为了在实验中能掌握使用 LLVM 提供的功能，需要了解 LLVM 的接口函数。下面简单介绍使用 LLVM 提供的接口函数 API。

使用自定义的初始化函数 InitializeModuleAndPassManager，为 LLVM 的全局变量构建实例，用于中间代码生成：

1) theContext 生成 IR 中的上下文内容；
2) theModule IR 代码及环境、架构相关的所有内容；
3) builder IR 代码的构造器；

theFPM 为 LLVM 的代码优化的管理器，它通过 add 方法增加优化和代码分析的模块 Pass，初始化后需要调用其 run 方法进行优化。

LLVM 初始化部分及 InitializeModuleAndPassManager 函数的代码如下：

```
1.  using namespace llvm;
2.  std::unique_ptr<LLVMContext> theContext;  //上下文对象
1.  std::unique_ptr<Module> theModule;        //模块对象，包含所有其他 IR 对象的容器
2.  std::unique_ptr<IRBuilder<>> builder;     //构造器对象，协助产生 LLVM 指令
3.  std::map<std::string, AllocaInst *> namedValues;    // 命名变量
4.  std::unique_ptr<legacy::FunctionPassManager> theFPM;// 优化管理器
5.  void InitializeModuleAndPassManager() {             // 初始化模块和遍管理器
3.  theContext = std::make_unique<LLVMContext>();       // 打开一个上下文
4.  theModule = std::make_unique<Module>("test", *theContext);  // 打开模块
5.  // 创建模块对应的构造器对象 Create a new builder for the module.
6.  builder = std::make_unique<IRBuilder<>>(*theContext);
7.  // 创建遍管理器对象 Create a new pass manager attached to it.
8.  theFPM = std::make_unique<legacy::FunctionPassManager>(theModule.get());
9.  // 内存操作提升为寄存器 Promote allocas to registers.
10. theFPM->add(createPromoteMemoryToRegisterPass());
11. // 简单窥孔优化 simple "peephole" optimizations and bit-twiddling optzns.
12. theFPM->add(createInstructionCombiningPass());
13. // 重关联表达式 Reassociate expressions.
14. theFPM->add(createReassociatePass());
15. // 消除公共子表达式 Eliminate Common SubExpressions.
16. // TheFPM->add(createReassociatePass());
17. // 控制流程图简化 Simplify the control flow graph.
18. theFPM->add(createGVNPass());
19. theFPM->add(createCFGSimplificationPass());
20. theFPM->doInitialization();
21. }
```

借助 LLVM API，可以逐条生成 IR 指令。实验中需要使用的 API 如下，另外，可以进一步查阅官方文档理解其分类和含义。

```
6.  AllocaInst * CreateAlloca(Type *Ty, Value *ArraySize = nullptr, const Twine &Name =
    "")
7.  // 产生一条内存指令 alloca（属于指令产生）（Instruction creation，Memory）；用于声明变量;
    llvm/IRBuilder.h
8.  StoreInst * CreateStore(Value *Val, Value *Ptr, bool isVolatile = false)
9.  //类似，产生内存指令 store，将 Val 的值赋给 Ptr 指向的变量
```

```
LoadInst * CreateLoad(Type *Ty, Value *Ptr, bool isVolatile, const Twine &Name = "")

//产生指令内存指令 load，取变量 Ptr 的值
CallInst * CreateCall(FunctionType *FTy, Value *Callee, ArrayRef<Value *> Args = std
::nullopt, const Twine &Name = "", MDNode * FPMathTag = nullptr)
//产生函数调用指令（属于基本块结束指令）；可重载，无参数可用 CreateCall(calleeF)
Value * CreateICmpEQ(Value *LHS, Value *RHS, const Twine &Name = "")
//产生比较指令
BranchInst * CreateCondBr(Value *Cond, BasicBlock *True, BasicBlock *False, MDNode *
BranchWeights = nullptr, MDNode *Unpredictable = nullptr)
//产生条件指令（属于基本块结束指令），'br Cond,TrueDest,FalseDest'
void SetInsertPoint(BasicBlock *TheBB)
//生成器配置方法（Builder configuration），指定产生的指令被添加到基本块的末尾
BasicBlock * GetInsertBlock() const
//属于生成器配置方法（Builder configuration），获取当前操作的基本块指针；
IntegerType * getInt32Ty()
//属于类型创建方法（Type creation），获得 32 位整数的类型
Constant * ConstantInt::get(Type *Ty, const APInt& V)
//属于杂项创建方法（Miscellaneous creation methods），可重载，常用于获取常量值
static BasicBlock * llvm::BasicBlock::Create(LLVMContext & Context,const Twine & Nam
e = "",Function * Parent = nullptr,BasicBlock * InsertBefore = nullptr )
//属于基本块对象的创建方法，创建一个名字为 Name 的基本块。BasicBlock.h
```

A.8.3 LLVM 接口函数的使用举例

1. 常量

```
Value *const_1 = ConstantInt::get(*theContext, APInt(32, 1, true));
```

创建 int 类型常量 1，其中除上下文指针 theContext 外，还需要指定该常量对应的宽度 32，并通过 true 设置有符号类型。

变量与类型

2. 变量

```
AllocaInst *alloca_a = builder->CreateAlloca(Type::getInt32Ty(*theContext), nullptr
, "a");
```

创建变量需要使用构造器 builder，传入类型 Type::getInt32Ty(*theContext)，参数 nullptr 表示 ArraySize 为空，"a"表示助记词。

3. 表达式计算

表达式计算之前，需要先加载变量，然后再计算，存储。

加载使用 CreateLoad，创建加载变量指令，传入类型、变量地址和助记词；

计算使用 CreateAdd，创建运算符指令，传入两个操作数和助记符；

存储使用 CreateStore，创建存储指令，传入需要存储的值和存储地址。

```
//计算 a+b
//产生 load 指令，取出 a、b
Value*load_a = builder->CreateLoad(alloca_a->getAllocatedType(),alloca_a, "a");
```

```
4.  Value *load_b = builder->CreateLoad(alloca_b->getAllocatedType(), alloca_b, "b");
5.  //产生加法指令，计算结果存入临时
6.  Value *a_add_b = builder->CreateAdd(load_a, load_b, "add");
7.  //将结果存入 a 变量
8.  builder->CreateStore(a_add_b, alloca_a);
```

4. 函数

包括函数实现、函数调用两部分。

函数实现，需要先设置返回值类型、参数类型来创建函数，然后设置函数的参数信息。

```
1.  std::vector<Type *> argsTypes;      //参数类型
2.  std::vector<std::string> argNames; //参数名
3.  //无参，所以不 push 内容
4.  //得到函数类型
5.  FunctionType *ft = FunctionType::get(retType, argsTypes, false);
6.  //创建函数
7.  Function *f =  Function::Create(ft, Function::ExternalLinkage, "inc", theModule.get(
    ));
```

另外，函数入口为第一个基本块，要在函数开始时创建，如果函数有返回值，需要使用 builder->CreateRet 创建返回指令。

5. 分支结构

分支结构中，有两个关键指令 CreateCondBr 和 CreateBr。 前者根据 condVal 跳转，后者为无条件跳转。 可以根据语义逻辑，设计在合适的情况跳转，实现 if、while 等结构。

```
1.  //创建判断结果 condVal
2.  Value *condVal = builder->CreateICmpNE(compare_a_0, Constant::getNullValue(compare_a
    _0->getType()), "cond");
3.  //创建条件为真和假应跳转的两个基本块
4.  BasicBlock *thenb = BasicBlock::Create(*theContext, "then", f);
5.  BasicBlock *ifcontb = BasicBlock::Create(*theContext, "ifcont");
6.  //创建条件跳转指令，根据 condVal 跳转，真为 thenb 否则为 ifcontb
7.  builder->CreateCondBr(condVal, thenb, ifcontb);
8.  builder->SetInsertPoint(thenb);    //进入 thenb 基本块
9.  builder->CreateBr(ifcontb);  //创建无条件转移指令，转向基本块 ifcontb
10. f->getBasicBlockList().push_back(ifcontb);  //将基本块 ifcontb 插入函数中
```

利用上面的 API 函数，在中间代码的生成实验中，根据语言的语法规则，可以完成语法结构对应的分析函数 codegen，生成需要的中间代码。之后，在分析器的 main 函数中，先调用 InitializeModuleAndPassManager()初始化；然后调用不同语法成分的中间代码生成函数，最后将中间代码输出到文件中。

下面的例子，直接在 main 函数中完成简单的条件语句、函数调用语句的中间代码生成。

```
1.  int main(int argc, char *argv[]) {
2.  InitializeModuleAndPassManager();
3.  //输出函数 putchar
```

```
4.  std::vector<Type *> putArgs;
5.  putArgs.push_back(Type::getInt32Ty(*theContext));
6.  FunctionType *putType = FunctionType::get(builder->getInt32Ty(), putArgs, false);
7.  Function *putFunc = Function::Create(putType, Function::ExternalLinkage,"putchar", t
    heModule.get());
8.  //输入函数 getchar
9.  std::vector<Type *> getArgs;
10. FunctionType *getType = FunctionType::get(builder->getInt32Ty(), getArgs, false);
11. Function *getFunc = Function::Create(getType, Function::ExternalLinkage, "getchar",
    theModule.get());
12. //根据输入的单字符，判断，如果是'a'，则输出'Y'，否则输出'N'。
13. //设置返回类型
14. //***************begin***************
15. Type *retType = Type::getInt32Ty(*theContext);
16. std::vector<Type *> argsTypes;       //参数类型
17. std::vector<std::string> argNames; //参数名
18. FunctionType *ft = FunctionType::get(retType, argsTypes, false);//类型
19. Function *f = Function::Create(ft, Function::ExternalLinkage, "main", theModule.get(
    )); //创建函数
20. unsigned idx = 0;
21. for (auto &arg : f->args()) {
22. arg.setName(argNames[idx++]);  //处理函数 f 的参数
23. }
24. //创建第一个基本块，函数入口
25. BasicBlock *bb = BasicBlock::Create(*theContext, "entry", f);
26. builder->SetInsertPoint(bb);
27. AllocaInst *alloca_a = builder->CreateAlloca(Type::getInt32Ty(*theContext), nullptr,
     "a");  // 创建变量 a
28. Value *const_0 = ConstantInt::get(*theContext, APInt(32, 0, true));//常量 0
29. builder->CreateStore(const_0, alloca_a);  //初始化 a
30. Function *calleeF = theModule->getFunction("getchar");
31. std::vector<Value *> argsV;  //处理参数
32. Value *callgetchar = builder->CreateCall(calleeF, argsV, "callgetchar");
33. builder->CreateStore(callgetchar, alloca_a);
34. // if 结构
35. Value *load_a2 =  builder->CreateLoad(alloca_a->getAllocatedType(), alloca_a, "a");
    // 加载变量 a
36. Value *const_a = ConstantInt::get(*theContext, APInt(32, 'a', true)); //得到常量'a'
37. Value *compare_a_a =  builder->CreateICmpEQ(load_a2, const_a, "comp");
38. //创建条件为真和假应跳转的两个基本块
39. BasicBlock *thenb = BasicBlock::Create(*theContext, "then", f);
40. BasicBlock *elseb = BasicBlock::Create(*theContext, "else");
41. BasicBlock *ifcontb = BasicBlock::Create(*theContext, "ifcont");
42. builder->CreateCondBr(compare_a_a, thenb, elseb); //创建条件转移指令
```

```
43.  builder->SetInsertPoint(thenb);  //进入 thenb 基本块，增加块内指令
44.  Value *const_Y = ConstantInt::get(*theContext, APInt(32, 'Y', true));
45.  argsV.clear();  //准备参数
46.  argsV.push_back(const_Y);
47.  Function *calleeP = theModule->getFunction("putchar");
48.  builder->CreateCall(calleeP, argsV, "callputchar");  //产生函数调用指令
49.  builder->CreateBr(ifcontb);   // 无条件转移到基本块 ifcontb
50.  f->getBasicBlockList().push_back(elseb);  //创建 elseb 基本块 插入函数中
51.  builder->SetInsertPoint(elseb);  //进入基本块 elseb
52.  Value *const_N = ConstantInt::get(*theContext, APInt(32, 'N', true));
53.  argsV.clear();//准备参数
54.  argsV.push_back(const_N);
55.  builder->CreateCall(calleeP, argsV, "callputchar"); //产生函数调用指令
56.  builder->CreateBr(ifcontb);  //无条件转移到基本块 ifcontb
57.  f->getBasicBlockList().push_back(ifcontb);  //创建 ifcontb 插入函数 f
58.  builder->SetInsertPoint(ifcontb);  //进入基本块 infcontb
59.  //***************end***************
60.  builder->CreateRet(const_0);  //设置返回值
61.  verifyFunction(*f);//检查函数 f 指令是否合法
62.  // theFPM->run(*f);// Run the optimizer on the function.
63.  theModule->print(outs(), nullptr);  //输出指令到文件
64.  return 0;
65.  }
```

语义分析可以根据语法树对应节点，设计完成中间代码生成函数 codegen，填充语义分析过程。

例如声明抽象节点类 Node，其中语法分析函数为 parse，中间代码生成函数为 codegen，具体虚函数实现由派生的语法节点类完成。

```
1.   class Node {
2.   public:
3.   int line;
4.   std::string getNodeName() { return "node"; }
5.   virtual ~Node() {}
6.   virtual int parse() { return 0; }
7.   virtual int handle() { return 0; }
8.   virtual Value *codegen();
9.   };
```

例如表达式类 NExpression 派生自 Node，进一步整型常量类派生自表达式，为 NInteger，其中需要完成相应的语法分析和中间代码生成函数。

```
1.   /*Expressions*/
2.   class NExpression : public Node {
3.   public:
4.   std::string name;
5.   std::string getNodeName() { return "Exp"; }
6.   virtual int parse() { return 0; }
```

```
7.  virtual int handle() { return 0; }
8.  virtual Value *codegen();
9.  };
10. class NInteger : public NExpression {
11. public:
12. int value;
13. NInteger(int value) : value(value) {}
14. int parse();
15. int handle() { return 0; }
16. Value *codegen();
17. };
```

NInteger 的 codegen 函数可以参考代码如下：

```
Value *NInteger::codegen() {
  return ConstantInt::get(*theContext, APInt(32, value, true));
}
```

类似的，标识符也是表达式，对应声明及 codegen 函数参考如下：

```
1.  class NIdentifier : public NExpression {
2.  public:
3.  NIdentifier(const std::string &name) { this->name = name; }
4.  int parse();
5.  Value *codegen();
6.  };
7.  Value *NIdentifier::codegen() {
8.  // this function is to get the var that has been alloced.
9.  AllocaInst *A = namedValues[name];
10. if (!A) {  // not defined
11.     for (auto item = theModule->global_begin();
12.          item != theModule->global_end();++item) {
13.           GlobalVariable *gv = &*item;
14.           if (gv->getName() == name) {  //found the identifier in globalvars
15.              return builder->CreateLoad(gv); //load the variable and return
16.           }  //end if
17.      }  //end for
18.      printSemanticError(1, line, "Undefined variable " + name);
19.      return LogErrorV("Unknown variable name");  //return ErrorValue
20. }
21. // Load the value and return A as the Value;
22. return builder->CreateLoad(A->getAllocatedType(), A, name.c_str());
23. }
```

其他语法成分根据具体语言的语义进行类似翻译。

最后，在 main 函数中调用语法树根节点对应的 codegen 函数，进而根据子节点成员，再调用相应类的 codegen 函数，将所有 llvm 指令写入 theModule 对象中保存，最后调用 print 函数输出到文件中。

```
…
InitializeModuleAndPassManager();
If(p->codegen()){
   theModule->print(outs(),nullptr);
}
…
```