3 版本控制与代码风格

二、词法分析

- 2.1 概述
- 2.2 token的正则表达式描述
 - 2.2.1 C语言关键字
 - 2.2.2 数据表示
 - 2.2.3 运算符
 - 2.2.4 注释及其他
- 2.3 词法分析器具体实现
- 2.4 结果示例

三、语法分析与AST生成

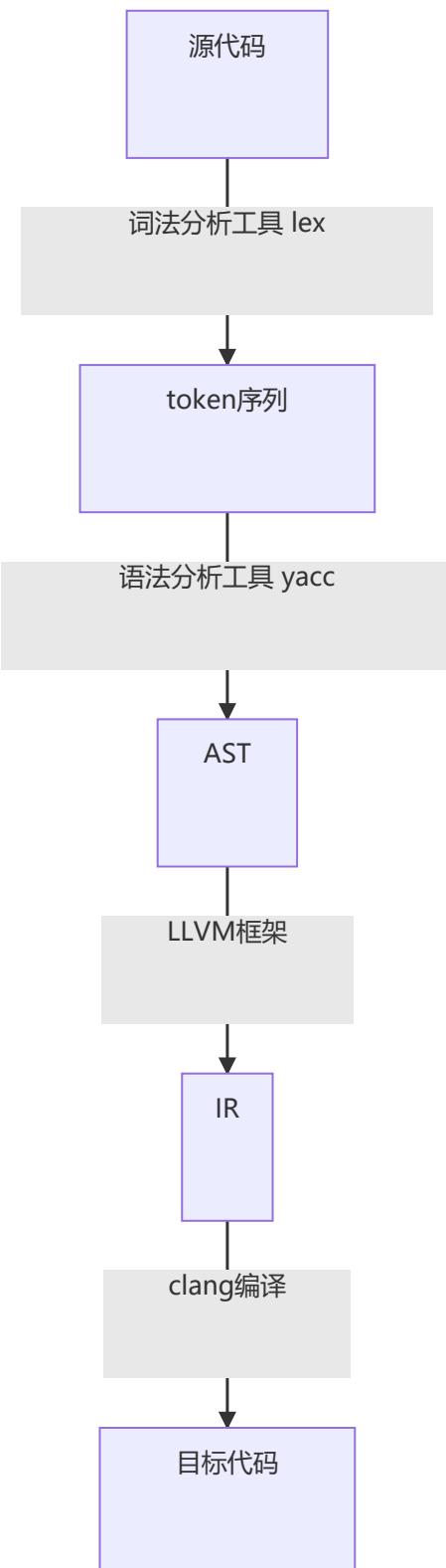
- 3.1 概述
- 3.2 AST节点定义
 - 3.2.1 basic.hpp
 - 3.2.2 type.hpp
 - 3.2.2.1 类型系统基类
 - 3.2.2.2 基本类型
 - 3.2.2.3 数组与指针
 - 3.2.2.4 结构与联合
 - 3.2.3 expression.hpp
 - 3.2.3.1 表达式基类
 - 3.2.3.2 数值常量
 - 3.2.3.3 标识符
 - 3.2.3.4 函数调用
 - 3.2.4 statement.hpp
 - 3.2.4.1 语句基类
 - 3.2.4.2 复合语句
 - 3.2.4.3 表达式语句
 - 3.2.4.4 选择语句
 - 3.2.4.5 迭代语句
 - 3.2.4.6 跳转语句
 - 3.2.4.7 标签语句
 - 3.2.5 declaration.hpp

- 3.2.5.1 声明基类
 - 3.2.5.2 变量声明
 - 3.2.5.3 形参声明
 - 3.2.5.4 函数声明
 - 3.2.5.5 类型声明
 - 3.2.6 program.hpp
 - 3.3 语法分析器具体实现
 - 3.3.1 yacc源文件定义段说明
 - 3.3.2 ANSI C语法BNF描述
 - 3.3.2.1 程序结构
 - 3.3.2.2 声明
 - 3.3.2.3 类型
 - 3.3.2.4 语句
 - 3.3.2.5 表达式
 - 3.3.3 常量折叠
 - 3.4 AST可视化示例
- 四、中间代码生成
- 4.1 概述
 - 4.2 LLVM框架简介
 - 4.3 codegen管理器定义
 - 4.4.1 type.cpp
 - 4.4.1.1 基本类型
 - 4.4.1.2 数组与指针
 - 4.4.1.3 结构与联合
 - 4.4.2 expression.cpp
 - 4.4.2.1 数值常量
 - 4.4.2.2 字符串常量
 - 4.4.2.3 标识符
 - 4.4.2.4 函数调用
 - 4.4.2.5 通用表达式
 - 4.4.3 statement.cpp
 - 4.4.3.1 复合语句
 - 4.4.3.2 表达式语句
 - 4.4.3.3 选择语句
 - 4.4.3.4 迭代语句
 - 4.4.3.5 跳转语句
 - 4.4.3.6 标签语句
 - 4.4.4 declaration.cpp
 - 4.4.4.1 变量声明
 - 4.4.4.2 函数声明
 - 4.4.4.3 类型声明
 - 4.4.5 program.cpp
- 五、静态语义检查
- 5.1 定义检查
 - 5.2 控制流语法检查
 - 5.3 类型检查
- 六、宏
- 七、测试
- 7.1 实验要求测试点
 - 7.1.1 快速排序
 - 7.1.2 矩阵乘法
 - 7.1.3 选课助手

一、序言

1.1 概述

本实验要求实现一个现代高级语言的编译器，我实现的编译器能够解析C语言语法的子集，并生成能够在目标机上运行的二进制程序。其工作流程如下：



1.2 开发环境

操作系统: Ubuntu 20.04 for x86-64

编译器实现部分:

- 编程语言: C++(11)
- 依赖工具: flex 2.6.4, bison 3.5.1, llvm 10.0.0

AST可视化部分:

- 编程语言: C++(11), Python(3.8.10)
- 依赖第三方库: graphviz 0.19.2

1.3 代码说明

1.3.1 源码目录结构

```
1  compiler
2  └── Dockerfile
3  └── README.md
4  └── src
5  |   ├── ast
6  |   |   ├── basic.hpp
7  |   |   ├── declaration.hpp
8  |   |   ├── expression.hpp
9  |   |   ├── program.hpp
10 |   |   ├── statement.hpp
11 |   |   └── type.hpp
12 |   ├── codegen
13 |   |   ├── codegen.cpp
14 |   |   ├── codegen.hpp
15 |   |   ├── declaration.cpp
16 |   |   ├── expression.cpp
17 |   |   ├── program.cpp
18 |   |   ├── statement.cpp
19 |   |   └── type.cpp
20 |   └── ecc.l
21 |   └── ecc.y
22 |   └── macro
23 |   |   ├── macro.cpp
24 |   |   └── macro.hpp
25 |   └── main.cpp
26 |   └── Makefile
27 |   └── utils
28 |       ├── json2dot.py
29 |       ├── visualizer.cpp
30 |       └── visualizer.hpp
31 └── test
32     ├── auto-advisor.c
33     ├── demo
34     |   ├── demo.c
35     |   └── header.h
36     └── Makefile
37     ├── matrix-multiplication.c
38     └── quicksort.c
```

- `ecc.l` 和 `ecc.y` 分别为词法分析文件和语法分析文件
- `main.cpp` 为编译器主入口
- `ast` 目录下文件均为抽象语法树的节点类型定义
- `codegen` 目录下文件均用于生成中间代码
- `utils` 目录下文件均用于AST可视化

`demo` 文件夹中的附件展示了本编译器可支持的**所有**语法规，如有需要可以阅读其中的 `demo.c`，使用本实验实现的编译器可以编译并正常工作

`Dockerfile` 用于构建可以编译本实验代码的环境，方便测试

1.3.2 代码运行

在代码根目录下执行 `docker build -t test .` 生成名为 `test` 的镜像，运行 `docker run -it /bin/bash` 进入 docker，执行 `make` 即可编译生成名为 `ecc` 的编译器程序。

运行 `./ecc FILENAME` (`FILENAME` 为合法的C源代码文件)，会自动编译生成名为 `a.out` 的可执行文件。如果不想使用默认的名字，可以用 `-o` 参数指定输出文件名；还可以使用 `-E` 选项输出宏展开后的源文件。例如 `./ecc src.c -o src -E src.i` (用法均同gcc)

编译器在生成AST后会将结果写为一个json文件，存在 `./tmp` 目录下，如果需要查看AST的可视化效果，可以在 `src` 目录下执行 `python3 utils/json2dot.py <json filename>` 生成对应的可视化png图片

如果需要进行测试，在执行 `make` 之后再执行 `make test` 会自动编译三个测试点源码并运行三个测试程序，也可以自行使用 `./ecc ...` 进行手动编译

1.3.3 版本控制与代码风格

实验全程使用 `git` 进行版本控制，仓库地址：<https://github.com/Esifiel/compiler>。此仓库于实验验收结束之后公开，此前均处于 `private` 状态

截至2022年5月18日，仓库共有47个commit

在代码风格上，

1. 如1.3.1节所示，对于每一种不同的语法树节点都进行了模块的分割，并且再细分为AST定义和中间代码生成两个不同的方面
2. 在各个模块中，命名规则基本遵循：变量名全小写、函数名除第一个单词外首字母大写、类名单词全首字母大写

二、词法分析

从第二节开始，为了篇幅简洁，所有讲述实现的章节中只贴出部分实现用于举例，**不代表同类型的其他内容没有实现**，完整实现详见附件

2.1 概述

词法分析器是从源代码生成token序列的中间程序。本编译器的词法分析部分借助lex实现。

2.2 token的正则表达式描述

2.2.1 C语言关键字

类型	关键字
类型定义	"char", "short", "int", "long", "float", "double" "struct", "enum", "union", "void", "unsigned", "signed"
存储修饰符	"auto", "register", "extern", "static"
跳转语句	"switch", "else", "if", "break", "return" "continue", "goto", "case", "default"
循环语句	"for", "do", "while"
类型修饰符	"const", "volatile"
其他	"typedef", "sizeof"

2.2.2 数据表示

类型	正则表达式
十进制长整数	-?(0 ([1-9][0-9]*))
八进制长整数	-?0[0-7]*
十六进制长整数	-?0[xX][0-9a-fA-F]+
双精度浮点数	-?(0 ([1-9][0-9]*)).([0-9]+)?
标识符	[A-Za-z_][A-Za-z0-9_]*
常规字符	\.\'
转义字符	\\\[abfnrtv\\\"\\]\\\'
八进制表示字符	\\\'0[0-7]{1,3}\\'
十六进制表示字符	\\\'x[0-9a-fA-F]{1,2}\\'
字符串常量	(\"([^\"] \\\\\")*)+

2.2.3 运算符

类型	运算符
算术运算符	"+", "-", "*", "/", "%", "++", "--"
位运算符	"&", " ", "~", "^", "<<", ">>"
逻辑运算符	"&&", " ", "!"
赋值运算符	"=", "+=", "-=", "*=", "/=", "%=", "&=", " =", "^=", "<<=", ">>="
关系运算符	"<", ">", "<=", ">=", "!=" , "=="
其他	"?", ":" (条件表达式), ":" (逗号运算符) "." (结构体成员访问), "->" (结构体指针访问) "&" (取地址), "*" (解引用), "sizeof" (取类型大小) "(", ")" (强制类型转换), "[", "]" (下标访问)

2.2.4 注释及其他

	说明
";"	语句结尾
"..."	可变参数标志
"{", "}"	复合语句标志
[\t\n]	空白符
"/\/*"	单行注释
"/**([^*] \""+[^/*])***/"	可跨行注释
\\$	续行符

2.3 词法分析器具体实现

大多数token被正则表达式匹配到时，都直接返回对应的标识符，交由语法分析器进一步处理，例如：

```
1  "double"           { return DOUBLE; }
2  "int"              { return INT; }
3  "struct"           { return STRUCT; }
```

对于数值常量，需要使用 `strtol` 函数转为对应的数值并存入yyval：

```
1  -?0[0-7]*          {
2      debug();
3      yyval.num.longValue = strtol(yytext, NULL,
4          8);
5      return NUMLONG;
6  -?0[xX][0-9a-fA-F]+ {
7      debug();
8      yyval.num.longValue = strtol(yytext, NULL,
9          16);
10     return NUMLONG;
11 -?(0|([1-9][0-9]*))(\.[0-9]+)? {
12     debug();
13     yyval.num.doubleValue = atof(yytext);
14     return NUMDOUBLE;
15 }
```

yyval.num是一个union类型的值，具体定义见第三节

对于字符或字符串常量，主要是需要对转义字符进行处理，我自己实现了一个replace函数，用于对转义字符文本进行替换：

```
1  \'\\x[0-9a-fA-F]{1,2}\'
2                                {
3                                     // hexadecimal char
4                                     debug();
5                                     yytext[strlen(yytext) - 1] = '\0';
6                                     yyval.num.charValue = strtol(yytext + 3,
7                                         NULL, 16);
```

```

6                         return NUMCHAR;
7                     }
8     ("([^\"]|\\\"")*\""+ {
9                     // string const
10                    debug();
11                    // strip "
12                    yyval.stringValue = new
13                    string(string(yytext).substr(1, strlen(yytext) - 2));
14                    // multiple strings will be combined
15                    replace(*yyval.stringValue, "\\"\"", "");
16                    // translate escape char
17                    replace(*yyval.stringValue, "\\a", "\a");
18                    replace(*yyval.stringValue, "\\b", "\b");
19                    replace(*yyval.stringValue, "\\f", "\f");
20                    replace(*yyval.stringValue, "\\n", "\n");
21                    replace(*yyval.stringValue, "\\r", "\r");
22                    replace(*yyval.stringValue, "\\t", "\t");
23                    replace(*yyval.stringValue, "\\v", "\v");
24                    replace(*yyval.stringValue, "\\\\"", "\\\"");
25                    replace(*yyval.stringValue, "\\\\'", "\\'");
26                    replace(*yyval.stringValue, "\\\\\\", "\\\"");
27                    return STRING;
}

```

如果是空白字符、注释或续行符，就都忽略：

```

1 [ \t\n]          {}
2 /*.*             {}
3 /*([^\*]|*+[/*])*/*+ /"  {}
4 \\$              {}

```

2.4 结果示例

```
token 0:      void
token 1:      quicksort
token 2:      (
token 3:      int
token 4:      A
token 5:      [
token 6:      ]
token 7:      ,
token 8:      int
token 9:      len
token 10:     )
token 11:     {
token 12:     int
token 13:     i
token 14:     ,
token 15:     j
token 16:     ,
token 17:     pivot
token 18:     ,
token 19:     temp
token 20:     ;
token 21:     if
token 22:     (
token 23:     len
token 24:     <
token 25:     2
token 26:     )
token 27:     return
token 28:     ;
token 29:     pivot
token 30:     =
token 31:     A
token 32:     [
token 33:     len
token 34:     /
token 35:     2
token 36:     ]
token 37:     ;
token 38:     for
token 39:     (
token 40:     i
token 41:     =
token 42:     0
token 43:     ,
token 44:     j
token 45:     =
token 46:     len
token 47:     -
token 48:     1
token 49:     ;
token 50:     ;
token 51:     i
token 52:     ++
token 53:     ,
token 54:     j
token 55:     --
token 56:     )
token 57:     {
token 58:     while
token 59:     (
token 60:     A
```

三、语法分析与AST生成

3.1 概述

语法分析器的作用是解析token序列，判断程序的语法结构是否符合语法规则，并为程序构建语法树。本编译器的语法分析部分借助yacc实现。本实验中所有C语言相关语法的地方均指ANSI C（即标准C、C89），不涉及后续标准如C99等添加的新语法。

3.2 AST节点定义

这一节描述代码中 `ast` 目录下的各个头文件，主要涉及各种节点的类定义

3.2.1 basic.hpp

在 `basic.hpp` 中定义了最基本的AST节点 `Node` 类：

```
1 class Node
2 {
3     public:
4         Node() {}
5         virtual string getName() = 0;
6         virtual llvm::Value *CodeGen(CodeGenerator &context) { return nullptr; };
7 }
```

其中，

- `getName` 方法返回节点对应的名称
- `CodeGen` 方法用于生成节点对应的中间代码

CodeGenerator类为自定义类，用于中间代码生成，Value类为llvm内部类，具体定义与说明见第四节

此外，还需要定义一些标记操作符、值类型和类型的枚举值，方便后续代码的编写：

```
1 enum op_type { OP_NONE, OP_EQ, OP_ADD, OP_ADDRESSOF, OP_ASSIGN, OP_COMMA,
2     OP_INDEX, ... };
3 enum val_type { VAL_NONE, VAL_CHAR, ... };
4 enum type_type { TYPE_NONE, TYPE_CHAR, TYPE_STRUCT, TYPE_ARRAY, TYPE_QUALIFIER,
... };
```

其中需要说明的是，`OP_NONE` 在后续代码中用于表示一个叶表达式，即一个常量或标识符，不需要解析操作符。

3.2.2 type.hpp

`type.hpp` 负责定义语言的类型系统。

首先需要说明的是，在C语言中定义一个变量时，标识符前可能会由多个部分组成，比如 `static const unsigned int *p;`。从C语言语法的角度来看：

- 基本类型名（如int、double，包括unsigned和signed前缀）以及用户自定义的类型名（struct a, union b, enum c以及typedef定义的别名）称为"Type"
- const、volatile称为"Qualifier"，即类型描述符
- static等描述变量作用范围的称为存储描述符，我认为也可以算是"Qualifier"的一部分
- "Qualifier"和"Type"是同级的，但是指针描述符 `*` 是和标识符（Identifier）同级的，这也是为什么 `int *p, q;` 定义的q是int类型，`int *p, *q;` 定义的q才是一个int的指针

为了将 `*` 所描述的指针类型和其他描述符都加入"Type"中，我在实现的时候除了为"类型"定义了 `TypeSpecifier` 类，同时为了顺应语法规则还定义了 `Qualifier` 类，并且为 `Identifier` 类（见3.2.3节）也定义了 `Qualifier` 类的成员，但最终这些信息都会集中到 `TypeSpecifier` 对象中，`Qualifier` 类只是作为一个中间产物，不参与中间代码生成。

3.2.2.1 类型系统基类

`Qualifier` 类定义：

```

1  class Qualifier : public Node
2  {
3  public:
4      bool isconst, isvolatile;
5      uint64_t pcnt;
6
7      Qualifier() : isconst(false), isvolatile(false), pcnt(0) {}
8      Qualifier(uint64_t i) : isconst(false), isvolatile(false), pcnt(i) {}
9
10     virtual string getName() { return "\"Qualifier\""; }
11 };

```

其中，

- `isconst`, `isvolatile` 标记该类型是否有对应的修饰
- `pcnt` 记录指针的级别，即类型定义中 `*` 的个数

`isvolatile` 字段是出于扩展考虑加入的，在本实验中目前暂未实现 `volatile` 关键字

`TypeSpecifier` 类定义：

```

1  class TypeSpecifier : public Node
2  {
3  public:
4      enum type_type type;
5      bool isunsigned;
6      Qualifier *qual;
7
8      TypeSpecifier(enum type_type t) : type(t), isunsigned(false), qual(nullptr)
9      {}
9      TypeSpecifier(enum type_type t, Qualifier *q) : type(t), isunsigned(false),
10      qual(q) {}
11
11     virtual string getName() { return "\"TypeSpecifier\""; }
12     virtual Type *getType(CodeGenerator &ctx) = 0;
13     virtual TypeSpecifier *getRootType() = 0;
14     virtual uint64_t getSize() = 0;
15     virtual bool isAggregateType() = 0;
16     virtual bool isIterableType() = 0;
17 };

```

`TypeSpecifier` 类的成员变量最终会包含所有的类型定义信息。除此之外，关于几个成员函数的说明：

- `getType` 方法用于返回该类型在 LLVM 中的对应描述，类型为 `llvm::Type *`，具体定义见第四节
- `getRootType` 方法用于返回该类型的基本类型，用于获取数组或指针
- `getSize` 方法返回该类型的大小，用于实现 `sizeof` 运算符
- `isAggregateType` 方法返回该类型是否是一个聚合类型，即是否是结构体或联合体
- `isIterableType` 方法返回该类型是否是一个可枚举类型，即是否是数组或指针

3.2.2.2 基本类型

以 `int` 类型为例：

```

1  class IntType : public TypeSpecifier
2  {
3  public:
4      IntType() : TypeSpecifier(TYPE_INT) {}
5      IntType(Qualifier *q) : TypeSpecifier(TYPE_INT, q) {}
6
7      virtual string getName() { return "\"IntType\""; }
8      virtual TypeSpecifier *getRootType() { return this; }
9      virtual uint64_t getSize() { return sizeof(int); }
10     virtual bool isAggregateType() { return false; }
11     virtual bool isIterableType() { return false; }
12 };

```

3.2.2.3 数组与指针

数组和指针都继承自可枚举类型，其定义如下：包含一个基本类型 basictype

```

1  class IterableType : public TypeSpecifier
2  {
3  public:
4      TypeSpecifier *basictype;
5
6      IterableType(TypeSpecifier *bt, enum type_type type) : basictype(bt),
7      TypeSpecifier(type) {}
8
9      virtual TypeSpecifier *getRootType() { return basictype->getRootType(); }
10     virtual bool isAggregateType() { return false; }
11     virtual bool isIterableType() { return true; }
12 };

```

数组类型：继承自 IterableType，需要再多一个成员变量 size 表示大小。同时 getSize 方法的实现也有别于指针类型，其值为数组大小 * 基本类型大小

```

1  class MyArrayType : public IterableType
2  {
3  public:
4      uint64_t size;
5
6      MyArrayType(TypeSpecifier *t) : IterableType(t, TYPE_ARRAY), size(0) {}
7      MyArrayType(TypeSpecifier *t, uint64_t sz) : IterableType(t, TYPE_ARRAY),
8      size(sz) {}
9
10     virtual string getName() { return "\"MyArrayType\""; }
11     virtual uint64_t getSize() { return size * basictype->getSize(); }
12 };

```

3.2.2.4 结构与联合

结构体与联合体都继承自聚合类型，其定义如下：包含此类型声明的各个成员信息

```

1  class AggregateType : public TypeSpecifier
2  {
3      public:
4          string name;
5          vector<pair<vector<TypeSpecifier *> *, vector<Identifier *> *> *members;
6
7          AggregateType(enum type_type t) : name(""), members(nullptr),
8          TypeSpecifier(t) {}
9
10         virtual string getName() { return "\AggregateType\"; }
11         virtual bool isAggregateType() { return true; }
12         virtual bool isIterableType() { return false; }
13         pair<TypeSpecifier *, Identifier *> getMemberDef(Identifier *id);
14     };

```

成员信息统一用 `members` 变量记录，该成员类型比较复杂，之所以定义中全是指针是为了方便AST的构建，其表示的是一系列（类型，标识符）对的数组。以下面的结构体定义为例：

```

1  struct st {
2      int a, *b;
3      long c;
4      double d;
5  }

```

该结构体成员在 `members` 中记录为如下结构：

```

1  vector {
2      (
3          vector { IntType      , MyPointerType },
4          vector { Identifier a, Identifier b  }
5      ),
6      (LongType,   Identifier c),
7      (DoubleType, Identifier d)
8  }

```

`getMemberDef` 方法是一个辅助函数，用于返回一对结构体或联合体中的成员声明，包括类型与标识符

3.2.3 expression.hpp

`expression.hpp` 负责定义表达式节点。

3.2.3.1 表达式基类

通用 `Expression` 节点定义如下：

```

1  class Expression : public Node
2  {
3      public:
4          Expression *left;
5          Expression *right;
6          Expression *addition;
7          enum op_type op;
8          TypeSpecifier *type;
9
10         // terminal expr
11         Expression() : left(nullptr), right(nullptr), addition(nullptr),
12         op(OP_NONE), type(nullptr) {}
13         // unary expr

```

```

13     Expression(Expression *l, enum op_type o) : left(l), right(nullptr),
14     addition(nullptr), op(o), type(nullptr) {}
15     // binary expr
16     Expression(Expression *l, Expression *r, enum op_type o) : left(l),
17     right(r), addition(nullptr), op(o), type(nullptr) {}
18     // trinary expr
19     Expression(Expression *l, Expression *r, Expression *a, enum op_type o) :
20     left(l), right(r), addition(a), op(o), type(nullptr) {}
21     // for type casting
22     Expression(TypeSpecifier *t) : left(nullptr), right(nullptr),
23     addition(nullptr), op(OP_NONE), type(t) {}
24
25     virtual string getName() { return "\"Expression\""; }
26     virtual llvm::Value *CodeGen(CodeGenerator &ctx);
27 };

```

其中包含五个构造函数，不接受任何参数的用于 Identifier 和 Number 等叶表达式节点，接受一、二、三个子表达式节点的分别用于单目、双目、三目表达式的构建，接受一个类型参数的用于强制类型转换表达式

3.2.3.2 数值常量

该类实际上是对一个8字节内存空间按照不同类型进行解读：

```

1  class Number : public Expression
2  {
3  public:
4      uint8_t buf[8];
5      enum val_type valtype;
6
7      Number() : valtype(VAL_NONE) {}
8      Number(union union_num u, TypeSpecifier *t, enum val_type v) :
9      Expression(t), valtype(v) { memcpy(buf, &u, 8); }
10
11     virtual string getName() { return "\"Number\""; }
12
13     uint8_t charView() { return *(uint8_t *)buf; }
14     uint16_t shortView() { return *(uint16_t *)buf; }
15     uint32_t intView() { return *(uint32_t *)buf; }
16     uint64_t longView() { return *(uint64_t *)buf; }
17     float_t floatView() { return *(float_t *)buf; }
18     double_t doubleView() { return *(double_t *)buf; }
19 };

```

3.2.3.3 标识符

标识符可以拥有初始化表达式 init，同时也包含可能存在的指针描述符 qual

```

1  class Identifier : public Expression
2  {
3      public:
4          string name;
5          Expression *init;
6          Qualifier *qual;
7
8          Identifier(string v) : name(v), init(nullptr), qual(nullptr) {}
9          Identifier(string v, Expression *ini) : name(v), init(ini), qual(nullptr) {}
10
11         virtual string getName() { return "\"Identifier\""; }
12     };

```

3.2.3.4 函数调用

函数调用包含调用函数名 `name` 和一个参数表达式列表 `varlist` :

```

1  class FunctionCall : public Expression
2  {
3      public:
4          Expression *name;
5          vector<Expression *> *varlist;
6
7          FunctionCall(Expression *n, vector<Expression *> *l) : name(n), varlist(l)
8          {}
9
10         virtual string getName() { return "\"FunctionCall\""; }
11     };

```

3.2.4 statement.hpp

`statement.hpp` 负责定义各种语句节点。

3.2.4.1 语句基类

通用 `Statement` 节点定义如下:

```

1  class Statement : public Node
2  {
3      public:
4          Statement *tail;
5          Statement *next;
6
7          Statement() : tail(nullptr), next(nullptr) {}
8
9          virtual string getName() { return "\"Statement\""; }
10     };

```

`Statement` 节点在AST中是用left-child-right-sibling方法表示的，因此每个节点只需要指向下一条语句的指针 `next` 即可。由于一个语句链的最后一条语句也是很常用的，因此这里我还加了一个 `tail` 指针，方便快速定位

3.2.4.2 复合语句

在ANSI C语法中，复合语句可以按序包含两个内容：首先是一系列变量或类型声明 `decl`，然后是一系列语句 `stmt`：

```
1 class CompoundStatement : public Statement
2 {
3     public:
4         Declaration *decl;
5         Statement *stmt;
6
7         CompoundStatement(Declaration *d, Statement *s) : decl(d), stmt(s) {}
8
9         virtual string getName() { return "\"CompoundStatement\""; }
10        virtual llvm::Value *CodeGen(CodeGen &ctx);
11    };
```

3.2.4.3 表达式语句

表达式语句即一个以分号结尾的、包含一个表达式（可以为空）`expr` 成员的语句：

```
1 class ExpressionStatement : public Statement
2 {
3     public:
4         Expression *expr;
5
6         ExpressionStatement() : expr(nullptr) {}
7         ExpressionStatement(Expression *e) : expr(e) {}
8
9         virtual string getName() { return "\"ExpressionStatement\""; }
10    };
```

3.2.4.4 选择语句

选择语句的基本成员包括一个条件表达式 `cond` 和满足条件时执行的语句链 `stmt`：

```
1 class SelectionStatement : public Statement
2 {
3     public:
4         Expression *cond;
5         Statement *stmt;
6
7         SelectionStatement(Expression *c, Statement *s) : cond(c), stmt(s) {}
8
9         virtual string getName() { return "\"SelectionStatement\""; }
10    };
```

选择语句有两种：if-else语句和switch-case语句。对于if-else语句而言，`stmt` 成员表示的语句链是if子句的内容和else子句的内容；对于switch-case语句而言，`stmt` 成员表示的语句链是各个case语句。

3.2.4.5 迭代语句

迭代语句（即循环语句）的基本成员包括一个循环条件 `cond` 和满足条件时执行的语句链 `stmt`：

```

1  class IterationStatement : public Statement
2  {
3      public:
4          Expression *cond;
5          Statement *stmt;
6
7          IterationStatement(Expression *c, Statement *s) : cond(c), stmt(s) {}
8
9          virtual string getName() { return "\"IterationStatement\""; }
10     };

```

循环语句有三种：while语句、do-while语句和for语句。对于前两者，只需要上述两个成员即可表示，对于for语句，还需要额外的初始化表达式 `init` 和每一轮循环后都执行的表达式 `end`：

```

1  class ForStatement : public IterationStatement
2  {
3      public:
4          Expression *init;
5          Expression *end;
6
7          ForStatement(Expression *i, Expression *c, Expression *e, Statement *s) :
8              init(i), IterationStatement(c, s), end(e) {}
9
10         virtual string getName() { return "\"ForStatement\""; }
11     };

```

3.2.4.6 跳转语句

跳转语句包括return语句、break语句、continue语句，goto语句

return语句有一个表达式成员 `res`，该成员可以为空：

```

1  class ReturnStatement : public Statement
2  {
3      public:
4          Expression *res;
5
6          ReturnStatement() : res(nullptr) {}
7          ReturnStatement(Expression *r) : res(r) {}
8
9          virtual string getName() { return "\"ReturnStatement\""; }
10     };

```

break语句和continue语句不需要额外的成员变量，在中间代码生成时会根据上下文进行实现。

goto语句需要一个string成员 `name` 指明要跳转的标签：

```

1  class GotoStatement : public Statement
2  {
3      public:
4          string label;
5
6          GotoStatement(string s) : label(s) {}
7
8          virtual string getName() { return "\"GotoStatement\""; }
9     };

```

3.2.4.7 标签语句

标签语句包括case语句和标签定义。

case语句的开关值必须是一个 **Number** 类型的数值常量：

```
1 class CaseStatement : public Statement
2 {
3     public:
4         Number *val;
5
6         CaseStatement(Number *e) : val(e) {}
7
8         virtual string getName() { return "\"CaseStatement\""; }
9     };
```

需要说明的是，在本实验中，为了方便起见，认为case语句的含义仅限于case本身，不包括后续的子语句，所以 **CaseStatement** 类没有 **Statement** 类的成员；同时，由于case语句必须和switch语句配合使用，所以认为case语句下的子语句归switch语句所属，在为switch语句生成中间代码时统一都生成出来。

标签定义包含一个string类型的标签名：

```
1 class LabelStatement : public Statement
2 {
3     public:
4         string label;
5
6         LabelStatement(string s) : label(s) {}
7
8         virtual string getName() { return "\"LabelStatement\""; }
9     };
```

3.2.5 declaration.hpp

declaration.hpp 负责定义各种声明。

3.2.5.1 声明基类

通用 **Declaration** 节点定义如下，和 **Statement** 类类似，也是一个链表结构：

```
1 class Declaration : public Node
2 {
3     public:
4         Declaration *tail;
5         Declaration *next;
6
7         Declaration() : tail(nullptr), next(nullptr) {}
8
9         virtual string getName() { return "\"Declaration\""; }
10    };
```

3.2.5.2 变量声明

变量声明的结构类似结构体的成员定义，也是由 **TypeSpecifier** 类和 **Identifier** 类列表组成的序对：

```

1  class VariableDeclaration : public Declaration
2  {
3  public:
4      vector<TypeSpecifier *> *types;
5      vector<Identifier *> *ids;
6
7      VariableDeclaration(vector<TypeSpecifier *> *t, vector<Identifier *> *i) :
8          types(t), ids(i) {}
9
10     virtual string getName() { return "\"VariableDeclaration\""; }
11 };

```

3.2.5.3 形参声明

函数的形参声明类似变量声明，但是是类型和标识符的一一对应（也有可能只有类型），不存在一个类型对应多个表示符的情况，所以实验中也作为单独一个类来实现：

```

1  class Parameter : public Declaration
2  {
3  public:
4      TypeSpecifier *type;
5      Identifier *id;
6      bool isvariableargs;
7
8      Parameter(TypeSpecifier *t) : type(t), id(nullptr), isvariableargs(false) {}
9      Parameter(TypeSpecifier *t, Identifier *i) : type(t), id(i),
10         isvariableargs(false) {}
11
12     virtual string getName() { return "\"Parameter\""; }
13 };

```

在C语言语法中，参数列表的推导能够识别可变参数标识`...`，因此这里还为 `Parameter` 类加上一个 `bool`类型的变参标志

3.2.5.4 函数声明

函数声明包括四个部分：返回类型 `rettype`、函数名 `id`、参数列表 `params`、函数体 `stmts`

```

1  class FunctionDeclaration : public Declaration
2  {
3  public:
4      TypeSpecifier *rettype;
5      Identifier *id;
6      Parameter *params;
7      CompoundStatement *stmts;
8
9      FunctionDeclaration(TypeSpecifier *t, Identifier *n, Parameter *p,
10         CompoundStatement *s) : rettype(t), id(n), params(p), stmts(s) {}
11
12     virtual string getName() { return "\"FunctionDeclaration\""; }
13 };

```

3.2.5.5 类型声明

类型声明包含一个 `TypeSpecifier` 类成员，主要用于非基本类型的声明：

```

1  class TypeDeclaration : public Declaration
2  {
3      public:
4          TypeSpecifier *type;
5
6          TypeDeclaration() : type(nullptr) {}
7          TypeDeclaration(TypeSpecifier *t) : type(t) {}
8
9          virtual string getName() { return "\"TypeDeclaration\""; }
10     };

```

在构造AST时，如果要声明的类型是一个基本类型，会使用空构造函数，并输出一个warning（按照gcc的逻辑），在生成中间代码时会忽略它

3.2.6 program.hpp

Program节点定义如下：

```

1  class Program : public Node
2  {
3      public:
4          Declaration *decl;
5
6          Program(Declaration *l) : decl(l) {}
7
8          virtual string getName() { return "\"Program\""; }
9          virtual llvm::Value *CodeGen(CodeGenerator &ctx);
10     };

```

Program是整个AST的根节点，一个C源代码程序是由各种声明构成的，所以Program节点的成员是一个指向首个Declaration节点的指针

3.3 语法分析器具体实现

3.3.1 yacc源文件定义段说明

- 使用 `%expect 1` 声明编译器语法只接受一个shift-reduce冲突，即由于if语句可选的else子句造成的冲突。yacc默认执行shift，即if-else的配对遵循就近原则
- 使用 `%union { ... }` 定义yyval，除了数值常量，都是各个AST节点类的指针
- 使用 `%token ...` 定义语法中的终结符
- 使用 `%type<...> ...` 将yyval中的成员与语法元素绑定
- 使用 `%start program` 指定语法分析以 `program` 为start symbol
- 定义AST的根节点 `Program *program;`，以及需要用到的一些字典结构：

```

1  map<string, Number *> constvar; // 声明为const的变量
2  map<string, AggregateType *> aggrdef; // 聚合类型定义
3  map<string, TypeSpecifier *> typealias; // typedef定义的类型别名

```

3.3.2 ANSI C语法BNF描述

BNF描述参考的是 http://www.cs.man.ac.uk/~pjy/bnf/c_syntax.bnf，在实验中略做了一些修改

由于完整语法很长，.y文件的内容非常复杂，没有办法很具体的逐条描述，所以下面只对部分语法和AST构建的思路做简单的文字介绍，没有把每条规则具体需要执行的代码贴上来

3.3.2.1 程序结构

一个C语言程序由各种声明组成：函数声明（function-definition）、变量声明（decl）、类型声明（decl）。所有声明构成一个声明列表（translation-unit）

```
1 program : translation-unit
2 ;
3 translation-unit : external-decl
4         | translation-unit external-decl
5 ;
6 external-decl : function-definition
7         | decl
8 ;
```

3.3.2.2 声明

函数声明：在C语言语法中，一个函数定义由三个部分组成：返回类型（decl-specs）、声明符（declarator）、函数体（一个复合语句，compound-stat）。返回类型可以省略，缺省值为int类型

```
1 function-definition : decl-specs declarator compound-stat
2             | declarator compound-stat
3 ;
```

变量声明和类型声明：

- 变量声明需要指定变量类型（decl-specs），以及一个声明符列表（init-declarator-list），此列表可能包含多个变量、每个变量可能会有初始化表达式
- 类型声明包含结构体等的定义（decl-specs），或者是通过typedef定义的类型别名（IDENTIFIER）

```
1 decl : decl-specs init-declarator-list DELIM
2     | decl-specs DELIM
3     | TYPEDEF decl-specs IDENTIFIER DELIM
4 ;
```

声明符（declarator）：用于对一个标识符进行各种描述。同一个类型的变量声明可以写在一个声明符列表（init-declarator-list）中，声明符之间用逗号（COMMA）连接。每个声明符都可以有初始化表达式（initializer）

```
1 init-declarator-list : init-declarator
2             | init-declarator-list COMMA init-declarator
3             ;
4 init-declarator : declarator
5             | declarator ASSIGN initializer
6 ;
```

声明符的修饰包括指针修饰（pointer）和类型修饰（type-qualifier），一个声明符前可以有若干个*（MULORDEREREFERENCE），每个*都可以有自己的类型修饰。

声明符的主体有很多种：

- 普通变量：IDENTIFIER
- 数组：由中括号（LB、RB）包围的常量表达式（const-exp），或者是不带常量表达式，即省略最后一维大小的作为参数的数组
- 函数：由小括号（LP、RP）包围的参数列表（param-type-list，语法类似变量声明），或者没有参数的函数

所有描述性单元最终都会集中到主体的类型定义中。

```
1 declarator : pointer direct-declarator
```

```

2           | direct-declarator
3           ;
4 direct-declarator : IDENTIFIER
5           | direct-declarator LB const-exp RB
6           | direct-declarator LB RB
7           | direct-declarator LP param-type-list RP
8           | direct-declarator LP RP
9           ;
10      pointer : MULORDEREference type-qualifier-list
11      | MULORDEREference
12      | MULORDEREference type-qualifier-list pointer
13      | MULORDEREference pointer
14      ;
15      type-qualifier-list : type-qualifier
16      | type-qualifier-list type-qualifier
17      ;

```

3.3.2.3 类型

一个类型可以有三个部分：存储修饰符（storage-class-specifier，如extern、static）、主类型（type-spec，如int、double、struct st）、类型修饰符（type-qualifier，如const），并且主类型可以省略，缺省值为int类型

```

1 decl-specs : storage-class-specifier
2           | storage-class-specifier decl-specs
3           | type-spec
4           | type-spec decl-specs
5           | type-qualifier
6           | type-qualifier decl-specs
7           ;

```

结构体和联合体：聚合类型的定义可以有两个部分：类型名称（IDENTIFIER）、成员定义（struct-decl-list）。两者可以省略其一，省略类型名称时为匿名结构或联合，省略成员定义时认为是前向声明

成员定义的语法类似变量声明。

```

1 struct-or-union-spec : struct-or-union IDENTIFIER LC struct-decl-list RC
2           | struct-or-union LC struct-decl-list RC
3           | struct-or-union IDENTIFIER
4           ;

```

3.3.2.4 语句

C语言语法将语句分为标签语句、表达式语句、复合语句、选择语句、迭代语句、跳转语句几大类：

```

1 stat : labeled-stat
2   | exp-stat
3   | compound-stat
4   | selection-stat
5   | iteration-stat
6   | jump-stat
7   ;

```

标签语句：switch语句中使用的case和default都属于标签语句，还包括由标识符（IDENTIFIER）和冒号（COLON）的自定义标签

```

1   labeled-stat : IDENTIFIER COLON stat
2           | CASE const-exp COLON
3           | DEFAULT COLON
4           ;

```

表达式语句：内容为一个表达式，也可以为空

```

1   exp-stat : exp DELIM
2           |      DELIM
3           ;

```

复合语句：由一对花括号 (LC、 RC) 包裹的一系列子声明 (decl-list) 和子语句 (stat-list)

```

1   compound-stat : LC decl-list stat-list RC
2           | LC          stat-list RC
3           | LC decl-list          RC
4           | LC          RC
5           ;

```

选择语句：if语句和switch语句都属于选择语句，if语句还有可选的else子句

```

1   selection-stat : IF LP exp RP stat
2           | IF LP exp RP stat ELSE stat
3           | SWITCH LP exp RP stat
4           ;

```

迭代语句：包括while、do-while和for，其中for语句的三个表达式元素都是可以为空的

```

1   iteration-stat : WHILE LP exp RP stat
2           | DO stat WHILE LP exp RP DELIM
3           | FOR LP exp DELIM exp DELIM exp RP stat
4           | FOR LP exp DELIM exp DELIM      RP stat
5           | FOR LP exp DELIM      DELIM exp RP stat
6           | FOR LP      DELIM exp DELIM exp RP stat
7           | FOR LP      DELIM exp DELIM      RP stat
8           | FOR LP      DELIM      DELIM exp RP stat
9           | FOR LP      DELIM      DELIM exp RP stat
10          | FOR LP      DELIM      DELIM      RP stat
11          ;

```

跳转语句：包括goto、break、continue、return

```

1   jump-stat : GOTO IDENTIFIER DELIM
2           | CONTINUE DELIM
3           | BREAK DELIM
4           | RETURN exp DELIM
5           | RETURN DELIM
6           ;

```

3.3.2.5 表达式

由于表达式种类太多，且含义比较简单，就不一一介绍了。简单来说，在BNF描述中，表达式都按照对应运算符优先级从低到高的顺序进行推导。此外，2.2.3节中提到的运算符在本实验中均有实现，详见.y源码。

3.3.3 常量折叠

在进行语法分析生成AST时，会进行常量折叠（Constant Folding），所有表达式节点由自定义函数 `calculate` 接管，该函数有用于单目运算符、双目运算符、三目运算符的三个重载。该函数会对操作数均为常数或`const`变量的可直接的表达式进行计算合并，并直接将节点生成为 `Number` 类节点；如果判断为无法合并，就仍然生成常规 `Expression` 节点。

`const`变量的来源：3.3.1节中提到了用于存储`const`变量的字典 `constvar`，在为变量定义生成AST时，会判断变量是否被定义为`const`且是一个数值常量，如果是，就存入字典中：

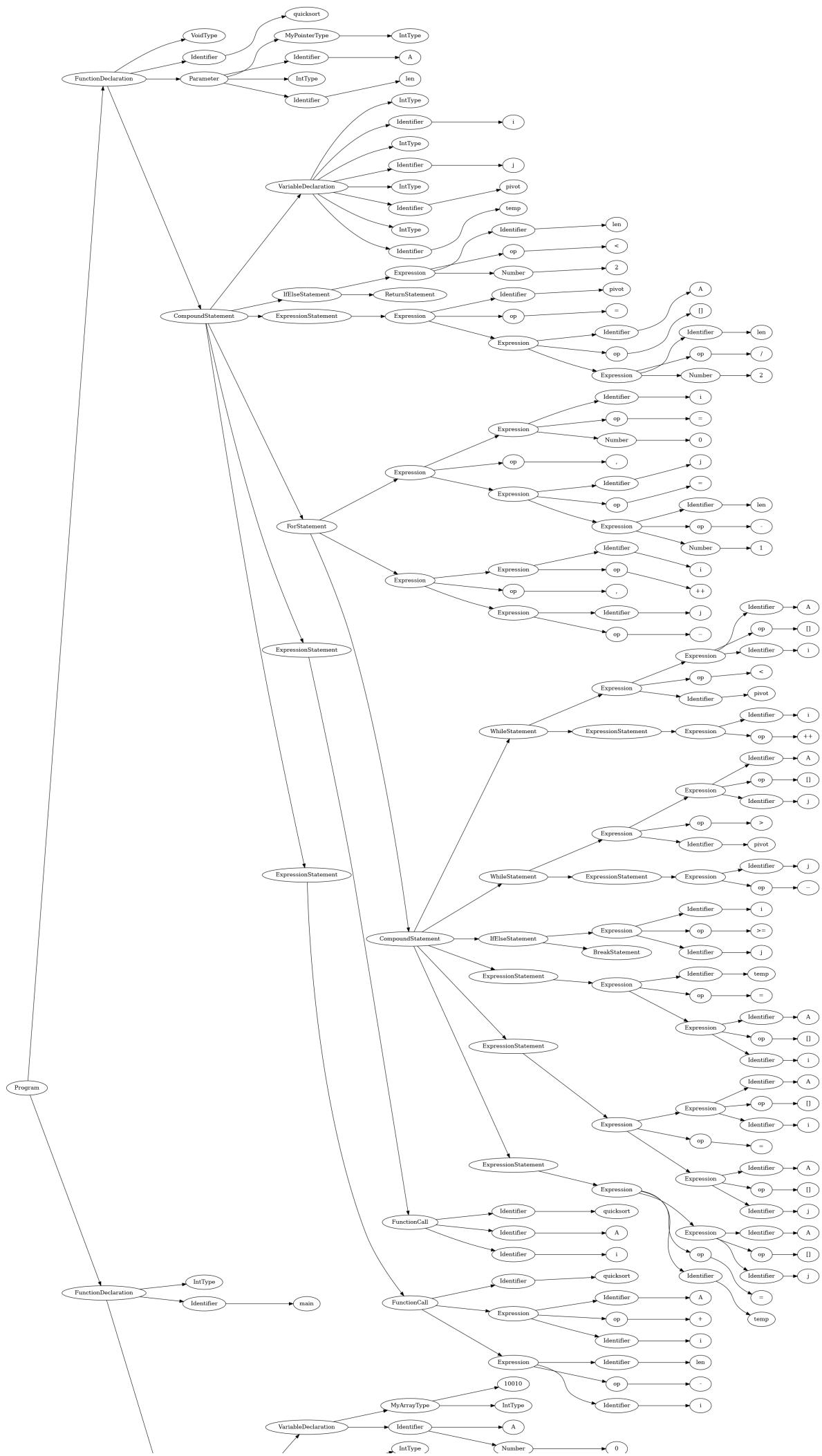
```
1  if (
2      var->qual && \
3      var->qual->isconst && \
4      !type->isIterableType() && \
5      !type->isAggregateType()
6  )
7      constvar[var->name] = (Number *) (var->init);
```

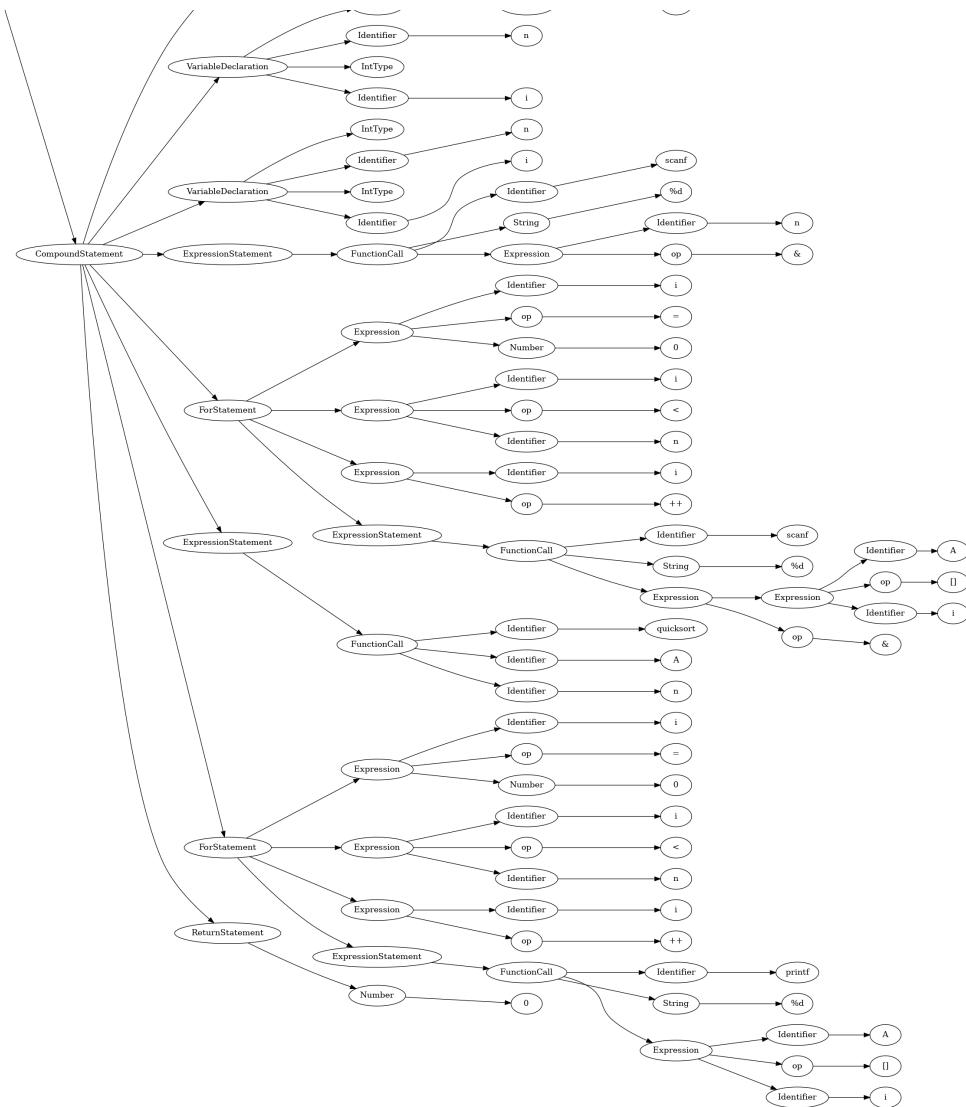
注意这里变量的初始化表达式也是被折叠过的，所以是一个 `Number` 类的指针

`calculate` 函数进行折叠计算时，还会涉及到隐式类型转换的问题。虽然C语言有很多的数值类型，但是从目前gcc的实现来看，常浮点数都认为是`double`，常整数在`int`范围内时认为是`int`（常字符也会认为是`int`），超过`int`范围时认为是`long`，因此我在实现时就做的比较简单了：

- 对于常量的比较运算和逻辑运算，折叠为一个 `LongType` 类型的结果
- 对于算术运算，如果操作数至少有一个是浮点类型，就折叠为一个 `DoubleType` 类型的结果，否则就同样折叠为一个 `LongType` 类型的结果

3.4 AST可视化示例





四、中间代码生成

4.1 概述

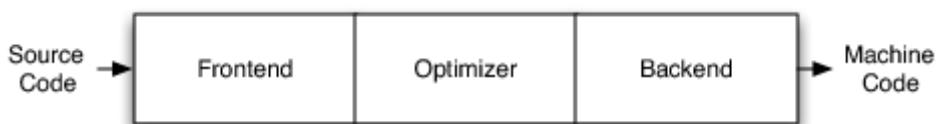
本实验采用的技术路线中，需要将AST转化为中间代码表示，再转化为目标可执行文件。中间代码是一种复杂性介于源程序语言和机器语言的一种表示形式，这一步将借助LLVM框架生成LLVM定义的中间表示：LLVM IR。

生成LLVM IR（源文件后缀）后，使用clang编译生成可执行文件即可，编译命令如下：

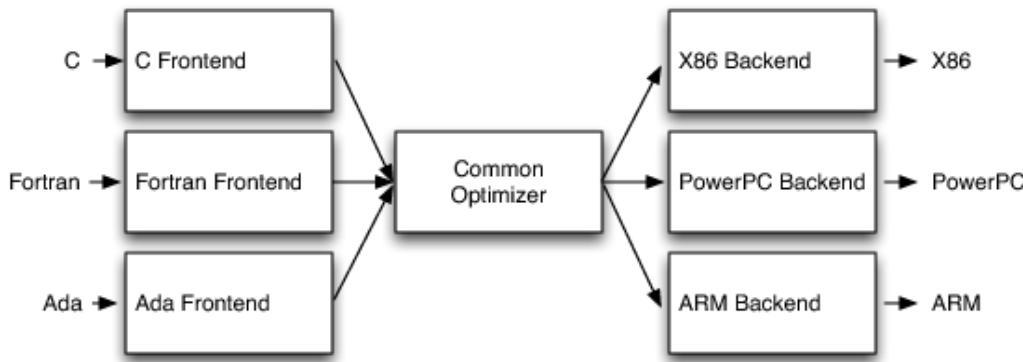
```
1 clang -O3 --target='llvm-config --host-target' <objname>.ll -o <objname>
```

4.2 LLVM框架简介

LLVM是一个用于建立编译器的基础框架，使用C++编写。该项目的目的是对于任意的编程语言都可以构建一个包括编译时、链接时、执行时等的语言执行器。



传统编译器架构



LLVM架构编译器

LLVM有很多子项目，本实验的中间代码生成部分只使用了其核心库。LLVM核心库提供了为不同前端和后端使用统一中间代码的生成支持，生成的代码表示称为 LLVM Intermediate Representation（LLVM IR）。

同时，LLVM核心库还提供了一个现代的源和目标无关的优化器，可以使用它对 LLVM IR 运行指定的优化或分析（LLVM Pass），在本实验的代码优化部分将会用到（目前暂未实现）

使用 LLVM IR 编程时会涉及到一些基本概念，在这里有必要事先进行说明：

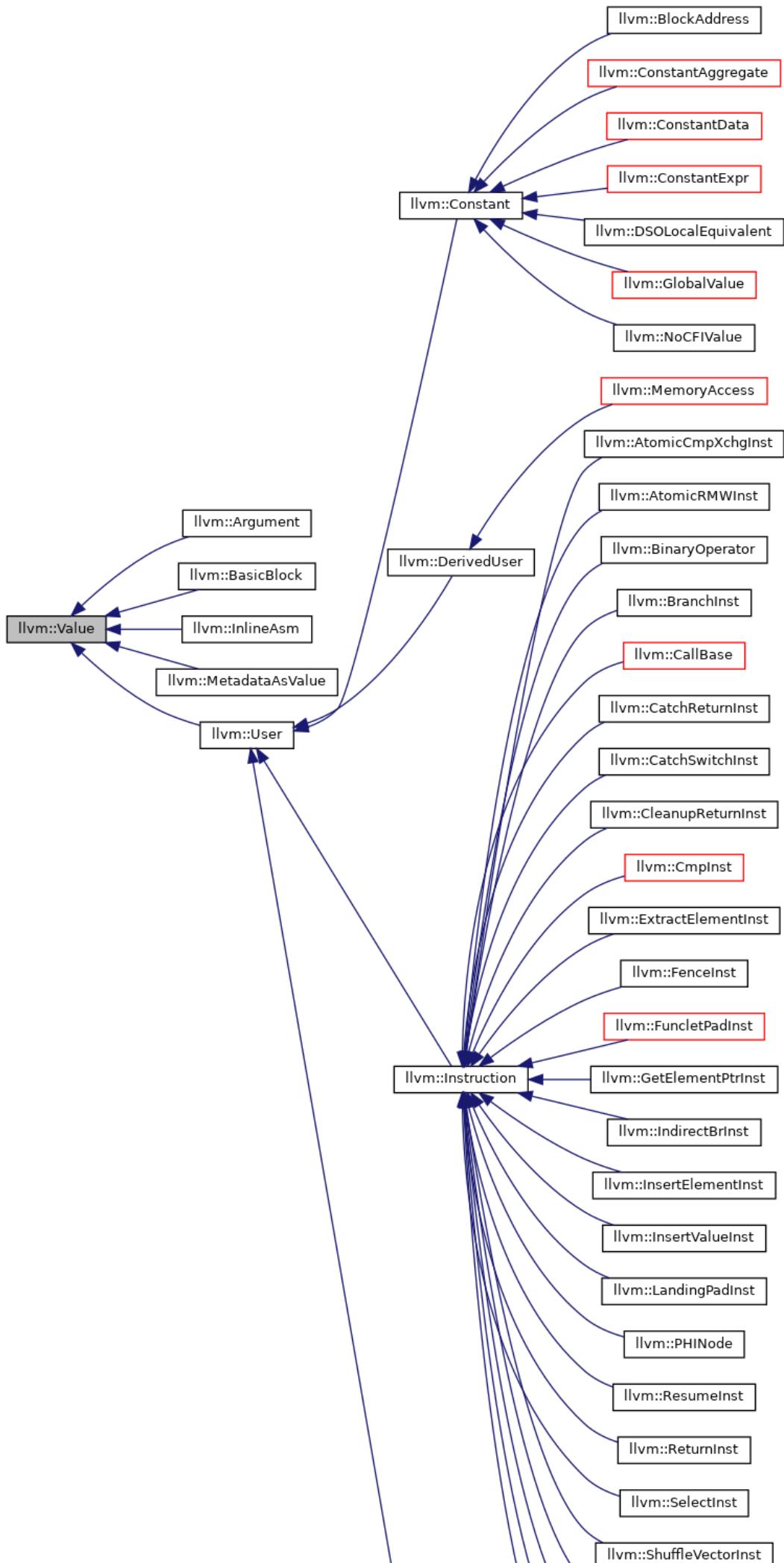
- Module： LLVM 的 `Module` 类的含义和编程语言的源文件类似，只是在 `Module` 中代码是用 IR 来表示的。在中间代码生成的过程中，生成的 IR 指令会不断添加到 `Module` 对象中，最终可以通过 `Module` 类提供的 API 来存入文件
- Function： LLVM 的 `Function` 类的含义和编程语言中的函数概念是一样的
- BasicBlock： LLVM 的 `BasicBlock` 类的含义和计算机体系结构中基本块（basic block）的概念是一样的。 LLVM IR 要求一个 `BasicBlock` 必须以终端指令（Terminator Instructions）结尾。（常见的终端指令即跳转型指令，如 `ret(urn)`、`br(anch)` 等，但终端指令并不仅限于这些）
- Instruction： LLVM 的 `Instruction` 类对应 LLVM IR 的每一条指令

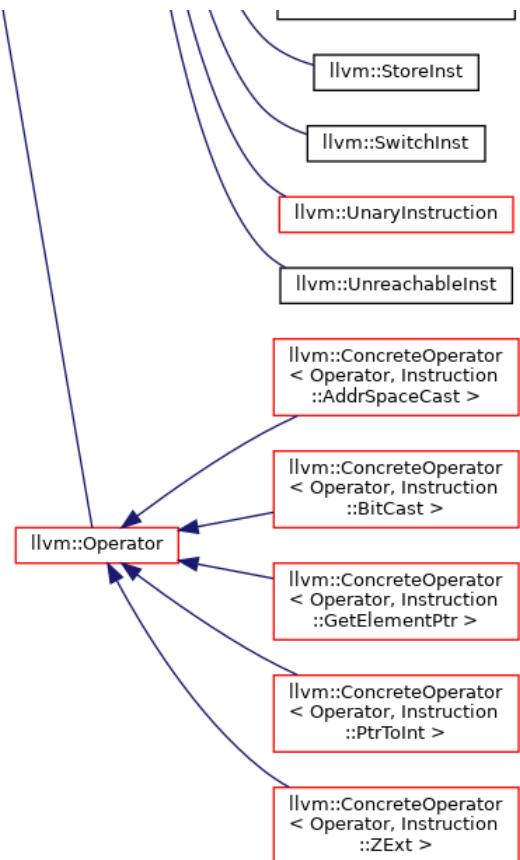
LLVM IR 编程基本流程如下：

1. 创建一个 `Module`
2. 在 `Module` 中添加 `Function`
3. 在 `Function` 中添加 `BasicBlock`
4. 在 `BasicBlock` 中添加 IR 指令

使用 LLVM 的 API 生成 IR 时，也有一些常用的类，在 LLVM 的 [文档](#) 中有详细说明，这里主要说明两个最重要的类：

1. `Value` 类：用来表示一个具有类型的值。下图是 LLVM 文档中展示的该类的继承关系，可以看到很多常用类都继承自 `Value`，包括 `BasicBlock`、`Constant`、`Instruction`，在 LLVM 中可以认为“一切皆 `Value`”，这也是为什么 AST 节点的 `CodeGen` 方法的返回值统一是 `Value *` 类型





2. **Type** 类: LLVM 定义的类型。在 LLVM 中, 对于一个有类型的值, 可以使用 `Value::getType()` 方法获取到它对应的类型。**Type** 的分类有很多, 下面是 LLVM 源码中关于不同类型的枚举值定义:

```

1 enum TypeID {      // PrimitiveTypes - make sure LastPrimitiveTyID stays up
2     to date.
3     VoidTyID = 0,    ///< 0: type with no size
4     HalfTyID,        ///< 1: 16-bit floating point type
5     FloatTyID,       ///< 2: 32-bit floating point type
6     DoubleTyID,      ///< 3: 64-bit floating point type
7     X86_FP80TyID,    ///< 4: 80-bit floating point type (X87)
8     FP128TyID,       ///< 5: 128-bit floating point type (112-bit
9     mantissa)
10    PPC_FP128TyID,   ///< 6: 128-bit floating point type (two 64-bits,
11     PowerPC)
12    LabelTyID,        ///< 7: Labels
13    MetadataTyID,     ///< 8: Metadata
14    X86_MMXTyID,      ///< 9: MMX vectors (64 bits, X86 specific)
15    TokenTyID,        ///< 10: Tokens
16
17    // Derived types... see DerivedTypes.h file.
18    // Make sure FirstDerivedTyID stays up to date!
19    IntegerTyID,       ///< 11: Arbitrary bit width integers
20    FunctionTyID,      ///< 12: Functions
21    StructTyID,        ///< 13: Structures
22    ArrayTyID,         ///< 14: Arrays
23    PointerTyID,       ///< 15: Pointers
24    VectorTyID         ///< 16: SIMD 'packed' format, or other vector type
25 };

```

4.3 codegen 管理器定义

生成中间代码时需要有统一的上下文管理器和辅助工具, 因此首先需要定义一个 **CodeGenerator** 类:

```

1   class CodeGenerator
2   {
3     public:
4       // necessary data structure
5       LLVMContext ctx;
6       Module *module;
7       IRBuilder<> builder;
8       // auxiliary
9       Function *curFunction;           // current defined function
10      map<string, Function *> functions; // functions' def
11      list<map<string, Value *>> blocks; // local environments
12      map<string, Value *> globals; // global variable
13      list<pair<BasicBlock *, BasicBlock *>> jumpctx; // for jump statement
14      bool isglobal; // global definition
15      bool isleft; // left value
16      map<string, vector<string>> structtypes; // record members' name
17      map<string, AggregateType *> structvars; // record struct types' def
18      map<string, BasicBlock *> labels; // record labels' def
19
20     CodeGenerator();
21     ~CodeGenerator();
22
23     // for debug
24     void dump();
25     void error(string msg);
26     void warning(string msg);
27
28     // utils
29     Value *CreateCast(Value *V, Type *DestTy);
30     Value *GetVar(string name);
31     Function *GetFunction(string name);
32     Value *CreateUnaryExpr(Value *a, enum op_type op);
33     Value *CreateBinaryExpr(Value *a, Value *b, enum op_type op);
34     Constant *Num2Constant(Number *num);
35   };

```

首先是必需的成员变量：

- `LLVMContext ctx`：LLVM上下文，用于初始化 `Module` 和 `IRBuilder`，使二者协同工作
- `Module *module`：LLVM模块，用于存放和输出生成的IR
- `IRBuilder<> builder`：`IRBuilder` 类集成了很多LLVM常用类API的封装，例如获取类型、获取值、创建IR指令等等，codegen的大部分操作可以借助它完成，当然，也可以使用原本未封装的各个类API，都是十分便捷的。

然后是在codegen过程中需要记录的一些上下文信息：

- `Function *curFunction`：当前指令插入点所在的函数
- `map<string, Function *> functions`：该变量记录codegen过程中声明的函数，是函数的符号表
- `list<map<string, Value *>> blocks`：该变量记录块结构的内容和嵌套信息。

该列表初始时会有一个顶层块：

```
1   blocks.push_front(map<string, Value *>());
```

每当进入一个新的块（复合语句）时，就向列表**头部**加入新的块，退出时则删除列表头部的块。在codegen过程中声明的变量等都会保存在当前列表头部的块中，生命周期由块来决定。需要获取变量时，按顺序遍历列表中的块，即按从内到外的层次遍历，直到找到一个符合要求的变量

- `list<pair<BasicBlock *, BasicBlock *>> jumpctx`：该变量记录当前所在块的外层执行体的起始和结束，用于跳转语句确定跳转的目标基本块
- `bool isglobal`：用于标识当前环境为全局，当该标志置位时，声明的变量均为全局变量
- `bool isleft`：用于标识当前目标值为左值，当该标志置位时，对目标值的codegen会返回一个C语言概念下的左值的指针，方便创建IR的 `store` 指令
- `map<string, vector<string>> structtypes`、`map<string, AggregateType *> structvars`：存放结构体成员名称和定义的字典。在LLVM IR中，对结构体的访问和访问数组和指针是一致的，第一个成员的下标为0、第二个成员的下标为1，以此类推，因此在实现结构体访问的运算符时需要知道成员在定义中的位置
- `map<string, BasicBlock *> labels`：存放程序中的标签定义，方便goto语句直接获取目标 `BasicBlock`

在 `CodeGenerator` 的构造函数中，会完成

```
1 CodeGenerator::CodeGenerator() : builder(ctx)
2 {
3     module = new Module("main", ctx);
4     ...
5 }
```

此外，还会声明（declare）一些在程序中需要用到的C语言库函数，声明过程和使用LLVM IR API定义一个普通函数是一样的，区别只在于我们不需要给库函数定义（define）函数体。以声明printf函数为例：

```
1 vector<Type *> args;
2 args.push_back(Type::getInt8PtrTy(ctx));
3 FunctionType *functype = FunctionType::get(Type::getInt32Ty(ctx), args, true);
4 Function *func = Function::Create(functype, Function::ExternalLinkage, "printf",
5 module);
6 func->setCallingConv(CallingConv::C);
7 functions["printf"] = func;
```

1. 首先需要一个存放参数类型的vector，由于printf是变参函数，只有第一个参数格式化字符串是固定的，因此只需要使用 `Type` 的 `get` 方法获取一个 `i8 *` 的类型对象并push到vector中即可
2. 然后根据参数列表获取一个 `FunctionType` 类对象作为函数类型，同样使用 `get` 方法来构造即可，第一个参数指明函数返回值类型为 `i32`，第二个参数为参数列表，第三个参数表示该函数是否是一个可变参数的函数
3. 然后根据函数类型获取一个 `Function` 类对象，使用 `Create` 方法创建，第一个参数为函数类型，第二个参数为函数的链接属性，第三个参数是函数名，第四个参数指明在某个 `Module` 对象加入此函数声明
4. 使用 `setCallingConv` 方法将生成的函数的调用规范定义为C语言规范
5. 将生成的函数记录到上下文中

完成声明后编译器就可以识别源码中对这些库函数的调用，并且在最终生成目标可执行文件时链接到真正的库函数

还有自定义的一些辅助函数，具体实现见源码：

- `Value *CreateCast(Value *V, Type *DestTy)`：用于类型转换
- `Value *GetVar(string name)`：用于根据变量名获取变量值
- `Function *GetFunction(string name)`：用于根据函数名获取函数定义
- `Value *CreateUnaryExpr(Value *a, enum op_type op)`：用于生成单目表达式

- `Value *CreateBinaryExpr(Value *a, Value *b, enum op_type op)` : 用于生成双目表达式
- `Constant *Num2Constant(Number *num)` : 用于将我们自定义的 `Number` 类转化为 LLVM 的 `Constant` 类

对于表达式生成，这里举几个典型例子来简单描述一下：

- 假如一个加法表达式的两个操作数 `a` 和 `b` 都是整数类型，则使用如下调用生成IR:

```
1 builder.CreateBinOp(Instruction::BinaryOps::Add, a, b)
```

- 假如一个加法表达式的两个操作数 `a` 和 `b` 至少有一个是浮点类型，则首先将两者都cast成浮点类型，并使用如下调用生成IR:

```
1 builder.CreateBinOp(Instruction::BinaryOps::FAdd, a, b)
```

- 假如一个加法表达式的操作数 `a` 是指针类型，操作数 `b` 是一个整数，则使用 `CreateGep` 方法生成IR:

```
1 builder.CreateGEP(a, b)
```

GEP全称GetElementPtr，是LLVM IR的一条指令，用于指针计算，例如对于一个int类型的指针 `p`，要访问它指向的下一块内存，使用 `getelementptr i32, i32* %p, i32 1` 即可。该指令的泛用性很高，也用于实现结构体成员的访问

- 假如一个减法表达式的两个操作数 `a` 和 `b` 都是指针类型，则使用 `CreatePtrDiff` 方法生成IR:

```
1 builder.CreatePtrDiff(a, b)
```

- 假如一个小于表达式的两个操作数 `a` 和 `b` 都是指针类型，则需要使用 `CreatePtrToInt` 将其指向的地址转为整型值，用如下调用生成IR:

```
1 builder.CreateICmp(
2     CmpInst::ICMP_ULT,
3     builder.CreatePtrToInt(a, Type::getInt64Ty(ctx)),
4     builder.CreatePtrToInt(b, Type::getInt64Ty(ctx))
5 )
```

在3.2.1节中提到，每个AST节点都有自己的 `CodeGen` 函数，这一节会分模块描述各自的实现。

4.4.1 type.cpp

类型类没有对应的中间代码，不需要 `CodeGen` 函数，但是需要一个 `getType` 方法，将我们自己定义的类型转换成等价的 LLVM Type

4.4.1.1 基本类型

以int类型为例，直接使用 `builder` 的API返回一个 `Int32` 即可：

```
1 Type *IntType::getType(CodeGen &ctx)
2 {
3     return ctx.builder.getInt32Ty();
4 }
```

4.4.1.2 数组与指针

使用LLVM中对应的类型 `ArrayType` 和 `PointerType` 的 `get` 方法获取对应LLVM Type，数组类型需要指定大小

```

1 Type *MyArrayType::getType(CodeGen &ctx)
2 {
3     return ArrayType::get(basictype->getType(ctx), size);
4 }
5
6 Type *MyPointerType::getType(CodeGen &ctx)
7 {
8     return PointerType::get(basictype->getType(ctx), 0);
9 }

```

4.4.1.3 结构与联合

聚合类型是用户定义的，其定义会在codegen过程中存入 module，可以借助 module 的 `getTypeByName` 方法，直接使用类型名获取对应LLVM Type：

```

1 Type *MyStructType::getType(CodeGen &ctx)
2 {
3     return ctx.module->getTypeByName(name);
4 }

```

4.4.2 expression.cpp

4.4.2.1 数值常量

使用定义好的 `Number` 转 `Constant` 函数即可：

```

1 Value *Number::CodeGen(CodeGen &ctx)
2 {
3     return ctx.Num2Constant(this);
4 }

```

4.4.2.2 字符串常量

字符串常量在一般程序内存布局中位于全局只读区域，因此在我的实现中也记录为一个全局变量。在 codegen 时，首先查找 `Module` 对象中是否存有对应的全局变量，如果没有就创建一个对应的 `GlobalVariable` 对象。虽然定义时的类型是一个 `Array` 类型，但在 LLVM IR 体系中，一个数组变量并不等价于其首元素的地址，为了让数组和指针统一，所以在返回的时候会用 `CreateGEP` 将数组转为其首元素的指针

```

1 Value *String::CodeGen(CodeGen &ctx)
2 {
3     Value *var = ctx.module->getGlobalVariable(val);
4     if (!var)
5     {
6         Constant *str = ConstantdataArray::getString(ctx.ctx, val);
7         var = new GlobalVariable(*ctx.module, str->getType(), true,
8             GlobalValue::PrivateLinkage, str, val);
8     }
9     // return a string variable as char * but not char array
10    return ctx.builder.CreateGEP(var, {ctx.builder.getInt32(0),
11        ctx.builder.getInt32(0)} );
11 }

```

4.4.2.3 标识符

标识符的codegen和当前上下文要求的值类型有关。

在LLVM IR中，声明一个本地变量相当于使用 `alloca` 指令为变量分配一块内存空间，所以在保存时实际上存的是这块空间的地址。假如当前上下文的 `isleft` 置位，要求返回一个左值，就可以直接将获取到的地址返回；如果要求返回右值，则需要用 `CreateLoad` 方法取出地址上存放给的值；如果变量是数组类型的，那么和前面提到的字符串常量一样，需要转化成首元素地址返回

```
1 Value *Identifier::CodeGen(CodeGenerator &ctx)
2 {
3     Value *var = ctx.GetVar(name);
4     if (!var)
5         ctx.error(string("variable ')") + name + string("' not found"));
6
7     if (ctx.isleft)
8         // return as left value
9     return var;
10    else
11    {
12        // return as right value
13        if (var->GetType()->getPointerElementType()->isArrayTy())
14            // if the variable is ArrayType, return as a pointer type (pointer
15            // of first element)
16            return ctx.builder.CreateGEP(var, {ctx.builder.getInt32(0),
17                                              ctx.builder.getInt32(0)});
18        return ctx.builder.CreateLoad(var);
19    }
20 }
```

4.4.2.4 函数调用

对于一个函数调用，获取到对应 `Function` 对象 `func` 并构造好一个 `vector<Value *>` 类型的参数列表 `args` 后即可使用 `CreateCall(func, args)` 创建一个 `call` 指令，具体实现就不赘述了。

4.4.2.5 通用表达式

大多数运算符都可以通过前面提到过的API实现，这里再介绍两个API：

- 当对一个数组类型变量进行解引用时，可以使用 `CreateExtractValue(array, idx)` 创建对应指令
- 三目运算符（选择运算）在LLVM IR有完全一致的表达，即 `select` 指令，可以使用 `CreateSelect(condition, choice1, choice2)` 创建，需要注意两个选项表达式的类型要完全一致

4.4.3 statement.cpp

语句没有返回值，所以所有语句类型的codegen均返回 `nullptr`

4.4.3.1 复合语句

复合语句的codegen就是对其中的声明和子语句依次codegen

需要注意的是，当开始codegen时，需要添加一层新的block；当退出codegen时，需要将当前block清除：

```

1 Value *CompoundStatement::CodeGen(CodeGenerator &ctx)
2 {
3     // new level block
4     ctx.blocks.push_front(map<string, Value *>());
5     ...
6     // leave the current block
7     ctx.blocks.pop_front();
8
9     return nullptr;
10 }

```

4.4.3.2 表达式语句

表达式语句的codegen就是对其中的表达式进行codegen

4.4.3.3 选择语句

以if-else语句的codegen为例说明为语句创建基本块和指令的思路：

首先，需要为语句结构的每一个部分创建基本块，if-else语句有条件判断 `if.cond`、if子句 `if.then`、else子句 `if.else`，还有语句的出口 `if.out`。然后在当前的指令插入点（上一个语句的出口）处创建一个无条件跳转 `br` 指令，跳到条件判断这个块中。

```

1 Value *IfElseStatement::CodeGen(CodeGenerator &ctx)
2 {
3     BasicBlock *ifcond = BasicBlock::Create(ctx.ctx, "if.cond",
4     ctx.curFunction);
5     BasicBlock *ifthen = BasicBlock::Create(ctx.ctx, "if.then",
6     ctx.curFunction);
7     BasicBlock *ifelse = BasicBlock::Create(ctx.ctx, "if.else",
8     ctx.curFunction);
9     BasicBlock *ifout = BasicBlock::Create(ctx.ctx, "if.out", ctx.curFunction);
10    ctx.builder.CreateBr(ifcond);
11
12    ...
13
14    return nullptr;
15 }

```

接下来为每个基本块创建指令，首先需要用 `SetInsertPoint` 将指令插入点设置到当前基本块：

在条件判断中，对条件表达式进行codegen后，可以根据返回值进行条件跳转：

`CreateCondBr(cmpres, ifthen, ifelse);`

```

1     // branch condition
2     ctx.builder.SetInsertPoint(ifcond);
3     Value *cmpres = cond->CodeGen(ctx);
4     if (cmpres->GetType()->isPointerTy() || cmpres->GetType()-
5         >getIntegerBitWidth() != 1)
6         // logical comparision is omitted
7         cmpres = ctx.CreateBinaryExpr(
8             cmpres,
9             cmpres->GetType()->isPointerTy() ? dyn_cast<Value>
10                (ConstantPointerNull::get(dyn_cast<PointerType>(cmpres->GetType()))):
11                ConstantInt::get(cmpres->GetType(), 0),
12                OP_NEQ);
13     ctx.builder.CreateCondBr(cmpres, ifthen, ifelse);

```

在对两个不同子句进行codegen后，都需要无条件跳转到语句出口，但是考虑到其中可能存在return语句，即此时的基本块已经包含一个终端指令 `ret`，没有必要再跳转到 `if.out` 了，所以创建无条件跳转之前会先判断当前基本块的 `getTerminator` 方法返回值是否为空

```
1 // if-statement
2 ctx.builder.SetInsertPoint(ifthen);
3 stmt->CodeGen(ctx);
4 // if current block has not terminated, go if.out
5 if (!ctx.builder.GetInsertBlock()->getTerminator())
6     ctx.builder.CreateBr(ifout);
7
8 // else-part
9 ctx.builder.SetInsertPoint(ifelse);
10 if (stmt->next)
11     stmt->next->CodeGen(ctx);
12 if (!ctx.builder.GetInsertBlock()->getTerminator())
13     ctx.builder.CreateBr(ifout);
```

最后，将指令插入点转到语句出口 `if.out` 处，由下一条语句继续向下进行codegen：

```
1 // go out
2 ctx.builder.SetInsertPoint(ifout);
```

对于switch语句，有专用的一些API需要介绍

switch指令由 `CreateSwitch` 方法创建，两个参数分别是switch的条件和default分支。不过由于codegen时是按顺序解析，所以创建时先将default分支设置为空：

```
1 Value *condexp = cond->CodeGen(ctx);
2 SwitchInst *swinst = ctx.builder.CreateSwitch(condexp, nullptr);
```

接下来对switch中的每个子语句进行codegen，如果遇到case语句则需要创建新的基本块。

如果 `CaseStatement` 的 `val` 成员非空，表示这是一个case，使用 `SwitchInst` 的 `addCase` 方法将 case的表达式值加入switch指令的选择中（此表达式的类型要和switch条件的类型完全一致，所以这里需要一个隐式类型转换）；如果为空，表示这是default分支，使用 `setDefaultDest` 方法为switch指令设置default分支

还需要注意的是，有些时候我们使用switch语句时，会特意让多个case连在一起执行相同的逻辑，而不是每个case都使用break语句中断，此时case基本块就缺少了终端指令，所以在指令插入点转到下一个基本块前，要为当前块加上一个跳转到下一个块的 `br` 指令

```
1 if (((CaseStatement *)s)->val)
2 {
3     BasicBlock *switchcase = BasicBlock::Create(ctx.ctx, "switch.case",
4         ctx.curFunction);
5     swinst->addCase(
6         dyn_cast<ConstantInt>(
7             ctx.CreateCast(
8                 ctx.Num2Constant(((CaseStatement *)p)->val), condexp-
9 >getType())),
10        switchcase);
11     if (!ctx.builder.GetInsertBlock()->getTerminator())
12         ctx.builder.CreateBr(switchcase);
13     ctx.builder.SetInsertPoint(switchcase);
14 }
```

```

14  {
15      BasicBlock *switchdefault = BasicBlock::Create(ctx.ctx, "switch.default",
16          ctx.curFunction);
17      swinst->setDefaultDest(switchdefault);
18      if(!ctx.builder.GetInsertBlock()->getTerminator())
19          ctx.builder.CreateBr(switchdefault);
20      ctx.builder.SetInsertPoint(switchdefault);
21  }

```

4.4.3.4 迭代语句

迭代语句的codegen思路和if-else完全一致，唯一的不同点是，由于break和continue语句的存在，在codegen开始时需要将迭代条件判断块 `cond` 和迭代语句出口 `out` 记录在上下文中：

```
1 ctx.loopctx.push_back(pair<BasicBlock *, BasicBlock *>(whilecond, whileout));
```

4.4.3.5 跳转语句

return语句可以通过 `CreateRet` 或 `CreateRetVoid` 来创建，如果函数有返回值，需要将待返回的值转为函数定义的返回类型：

```

1 Value *ReturnStatement::CodeGen(CodeGenerator &ctx)
2 {
3     if (res)
4         // correct the ret val type and do a type cast
5         ctx.builder.CreateRet(ctx.CreateCast(res->CodeGen(ctx), ctx.curFunction-
6             >getReturnType()));
6     else
7         ctx.builder.CreateRetVoid();
8
9     return nullptr;
10 }

```

break语句和continue语句直接根据当前基本块上下文进行跳转即可，break语句的跳转目标是上层语句出口，continue语句的跳转目标是下一轮的迭代条件判断：

```

1 Value *BreakStatement::CodeGen(CodeGenerator &ctx)
2 {
3     ctx.builder.CreateBr(ctx.loopctx.back().second);
4
5     return nullptr;
6 }
7
8 Value *ContinueStatement::CodeGen(CodeGenerator &ctx)
9 {
10     ctx.builder.CreateBr(ctx.loopctx.back().first);
11
12     return nullptr;
13 }

```

goto语句的codegen根据目标标签名称 `label` 查找到对应基本块进行跳转即可：

```
1 ctx.builder.CreateBr(ctx.labels[label]);
```

4.4.3.6 标签语句

创建一个指定名称的基本块并记录：

```
1 BasicBlock *labelblock = BasicBlock::Create(ctx.ctx, label, ctx.curFunction);
2 ctx.labels[label] = labelblock;
```

4.4.4 declaration.cpp

4.4.4.1 变量声明

声明一个本地变量：使用 `CreateAlloca` 分配空间即可

```
1 ctx.builder.CreateAlloca(type, 0, varname);
```

如果该变量有初始化表达式，则对表达式做codegen后使用 `store` 指令将结果存入分配的空间即可

声明一个普通全局变量：使用 `GlobalVariable` 类

```
1 GlobalVariable *v = new GlobalVariable(
2     *ctx.module,
3     type,
4     var->qual && var->qual->isconst ? true : false,
5     GlobalValue::PrivateLinkage,
6     0,
7     varname
8 );
```

其构造函数的参数依次为：`Module` 类对象、变量类型、变量是否是常值、变量的链接属性、初始值、变量名

LLVM IR要求全局变量必须有初始值，可以用 `GlobalVariable::setInitializer` 来设置

4.4.4.2 函数声明

函数的定义和4.3节中提到的对于C库函数的声明的步骤是一样的，只是在声明后还需要创建基本块并对函数体进行codegen

4.4.4.3 类型声明

中间代码生成部分的类型声明不包括`typedef`语句，类型别名在生成AST时就已经替换为原来的类型了。这里主要是指结构体的声明：

```
1 MyStructType *mst = (MyStructType *)type;
2 // create struct type for assigned name or implement the member declaration
3 StructType *stype = ctx.module->getTypeByName(mst->name);
4 if (!stype)
5     stype = StructType::create(ctx.ctx, mst->name);
6 ctx.structtypes[mst->name] = vector<string>();
7
8 if (mst->members)
9 {
10     // set members
11     vector<Type *> elements;
12     for (auto &p : *mst->members)
13     {
14         for (auto &q : *p->first)
15             elements.push_back(q->getType(ctx));
16         // record field name in context
17         for (auto &q : *p->second)
18             ctx.structtypes[mst->name].push_back(q->name);
19     }
20     stype->setBody(elements);
```

```
21     }
22 // if no member, just a forward declaration, set as opaque type (default)
```

由于结构体存在没有成员定义的前向声明用法，因此声明时首先会使用 `Module` 对象的 `getTypeByName` 方法查找当前上下文中是否存在对应的声明，如果没有就使用 `StructType::create` 创建一个。然后获取一个成员类型vector，通过 `setBody` 方法为结构体设置成员定义，注意LLVM IR的结构体成员是没有名字的，所以在codegen时要手动收集这个信息，在访问时才可以根据成员名找到对应成员所处的位置

4.4.5 program.cpp

这是整个代码生成的顶层部分，遍历程序中的声明链依次做codegen即可。

在顶层的变量声明即全局变量，需要将 `isglobal` 标志置位；对于函数声明，做完codegen之后需要调用 `verifyFunction` 函数检查一下是否符合LLVM IR的语法要求：

```
1 Value *Program::CodeGen(CodeGen &ctx)
2 {
3     for (Declaration *p = decl; p; p = p->next)
4     {
5         if (p->getName() == "\"VariableDeclaration\"")
6         {
7             ctx.isglobal = true;
8             p->CodeGen(ctx);
9             ctx.isglobal = false;
10        }
11        else if (p->getName() == "\"FunctionDeclaration\"")
12        {
13            Value *ret = p->CodeGen(ctx);
14            verifyFunction(*(Function *)ret, &errs());
15        }
16        else
17            p->CodeGen(ctx);
18    }
19
20    return nullptr;
21 }
```

五、静态语义检查

在我的实现中，静态语义检查是穿插在语法分析和中间代码生成两个部分中的，并没有作为单独一个模块。这一节主要讲述检查的一些规则。

5.1 定义检查

1. 类型别名是否重复定义：在解析到一条typedef语句时，首先在存储类型别名的字典 `typealias` 中搜索此别名，如果搜索到了，就认为是别名重定义，编译器将抛出错误（在语法分析时执行）
2. 类型别名使用时是否还未定义：词法分析器只会在类型别名已经定义的情况下将一个标识符返回为token `TYPENAME`，否则会返回token `IDENTIFIER`，如果程序意图将未定义的别名用作类型名，编译器会认为是语法错误（在语法分析时执行）
3. 函数、变量、标签是否重复定义或在使用时还未定义：在进行一个新的函数声明前，首先在记录中搜索该函数名，如果找到则说明是重复定义，抛出错误；执行函数调用时，同样检查是否存在该函数，如果没找到，抛出错误（在中间代码生成时执行）。对变量和标签的检查同理。

4. switch子句中是否有声明：在gcc的实现中，switch子句中的声明不会被执行，gcc会抛出一个warning，所以我也进行了同样的实现（在中间代码生成时执行）
5. 函数返回值类型是否与定义一致：如果不一致，编译器会进行隐式类型转换，并抛出一个warning告知用户（在中间代码生成时执行）

5.2 控制流语法检查

这部分主要是检查一些关键字是否搭配正确，如case、default关键字是否是在switch语句中使用、continue关键字是否在迭代语句中使用、break关键字是否在迭代语句或switch语句中使用（在中间代码生成时执行）

5.3 类型检查

1. 赋值运算、++、--的目标值是否是左值：编译器会判断目标值的类型，如果不是 Identifier，且不是指针解引用、数组下标访问、结构体成员访问中的一种表达式，则抛出错误（在语法分析时执行）
2. 下标访问和解引用的对象是否是数组或指针：如果该对象是 Number（在语法分析时执行）、或经过解析后不是数组或指针类型（在中间代码生成时执行），则抛出错误
3. 函数调用的对象是否是函数：如果该对象是 Number（在语法分析时执行）、或经过解析后没有在已定义函数列表中找到（在中间代码生成时执行），则抛出错误
4. 数组定义的大小是否为常量：由于常量表达式会进行折叠，因此直接检查下标是否是 Number 类型即可，如果不是则抛出错误（在语法分析时执行）
5. 算术等运算符操作数类型是否不符合要求：例如是否对非整型值进行位运算、模运算的模是非整型值等各种错误（在中间代码生成时执行）

六、宏

在验收要求指出的进阶主题中，除了上文提到过的结构体，我还实现了一点简单的宏功能。由于宏不属于C语言语句而是编译器预处理指令，因此单独作为一个模块来描述。

目前只实现了 #define 和 #include 两条指令，且：

- #define 只能够定义一般常量，无法定义带参数宏
- #include 可以进行文件内容的替换，但以 <> 包围的文件名不会进行解析（因为本实验实现的编译器暂不能到达解析C语言系统头文件中某些语法的水平），以 " " 包围的、符合本实验语法解析器语法定义的文件能够递归进行宏展开
- 其余的如条件编译选项 #ifdef、#ifndef、#else、#endif（因此也还不能解决循环包含的问题）以及 #pragma（特定于计算机或特定于操作系统）、## 字符串化等等均没有实现

实现原理就是在进行语法分析前先扫描一遍文件，收集宏定义信息，对于 #include，递归读取文件并写入展开后的输出；对于 #define，将定义好的常量收集起来，调用 yylex()，对识别到的宏定义标识符进行文本替换，并写入文件。

七、测试

7.1 实验要求测试点

为了简洁，这里不再对三个问题进行描述，直接贴出解决问题的代码、编译结果以及测试结果，同时，由于生成的IR比较长，所以也只展示一份，其他的会放在附件中。

7.1.1 快速排序

编译测试目标C源文件:

```
1 void quicksort(int A[], int len)
2 {
3     int i, j, pivot, temp;
4
5     if (len < 2)
6         return;
7
8     pivot = A[len / 2];
9     for (i = 0, j = len - 1; ; i++, j--)
10    {
11        while (A[i] < pivot)
12            i++;
13        while (A[j] > pivot)
14            j--;
15        if (i >= j)
16            break;
17        temp = A[i];
18        A[i] = A[j];
19        A[j] = temp;
20    }
21    quicksort(A, i);
22    quicksort(A + i, len - i);
23 }
24
25 int main()
26 {
27     int A[10010] = {0};
28     int n, i;
29
30     scanf("%d", &n);
31     for (i = 0; i < n; i++)
32         scanf("%d", &A[i]);
33
34     quicksort(A, n);
35
36     for (i = 0; i < n; i++)
37         printf("%d\n", A[i]);
38
39     return 0;
40 }
```

编译结果:

```
esifiel@ubuntu:~/compiler/src$ ./ecc -o obj/quicksort ..//test/quicksort.c
[+] parse done.
[+] visualization done.
[+] target code generated.
[+] compilation done.
esifiel@ubuntu:~/compiler/src$ ll obj
总用量 40
drwxrwxr-x 2 esifiel esifiel 4096 5月 18 13:29 ./
drwxrwxr-x 8 esifiel esifiel 4096 5月 18 13:28 ../
-rwxrwxr-x 1 esifiel esifiel 16392 5月 18 13:29 quicksort*
-rw-rw-r-- 1 esifiel esifiel 2472 5月 18 13:29 quicksort.bc
-rw-rw-r-- 1 esifiel esifiel 6919 5月 18 13:29 quicksort.ll
```

生成的LLVM IR:

```
1 ; ModuleID = 'main'
2 source_filename = "main"
3 target triple = "x86_64-pc-linux-gnu"
4
5 @stdin = external global i8*
6 @"%d" = private constant [3 x i8] c"%d\00"
7 @"%d\0A" = private constant [4 x i8] c"%d\0A\00"
8
9 declare i32 @printf(i8*, ...)
10
11 declare i32 @scanf(i8*, ...)
12
13 declare i32 @puts(i8*)
14
15 declare i32 @strlen(i8*)
16
17 declare i32 @strcmp(i8*, i8*)
18
19 declare i8* @strcpy(i8*, i8*)
20
21 declare i8* @calloc(i64, i64)
22
23 declare i8* @malloc(i64)
24
25 declare i64 @read(i32, i8*, i64)
26
27 declare i8* @fgets(i8*, i64, i8*)
28
29 declare i32 @atoi(i8*)
30
31 declare void @free(i8*)
32
33 declare i32 @exit(i32)
34
35 define void @quicksort(i32* %A, i32 %len) {
36 entry:
37     %A1 = alloca i32*
38     store i32* %A, i32** %A1
39     %len2 = alloca i32
40     store i32 %len, i32* %len2
41     %i = alloca i32
42     %j = alloca i32
43     %pivot = alloca i32
44     %temp = alloca i32
45     br label %if.cond
46
47 if.cond:                                ; preds = %entry
48     %0 = load i32, i32* %len2
49     %1 = zext i32 %0 to i64
50     %2 = icmp slt i64 %1, 2
51     br i1 %2, label %if.then, label %if.else
52
53 if.then:                                  ; preds = %if.cond
54     ret void
55
56 if.else:                                  ; preds = %if.cond
57     br label %if.out
58
```

```

59    if.out:                                ; preds = %if.else
60        %3 = load i32*, i32** %A1
61        %4 = load i32, i32* %len2
62        %5 = zext i32 %4 to i64
63        %6 = sdiv i64 %5, 2
64        %7 = getelementptr i32, i32* %3, i64 %6
65        %8 = load i32, i32* %7
66        store i32 %8, i32* %pivot
67        br label %for.init
68
69    for.init:                                ; preds = %if.out
70        store i32 0, i32* %i
71        %9 = load i32, i32* %len2
72        %10 = zext i32 %9 to i64
73        %11 = sub i64 %10, 1
74        %12 = trunc i64 %11 to i32
75        store i32 %12, i32* %j
76        br label %for.cond
77
78    for.cond:                                ; preds = %for.end, %for.init
79        br label %for.loop
80
81    for.loop:                                ; preds = %for.cond
82        br label %while.cond
83
84    for.end:                                ; preds = %if.out9
85        %13 = load i32, i32* %i
86        %14 = add i32 %13, 1
87        store i32 %14, i32* %i
88        %15 = load i32, i32* %j
89        %16 = sub i32 %15, 1
90        store i32 %16, i32* %j
91        br label %for.cond
92
93    for.out:                                ; preds = %if.then7
94        %17 = load i32*, i32** %A1
95        %18 = load i32, i32* %i
96        call void @quicksort(i32* %17, i32 %18)
97        %19 = load i32*, i32** %A1
98        %20 = load i32, i32* %i
99        %21 = getelementptr i32, i32* %19, i32 %20
100       %22 = load i32, i32* %len2
101       %23 = load i32, i32* %i
102       %24 = sub i32 %22, %23
103       call void @quicksort(i32* %21, i32 %24)
104       ret void
105
106    while.cond:                            ; preds = %while.loop,
107        %25 = load i32, i32* %pivot
108        %26 = load i32*, i32** %A1
109        %27 = load i32, i32* %i
110        %28 = getelementptr i32, i32* %26, i32 %27
111        %29 = load i32, i32* %28
112        %30 = icmp slt i32 %29, %25
113        br i1 %30, label %while.loop, label %while.out
114
115    while.loop:                            ; preds = %while.cond

```

```

116    %31 = load i32, i32* %i
117    %32 = add i32 %31, 1
118    store i32 %32, i32* %i
119    br label %while.cond
120
121    while.out:                                ; preds = %while.cond
122    br label %while.cond3
123
124    while.cond3:                                ; preds = %while.loop4,
125    %while.out
126    %33 = load i32, i32* %pivot
127    %34 = load i32*, i32** %A1
128    %35 = load i32, i32* %j
129    %36 = getelementptr i32, i32* %34, i32 %35
130    %37 = load i32, i32* %36
131    %38 = icmp sgt i32 %37, %33
132    br i1 %38, label %while.loop4, label %while.out5
133
134    while.loop4:                                ; preds = %while.cond3
135    %39 = load i32, i32* %j
136    %40 = sub i32 %39, 1
137    store i32 %40, i32* %j
138    br label %while.cond3
139
140    while.out5:                                ; preds = %while.cond3
141    br label %if.cond6
142
143    if.cond6:                                  ; preds = %while.out5
144    %41 = load i32, i32* %j
145    %42 = load i32, i32* %i
146    %43 = icmp sge i32 %42, %41
147    br i1 %43, label %if.then7, label %if.else8
148
149    if.then7:                                  ; preds = %if.cond6
150    br label %for.out
151
152    if.else8:                                  ; preds = %if.cond6
153    br label %if.out9
154
155    if.out9:                                  ; preds = %if.else8
156    %44 = load i32*, i32** %A1
157    %45 = load i32, i32* %i
158    %46 = getelementptr i32, i32* %44, i32 %45
159    %47 = load i32, i32* %46
160    store i32 %47, i32* %temp
161    %48 = load i32*, i32** %A1
162    %49 = load i32, i32* %i
163    %50 = getelementptr i32, i32* %48, i32 %49
164    %51 = load i32*, i32** %A1
165    %52 = load i32, i32* %j
166    %53 = getelementptr i32, i32* %51, i32 %52
167    %54 = load i32, i32* %53
168    store i32 %54, i32* %50
169    %55 = load i32*, i32** %A1
170    %56 = load i32, i32* %j
171    %57 = getelementptr i32, i32* %55, i32 %56
172    %58 = load i32, i32* %temp
173    store i32 %58, i32* %57

```

```

173     br label %for.end
174 }
175
176 define i32 @main() {
177 entry:
178     %A = alloca [10010 x i32]
179     %n = alloca i32
180     %i = alloca i32
181     %0 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x
182     i8]* @"%d", i32 0, i32 0), i32* %n)
183     br label %for.init
184
185     for.init:                                ; preds = %entry
186         store i32 0, i32* %i
187         br label %for.cond
188
189     for.cond:                                ; preds = %for.end, %for.init
190         %1 = load i32, i32* %n
191         %2 = load i32, i32* %i
192         %3 = icmp slt i32 %2, %1
193         br i1 %3, label %for.loop, label %for.out
194
195     for.loop:                                ; preds = %for.cond
196         %4 = load i32, i32* %i
197         %5 = getelementptr [10010 x i32], [10010 x i32]* %A, i32 0, i32 %4
198         %6 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x
199         i8]* @"%d", i32 0, i32 0), i32* %5)
200         br label %for.end
201
202     for.end:                                ; preds = %for.loop
203         %7 = load i32, i32* %i
204         %8 = add i32 %7, 1
205         store i32 %8, i32* %i
206         br label %for.cond
207
208     for.out:                                ; preds = %for.cond
209         %9 = getelementptr [10010 x i32], [10010 x i32]* %A, i32 0, i32 0
210         %10 = load i32, i32* %n
211         call void @quicksort(i32* %9, i32 %10)
212         br label %for.init1
213
214     for.init1:                               ; preds = %for.out
215         store i32 0, i32* %i
216         br label %for.cond2
217
218     for.cond2:                               ; preds = %for.end4,
219     %for.init1
220         %11 = load i32, i32* %n
221         %12 = load i32, i32* %i
222         %13 = icmp slt i32 %12, %11
223         br i1 %13, label %for.loop3, label %for.out5
224
225     for.loop3:                               ; preds = %for.cond2
226         %14 = load i32, i32* %i
227         %15 = getelementptr [10010 x i32], [10010 x i32]* %A, i32 0, i32 %14
228         %16 = load i32, i32* %15
229         %17 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
230         i8]* @"%d\0A", i32 0, i32 0), i32 %16)

```

```

227     br label %for.end4
228
229     for.end4:                                ; preds = %for.loop3
230         %18 = load i32, i32* %i
231         %19 = add i32 %18, 1
232         store i32 %19, i32* %i
233         br label %for.cond2
234
235     for.out5:                                ; preds = %for.cond2
236     ret i32 0
237 }
```

运行测试程序，结果如下。单次运行通过所有测试点，运行100次并搜索fail的输出为空：

```

esifiel@ubuntu:~/compiler/src$ ..//test/tester/quicksort/linux-amd64 ./obj/quicksort
fixed case 0 (size 0)...pass!
fixed case 1 (size 1)...pass!
fixed case 2 (size 2)...pass!
fixed case 3 (size 2)...pass!
fixed case 4 (size 3)...pass!
fixed case 5 (size 3)...pass!
fixed case 6 (size 3)...pass!
fixed case 7 (size 3)...pass!
fixed case 8 (size 3)...pass!
fixed case 9 (size 4)...pass!
fixed case 10 (size 9)...pass!
fixed case 11 (size 9)...pass!
fixed case 12 (size 10000)...pass!
fixed case 13 (size 10000)...pass!
fixed case 14 (size 4096)...pass!
randomly generated case 0 (size 10000)...pass!
randomly generated case 1 (size 10000)...pass!
randomly generated case 2 (size 10000)...pass!
randomly generated case 3 (size 10000)...pass!
randomly generated case 4 (size 10000)...pass!
randomly generated case 5 (size 10000)...pass!
randomly generated case 6 (size 10000)...pass!
randomly generated case 7 (size 10000)...pass!
randomly generated case 8 (size 10000)...pass!
randomly generated case 9 (size 10000)...pass!
-----
2022-05-18 13:37:10.949
esifiel@ubuntu:~/compiler/src$ for i in `seq 1 100` ; do (..//test/tester/quicksort/linux-amd64 ./obj/quicksort | grep -i fail); done
esifiel@ubuntu:~/compiler/src$ |
```

7.1.2 矩阵乘法

编译目标：

```

1 int a[30][40];
2 int b[30][40];
3 int c[30][40];
4
5 int main()
6 {
7     int ma, na, mb, nb;
8     int i, j, k;
9
10    scanf("%d %d", &ma, &na);
11    for (i = 0; i < ma; i++)
12        for (j = 0; j < na; j++)
13            scanf("%d", &a[i][j]);
14
15    scanf("%d %d", &mb, &nb);
```

```

16     for (i = 0; i < mb; i++)
17         for (j = 0; j < nb; j++)
18             scanf("%d", &b[i][j]);
19
20     if (na != mb)
21     {
22         puts("Incompatible Dimensions");
23         exit(0);
24     }
25     else
26     for (i = 0; i < ma; i++)
27         for (j = 0; j < nb; j++)
28         {
29             c[i][j] = 0;
30             for (k = 0; k < na; k++)
31                 c[i][j] += a[i][k] * b[k][j];
32         }
33
34     for (i = 0; i < ma; i++)
35     {
36         for (j = 0; j < nb; j++)
37             printf("%10d", c[i][j]);
38         puts("");
39     }
40
41     return 0;
42 }
```

编译后运行测试程序的结果：

```

esifiel@ubuntu:~/compiler/src$ ./test/tester/matrix-multiplication/linux-amd64 ./obj/matrix-multiplication
fixed case 0 (size [1x1]x[1x1])...pass!
fixed case 1 (size [1x1]x[2x1])...pass!
fixed case 2 (size [1x4]x[4x1])...pass!
fixed case 3 (size [4x1]x[1x4])...pass!
fixed case 4 (size [1x25]x[25x1])...pass!
randomly generated case 0 (size [20x20]x[20x20])...pass!
randomly generated case 1 (size [20x20]x[20x20])...pass!
randomly generated case 2 (size [20x20]x[20x20])...pass!
randomly generated case 3 (size [20x20]x[20x20])...pass!
randomly generated case 4 (size [20x20]x[20x20])...pass!
randomly generated case 5 (size [20x20]x[20x20])...pass!
randomly generated case 6 (size [20x20]x[20x20])...pass!
randomly generated case 7 (size [20x20]x[20x20])...pass!
randomly generated case 8 (size [20x20]x[20x20])...pass!
randomly generated case 9 (size [20x20]x[20x20])...pass!
-----
2022-05-18 13:44:03.027
esifiel@ubuntu:~/compiler/src$ for i in `seq 1 100`; do (./test/tester/matrix-multiplication/linux-amd64 ./obj/matrix-multiplication | grep -i fail); done
esifiel@ubuntu:~/compiler/src$
```

7.1.3 选课助手

编译目标：

```

1  int split(char *src, char delim, char *vec[])
2  {
3      int cnt = 0;
4      int len = strlen(src);
5      int p = 0, q = 0;
6
7      while (p < len && q < len)
8      {
9          while (src[q] != delim && q < len)
10              q++;
11          src[q] = 0;
```

```

12         vec[cnt++] = &src[p];
13         p = q + 1;
14     }
15
16     return cnt;
17 }
18
19 const int MAXLEN = 1000;
20 const int MAXN = 100;
21
22 char name[MAXN][10];
23 int credit[MAXN];
24 struct dep
25 {
26     char name[10];
27     struct dep *another;
28     struct dep *next;
29 } * dependency[MAXN];
30 char grade[MAXN];
31
32 void extract(char pyfa[][MAXLEN], int n)
33 {
34     char *vec[MAXN] = {0};
35     int i;
36
37     for (i = 0; i < n; i++)
38     {
39         int cnt = split(pyfa[i], '|', vec);
40         // course name
41         strcpy(name[i], vec[0]);
42         // course credit
43         credit[i] = atoi(vec[1]);
44         // dependency
45         if (vec[2][0])
46         {
47             // for ; part
48             char *subvec1[MAXLEN] = {0};
49             // for , part
50             char *subvec2[MAXLEN] = {0};
51
52             struct dep *p, *tmp;
53             int groups, items, ii, jj;
54
55             // dummy head
56             dependency[i] = (struct dep *)calloc(sizeof(struct dep), 1);
57             p = dependency[i];
58
59             groups = split(vec[2], ';', subvec1);
60             for (ii = 0; ii < groups; ii++)
61             {
62                 p->another = (struct dep *)calloc(sizeof(struct dep), 1);
63                 tmp = p->another;
64
65                 items = split(subvec1[ii], ',', subvec2);
66                 for (jj = 0; jj < items; jj++)
67                 {
68                     tmp->next = (struct dep *)calloc(sizeof(struct dep), 1);
69                     strcpy(tmp->next->name, subvec2[jj]);

```

```

70             tmp = tmp->next;
71         }
72
73         p = p->another;
74     }
75 }
76 // course grade
77 if (vec[3][0] && vec[3][0] != '\n')
78     grade[i] = vec[3][0];
79 }
80 }
81
82 int main()
83 {
84     double gpa = 0;
85     int total = 0, try = 0, taken = 0, remaining = 0;
86     char pyfa[MAXN][MAXLEN];
87     int cnt;
88     int i, j;
89
90     // get input
91     for (cnt = 0; fgets(pyfa[cnt], MAXLEN - 1, stdin), strcmp(pyfa[cnt], "\n");
92         cnt++)
93         ;
94
95     // get info
96     extract(pyfa, cnt);
97
98     for (i = 0; i < cnt; i++)
99     {
100         total += credit[i];
101         if (grade[i] && grade[i] != 'F')
102         {
103             taken += credit[i];
104             switch (grade[i])
105             {
106                 case 'A':
107                     gpa += credit[i] * 4;
108                     break;
109                 case 'B':
110                     gpa += credit[i] * 3;
111                     break;
112                 case 'C':
113                     gpa += credit[i] * 2;
114                     break;
115                 case 'D':
116                     gpa += credit[i] * 1;
117                     break;
118                 default:
119                     break;
120             }
121         }
122         else if (grade[i] == 'F')
123             try += credit[i];
124     }
125     try += taken;
126     remaining = total - taken;
127     if (try != 0)

```

```

127         gpa /= try;
128     else
129         gpa = 0;
130     printf("GPA: %.1f\n", gpa);
131     printf("Hours Attempted: %u\n", try);
132     printf("Hours Completed: %u\n", taken);
133     printf("Credits Remaining: %u\n\n", remaining);
134     puts("Possible Courses to Take Next");
135     if (remaining == 0)
136         puts(" None - Congratulations!");
137     else
138     {
139         // find recommend
140         struct dep *p, *prev, *cur, *tmp;
141         // for each course
142         for (i = 0; i < cnt; i++)
143             // if the credits got
144             if (grade[i] && grade[i] != 'F')
145                 // for other courses
146                 for (j = 0; j < cnt; j++)
147                     // delete the course from their dependencies
148                     if (dependency[j])
149                         for (p = dependency[j]->another; p; p = p->another)
150                             for (prev = p, cur = p->next; cur;)
151                                 if (!strcmp(cur->name, name[i]))
152                                     {
153                                         prev->next = cur->next;
154                                         tmp = cur;
155                                         cur = cur->next;
156                                         free(tmp);
157                                         break;
158                                     }
159                                 else
160                                     {
161                                         prev = cur;
162                                         cur = cur->next;
163                                     }
164
165         for (i = 0; i < cnt; i++)
166             if (!grade[i] || grade[i] == 'F')
167                 if (!dependency[i])
168                     printf(" %s\n", name[i]);
169                 else
170                     for (p = dependency[i]->another; p; p = p->another)
171                         if (!p->next)
172                         {
173                             printf(" %s\n", name[i]);
174                             break;
175                         }
176     }
177
178     return 0;
179 }
```

由于我并不了解算法相关的知识，上述代码可能写得非常繁琐。找到推荐修读课程的主要思路就是根据输入提取信息，将课程依赖关系存为链表，再对所有已修读的课程，从其他课程的依赖中删去他们，最后检查依赖是否为空。

同时，此测试点代码由于AST节点过多，在我的Ubuntu虚拟机上不能完整可视化，可能会导致机器卡顿或重启。对此代码的AST json文件运行可视化脚本请谨慎

编译后运行测试程序结果：

```
esifiel@ubuntu:~/compiler/src$ ./test/tester/auto-advisor/linux-amd64 ./obj/auto-advisor
fixed case 0...pass!
fixed case 1...pass!
fixed case 2...pass!
fixed case 3...pass!
randomly generated case 0...pass!
randomly generated case 1...pass!
randomly generated case 2...pass!
randomly generated case 3...pass!
randomly generated case 4...pass!
randomly generated case 5...pass!
randomly generated case 6...pass!
randomly generated case 7...pass!
randomly generated case 8...pass!
randomly generated case 9...pass!
-----
2022-05-18 14:06:56.551
esifiel@ubuntu:~/compiler/src$ for i in `seq 1 100`; do (./test/tester/auto-advisor/linux-amd64 ./obj/auto-advisor
| grep -i fail); done
esifiel@ubuntu:~/compiler/src$ |
```

八、总结

在本实验中我设计并完成了一个C语言子集的编译器，支持C语言中大部分常用的语法，目前已经实现的内容包括但不限于：

- C语言类型系统，包括基本类型、数组、指针、struct
- 多维数组、多级指针、指针数组声明
- const、extern、static修饰符
- C语言所有运算符，支持指针、数组的比较与加减运算
- C语言所有语句
- 结构体的定义与使用，包括结构体和数组、指针的结合使用
- 函数声明
- typedef关键字
- 简单的宏定义

暂未实现的语法包括但不限于：

- u、L、e、f等常量修饰符
- 指向const的指针、函数指针等复杂指针声明
- volatile、register修饰符
- struct的位域语法
- 复杂的宏语法

在代码优化方面，由于时间关系，暂时只实现了AST层面的常量折叠，后续打算学习LLVM Pass，通过它来进行其他优化（虽然clang编译IR时也会进行优化，但是还是希望这个工作可以由我自己完成一部分）