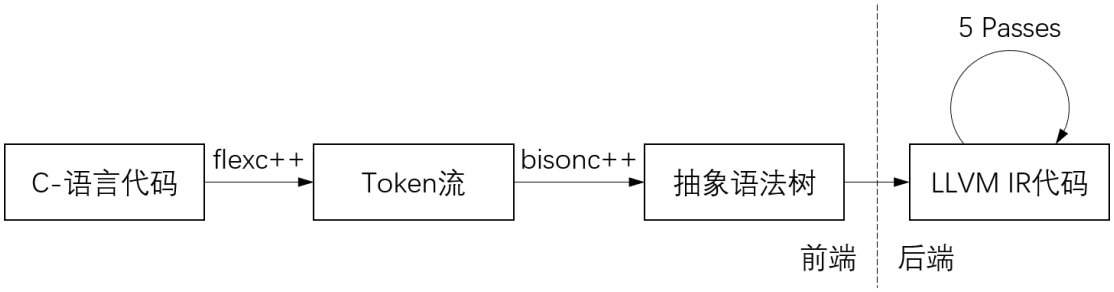
编译原理课程设计报告

161810129 董世晨　161810130 崔明暄　161810104 翁静

一、 总体设计

本课程设计实现了一个类似于C/C++语言的编译器。前端使用flexc++进行词法分析，bisonc++进行语法分析，最后使用自建的AST生成LLVM IR代码；后端实现了5个基于LLVM的Pass对IR代码进行优化。系统设计可以用下图概括：



二、 前端：从 C-语言代码到 IR 中间代码

1. 前言

本课程设计的前端由董世晨（本章中称“我”）独立完成，未参考任何除官方文档之外的代码，项目使用Git进行版本控制，已在GitHub上开源，并愿意长期维护，链接如下：

https://github.com/ClubieDong/C-Compiler

为了避免重复，我省略了对于flexc++、bisonc++、LLVM的介绍，相关文档参见：

flexc++：https://fbb-git.gitlab.io/flexcpp/manual/flexc++.html

bisonc++：https://fbb-git.gitlab.io/bisoncpp/manual/bisonc++.html

LLVM：https://llvm.org/docs/

C++：https://en.cppreference.com/w/

2. 概览

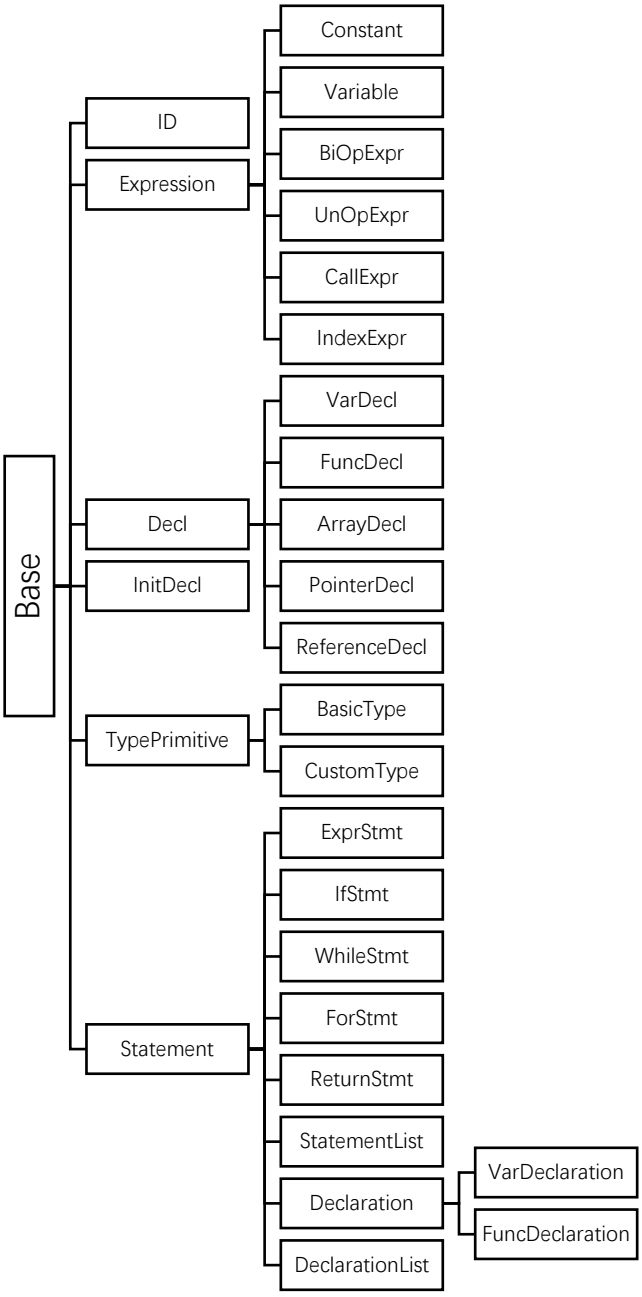
我设计的前端实现了所有课程设计要求的语法和语言特性，除此之外：

1) 支持bool、char、short、int、long、float、double类型
2) 支持true、false表示的bool常量，十进制、二进制、八进制、十六进制表示的int常量，小数、科学计数法表示的double常量，单引号字符表示的char常量
3) 数组大小常量合并（如int a[5*2];）
4) 允许在任意位置定义变量，而不局限于C-语言要求的代码块开始处
5) 允许一次性定义多个变量，并初始化（如int a, b=1, c=2;）
6) 允许全局变量初始化，初始化的代码在执行main函数之前执行
7) 支持“//”为首的单行注释
8) 实现了类似于C/C++的for循环
9) 支持所有C++基本运算符，包括加(+)、减(-)、乘(*)、除(/)、取余(%)、左移(<<)、右移(>>)、按位与(&)、按位或(|)、按位异或(^)、按位取反(~)、前缀递增(++)、前缀递减(--)、后缀递增(++)、后缀递减(--)、正号(+)、负号(-)、解引用(*)、取地址(&)、逻辑取反(!)、小于(<)、大于(>)、小于等于(<=)、大于等

于(>=)、等于(==)、不等于(!=)、赋值(=)，以及十个组合赋值运算符(+=、-=、*= /=、%=、<<=、>>=、&=、|=、^=)

10) 允许函数签名的参数列表为空，而不必显示指明void
11) 支持指针、引用、函数指针，以及支持任意复杂的合法的指针、引用、函数指针、数组的相互嵌套，引用用于与C++一致的语法和语义
12) 对于语法分析，可输出接受到的非法Token以及当前状态下可接受的所有合法Token；对于语义分析，可输出支持多达48种不同类型的错误/警告提示；精确到行和列的错误定位
13) 实现了不同整型、不同浮点型、不同指针、不同引用之间的自动转换和非法转换检测、不同类型数字之间进行运算支持自动类型提升（行为类似C++）
14) 支持链接任意外部函数，如int getchar()和int putchar(int ch)，基于这两个函数实现了测试所需的output函数和input函数

3. 抽象语法树的设计



上图展示了抽象语法树的继承关系，其中所有的类都在ast命名空间下。值得解释的有：BiOpExpr类表示二元运算表达式类，UnOpExpr表示一元运算表达式类，Decl类及其派生类表示对变量/函数类型的指针、数组、引用嵌套关系的描述。InitDecl类包含了一个Decl对象以及对所定义的变量的初始化表达式。CustomType类原本设计用于表示用户自定义类型（即class/struct），但由于时间关系，来不及实现，故此类没有被用到。

具体实现见https://github.com/ClubieDong/C-Compiler/tree/main/src/AST。

4. 符号表的设计

符号表在从AST生成IR代码阶段生成并查询。符号表是树状结构的，每个节点都是ast::SymbolTable类的一个对象表示，每个非根节点保存了其父节点的指针，每个节点拥有0个、1个或多个子节点，并持有其所有子节点的所有权。每个节点对应了C-代码中的一个局部代码块，记录了其中定义的变量的LLVM类型。

符号表使用std::map<std::string, llvm::Value *>类型来记录变量。在添加符号时保证了变量名不可重复；在搜索某个名称时，会尝试从当前节点开始递归地向上查找，直到找到根节点。这样实现了类似于C/C++的变量作用域和局部变量覆盖全局变量的语言特性。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/SymbolTable.hpp。

5. 七种不同的数字类型

我实现了bool、char、short、int、long五种不同的整数类型，分别为1位、8位、16位、32位和64位，除了bool类型，其余均为有符号整数。我还实现了32位的float和64位的double类型。另外还有一个特殊类型void，用于函数的返回值，或表示函数参数列表为空。这些类型在ast::TypePrimitive中定义，由于都有对应的LLVM类型，实现比较容易。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/TypePrimitive.hpp#L16。

6. 四种类型的常量

我实现了使用true或false表示bool类型，使用十进制、二进制、八进制、十六进制表示int类型，使用小数或科学计数法表示double类型，使用单引号字符表示char类型。这些常量的识别都可以在词法分析时完成。

二进制使用“0b”作为前导；八进制使用“0”作为前导，如0123；十六进制使用“0x”作为前导。这些前导和十六进制的A-F大小写不敏感。有一个值得注意的点是：如何让0123被识别为八进制，而不是十进制？根据文档，flex优先匹配更长的字符串，如果长度相等，则匹配靠前的规则。因此，我将识别八进制的规则写在识别十进制之前，就可以正确识别八进制。

对于浮点数类型的常量则比较复杂。我设计的正则表达式支持省略小数点前后其中一个，如“.123”“123.”等，以及科学计数法，如“.12E+5”“1.e-0”等。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Scanner/Scanner.l#L31。

7. 数组大小常量合并

定义数组时，数组长度不必严格为一个常量，而可以是对常量进行任意的运算组合，如“int a[('Z'-'A'+1)*2];”。这样方便了在使用C-语言时不必手动计算数组长度。

为了实现这一点，我在定义文法时允许数组长度为任意表达式，在语义分析阶段借助LLVM的IRBuilder进行自动常量合并，检测该表达式推导出的结果是否为一个整型常量。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Parser/Parser.y#L139和https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Decl.hpp#L151。

8. 类型转换

我实现了一个名为CastLLVMType的函数，可以尝试将变量从原类型转换到任意指定的类型。如果可以安全转换，则执行转换；如果触发了隐式转换，则输出警告信息；如果不能进行转换，则输出报错信息。具体转换规则如下：

1) 若目标类型为引用类型

   原类型必须和目标类型完全一致，且也是引用类型，除此之外均会输出报错信息。

2) 若目标类型和原类型都为指针类型

   若目标类型和原类型完全相同，则不必转换；否则进行转换，但是输出“隐式指针类型转换”的警告信息。

3) 若目标类型和原类型都为整数类型或都为浮点数类型

   若目标类型和原类型完全相同，则不必转换；否则进行转换，保证真值不变，但是输出“隐式整形/浮点型类型转换”的警告信息。

4) 若目标类型和原类型一个是整数类型，另一个是浮点数类型

   进行转换，尽量保证真值相近（四舍五入等），输出“隐式整形/浮点型类型转换”的警告信息。

除了以上这四种，其余转换都会被拒绝，并输出报错信息。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Expressions.hpp#L27。

9. 类型提升

LLVM IR代码的运算符对两侧的操作数类型有严格要求，必须类型相同才能进行运算。因此，我实现了一个名为CastToCommonType的函数，可以将二元运算符两侧的变量在尽量不损失精度和范围的情况下，转换到一个公共的类型，从而进行运算。在C++标准中，此操作叫做“类型提升”，我实现的类型提升和C++标准中的几乎一致。具体转换规则如下：

1) 若两操作数一个是整数类型，另一个是浮点数类型

   都转换为double类型。

2) 若两操作数都是整数类型，或都是浮点数类型

   将表示范围较小的那个转换为表示范围较大的那个的类型。

比如，int与float相加得到的结果为double类型，short和char相加得到的结果为short类型。上述的类型转换均使用CastLLVMType函数进行实现。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Expressions.hpp#L114。

10. 定义指针、引用、数组以及函数指针的文法设计

我实现了对任意对象定义指针、引用、数组，并且支持任意复杂的嵌套，如：

```
int (*a)[10];          // a 是一个指向 int[10]的指针
int *b[10];            // b 是一个包含了 10 个 int*的数组
int *&c;               // c 是一个 int*的引用
int *(*(&d)[5])[10];   // 很复杂，但合法
```

这一部分的文法设计、语义检查和IR代码生成我认为是整个课程设计种最具有挑战性、并最巧妙的部分。我观察到，对于每个变量的定义，分为三个部分：

1) 变量的名字

变量定义的最内层就是变量的名字，我使用ast::VarDecl和ast::FuncDecl两个类来表示，前者表示定义变量，后者表示定义函数或函数指针。

2) 变量的基本类型

写在最前面的那个类型就是变量的基本类型，它不包含任何修饰符，是上文提到的七种不同数字类型和void类型的其中一种。我使用ast::TypePrimitive类来表示。

3) 类型修饰

比如，上面的第二个例子，int *[100]是int *的数组修饰，长度为100，而int *又是int的指针修饰。类型修饰部分就是从基本类型开始，层层向外嵌套。我使用ast::ArrayDecl类表示数组修饰，ast::PointerDecl类表示指针修饰，ast::ReferenceDecl类表示引用修饰。这些以Decl结尾的类，都继承自ast::Decl类。

那么，文法的设计就可以从变量的名字开始，递归地向外包裹类型修饰。值得注意的是，由于后缀表达式的优先级是高于前缀表达式的（也就是数组修饰的优先级高于指针修饰），文法需要拆分成前缀修饰和后缀修饰两部分来明确指出这一优先级关系，另外也要支持括号来允许用户调整优先级。

对于函数指针的定义也是类似，只不过在名字后面需要接上参数列表部分。函数指针对应的非终结符需要和变量单独区分，因为这个非终结符需要被应用到函数的定义上。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Parser/Parser.y#L130和https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Decl.hpp。

11. 引用的实现

引用本质上就是指针的语法糖。在我设计的编译器内部，所有引用类型都被转换为相应的LLVM的指针类型进行记录，另外还需要记录一个名为IsRef的bool类型，用于表示LLVM的指针类型是真实的指针，还是引用的内部实现。所以，我设计了以下类，作为ast::Expression类及其所有派生类的CodeGen函数的返回结果（也就是表达式的计算结果）：

```
class SymbolTable::Symbol
{
public:
    llvm::Value *Value;
    bool IsRef;
    // 省略其余部分
};
```

比如，int&的内部表示为int*，IsRef为true；int*的内部表示为int*，IsRef为false；int的内部表示为int，IsRef为false。std::Variable类在符号表中进行名字搜索时，返回的结果都是引用类型的，它的直观解释是，当你定义了一个变量，之后使用这个变量实际上是在使用它的引用。

当然，大部分普通运算符接受的操作数不需要是引用类型，只需要普通的类型，这就需要一个隐式转换。比如，使int&和int&相加，就需要将两个操作数都转换为int类型。这其实是将int*中的内容读取出来。我设计了一个名为Dereference的函数进行引用类型的去除。它对于非引用类型，直接返回自身；对于引用类型，执行Load指令，将指针指向的内容读取出来。

这种语法糖巧妙的设计使得指针、引用、赋值、初始化相关的IR代码生成变得非常简洁，下文会对这几个部分进行详细阐述。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Expressions.hpp#L17和https://github.com/ClubieDong/C-Compiler/blob/main/src/SymbolTable.hpp#L41。

12. 变量和数组的实现及IR代码生成

由于LLVM使用了SSA形式（参见https://en.wikipedia.org/wiki/Static_single_assignment_form），变量的保存需要全部放到栈中，虽然这样子看上去效率较低，但是有mem2reg优化。

对于变量的定义，我在当前位置（尽管LLVM文档中将分配栈空间的指令放到了函数最前面的位置，但我没有发现任何这么做的必要）插入了分配栈空间的指令，并进行初始化操作，将获得的栈指针作为引用存入符号表。

对于数组的定义比较复杂，首先使用LLVM中的数组类型开辟一块相应大小的栈空间，再在栈中开辟一个指针，这个指针指向数组的起始位置，类型为数组元素类型。当时这样设计比较匆忙，没有仔细考虑是否有更为简单的方式，若有，请告知。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Decl.hpp#L248。

13. C++的所有37种基本运算符的文法设计

我实现了所有C++基本运算符，它们的优先级从高到低依次为：

1) 后缀递增(++)、后缀递减(--)
2) 前缀递增(++)、前缀递减(--)、正号(+)、负号(-)、解引用(*)、取地址(&)、按位取反(~)、逻辑取反(!)
3) 乘(*)、除(/)、取余(%)
4) 加(+)、减(-)
5) 左移(<<)、右移(>>)
6) 小于(<)、大于(>)、小于等于(<=)、大于等于(>=)
7) 等于(==)、不等于(!=)
8) 按位与(&)
9) 按位异或(^)
10) 按位或(|)
11) 赋值(=)、十个组合赋值运算符(+=、-=、*= /=、%=、<<=、>>=、&=、|=、^=)

其中，所有赋值运算符是右结合的，其余二元运算符都是左结合的。这个优先级的设计和C++保持了完全一致，具体可见https://en.cppreference.com/w/cpp/language/operator_precedence。

和课程设计指导书不同的是，我把赋值运算看作一个普通的运算符，只不过它的左操作数必须是引用类型，赋值运算的返回值就是左操作数的引用。这样和C++保持一致，充分利用了“引用”这个语言特性，优点是可以进行连等，如a=b=c=1。

写出满足这么多运算符优先级和结合性的文法本是一件令人痛苦的事情，但好在：

1) 对于二元运算符，bisonc++提供了定义二元运算符优先级和结合性的功能。在定义Token时，越先定义的Token优先级越低，越后定义的Token优先级越高，同一行中定义的Token优先级一致。对于优先级一致的Token，可以使用“%left”或“%right”代替“%token”，来实现左结合性和右结合性；
2) 对于一元运算符，通过仔细观察上文给出的C++运算符优先级，我发现所有的后缀一元运算符是优先级最高的，所有的前缀一元运算符是优先级次高的，高于所有二元运算符，仅次于后缀运算符。

基于以上两点，设计文法时只需要设计“PrimaryExpression”用于表示常量和括号表达式，“PostExpression”用于表示只有后缀运算符的表达式，“PreExpression”用于表示前缀和后缀运算符的表达式，以及“Expression”用于表示有所有（即前缀、后缀和二元）运算符的表达式。不需要像课程设计指导书中那样，对于每一级优先级，都设置一个非终结符。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Parser/Parser.y#L32。

14. 二元运算符IR代码生成

二元运算符基本可以分为以下四类：

1) 比较运算符

小于(<)、大于(>)、小于等于(<=)、大于等于(>=)、等于(==)、不等于(!=)

2) 数字运算符

加(+)、减(-)、乘(*)、除(/)、取余(%)

3) 位运算符

左移(<<)、右移(>>)、按位与(&)、按位或(|)、按位异或(^)

4) 赋值运算符

赋值(=)，以及十个组合赋值运算符(+=、-=、*= /=、%=、<<=、>>=、&=、|=、^=)

对于第一类的比较运算符，操作数可以是整型、浮点型和指针。若操作数为指针，则需要转换成相应位数的整型后进行比较；否则应用CastToCommonType转换到公共类型后进行比较。

对于第二类的数字运算符，除了取余运算符只支持整型，其余四个支持整型和浮点型。将操作数应用CastToCommonType转换到公共类型后进行运算。而对于加减运算符，允许左操作数为指针，右操作数为整数类型，此时需要使用LLVM提供的GetElementPointer来完成指针的偏移量计算。

对于第三类的位运算符，操作数必须为整型。移位运算符允许操作数位数不同，应用CastToCommonType转换到公共类型后进行运算；而按位逻辑运算符必须保证左右两操作数是位数相同的整型。

对于第四类中的赋值运算符(=)，它的左操作数必须为引用类型。首先使用CastLLVMType将右操作数转换为左操作数的非引用类型，然后生成Store指令进行赋值。

对于第四类中的组合赋值运算符，将它拆解为运算和赋值两步，如“a+=1”拆解为“a=a+1”，这里可以使用递归调用来简化逻辑并复用代码。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Expressions.hpp#L257。

15. 一元运算符IR代码生成

对于正负号运算符(+)(-)，允许操作数类型为整型和浮点型，正号运算符无需任何操作，负号运算符只需取相反数。

对于解引用运算符(*)，操作数必须为指针类型，运算结果为引用类型。得益于巧妙的引用的设计，解引用运算符无需任何指令（除了Dereference操作），只需将IsRef设为true即可。

对于取地址操作符(&)，操作数必须为引用，运算结果为非引用类型。再次得益于巧妙的引用的设计，取地址操作符也无需任何指令，只需将IsRef设为false即可。

对于前置/后置的递增/递减运算符(++)(--)，操作数必须为整型或指针类型的引用。

可以把此类运算符进行拆解，如++a可拆解为a+=1或a=a+1。值得注意的是，对于指针类型的递增或递减需要使用GetElementPointer指令来进行偏移量的计算；另外，前置递增/递减运算符返回的是原操作数的引用，而后置递增/递减运算符返回的中间临时变量的非引用类型。

对于按位取反运算符(~)，操作数必须为整型，只需将操作数与-1进行异或运算即可。

对于逻辑取反运算符(!)，操作数必须为整型。首先进行等价转换，!x等价于~(x!=0)，也等价于(x!=0)^-1，然后就可以用不等于比较指令和异或指令进行执行。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Expressions.hpp#L495。

16. 任意位置定义变量，且允许一次定义多个变量

和课程设计指导书不同，我把定义变量看作一条普通的语句，和表达式语句、If语句等一样，这样就可以在任意位置定义变量。

在一行中定义多个变量时，它们的基本类型相同，变量的名字、类型修饰和初始化表达式各自不同。语法结构可以直观理解为如下：

<基本类型> (<带修饰的变量名>['=' <初始化表达式>])+ ';'

另外一个神奇的应用是在一行中定义多个函数指针，它们的返回值相同，如：

```
int (*f)(int a), *g(bool b);
```

上行中首先定义了一个int f(int a)函数指针，然后定义了一个int *g(bool b)的函数。这种语法比较神奇，在C/C++中都有支持。但它的实现是很自然的，利用前文所提到的方法已经自然可以实现。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Parser/Parser.y#L130。

17. 初始化全局变量

全局变量的初始化需要在main函数之前被执行，在下面这个例子中，首先执行int f()，然后执行int main()，最终输出“ab”。

```
int f()
{
    putchar('a');
    return 1;
}
int a = f();
int main()
{
    putchar('b');
    return 0;
}
```

为了实现这样的效果，我在创建IR代码时，建立了一个内部的main函数，而将代码中的main函数生成IR代码时修改为__main__，在内部main函数中执行所有的初始化操作，退出前调用__main__函数。这样就可以做到在执行main函数之前初始化全局变量的效果。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Declarations.hpp#L153。

18. If语句、While语句、For语句和Return语句

生成If语句、While语句、For语句这些控制流语句的IR代码的基本思想是：先构建所需的基本块（如If语句需要构建Condition块、Then块、Else块、Merge块），然后在不同的基本块中递归地插入所需的代码，最后使用条件/无条件跳转将这些基本块连接起来即

可。

Return语句对应的IR代码就是Ret或RetVoid，值得注意的是，在同一代码块中，Return语句之后的代码不会被执行，因此需要创建一个没有前驱的基本块，将Return语句之后的代码放到该基本块中。这个基本块会在优化时被去掉。另一个值得注意的点是，需要在函数结尾的地方手动添加一个对应的Ret或RetVoid指令，避免用户没有写Return语句。因为在C/C++中，即使一个有返回值函数不是所有路径都有返回值，编译器不会报错，而只会报警告，我实现的编译器也有类似的行为。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Statements.hpp。

19. 精确到行和列的错误定位

我的警告/报错信息可以精确定位到一个Token的行号和列号范围。在词法分析阶段，返回每个Token时会顺便设置这个Token的行号和列号的范围。在语法分析阶段，AST的（几乎）每个与Token对应的节点都记录了该Token的位置。这样，在产生报错信息时，可以递归地向下寻找最近的Token所在的位置，从而给出精确的错误/警告定位。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/ErrorHandler.hpp#L9。

20. 语法分析阶段输出当前状态下可接受的所有合法Token

在语法分析阶段，可输出接受到的非法Token以及当前状态下可接受的所有合法Token。Bison提供了这样的接口，当定义%define parse.error verbose时，使用void yyerror (char const *)进行错误处理会提供一个用户友好的错误信息，包含了接受道的非法Token和当前状态下可接受的所有合法Token。可是我使用的bisonc++并没有实现这个功能，因此我阅读了bisonc++生成的parse.cc文件，自己实现了一个函数来实现这个功能。

令人沮丧的是，所有解析上下文无法语法所需的表格都被定义在了parse.cc文件的一个匿名命名空间中，无法从外部访问，而实现上述功能必须要访问这些表格，但parse.cc文件在每次编译都会被重新生成。为了解决这个问题，我把该函数写在Hack文件中，每次执行bisonc++生成parse.cc后，将Hack文件追加到parse.cc文件的末尾。

该函数的实现比较困难，主要思想是扫描bisonc++生成的s_state数组，里面包含了合法的状态转移。从中就可以获取出当前状态下合法的Token列表，进行去重、筛选等操作后就实现了所需的功能。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/Parser/Hack。

21. 支持链接到任意外部函数

由于我的编译器支持函数指针的定义，在生成函数指针对应的IR代码时，LLVM提供了一种名为ExternalLinkage的链接方式，可以将该函数链接到外部标准库。因此，在编写C-代码时，只需要写一个函数签名，省略函数体，就可以引用外部标准库中的函数了，用法如下所示：

```
int getchar();
int putchar(int ch);
```

只需以上两行代码，就可以进行实现对标准输入输出流的字符操作。再加上一些字符串和数字之间的转换代码，就能实现课程设计测试用例中所需的output和input函数了。

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/src/AST/Declarations.hpp#L110。

22. 课设的编译、运行、测试方式

本课程设计的编译、调试环境在Ubuntu20.04，需要的依赖项有：

1) LLVM 10.0.0
2) Flexc++ 2.07.07
3) Bisonc++ 6.03.00

可以使用以下指令安装：

```
sudo apt install llvm-dev
sudo apt install flexc++
sudo apt install bisonc++
```

使用Git克隆仓库到本地：

```
git clone https://github.com/ClubieDong/C-Compiler.git
```

在根目录下，build.sh脚本中包含了构建本课程设计的命令，该脚本会依次运行flexc++、bisonc++并追加Hack文件，最后使用clang++编译项目，可执行文件输出到build/exe。即，使用以下命令编译：

```
./build.sh
```

build/exe接受且仅接受一个命令行参数，表示输入的C-语言文件，成功编译后会生成.lex文件表示词法分析结果、.ast文件表示抽象语法树、.ir文件表示生成的IR中间代码。即，使用以下命令运行：

```
build/exe <C-语言文件>
```

或使用以下命令查看帮助：

```
build/exe -h
```

另外，我编写了test.sh脚本用于测试，该脚本会运行build/exe，并将输出的IR中间代码使用llvm-as转换为字节码，最后使用lli执行字节码。test.sh脚本接受一个参数，表示测试代码所在的文件夹。在tests/Basic文件夹中保存了课程设计指导书给出的11个基本测试用例和执行结果，如以下命令可以运行第1个测试用例：

```
./test.sh Basic/Test1
```

具体实现见https://github.com/ClubieDong/C-Compiler/blob/main/build.sh和https://github.com/ClubieDong/C-Compiler/blob/main/test.sh。

23. 感想

相比于其他学科的课设做一个XXX网站或者YYY管理系统，我（极其）更乐意实现一个编译器，一方面是因为它让我有一种作为技术型程序员的优越感，另一方面它是我从上学期的计算机组成原理设计CPU，到下学期的操作系统原理设计操作系统，自己构建一个计算机大厦之中的一环。再加上我对C++语言的热爱，让我有动力完成了2500行（不包含自动生成的代码）的课程设计和8000字的前端报告。

由于是从零开始，没有参考任何人的代码，本次课设大部分时间花在设计这一整套框架上。我本来还想实现类、成员函数、重载运算符等功能，但是由于时间关系，没来得及全部完成。即使如此，我设计的框架完全能够支持这些功能的实现。

最后我想赞美一下LLVM，它如此庞大的一个工程，却能设计地如此易用，到处可见软件设计的精妙（比如单纯比较llvm::Type*指针就可以判断两个类型是否相等、庞大的llvm::Value继承体系但是没有一个虚函数），完美体现了C++零成本抽象的精髓。

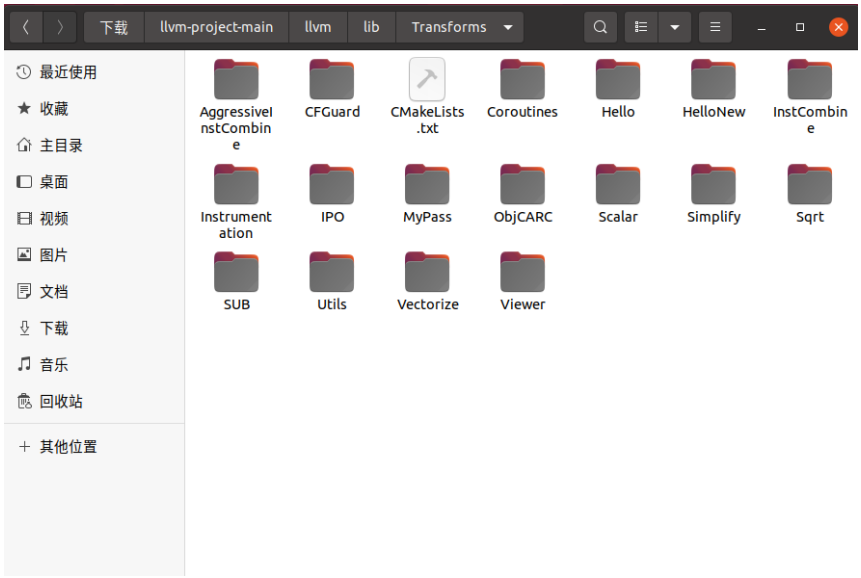
三、 后端：IR 中间代码的优化

后端由崔明暄（本章中称“我/我们”）实现。

1. Pass相关文件的组织结构

在LLVM中每个pass文件都是指遍历一次IR代码，并生成优化后的IR代码（或仅输出原

IR代码的特定信息）。LLVM的pass文件在llvm-project-main/llvm/lib/Transforms（实现文件）与llvm-project-main/llvm/include/llvm/Transforms（头文件）文件夹下（本文将以https://github.com/llvm/llvm-project项目的组织结构指定特定文件）。



当想添加一个Pass时，我们需要在llvm-project-main/llvm/lib/Transforms文件下创建相关的Pass文件夹，并在CMakeLists.txt文件内添加该文件夹。文件夹内部实现具体的Pass类，并创建CMakeLists.txt，将该Pass的cpp文件注册至llvm_library中。

所有的Pass都是llvm::Pass类的子类，在具体实现过程中，我们创建的类必须继承自ModulePass、CallGraphSCCPass、FunctionPass、LoopPass或RegionPass类。它们每一个都是llvm::Pass的子类并且有特定的功能，例如llvm::FunctionPass类中需要重写bool runOnFunction(llvm::Function &F) 函数，该Pass便会在每一个函数中调用该函数，以达到遍历每个函数的功能。

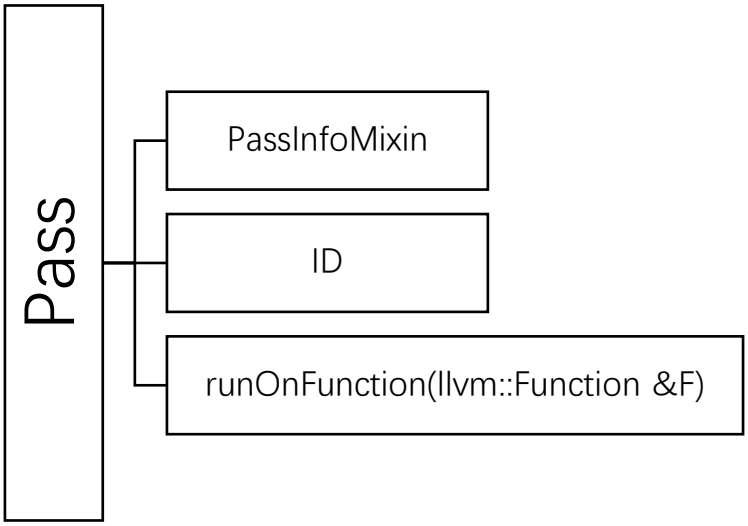
更多的Pass子类的功能介绍并不在本文的讨论范围中，具体请参照https://llvm.org/docs/WritingAnLLVMPass.html#writing-an-llvm-pass-looppass。本文主要涉及到llvm::FunctionPass类，涉及的操作也大多针对特定函数。

2. Pass文件分析

每个规范的Pass分为头文件和具体实现cpp文件两个部分，头文件中定义该Pass会涉及到的数据结构、操作、方法，而具体实现的cpp文件中完成优化、注册Pass等任务。下面将以实现llvm::FunctionPass的Pass为例（假设Pass名为Viewer），具体介绍其实现过程。

2.1. Pass 头文件

在规范的FunctionPass头文件中我们会定义继承自public llvm::PassInfoMixin<Viewer>的结构体（Viewer为结构体名）和继承自llvm::FunctionPass类的注册类（假设类名为LegacyViewer），具体的结构如下图所示。



PassInfoMixIn是LLVM提供的API，每一个Pass都必须采用该类实现接口。对于本文的例子来说，结构体Viewer为FunctionPass类提供接口的实现类（即LegacyViewer类中存在着Viewer类型的属性，假设其名为Imp）。Viewer定义如下：

```
struct Viewer : public llvm::PassInfoMixin<Viewer> {
  llvm::PreservedAnalyses run(llvm::Function &F,
                              llvm::FunctionAnalysisManager &);
  bool runOnFunction(llvm::Function &F);
};
```

此外，LegacyViewer也必须具有一个static char类型的ID，PassManager利用该ID寻找到特定的Pass，该ID的值并不需要我们去特别指定。

最后由于LegacyViewer继承自FunctionPass，我们需要对FunctionPass的runOnFunction方法进行重写，最终的Pass定义如下：

```
struct LegacyViewer : public llvm::FunctionPass {
  static char ID;
  LegacyViewer() : FunctionPass(ID) {}
  bool runOnFunction(llvm::Function &F) override;
  Viewer Impl;
};
```

2.2. Pass 实现

Pass的实现过程分为两个重要部分，一是Pass内算法的实现，二是Pass的注册。

Pass的内部算法由FunctionPass类的runOnFunction方法调用Imp的方法实现（返回值固定为false是因为本pass一定不会修改IR代码）：

```
bool LegacyViewer::runOnFunction(llvm::Function &F) {
  bool Changed = false;
  Impl.runOnFunction(F);
  return Changed;
}
```

而Imp对应的结构体Viewer也需要定义其方法，由于本Pass功能为显示每个函数的CFG图，我们调用llvm::Function类的方法显示函数的CFG图（需要当前环境下存在xdot与gv）：

```
bool Viewer::runOnFunction(llvm::Function &F) {
  F.viewCFG();
  return false;
}

PreservedAnalyses Viewer::run(llvm::Function &F,
```

```
                                llvm::FunctionAnalysisManager &) {
  bool Changed = false;
  runOnFunction(F);
  return (Changed ? llvm::PreservedAnalyses::none()
                  : llvm::PreservedAnalyses::all());
}
```

Pass的注册过程较为固定，我们唯一需要指定的就是pass调用时opt的参数名（下文会介绍），具体来说有如下代码：

```
static RegisterPass<LegacyViewer> X(/*PassArg=*/"viewer",
                                    /*Name=*/"Viewer",
                                    /*CFGOnly=*/true,
                                    /*is_analysis=*/false);
```

其中viewer便为opt的参数名。

其总的注册函数如下：

```
llvm::PassPluginLibraryInfo getViewerPluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "Viewer", LLVM_VERSION_STRING,
          [](PassBuilder &PB) {
            PB.registerPipelineParsingCallback(
                [](StringRef Name, FunctionPassManager &FPM,
                   ArrayRef<PassBuilder::PipelineElement>) {
                  if (Name == "Viewer") {
                    FPM.addPass(Viewer());
                    return true;
                  }
                  return false;
                });
          }};
}

extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
  return getViewerPluginInfo();
}

char LegacyViewer::ID = 0;

// 提供 opt 注册 pass
static RegisterPass<LegacyViewer> X(/*PassArg=*/"viewer",
                                    /*Name=*/"Viewer",
                                    /*CFGOnly=*/true,
                                    /*is_analysis=*/false);
```

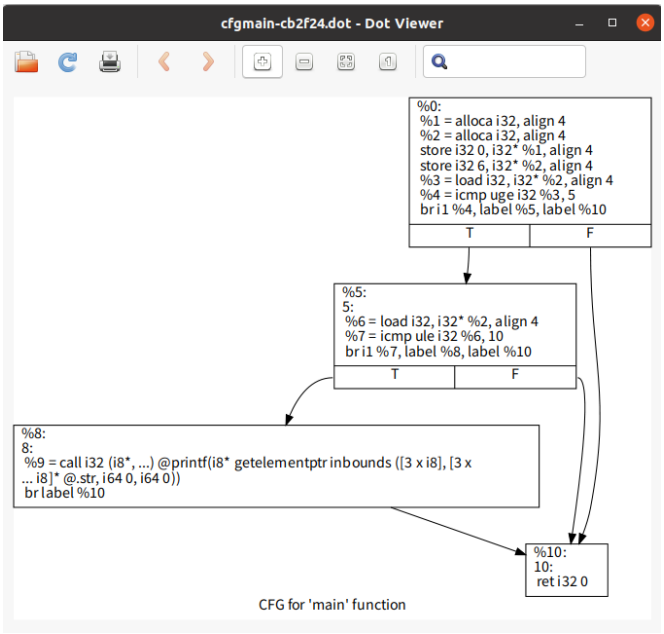
2.3. 运行 Pass

将Pass文件完成后，我们便需要重新编译整个LLVM项目（编译方法参见https://github.com/llvm/llvm-project），第一次编译耗时较久（由计算机性能决定，我们编译了2-3小时）。之后的每次编译仅需要在build文件夹下输入make指令，编译耗时约为一分钟（此时我们不需要编译整个项目，Pass文件在项目编译到39%时便已经编译完成）。

编译完成后，我们通过如下命令运行Pass（cmp.ll为输入的IR文件）：

```
opt -load ~/llvm-project-main/build/lib/Viewer.so -viewer -S cmp.ll -
o cmpOut.ll
```

运行后的结果如下：



3. Pass介绍

本文接下来将介绍我们实现的另外4种Pass，并针对某些Pass实现的方式做一定的说明。

3.1. Sub 运算的转换

此Pass的功能为：将代码中的如a-b替换为(a + ~b) + 1。此Pass主要是展示通过Pass在IR中进行代码替换的方式。其功能性代码如下：

```
bool MBASub::runOnBasicBlock(BasicBlock &BB) {
  bool Changed = false;

  // 遍历每个代码块中的指令
  for (auto Inst = BB.begin(), IE = BB.end(); Inst != IE; ++Inst) {

    // 跳过非二元运算符
    auto *BinOp = dyn_cast<BinaryOperator>(Inst);
    if (!BinOp)
      continue;

    // 跳过非整数运算
    unsigned Opcode = BinOp->getOpcode();
    if (Opcode != Instruction::Sub || !BinOp->getType()->isIntegerTy())
      continue;

    // 代码替换
    IRBuilder<> Builder(BinOp);

    // 创建 (a + ~b) + 1 替换原指令
    Instruction *NewValue = BinaryOperator::CreateAdd(
        Builder.CreateAdd(BinOp->getOperand(0),
                          Builder.CreateNot(BinOp->getOperand(1))),
        ConstantInt::get(BinOp->getType(), 1));

    // 替换函数
    ReplaceInstWithInst(BB.getInstList(), Inst, NewValue);
    Changed = true;
```

```
    // 更新此静态变量，表示改变的代码条数
    ++SubstCount;
  }
  return Changed;
}
```

由上面的代码很容易看出，遍历每个代码块、每一条指令LLVM都提供了相应的迭代器，我们在上面的Pass中做的工作主要是识别SUB指令并生成新的指令以替换它。其中创建的过程主要是通过IRBuilder来完成的，llvm::IRBuilder类封装了所有指令的产生函数。多数情况下，我们只要输入正确的操作数、调用指令生成函数便可以获得一个llvm::Value类型（llvm::Value为llvm的IR生成部分最重要的基类之一，详情参见llvm官方文档 https://llvm.org/doxygen/classllvm_1_1Value.html）的指针，其指向的内容即为生成的指令。随后我们只需要调用ReplaceInstWithInst函数便可以完成新老指令的替换。

当输入用例如下时：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = sub nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  ret i32 0
}
```

输出用例转换为：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = xor i32 %6, -1
  %8 = add i32 %5, %7
  %9 = add i32 %8, 1
  store i32 %9, i32* %4, align 4
  ret i32 0
}
```

可以看出，原文件产生%7的指令被替换为了三条对应的指令，Pass完成了IR代码的替换工作。

3.2. Mul 运算转换

乘法Pass的原理为将整数乘法乘以2的指数幂次的指令替换为左移指令，由于位操作快于乘法运算，该优化一定程度上来说为正优化。

其实现的代码如下：

```
bool MyPass::runOnBasicBlock(BasicBlock &BB) {
  bool Changed = false;
```

```
  for (auto Inst = BB.begin(), IE = BB.end(); Inst != IE; ++Inst) {

    // 跳过非二元运算
    auto *BinOp = dyn_cast<BinaryOperator>(Inst);
    if (!BinOp)
      continue;

    // 跳过非乘法运算.
    unsigned Opcode = BinOp->getOpcode();
    if (Opcode != Instruction::Mul || !BinOp->getType()->isIntegerTy())
      continue;

    int constIntValue, target;
    // 获得乘法的常数操作数，如没有则跳过
    if (ConstantInt* CI = dyn_cast<ConstantInt>(BinOp->getOperand(0)))
{
      if (CI->getBitWidth() <= 32) {
        constIntValue = CI->getSExtValue();
        target = 1;
      }
    }
    else if(ConstantInt* CI = dyn_cast<ConstantInt>(BinOp->getOperand(1
))){
      if (CI->getBitWidth() <= 32) {
        constIntValue = CI->getSExtValue();
        target = 0;
      }
    }
    else continue;

    // 判断该操作数是否为 2 的整数次方
    if(constIntValue < 1 || (constIntValue & (constIntValue - 1)) != 0)
 continue;

    int n = 0;
    // 获取左移位数
    do
    {
      constIntValue >>= 1;
      ++n;
    }while (constIntValue);

    // 代码替换
    IRBuilder<> Builder(BinOp);

    // 创建 a << n
    Instruction *NewValue = BinaryOperator::CreateShl(
        BinOp->getOperand(target),
        ConstantInt::get(BinOp->getType(), n - 1)
    );

    // 替换
    ReplaceInstWithInst(BB.getInstList(), Inst, NewValue);
    Changed = true;
    ++SubstCount;
  }
  return Changed;
}
```

上述Pass与Sub指令优化大体相同，但多出了一定的常数操作，由于LLVM自己实现了一套类JAVA标准库的操作，将常数转换为Int类型需要首先尝试将一个操作数转换为ConstantInt类型（两个操作数都不是时跳过该命令），然后将调用ConstIntValue 类的getSExtValue方法获取其对应的整形数字。

生成新指令、替换等工作就不再赘述，下面为输入用例：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  %6 = load i32, i32* %2, align 4
  %7 = mul nsw i32 %6, 2
  store i32 %7, i32* %3, align 4
  %8 = load i32, i32* %2, align 4
  %9 = mul nsw i32 %8, 4
  store i32 %9, i32* %4, align 4
  %10 = load i32, i32* %2, align 4
  %11 = mul nsw i32 %10, 256
  store i32 %11, i32* %5, align 4
  ret i32 0
}
```

其输出为：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  %6 = load i32, i32* %2, align 4
  %7 = shl i32 %6, 1
  store i32 %7, i32* %3, align 4
  %8 = load i32, i32* %2, align 4
  %9 = shl i32 %8, 2
  store i32 %9, i32* %4, align 4
  %10 = load i32, i32* %2, align 4
  %11 = shl i32 %10, 8
  store i32 %11, i32* %5, align 4
  ret i32 0
}
```

可以看出经优化后的代码%9由乘4变为了左移两位。

3.3. 代码块简化

代码块简化部分是这几个Pass中唯一涉及到改变代码块结构的，其主要分为三个部分：消除无用代码块、消除常数条件跳转、合并仅有单一前驱代码块。

其中，消除无用代码块代码如下：

```
bool Simplify::removeDeadBlocks(llvm::Function &F) {
  bool Changed = false;
  for (auto &BB : make_early_inc_range(F)) {
    // 消除既没有前驱也无后继的代码块
    if (&F.getEntryBlock() == &BB || !pred_empty(&BB))
```

```
      continue;
.
    for (BasicBlock *Succ : successors(&BB))
      Succ->removePredecessor(&BB);

    // 用 Undeftype 替换其中所有指令
    while (!BB.empty()) {
      Instruction &I = BB.back();
      I.replaceAllUsesWith(UndefValue::get(I.getType()));
      I.eraseFromParent();
    }
    //消除代码块
    BB.eraseFromParent();
    Changed = true;
  }
  return Changed;
}
```

此部分基本思路便是识别既没有前驱有没有后继的代码块，并将其中所有指令替换为空指令，最终消除代码块。

消除常数条件跳转部分代码如下：

```
bool Simplify::eliminateCondBranches(llvm::Function &F) {
  bool Changed = false;

  // 消除有常数跳转的指令
  for (auto &BB : F) {
    // 跳过没有以条件跳转结尾的指令
    BranchInst *BI = dyn_cast<BranchInst>(BB.getTerminator());
    if (!BI || !BI->isConditional())
      continue;

    // 跳过没有常数条件跳转的指令
    ConstantInt *CI = dyn_cast<ConstantInt>(BI->getCondition());
    if (!CI)
      continue;

    BasicBlock *RemovedSucc = BI->getSuccessor(CI->isOne());
    RemovedSucc->removePredecessor(&BB);

    // 将条件跳转替换为无条件跳转
    BranchInst::Create(BI->getSuccessor(CI->isZero()), BI);
    BI->eraseFromParent();
    Changed = true;
  }
  return Changed;
}
```

此部分代码思路也很简单，即识别常数跳转指令、替换指令两步。

合并仅有单一前驱代码块部分代码如下：

```
bool Simplify::mergeIntoSinglePredecessor(llvm::Function &F) {
  bool Changed = false;

  // 合并仅有单一前驱的代码块
  for (auto &BB : make_early_inc_range(F)) {
    BasicBlock *Pred = BB.getSinglePredecessor();
    // 确保 BB 仅有单一的前驱 Pred 并且 Pred 仅有单一的后继 BB
    if (!Pred || Pred->getSingleSuccessor() != &BB)
      continue;
```

```
    // 跳过自循环代码块
    if (Pred == &BB)
      continue;

    BB.replaceAllUsesWith(Pred);
    for (auto &PN : make_early_inc_range(BB.phis())) {
      PN.replaceAllUsesWith(PN.getIncomingValue(0));
      PN.eraseFromParent();
    }
    // 将 BB 中所有的指令移到 Pred 中
    for (auto &I : make_early_inc_range(BB))
      I.moveBefore(Pred->getTerminator());

    // 替换 Pred 的终结指令为 BB 的终结指令
    Pred->getTerminator()->eraseFromParent();
    BB.eraseFromParent();

    Changed = true;
  }

  return Changed;
}
```

此部分思路为：首先识别可合并的代码块，再将后继中每一天指令移动到其前驱中。

代码块合并的Pass思路较为简单，但需要注意的是，代码块的操作需要很多前置条件，否则便会引起问题。

3.4. 平方根倒数优化

平方根倒数优化是这三个Pass中最针对特定问题的一种优化，其原理为：当我们想求一个数的平方根倒数时，可以使用一个Magic Number来快速近似的求得其值。表述为C++代码如下：

```
float r_sqrt(float n) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;
    x2 = n * 0.5F;
    y = n;
    i = *(long *)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *)&i;
    y = y * (threehalfs - (x2 * y * y));
    y = y * (threehalfs - (x2 * y * y));
    return y;
}
```

所以，如果为了使用Pass去完成该优化，我们的任务就变成了：识别类似于a = 1/sqrt(n)的表达式，然后将其替换为a = r_sqrt(n)的形式（当然为了效率更快，我们采用内联的形式调用r_sqrt函数）。

不过在替换之前，我们需要得到r_sqrt函数的IR表述形式来方便我们的替换工作。于是，我们将r_sqrt函数经过clang以o3的方式编译为IR得到如下形式：

```
define dso_local float @r_sqrt(float %0) local_unnamed_addr #0 {
  %2 = fmul float %0, 5.000000e-01
  %3 = bitcast float %0 to i32
  %4 = lshr i32 %3, 1
  %5 = sub nsw i32 1597463007, %4
```

```
  %6 = bitcast i32 %5 to float

  %7 = fmul float %2, %6
  %8 = fmul float %7, %6
  %9 = fsub float 1.500000e+00, %8
  %10 = fmul float %9, %6

  %11 = fmul float %2, %10
  %12 = fmul float %10, %11
  %13 = fsub float 1.500000e+00, %12
  %14 = fmul float %10, %13
  ret float %14
}
```

这就是我们希望将平方根倒数希望替换的形式。

接下来需要解决的问题就是识别所有的a = 1/sqrt(n)的表达式，我的解决方案便是：识别call命令调用sqrt函数的指令，再寻找该指令所有的use点，最终识别出求倒数的部分，由于具体操作较为复杂，我们将中代码注释中解释具体过程：

```
bool Sqrt::runOnBasicBlock(BasicBlock &BB) {
  bool Changed = false;

  // 遍历代码块
  for (auto Inst = BB.begin(), IE = BB.end(); Inst != IE; ++Inst) {
    auto *CallOp = dyn_cast<CallInst>(Inst);
    if (!CallOp)
      continue;
    // 判断是否为 call sqrt 函数
    if(CallOp->getCalledFunction()->getName().equals("sqrt")){
      // 获取 sqrt 函数的参数
      Value* param = CallOp->arg_begin()->get();
      int count = 0;
      for(auto arg = CallOp->arg_begin(); arg < CallOp->arg_end(); arg+
+){
        count++;
        if(!arg->get()->getType()->isDoubleTy())
          count ++;
      }
      if(count > 1) continue;
      //我们需要存储所有的 use 点以保证此 Pass 的安全性
      bool canBeSubstituted = false;
      struct targetInsertPoint{
        int target;
        Instruction* insertPoint;
      };
      std::vector<targetInsertPoint> target_list;
      // 遍历所有 sqrt 的 use 点
      for(User *U: CallOp -> users()){
        // 判断是否可以被替换
        if(Instruction* Inst = dyn_cast<Instruction>(U)){
          canBeSubstituted = false;
          auto *nextInst = dyn_cast<BinaryOperator>(Inst);
          auto nextOp = nextInst->getOpcode();
          //use 点必须是 Fdiv 指令，且另一个操作数为 1
          if(nextOp != Instruction::FDiv)
            continue;
          //获取另一个操作数
          APFloat constFloatValue(0.0);
```

```
          int target;
          if (ConstantFP* CI = dyn_cast<ConstantFP>(nextInst->getOperan
d(0))) {
            constFloatValue = CI->getValue();
            target = 1;
          }
          else if(ConstantFP* CI = dyn_cast<ConstantFP>(nextInst->getOp
erand(1))){
            constFloatValue = CI->getValue();
            target = 0;
          }
          if(constFloatValue != APFloat(1.0)) continue;
          target_list.push_back({target, Inst});
          canBeSubstituted = true;
        }
        if(!canBeSubstituted) break;
      }
      if (!canBeSubstituted) continue;

      // 替换工作
      IRBuilder<> Builder(CallOp);
      // 该部分为构造部分，由于篇幅限制略去
      ..................
      auto *Inst1, *Inst2, *Inst3, *Inst4, *Inst5, *Inst6, *Inst7,
*Inst8, *Inst9, *Inst10, *Inst11, *Inst13, *Inst14, *Inst15;
      ..................
      // 替换 sqrt
      ReplaceInstWithInst(BB.getInstList(), Inst, Inst15);
      // 替换 use 点
      for(auto i : target_list){
        Instruction *newInst = BinaryOperator::CreateFSub(i.insertPoint
->getOperand(i.target), ConstantFP::get(param->getType(), 0.0));
        ReplaceInstWithInst(i.insertPoint, newInst);
      }
      Changed = true;
    }
    SubstCount += 2;
  }
  return Changed;
}
```

当上面的Pass输入用例为如下形式时：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca float, align 4
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store float 1.000000e+01, float* %2, align 4
  %4 = load float, float* %2, align 4
  %5 = fpext float %4 to double
  %6 = call double @sqrt(double %5) #3 ;替换部分
  %7 = fdiv double 1.000000e+00, %6
  %8 = fptrunc double %7 to float
  store float %8, float* %3, align 4
  %9 = load float, float* %3, align 4
  %10 = fpext float %9 to double
  %11 = load float, float* %2, align 4
  %12 = call float @r_sqrt(float %11)
```

```
  %13 = fpext float %12 to double
  %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8
], [7 x i8]* @.str, i64 0, i64 0), double %10, double %13)
  ret i32 0
}
```

其输出为：

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca float, align 4
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store float 1.000000e+01, float* %2, align 4
  %4 = load float, float* %2, align 4
  %5 = fpext float %4 to double
  %6 = fptrunc double %5 to float ;开始内联函数
  %7 = fmul float %6, 5.000000e-01
  %8 = bitcast float %6 to i32
  %9 = lshr i32 %8, 1
  %10 = sub nsw i32 1597463007, %9
  %11 = bitcast i32 %10 to float
  %12 = fmul float %7, %11
  %13 = fmul float %12, %11
  %14 = fsub float 1.500000e+00, %13
  %15 = fmul float %14, %11
  %16 = fmul float %7, %15
  %17 = fmul float %15, %16
  %18 = fsub float 1.500000e+00, %17
  %19 = fmul float %15, %18
  %20 = fpext float %19 to double
  %21 = fsub double %20, 0.000000e+00
  %22 = fptrunc double %21 to float ;内联结束
  store float %22, float* %3, align 4
  %23 = load float, float* %3, align 4
  %24 = fpext float %23 to double
  %25 = load float, float* %2, align 4
  %26 = call float @r_sqrt(float %25)
  %27 = fpext float %26 to double
  %28 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8
], [7 x i8]* @.str, i64 0, i64 0), double %24, double %27)
  ret i32 0
}
```

可以看出该Pass成功将r_sqrt函数内联进了main函数中，完成了其使用Magic Number优化平方根倒数的任务。