实验报告

实验四：Cminus-F 语言编译器的设计与实现

1.1 实验目的和要求

本实验的主要目的是通过实现 Cminus-F 语言的编译器，深入理解编译原理，特别是语义分析和 IR 生成的过程。具体要求如下：

1. 阅读并理解 Cminus-F 语言的语义规则。

2. 阅读 LightIR 核心类的介绍，理解其在编译过程中的作用。

3. 熟悉实验框架，理解如何使用框架以及注意事项。

4. 修改 `src/cminusfc/cminusf\_builder.cpp` 以实现自动 IR 产生算法，确保能正确编译任何合法的 Cminus-F 程序。

5. 在报告中解释设计思路、遇到的困难和解决方案。

1.2 实验环境

1. 硬件环境：鲲鹏开发板。

2. 软件环境：openEuler操作系统。

3. 开发工具：LLVM 编译器框架。

4. 测试环境：提供一组 Cminus-F 语言的测试代码。

1.3 实验内容

1. 实现 Cminus-f 语言的编译器，包括词法分析、语法分析和语义分析。

2. 生成 LLVM IR 代码，为后续的代码生成和优化打下基础。

1.4 实验过程

1.4.1 设计全局变量与宏定义

为了在编译过程中传递信息和状态，设计了全局变量和宏定义，包括：

- `res`：用于存储表达式的值。

- `arg`：用于存储参数指针，表示参数列表。

- `if\_return\_notadd`：用于判断返回值是否为地址。

同时，定义了一系列宏，方便代码实现，如获取整型和浮点型常量、类型判断等。

1.4.2 遇到的难点及解决方案

1.各个函数之间通过全局变量传递消息

分析其中各个函数都是返回值类型为void类型的函数，所以需要设计额外的全局变量进行相关信息的传递。

这里全局变量的设计绝对是难点，这里和其他同学进行讨论，最终设置了三个全局变量。

比如赋值语句val = expression，expression进行过visit操作之后，需要使用一个额外的全局变量res将其值存储起来。

Value\* res;

//ASTNum弄出来的值存在这里，因为所有函数的类型都是res的

Value \*arg; //参数指针，表示参数列表

bool if\_return\_notadd=false;//表示返回得到为地址还是返回值，用于后续判断

* res，针对的是val = expression等表达式，用于获得表达式的值
* parm，针对的是ASTParam，用于记录parm的值
* if\_return\_notadd,是针对于ASTVal节点，用于判断这个节点的编译后需不需要返回指针

2.进行语义分析 其中的语义检查过于繁琐，只处理了基本语义检查和一些常规语义检查。 并对其中的相关内容进行LOG输出，便于后续进行调试。

1. 当if语句和while语句的endBB基本块为空时翻译结果不符合语法规则
2. if语句和while语句的标号问题
3. ASTVar结点处理问题, 通过全局枚举变量 `lr` 记录节点是左值还是右值，区分处理。
4. 数组长度为0等相关情况
5. 数组下标报错问题：对于浮点数下标，转化为整型并检查是否为负数，必要时调用 `neg\_idx\_except` 函数。

-

1.5 实验设计

实验中详细讲解了各个语法结构的处理，包括程序声明、变量声明、函数声明、复合语句、表达式语句、选择语句、循环语句、返回语句、赋值语句、简单表达式、加法表达式、项等的处理。每个部分都涉及到具体的 LLVM IR 指令生成和语义规则的实现。

**如何设计全局变量与宏定义**

全局变量的存在意义是将在下层才能取到的信息带回上层，或者将上层的状态信息传递给下层，一般来说有这样的需要就应该创建一个对应的全局变量，在我的实验中，有以下全局变量以及宏定义：

* 设置一些宏用于方便代码实现，包括获取Int常量和Float常量，获取int32和Float这两种type，以及判断是不是整形、浮点型、指针
* #define CONST\_INT(num) \
* ConstantInt::get(num, module.get()) */\* 增加一个有关整型的宏 \*/*
* #define CONST\_FP(num) \
* ConstantFP::get((float)num, module.get())
* #define CONST\_ZERO(var\_type) \
* ConstantZero::get(var\_type, module.get())
* #define TYPTRINT \
* Type::get\_int32\_ptr\_type(module.get())
* #define TYPTRFLOAT \
* Type::get\_float\_ptr\_type(module.get())
* #define TYVOID \
* Type::get\_void\_type(module.get())
* #define TYINT32 \
* Type::get\_int32\_type(module.get())
* #define TYFLOAT \
* Type::get\_float\_type(module.get())
* #define checkInt(num)\
* num->get\_type()->is\_integer\_type()
* #define checkFloat(num) \
* num->get\_type()->is\_float\_type() */\* 浮点型判断 \*/*
* #define checkPointer(num) \
* num->get\_type()->is\_pointer\_type() */\* 指针类型判断 \*/*
* Value\* res;*//ASTNum弄出来的值存在这里，因为所有函数的类型都是res的*
* Value \*arg; *//参数指针，表示参数列表*
* **bool** if\_return\_notadd=false;*//表示返回得到为地址还是返回值，用于后续判断*
  + res，针对的是val = expression等表达式，用于获得表达式的值
  + parm，针对的是ASTParam，用于记录parm的值
  + if\_return\_notadd,是针对于ASTVal节点，用于判断这个节点的编译后需不需要返回指针

**遇到的难点以及解决方案**

* 当if语句和while语句的endBB基本块为空时翻译结果不符合语法规则

因为在函数中必须保证endBB或者其后继基本块中需要有return语句，故可以利用一个全局布尔变量is\_return来记录语句中是否有return语句。

* + 在if情况下，再利用if\_is\_return 和else\_is\_return 来记录if语句和else语句中是否有return语句，如果if语句和else语句（如果存在else语句）中均有return语句，则说明无论如何都会ret，则可以在endBB中也要加入return语句来确保语法正确。
  + 在while情况下，在function declaration的最后根据is\_return的值来确定是否在返回，若没有返回值，则根据函数的类型添加return语句
* if语句和while语句的标号问题

由于函数中可能出现多个if和while语句，故将需要将每次跳转的label区分开，有以下两种解决方案

* + 利用sprintf语句生成序号不同的labelname
  + 直接讲labelname设置为空，翻译时会自动填充如不同的label
* ASTVar结点处理问题

变量要区分是左值还是右值，通过全局枚举变量lr来记录该节点是左值还是右值，左值取变量地址，右值取变量的值。

通过node.expression判断参数是整型/浮点型还是数组类型。在处理右值时需要注意函数调用的形参在符号表均为指针类型（node.expression均为nullptr），故需要在整型/浮点型中需要特别处理形参是数组、指针的情况。

在处理取了下标的数组变量时，需要考虑数组下标是否合法，以及通过指针类型访问数组的情况。

* 数组下标报错问题

数组下标如果是浮点数，调用create\_fptosi接口转化为整型，再判断是否是负数，如果是负数需要调用neg\_idx\_except()，通过符号表得到该方法的地址然后使用接口create\_call调用。

**详细讲解每块内容**

**1.**program->declaration-list

确保数值常量被正确识别并转换为IR常量节点，为代码生成提供准确的数值表示，并在早期发现数据类型错误，避免后续编译问题

|  |
| --- |
| **void** **CminusfBuilder::visit**(ASTProgram &node) {  LOG(DEBUG) << "Program\n";  *//语义检查*  **if**(node.declarations.size()==0){  LOG(ERROR) << "程序缺少声明\n";  **return**;  }  **if**(!(node.declarations.back()->id=="main"&&node.declarations.back()->type==TYPE\_VOID))  {  LOG(ERROR) << "程序最后一个声明不是void main(void)\n";  **return**;  }  **for**(**auto** decl : node.declarations){  decl->accept(\***this**);*//solve decl*  }  } |

* **日志记录**：使用LOG(DEBUG)宏记录程序结点的开始，便于调试和跟踪程序的执行流程。
* **语义检查**
  + 首先，函数检查node.declarations是否为空，即程序中是否有声明。如果没有声明，使用LOG(ERROR)记录错误信息，并返回。
  + 其次，函数检查程序的最后一个声明是否是void main(void)函数。这是Cminus-f语言的一个要求，即主函数必须以void main(void)的形式存在。如果不是，同样记录错误信息并返回。
* **遍历声明**：使用范围基于for循环遍历node.declarations中的所有声明。对于每个声明，调用decl->accept(\*this)方法，这会根据声明的类型调用相应的visit函数进行处理。

**2.**num

|  |
| --- |
| **void** **CminusfBuilder::visit**(ASTNum &node) {  LOG(DEBUG) << "Num\n";  **if**(node.type==TYPE\_INT)  res=ConstantInt::get(node.i\_val,**module**.get());  **else** **if**(node.type==TYPE\_FLOAT)  res=ConstantFP::get(node.f\_val,**module**.get());  **else** LOG(ERROR) <<"缺少num\n";  } |

* **日志记录**：
  + LOG(DEBUG)宏记录处理数值常量节点的开始，便于调试和跟踪编译过程。
* **数值类型处理**：
  + 检查数值常量的类型，决定创建整数还是浮点数常量。
  + 若node.type为TYPE\_INT，则调用ConstantInt::get创建整数常量节点，结果存储在res。
  + 若node.type为TYPE\_FLOAT，则调用ConstantFP::get创建浮点数常量节点，结果存储在res。
* **错误处理**：
  + 若数值常量的类型既非整数也非浮点数，记录错误信息，指出缺少有效的数值类型。

**3.**var-declaration -> type-specifier ID | type-specifier ID [INTEGER]

|  |
| --- |
| **void** **CminusfBuilder::visit**(ASTVarDeclaration &node) {  LOG(DEBUG)<<"var-declration\n";  Type \*tmp;*//类型指针*  **if**(node.type==TYPE\_INT)tmp=TYINT32;  **else** **if**(node.type==TYPE\_FLOAT)tmp=TYFLOAT;  **else** LOG(ERROR)<<"变量声明中只有整型和浮点型可以使用\n";  *//通过判断其是否具有num，区分为数组和非数组*  **if**(node.num!=nullptr){  **if**(node.num->i\_val<=0)LOG(ERROR)<<"数组长度必须大于0\n";  **auto** \*arrayType=ArrayType::get(tmp,node.num->i\_val);*//根据数组长度得到类型*  **auto** init=CONST\_ZERO(tmp);*//全局变量初始化为0*  Value \*arrayAlloca;*//存储申请到的数组的地址*  **if**(scope.in\_global())arrayAlloca=GlobalVariable::create(node.id,**module**.get(),arrayType,false,init);  **else** arrayAlloca=builder->create\_alloca(arrayType);  scope.push(node.id,arrayAlloca);*//将获得的数组变量加入域*  }  **else**{  **auto** init=CONST\_ZERO(tmp);  Value \*varAlloca;*//存储申请到的变量的地址*  **if**(scope.in\_global())varAlloca=GlobalVariable::create(node.id,**module**.get(),tmp,false,init);  **else** varAlloca=builder->create\_alloca(tmp);  scope.push(node.id,varAlloca);*//将获得的变量加入域*  }  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录函数的调用，便于调试。
* **类型检查**：
  + 根据节点的类型指定变量的类型，整型对应TYINT32，浮点型对应TYFLOAT。
  + 如果类型既不是整型也不是浮点型，则记录错误。
* **数组与非数组声明**：
  + 检查是否声明了数组（通过node.num是否存在来判断）。
  + 对于数组声明：
    - 验证数组长度是否合法（大于0）。
    - 创建数组类型，并初始化为零。
    - 根据是否为全局变量，创建全局或局部存储空间。
    - 将数组变量添加到作用域。
  + 对于非数组声明：
    - 初始化变量为零。
    - 根据是否为全局变量，创建全局或局部存储空间。
    - 将变量添加到作用域。

**4.**fun-declaration—>type-specifier ID ( params ) compound-stmt

|  |
| --- |
| *//fun-declaration—>type-specifier ID ( params ) compound-stmt*  **void** **CminusfBuilder::visit**(ASTFunDeclaration &node) {  LOG(DEBUG) <<"fun-declaration\n";  Type \*retType;*//设定返回值类型*  **if**(node.type==TYPE\_INT)retType=TYINT32;  **else** **if**(node.type==TYPE\_FLOAT)retType=TYFLOAT;  **else** **if**(node.type==TYPE\_VOID)retType=TYVOID;  **else** LOG(ERROR)<<"函数返回类型只能为int,float,void";  *//根据函数声明，构造形参类型列表*  std::vector<Type \*>paramsType;  **for**(**auto** param:node.params){  **if**(param->isarray){  **if**(param->type==TYPE\_INT)paramsType.push\_back(TYPTRINT);  **else** paramsType.push\_back(TYPTRFLOAT);  }**else**{  **if**(param->type==TYPE\_INT)paramsType.push\_back(TYINT32);  **else** paramsType.push\_back(TYFLOAT);  }  }  **auto** funType =FunctionType::get(retType,paramsType);*//retType返回结构*  **auto** function=Function::create(funType,node.id,**module**.get());*//创建函数*  scope.push(node.id,function);  scope.enter();  **auto** bb=BasicBlock::create(**module**.get(),node.id+"\_entry",function);*//创建基本块*  builder->set\_insert\_point(bb);  std::vector<Value \*> args;*//创建vector存储实参*  **for**(**auto** arg=function->arg\_begin();arg != function->arg\_end(); arg++)args.push\_back(\*arg);  **for** (**int** i = 0; i < node.params.size(); i++)  { */\* 遍历形参列表 \*/*  **auto** param = node.params[i]; */\* 取出对应形参 \*/*  arg = args[i]; */\* 取出对应实参 \*/*  param->accept(\***this**); */\* 调用param的accept进行处理 \*/*  }  node.compound\_stmt->accept(\***this**); */\* 处理函数体内语句compound-stmt \*/*  *// 判断返回值的类型，根据对应的返回值类型，执行ret操作*    **if** (builder->get\_insert\_block()->get\_terminator() == nullptr)*//判断是否没有显式返回*  {  **if** (function->get\_return\_type()->is\_void\_type())*// 若没有显式返回，默认返回对应类型的0或void*  builder->create\_void\_ret();  **else** **if** (function->get\_return\_type()->is\_float\_type())  builder->create\_ret(CONST\_FP(0.0));  **else**  builder->create\_ret(CONST\_INT(0));  }  scope.exit(); *//退出此函数作用域*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录函数声明节点的处理开始。
* **返回值类型设置**：
  + 根据函数的返回类型（node.type），设置返回值类型为TYINT32（整型）、TYFLOAT（浮点型）或TYVOID（无返回值）。
  + 如果返回类型不是这三种之一，记录错误信息。
* **构造形参类型列表**：
  + 遍历函数的参数列表，根据每个参数是否为数组以及其类型，构造形参类型列表。
  + 数组参数的类型设置为指针类型（TYPTRINT或TYPTRFLOAT），非数组参数的类型设置为基本类型（TYINT32或TYFLOAT）。
* **创建函数**：
  + 使用FunctionType::get根据返回值类型和形参类型列表创建函数类型。
  + 使用Function::create创建函数实例，并将其添加到作用域中。
* **设置插入点**：
  + 创建一个新的基本块（BasicBlock）作为函数的入口点，并设置构建器的插入点到这个基本块。
* **处理形参**：
  + 遍历函数的形参和实参，对每个形参调用accept方法进行进一步处理。
* **处理函数体**：
  + 对函数体中的复合语句（compound-stmt）调用accept方法进行处理。
* **默认返回值处理**：
  + 如果函数没有显式的返回语句，根据函数的返回类型，添加默认的返回语句：
    - 无返回值的函数添加create\_void\_ret。
    - 浮点型返回值的函数添加create\_ret(CONST\_FP(0.0))。
    - 整型返回值的函数添加create\_ret(CONST\_INT(0))。
* **退出函数作用域**：
  + 处理完函数声明后，退出函数作用域。

**5.**param→type-specifier ID | type-specifier ID []

参数的存储空间分配、值的存储以及作用域的管理，为编译器的后续代码生成和语义分析提供了必要的参数信息

|  |
| --- |
| *//param→type-specifier ID | type-specifier ID []*  **void** **CminusfBuilder::visit**(ASTParam &node) {  LOG(DEBUG) <<"Param\n";  Value \*paramAlloca;*//分配参数的存储空间*  **if**(node.isarray){*//如果是数组*  **if**(node.type==TYPE\_INT)paramAlloca=builder->create\_alloca(TYPTRINT);  **else** **if**(node.type==TYPE\_FLOAT)paramAlloca=builder->create\_alloca(TYPTRFLOAT);  }**else** {*//如果不是数组*  **if**(node.type==TYPE\_INT)paramAlloca=builder->create\_alloca(TYINT32);  **else** **if**(node.type==TYPE\_FLOAT)paramAlloca=builder->create\_alloca(TYFLOAT);  }  builder->create\_store(arg,paramAlloca);*//arg通过全局变量传递*  scope.push(node.id,paramAlloca);*//将参数压入域中*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录函数参数节点的处理开始。
* **参数存储空间分配**：
  + 根据参数是否为数组以及其类型，分配相应的存储空间。
  + 如果参数是数组，分配一个指向整数或浮点数的指针类型的存储空间。
  + 如果参数不是数组，直接分配一个整数或浮点数类型的存储空间。
* **参数值存储**：
  + 使用builder->create\_store(arg, paramAlloca)将实参的值存储到分配的参数存储空间中，其中arg是通过全局变量传递的实参。
* **参数加入作用域**：
  + 使用scope.push(node.id, paramAlloca)将参数的标识符和其存储空间的地址压入作用域中，以便在函数体内可以访问和使用这些参数。

**6.**compound-stmt→{ local-declarations statement-list}

确保复合语句内的所有局部声明和语句被正确处理，并且维护作用域的正确性。

|  |
| --- |
| *//compound-stmt→{ local-declarations statement-list}*  **void** **CminusfBuilder::visit**(ASTCompoundStmt &node) {  LOG(DEBUG)<<"CompoundStmt\n";  scope.enter();  **for**(**auto** local : node.local\_declarations)local->accept(\***this**);  **for**(**auto** statement : node.statement\_list)statement->accept(\***this**);  scope.exit();  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录复合语句节点的处理开始，有助于调试和跟踪编译过程。
* **进入新作用域**：
  + 调用scope.enter()进入一个新的作用域。这是因为复合语句通常定义了自己的局部变量，这些变量应该只在复合语句内部可见。
* **处理局部声明**：
  + 遍历node.local\_declarations中的所有局部变量声明，并调用它们的accept方法。这会触发对每个局部变量声明的处理，例如为其分配存储空间并将其添加到当前作用域。
* **处理语句列表**：
  + 遍历node.statement\_list中的所有语句，并调用它们的accept方法。这会根据语句的类型调用相应的visit函数进行处理，例如处理变量赋值、函数调用等。
* **退出作用域**：
  + 处理完复合语句内的所有声明和语句后，调用scope.exit()退出当前作用域。这表示复合语句结束，与之相关联的局部变量将不再可见。

**7.**expression-stmt→expression ; | ;

|  |
| --- |
| *//expression-stmt→expression ; | ;*  **void** **CminusfBuilder::visit**(ASTExpressionStmt &node) {  LOG(DEBUG)<<"Expreesionstmt\n";  **if**(node.expression!=nullptr)node.expression->accept(\***this**);*//如果表达式不为空，说明其存在，处理*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录表达式语句节点的处理开始，这有助于调试和跟踪编译过程。
* **表达式处理**：
  + 检查node.expression是否不为nullptr，即判断是否有表达式存在。
  + 如果表达式存在，调用node.expression->accept(\*this)处理该表达式。这会根据表达式的类型调用相应的visit函数，执行表达式的语义分析和中间代码生成。

**8.**selection-stmt→ if ( expression ) statement| if ( expression ) statement else statement

确保if和if-else语句被正确处理，包括条件表达式的求值、条件跳转、以及if和else块的执行，正确地处理条件控制流,便于后续数据流的优化

|  |
| --- |
| *//selection-stmt→ if ( expression ) statement| if ( expression ) statement else statement*  **void** **CminusfBuilder::visit**(ASTSelectionStmt &node) {  LOG(DEBUG)<<"selection-stmt";  node.expression->accept(\***this**);*//solve statement*  **auto** resType=res->get\_type();*//selection-stmt type*  **if**(resType->is\_pointer\_type())res=builder->create\_load(res);  **else** **if**(resType->is\_integer\_type())res=builder->create\_icmp\_gt(res,CONST\_ZERO(TYINT32));  **else** **if**(resType->is\_float\_type())res=builder->create\_fcmp\_gt(res,CONST\_ZERO(TYFLOAT));    **auto** function=builder->get\_insert\_block()->get\_parent();  **auto** trueBB=BasicBlock::create(**module**.get(),"true",function);*//构造truebb*    **if**(node.else\_statement!=nullptr){  **auto** falseBB=BasicBlock::create(**module**.get(),"false",function);*//存在else构造falsebb*  builder->create\_cond\_br(res,trueBB,falseBB);  builder->set\_insert\_point(trueBB);*//符合if*  node.if\_statement->accept(\***this**);*//处理else*  **auto** tbb=builder->get\_insert\_block();*//将块加入*  builder->set\_insert\_point(falseBB);*//符合else*  node.else\_statement->accept(\***this**);  **auto** fbb=builder->get\_insert\_block();  **auto** tTerm=builder->get\_insert\_block()->get\_terminator();*//判断true是否存在ret*  **auto** fTerm=builder->get\_insert\_block()->get\_terminator();*//判断false是否存在ret*  **if**(tTerm==nullptr||fTerm==nullptr){*//如果不存在ret*  BasicBlock \*retBB;  retBB=BasicBlock::create(**module**.get(),"retbb",function);  builder->set\_insert\_point(retBB);  **if**(tTerm==nullptr)builder->set\_insert\_point(trueBB);  **else** **if**(fTerm==nullptr)builder->set\_insert\_point(falseBB);  builder->create\_br(retBB);  }  }**else**{*//不存在else*  **auto** retBB=BasicBlock::create(**module**.get(),"ret",function);  builder->create\_cond\_br(res,trueBB,retBB);*//设置跳转*  builder->set\_insert\_point(trueBB);  node.if\_statement->accept(\***this**);*//条件符合执行*  **if**(builder->get\_insert\_block()->get\_terminator()==nullptr)builder->create\_br(retBB);*//补充return*  builder->set\_insert\_point(retBB);*//后续return*  }  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录条件选择语句节点的处理开始，有助于调试和跟踪编译过程。
* **表达式求值**：
  + 调用node.expression->accept(\*this)处理if语句的条件表达式，结果存储在res中。
* **结果类型处理**：
  + 根据res的类型，进行不同的处理：
    - 如果是指针类型，使用create\_load加载其指向的值。
    - 如果是整数类型，使用create\_icmp\_gt与0比较。
    - 如果是浮点类型，使用create\_fcmp\_gt与0比较。
* **基本块创建**：
  + 创建trueBB基本块，用于条件为真时的代码。
  + 如果存在else语句，创建falseBB基本块，用于条件为假时的代码。
* **条件跳转**：
  + 使用create\_cond\_br根据条件表达式的结果进行条件跳转。
* if**语句处理**：
  + 设置插入点到trueBB，处理if语句的主体。
  + 如果if语句块中没有终止指令（如return），则创建一个跳转到retBB的无条件跳转指令。
* else**语句处理**（如果存在）：
  + 设置插入点到falseBB，处理else语句的主体。
  + 如果else语句块中没有终止指令，同样创建一个跳转到retBB的无条件跳转指令。
* **统一返回处理**：
  + 如果if或else块中没有返回语句，创建一个retBB基本块，并在每个块的末尾添加跳转到retBB的指令。
* **结束处理**：
  + 设置插入点到retBB，作为后续代码的继续点。

**9.**iteration-stmt→while ( expression ) statement

负责循环语句（即while循环）,包括条件判断、循环体执行以及循环的退出

|  |
| --- |
| *//iteration-stmt→while ( expression ) statement*  **void** **CminusfBuilder::visit**(ASTIterationStmt &node) {  LOG(DEBUG)<<"iteration-stmt";  **auto** function=builder->get\_insert\_block()->get\_parent();*//获得当前块对应的函数*  **auto** conBB=BasicBlock::create(**module**.get(),"condition",function);  **auto** loopBB=BasicBlock::create(**module**.get(),"loop",function);  **auto** retBB=BasicBlock::create(**module**.get(),"ret",function);  **if**(builder->get\_insert\_block()->get\_terminator()==nullptr)builder->create\_br(conBB);*//跳转至条件判断块*    builder->set\_insert\_point(conBB);*//构建条件判断块*  node.expression->accept(\***this**);*//处理表达式，结果存到expression*  **auto** resType=res->get\_type();*//表达式结果类型*  **if**(resType->is\_pointer\_type())res=builder->create\_load(res);  **else** **if**(resType->is\_integer\_type())res=builder->create\_icmp\_gt(res,CONST\_ZERO(TYINT32));  **else** **if**(resType->is\_float\_type())res=builder->create\_fcmp\_gt(res,CONST\_ZERO(TYFLOAT));  builder->create\_cond\_br(res,loopBB,retBB);*//条件跳转*  *//loop*  builder->set\_insert\_point(loopBB);  node.statement->accept(\***this**);  **if**(builder->get\_insert\_block()->get\_terminator()==nullptr)builder->create\_br(conBB);    builder->set\_insert\_point(retBB);*//return 块*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录循环语句节点的处理开始，有助于调试和跟踪编译过程。
* **基本块创建**：
  + 创建三个基本块：conBB用于条件判断，loopBB用于循环体，retBB用于循环结束后的退出点。
* **初始跳转**：
  + 如果当前基本块没有终止指令，添加一个无条件跳转到条件判断块conBB的指令。
* **条件判断**：
  + 设置插入点到conBB。
  + 处理while循环的条件表达式，并根据其类型进行适当的比较操作，结果存储在res中。
  + 根据res的值进行条件跳转：如果为真，跳转到循环体loopBB；如果为假，跳转到退出块retBB。
* **循环体执行**：
  + 设置插入点到loopBB。
  + 处理循环体内的语句。
* **循环继续**：
  + 如果循环体结束后没有终止指令，添加一个无条件跳转回条件判断块conBB的指令，以继续循环。
* **退出循环**：
  + 设置插入点到retBB，作为循环结束后的代码继续执行的点。

**10.**return-stmt→return ; | return expression

|  |
| --- |
| *//return-stmt→return ; | return expression*  **void** **CminusfBuilder::visit**(ASTReturnStmt &node) {  LOG(DEBUG)<<"return-stmt";  **auto** function = builder->get\_insert\_block()->get\_parent(); *//得到函数所在范围*  **auto** retType = function->get\_return\_type();*//得到函数return的类型*  **if** (retType->is\_void\_type())*//若返回值为void*  {  builder->create\_void\_ret(); *//构建void返回值*  **return**;*//直接返回*  }  *//非空*  node.expression->accept(\***this**); *//处理expression条件判断对应的表达式*  **auto** resType = res->get\_type(); *//获取表达式得到的结果类型*  *// 处理expression返回的结果*      **if** (retType->is\_integer\_type() && resType->is\_float\_type()){res = builder->create\_fptosi(res, TYINT32);LOG(WARNING) << "强制类型转换 float转int，返回值需要与定义一致";}  **if** (retType->is\_float\_type() && resType->is\_integer\_type()){res = builder->create\_sitofp(res, TYINT32);LOG(WARNING) << "强制类型转换 int转float，返回值需要与定义一致";}    *//创建return，将expression的结果进行返回*  builder->create\_ret(res);  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录return语句节点的处理开始，有助于调试和跟踪编译过程。
* **获取函数返回类型**：
  + 通过builder->get\_insert\_block()->get\_parent()获取当前正在处理的函数。
  + 通过function->get\_return\_type()获取该函数的返回类型。
* **处理void返回类型**：
  + 如果函数的返回类型为void，则直接创建一个void返回指令builder->create\_void\_ret()并结束函数处理。
* **处理非void返回类型**：
  + 如果return语句带有表达式，调用node.expression->accept(\*this)处理该表达式，并将结果存储在res中。
  + 获取表达式的结果类型resType。
* **类型转换**：
  + 如果函数期望的返回类型是整数，但表达式结果是浮点数，则将浮点数强制转换为整数。
  + 如果函数期望的返回类型是浮点数，但表达式结果是整数，则将整数强制转换为浮点数。
  + 在进行类型转换时，记录警告日志，提示类型不匹配。
* **创建返回指令**：
  + 使用builder->create\_ret(res)创建返回指令，将表达式的结果作为返回值。

**11.**var→ID | ID [expression]

确保变量和数组元素的访问被正确处理，包括索引的合法性检查、类型转换和地址或值的获取，能够正确地处理变量访问和数组索引，为生成正确的中间表示代码提供基础。

|  |
| --- |
| *//var→ID | ID [expression]*  **void** **CminusfBuilder::visit**(ASTVar &node) {  LOG(DEBUG)<<"var\n";  **auto** var = scope.find(node.id); *//从域中取出对应变量 ID*  **bool** if\_return\_lvalue = if\_return\_notadd; *// 判断是否需要返回地址*  if\_return\_notadd = false; *// 重置*  Value \*index = CONST\_INT(0); *//init index for array*  **if** (node.expression != nullptr)*//带有表达式,为数组*  {  node.expression->accept(\***this**); *// 处理expression，得到res*  **auto** res1 = res; *// 存储结果*  **if** (checkFloat(res1)){res1 = builder->create\_fptosi(res1, TYINT32); LOG(WARNING)<<"下标应为整数，float改为int";}*// 若是浮点数，则矫正为整数*    index = res1; *//表示数组下标*  *// 判断下标是否为负数*  **auto** function = builder->get\_insert\_block()->get\_parent(); *// 获取当前函数*  **auto** indexTest = builder->create\_icmp\_lt(index, CONST\_ZERO(TYINT32)); *// 测试index是否为负数*  **auto** failBB = BasicBlock::create(**module**.get(), node.id + "\_failTest", function); *// 创建fail块*  **auto** passBB = BasicBlock::create(**module**.get(), node.id + "\_passTest", function); *// 创建pass块*  builder->create\_cond\_br(indexTest, failBB, passBB); *// 如果为负，则跳转到fail块*  *// 下标为负数，调用neg\_idx\_except函数进行处理*  *// 一个负的下标会导致程序终止，需要调用框架中的内置函数`neg\_idx\_except` （该内部函数会主动退出程序，只需要调用该函数即可），但是对于上界并不做检查。*  builder->set\_insert\_point(failBB); *// fail块，即下标为负数*  **auto** fail = scope.find("neg\_idx\_except"); *// 取出neg\_idx\_except函数*  builder->create\_call(**static\_cast**<Function \*>(fail), {}); *// 调用neg\_idx\_except函数进行处理*  builder->create\_br(passBB); *// 跳转到pass块*  *// 下标合法*  builder->set\_insert\_point(passBB); *// pass块*  **if** (var->get\_type()->get\_pointer\_element\_type()->is\_array\_type()) *// 若为指向数组的指针*  var = builder->create\_gep(var, {CONST\_INT(0), index}); *// 得到对应数组元素的指针，即指向元素*  **else**  {  **if** (var->get\_type()->get\_pointer\_element\_type()->is\_pointer\_type()) *// 若为指针*  var = builder->create\_load(var); *// 则取出指针指向的元素*  var = builder->create\_gep(var, {index}); *// 得到对应元素的指针*  }  *//expression中有内容*  **if** (if\_return\_lvalue) *// 若要返回值*  {  res = var; *// 则返回var对应的地址*  if\_return\_notadd = false;  }  **else** res = builder->create\_load(var); *// 否则则进行load*  **return**;  }  **else**{ *// 处理不是数组的情况 var->ID*  **if** (if\_return\_lvalue)  {  res = var; *// 若要返回值,则返回var对应的地址*  if\_return\_notadd = false; *//重置*  }  **else**  { *// 数组的指针即a[]类型就返回数组的起始地址，否则load取值*  **if** (var->get\_type()->get\_pointer\_element\_type()->is\_array\_type()) *// 若指向数组，需要两个偏移取地址*  res = builder->create\_gep(var, {CONST\_INT(0), CONST\_INT(0)}); *//寻址，找到对应数组的首地址*  **else**  res = builder->create\_load(var); *// 否则则是普通指针，直接进行load获得其指向元素*  }  }  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录变量节点的处理开始，有助于调试和跟踪编译过程。
* **变量查找**：
  + 通过scope.find(node.id)从当前作用域中查找变量的存储位置。
* **地址返回标记**：
  + if\_return\_lvalue标记指示是否需要返回变量的地址而不是值。
* **数组索引处理**：
  + 如果node.expression不为空，表示访问的是数组元素。
  + 处理数组索引表达式，并将其结果存储在res1中。
  + 如果索引是浮点数，则将其转换为整数索引。
* **负索引检查**：
  + 创建两个基本块failBB和passBB，用于处理数组索引是否为负数的情况。
  + 如果索引为负数，调用neg\_idx\_except函数处理异常，并跳转到passBB。
* **数组元素访问**：
  + 如果变量是指向数组的指针，使用create\_gep获取数组元素的指针。
  + 如果变量是指针，先加载其指向的值，然后使用create\_gep获取元素的指针。
* **返回值处理**：
  + 如果if\_return\_lvalue为真，返回变量的地址。
  + 否则，如果访问的是数组，获取数组的起始地址或加载指针指向的值。
* **非数组变量处理**：
  + 如果node.expression为空，表示访问的是普通变量。
  + 根据if\_return\_lvalue标记返回变量的地址或加载其值。

**12.**var = expression**赋值语句**

|  |
| --- |
| *//var = expression 赋值语句*  **void** **CminusfBuilder::visit**(ASTAssignExpression &node) {  LOG(DEBUG) << "AssignExpression\n";  if\_return\_notadd = true; *//表示需要返回值*    *// 获取左值，右值*  node.var->accept(\***this**); *// 处理var*  **auto** left = res; *// left为左侧地址*  node.expression->accept(\***this**); *// 处理右侧expression*  **auto** right = res; *// 获得右侧结果*  *// 处理左值，右值类型冲突问题*  **auto** leftType = left->get\_type()->get\_pointer\_element\_type(); *// 获取var的类型*  */\* 若赋值语句左右类型不匹配，则进行匹配 \*/*  **if** (leftType == TYFLOAT && checkInt(right)){right = builder->create\_sitofp(right, TYFLOAT);LOG(WARNING)<<"赋值语句左右类型不匹配，右侧改为float";}  **if** (leftType == TYINT32 && checkFloat(right)){right = builder->create\_fptosi(right, TYINT32);LOG(WARNING)<<"赋值语句左右类型不匹配，右侧改为int";}  *// 赋值*  builder->create\_store(right, left);  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录赋值表达式节点的处理开始，有助于调试和跟踪编译过程。
* **标记返回值需求**：
  + 设置if\_return\_notadd为true，表示这个赋值表达式需要返回值。
* **处理左侧变量**：
  + 调用node.var->accept(\*this)处理赋值语句左侧的变量，获取其地址，并将其存储在left中。
* **处理右侧表达式**：
  + 调用node.expression->accept(\*this)处理赋值语句右侧的表达式，获取其结果，并将其存储在right中。
* **类型检查与转换**：
  + 获取左侧变量的类型leftType。
  + 如果左侧是浮点类型而右侧是整数类型，则将右侧转换为浮点类型，并记录警告日志。
  + 如果左侧是整数类型而右侧是浮点类型，则将右侧转换为整数类型，并记录警告日志。
* **执行赋值**：
  + 使用builder->create\_store(right, left)将右侧的值存储到左侧变量的地址中，完成赋值操作。

**13.**simple-expression→additive-expression relop additive-expression | additive-expression

|  |
| --- |
| *//simple-expression→additive-expression relop additive-expression | additive-expression*  **void** **CminusfBuilder::visit**(ASTSimpleExpression &node) {  LOG(DEBUG) << "SimpleExpression\n";  node.additive\_expression\_l->accept(\***this**); *// 处理左边的expression*  **auto** lres = res; *// 获取结果存到lres中*  **if** (node.additive\_expression\_r == nullptr)*// 若不存在右expression，则直接返回*  {  **return**;  }  node.additive\_expression\_r->accept(\***this**); *// 处理右边的expression*  **auto** rres = res; *// 结果存到rres中*  *// 需要确保两个表达式的类型相同，*  *//若存在浮点和整型混存，全部转换为浮点型*    **if** (checkInt(lres) && checkInt(rres))*//两边都是整数则进行int型操作*  {  **switch** (node.op)*//根据op构建代码*  {  *//调用icmp进行处理*  *// 比较的返回结果*  *// ast.hh中enum RelOp：*  *// <= 对应 OP\_LE,*  *// < 对应 OP\_LT,*  *// > 对应 OP\_GT,*  *// >= 对应 OP\_GE,*  *// == 对应 OP\_EQ,*  *// != 对应 OP\_NEQ*  **case** OP\_LE:  res = builder->create\_icmp\_le(lres, rres);  **break**;  **case** OP\_LT:  res = builder->create\_icmp\_lt(lres, rres);  **break**;  **case** OP\_GT:  res = builder->create\_icmp\_gt(lres, rres);  **break**;  **case** OP\_GE:  res = builder->create\_icmp\_ge(lres, rres);  **break**;  **case** OP\_EQ:  res = builder->create\_icmp\_eq(lres, rres);  **break**;  **case** OP\_NEQ:  res = builder->create\_icmp\_ne(lres, rres);  **break**;  }  }  **else**  { LOG(WARNING)<<"表达式类型不匹配，强制转换为float";  *// 存在浮点类型，若存在其中一个为int型，需要将其转为浮点数，再进行比较*  **if** (checkInt(lres)) *// 若左边是整数，则将其转为浮点数*  lres = builder->create\_sitofp(lres, TYFLOAT);  **if** (checkInt(rres)) *// 若右边是整数，则将其转为浮点数*  rres = builder->create\_sitofp(rres, TYFLOAT);  **switch** (node.op)  {  */\* 根据不同的比较操作，调用fcmp进行处理 \*/*  **case** OP\_LE:  res = builder->create\_fcmp\_le(lres, rres);  **break**;  **case** OP\_LT:  res = builder->create\_fcmp\_lt(lres, rres);  **break**;  **case** OP\_GT:  res = builder->create\_fcmp\_gt(lres, rres);  **break**;  **case** OP\_GE:  res = builder->create\_fcmp\_ge(lres, rres);  **break**;  **case** OP\_EQ:  res = builder->create\_fcmp\_eq(lres, rres);  **break**;  **case** OP\_NEQ:  res = builder->create\_fcmp\_ne(lres, rres);  **break**;  }  }  res = builder->create\_zext(res, TYINT32); *// 将得到的结果作为整数保存（可作为判断语句中的判断条件）*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录简单表达式节点的处理开始，有助于调试和跟踪编译过程。
* **处理左侧表达式**：
  + 调用node.additive\_expression\_l->accept(\*this)处理左侧的表达式，并将结果存储在lres中。
* **处理右侧表达式（如果存在）**：
  + 检查右侧表达式是否存在，如果存在，调用node.additive\_expression\_r->accept(\*this)处理右侧的表达式，并将结果存储在rres中。
* **类型统一**：
  + 确保两个表达式的类型相同。如果存在类型不匹配，将整型转换为浮点型，以便进行比较。
* **整数类型比较**：
  + 如果两个表达式的结果都是整数类型，根据关系运算符node.op的类型，使用create\_icmp\_\*方法生成相应的整数比较指令。
* **浮点类型比较**：
  + 如果至少有一个表达式的结果不是整数类型，将所有整数转换为浮点型，然后根据关系运算符node.op的类型，使用create\_fcmp\_\*方法生成相应的浮点比较指令。
* **结果扩展**：
  + 最后，使用create\_zext将比较结果扩展为TYINT32类型，以便作为判断语句中的判断条件。

**14.**additive-expression→additive-expression addop term | term

确保加法和减法表达式被正确处理，包括类型检查、类型转换和生成相应的算术指令

|  |
| --- |
| *//additive-expression→additive-expression addop term | term*  **void** **CminusfBuilder::visit**(ASTAdditiveExpression &node) {  LOG(DEBUG) << "AdditiveExpression\n";  **if** (node.additive\_expression == nullptr)  { *// 若无加减法运算*  node.term->accept(\***this**);  **return**; *// 则做乘除法*  }  node.additive\_expression->accept(\***this**); *// 处理左expression*  **auto** lres = res; *// 结果保存在lres中*  node.term->accept(\***this**); *// 处理右term*  **auto** rres = res; *// 结果保存在rres中*  *// 分为整型-整型，和存在浮点数类型，这两种情况进行讨论*  *// 若存在浮点数，则全部强制转换为浮点数实现*  **if** (checkInt(lres) && checkInt(rres))  { *//all int*  **switch** (node.op)  { */\* 根据对应加法或是减法，调用iadd或是isub进行处理 \*/*  **case** OP\_PLUS:  res = builder->create\_iadd(lres, rres);  **break**;  **case** OP\_MINUS:  res = builder->create\_isub(lres, rres);  **break**;  }  }  **else**  { LOG(WARNING)<<"需要强制类型转换为float";  */\* 若有一边是浮点类型，则需要先将另一边也转为浮点数，再进行处理 \*/*  **if** (checkInt(lres)) */\* 若左边是整数，则将其转为浮点数 \*/*  lres = builder->create\_sitofp(lres, TYFLOAT);  **if** (checkInt(rres)) */\* 若右边是整数，则将其转为浮点数 \*/*  rres = builder->create\_sitofp(rres, TYFLOAT);  **switch** (node.op)  { */\* 根据对应加法或是减法，调用fadd或是fsub进行处理 \*/*  **case** OP\_PLUS:  res = builder->create\_fadd(lres, rres);  **break**;  **case** OP\_MINUS:  res = builder->create\_fsub(lres, rres);  **break**;  } |

} }

* **日志记录**：
  + 使用LOG(DEBUG)记录加法表达式节点的处理开始，有助于调试和跟踪编译过程。
* **处理单个项**：
  + 如果没有加法或减法操作（即node.additive\_expression为空），则直接处理term并返回。
* **处理左侧表达式**：
  + 如果存在左侧表达式，调用node.additive\_expression->accept(\*this)处理它，并将结果存储在lres中。
* **处理右侧项**：
  + 调用node.term->accept(\*this)处理右侧项，并将结果存储在rres中。
* **整型操作**：
  + 如果左侧和右侧的结果都是整型，根据操作符（加法或减法），使用create\_iadd或create\_isub生成相应的整数加法或减法指令。
* **浮点操作**：
  + 如果任一侧的结果不是整型（即存在浮点数），则将整型转换为浮点型，并记录警告日志。
  + 根据操作符（加法或减法），使用create\_fadd或create\_fsub生成相应的浮点加法或减法指令。

**15.**term→term mulop factor | factor

确保乘法和除法表达式被正确处理，包括类型检查、类型转换和生成相应的算术指令

|  |
| --- |
| *//term→term mulop factor | factor*  **void** **CminusfBuilder::visit**(ASTTerm &node) {  LOG(DEBUG) << "Term\n";  **if** (node.term == nullptr)  { *//no term, only factor*  node.factor->accept(\***this**);  **return**; *//则处理元素*  }  node.term->accept(\***this**); *// 处理左term*  **auto** lres = res; *// 结果保存在lres中*  node.factor->accept(\***this**); *// 处理右factor*  **auto** rres = res; *// 结果保存在rres中*  **if** (checkInt(lres) && checkInt(rres))  { *//all int*  **switch** (node.op)  { */\* 根据对应乘法或是除法，调用imul或是idiv进行处理 \*/*  **case** OP\_MUL:  res = builder->create\_imul(lres, rres);  **break**;  **case** OP\_DIV:  res = builder->create\_isdiv(lres, rres);  **break**;  }  }  **else**  { LOG(WARNING)<<"需要强制类型转换为float";  *//将int转为float，再进行处理*  **if** (checkInt(lres))  lres = builder->create\_sitofp(lres, TYFLOAT);  **if** (checkInt(rres))  rres = builder->create\_sitofp(rres, TYFLOAT);  **switch** (node.op)  { *// 根据对应乘法或是除法，调用fmul或是fdiv进行处理*  **case** OP\_MUL:  res = builder->create\_fmul(lres, rres);  **break**;  **case** OP\_DIV:  res = builder->create\_fdiv(lres, rres);  **break**;  }  }} |

* **日志记录**：
  + 使用LOG(DEBUG)记录项（Term）节点的处理开始，有助于调试和跟踪编译过程。
* **处理单个因子**：
  + 如果没有左项（node.term为空），直接处理因子（node.factor）并返回。
* **处理左侧项**：
  + 如果存在左项，调用node.term->accept(\*this)处理左侧项，并将结果存储在lres中。
* **处理右侧因子**：
  + 调用node.factor->accept(\*this)处理右侧因子，并将结果存储在rres中。
* **整型操作**：
  + 如果左侧和右侧的结果都是整型，根据操作符（乘法或除法），使用create\_imul或create\_isdiv生成相应的整数乘法或除法指令。
* **浮点操作**：
  + 如果任一侧的结果不是整型（即存在浮点数），则将整型转换为浮点型，并记录警告日志。
  + 根据操作符（乘法或除法），使用create\_fmul或create\_fdiv生成相应的浮点乘法或除法指令。

16.call→ID ( args)

*//call→ID ( args)*

|  |
| --- |
| **void** **CminusfBuilder::visit**(ASTCall &node) {  LOG(DEBUG) << "Call\n";  **auto** function = **static\_cast**<Function \*>(scope.find(node.id)); *// 获取需要调用的函数*  **auto** paramType = function->get\_function\_type()->param\_begin(); *// 获取函数参数类型*  **if** (function == nullptr)LOG(ERROR)<<"函数"<<node.id<<"未定义\n";  *// 处理参数列表*  std::vector<Value \*> args; *// 创建args用于存储函数参数的值，构建调用函数的参数列表*  **for** (**auto** arg : node.args)  { */\* 遍历形参的列表 \*/*  arg->accept(\***this**); *//获取每一个参数对应的值*  **if** (res->get\_type()->is\_pointer\_type())  {  args.push\_back(res); *// 若参数是指针，则直接将值加入到参数列表*  }  **else**  { */\* 若是参数，则需要判断规定的形参与实参的类型是否符合。不符合则需要类型转换 \*/*  **if** (\*paramType == TYFLOAT && checkInt(res)){res = builder->create\_sitofp(res, TYFLOAT);LOG(WARNING)<<"函数参数类型不匹配，要求为float，自动改为float";}  **if** (\*paramType == TYINT32 && checkFloat(res)){res = builder->create\_fptosi(res, TYINT32);LOG(WARNING)<<"函数参数类型不匹配，要求为int，自动改为int";}  args.push\_back(res); *// 将参数的值加入到参数列表*  }  paramType++; */\* 查看下一个形参 \*/*  }  res = builder->create\_call(**static\_cast**<Function \*>(function), args); */\* 创建函数调用 \*/*  } |

* **日志记录**：
  + 使用LOG(DEBUG)记录函数调用节点的处理开始，有助于调试和跟踪编译过程。
* **获取函数**：
  + 通过scope.find(node.id)从当前作用域中获取需要调用的函数对象。
* **检查函数是否存在**：
  + 如果函数对象为空，记录错误信息，表示函数未定义。
* **处理参数列表**：
  + 创建一个args向量，用于存储函数调用的参数值。
  + 遍历node.args中的每个参数，调用arg->accept(\*this)处理每个参数表达式。
* **参数类型检查与转换**：
  + 对于每个参数，检查其类型是否与函数的形参类型匹配。
  + 如果参数是指针类型，则直接添加到参数列表。
  + 如果参数类型不匹配，进行类型转换（整数到浮点或浮点到整数），并记录警告日志。
* **创建函数调用**：
  + 使用builder->create\_call创建函数调用指令，将函数对象和参数列表传递给该指令。

### 降低IR中的冗余

1.经过后续测试，可以针对int型提升精度为float型写一个专门的函数，用来进行优化

2.如果能够在运算位置，设计成一个函数实现，可以进一步简化代码

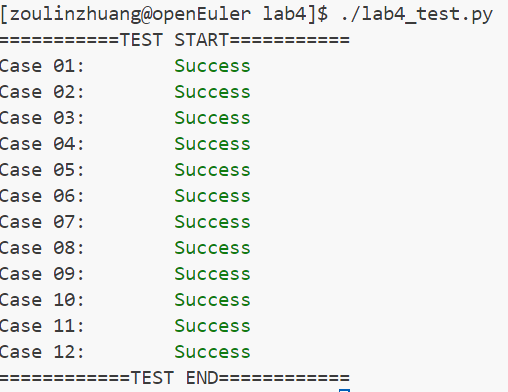
3.可以加入一个全局变量用于记录当前作用域的位置，从而不用再多次访问Function

1.6 实验结果

实验成功实现了 Cminus-F 语言的编译器，能够正确编译合法的 Cminus-F 程序，并生成相应的 LLVM IR 代码。

**性能测试**





1.7 实验难点

实验中遇到的难点包括全局变量的设计、复杂的语义分析、条件和循环语句的处理等。通过合理的设计和问题分解，成功解决了这些难点。

1.8 实验反馈

实验提高了对编译器工作原理的理解，特别是语义分析和 IR 生成的过程。同时，也加深了对 LLVM IR 和编译器优化技术的认识。

1.9 实验总结

1.深入理解了 cminus-f 语言的语义和 LLVM IR 的生成。

2.掌握了访问者模式在抽象语法树算法设计中的应用。

3.学会了使用 Scope 类管理变量作用域。

4.实践了毕昇编译器的循环优化和软件预取技术。

实验总体比较难，比较耗时，并且与lab5性能优化密切相关。在编写代码时，需要认认真真仔细阅读语义规则，往往很多问题就出现在语义规则没有完全符合实现导致的。在编写过程中，还需要阅读各个头文件，以明白各个模块提供的功能，可以找到特别的文件进行内容的书写，例如Value \*数据结构的定义。

姓名：邹林壮 学号：202208040412