编译原理第二次实验报告

组员：王琦琪 胡贤哲 李彦辰

实验任务

本次实验是在第一次实验的基础上增加对程序的语义分析，主要包括：

1. 参考CACT文法和语义规范，设计抽象语法树的节点属性，同时将属性计算代码插入节点遍历代码过程中，从而扩充.g4文件。
2. 通过多次遍历抽象语法树，实现对程序的语义分析，并检查程序是否符合语义规范，并报告错误。
3. 设计自己的IR实现，完善IR的打印输出功能，由于采用LLVM IR，不用实现单独的IR解释器来完成IR的解释执行。

实验项目代码结构

项目调用入口

首先整个项目的代码结构如下表格。

路径/文件	说明
cact-rie/apps/cact-rie.cc	主函数，程序的入口点。
cact-rie/tests/*	测试文件，包含用于验证项目功能的测试代码。
cact-front-end/grammar/CactLexer.g4	词法分析器，负责将输入的源代码分解为词法单元 tokens。
cact-front-end/grammar/CactParser.g4	语法分析器，负责根据语法规则分析词法单元，构建语法树。
cact-front-end/src/*	源文件，包含项目的主要实现代码。
cact-front-end/include/*	头文件，定义了项目中使用的各种接口和声明，包含：
└── **cact-base-visitor.h**	基础访问器类，用于访问抽象语法树。
└── **cact-parser-context.h**	语法分析器上下文，用于存储解析过程中的状态。
└── **cact-expr.h**	与表达式相关的属性和函数。
└── **cact-functions.h**	语义分析中与函数相关的属性。
└── **cact-operator.h**	语义分析中使用的运算符定义。
└── **cact-syntax-error-listener.h**	语法错误监听器，用于处理语法分析中的错误。
└── **cact-type.h**	语义分析中使用的类型定义。
└── **cact-typedef.h**	语义分析中使用的类型别名。
└── **cact-constant-variable.h**	与常量变量相关的属性和函数。
└── **local-var-collector.h**	本地变量收集器，用于收集本地变量。
└── **symbol-registry.h**	符号表类，用于语义分析。
└── **llvm-ir-formatter.h**	LLVM IR 格式化器，用于将 IR 代码格式化为可读的形式。
└── **symbol-registration-visitor.h**	符号注册visitor，one-pass。
└── **const-eval-and-expression-generation.h**	求值处理，one-pass。
└── **ir-generator.h**	中间表示生成的处理，one-pass。

主函数位于cact-rie/apps/cact-rie.cc，是整个项目代码的入口函数。主要功能为：

1. 从源文件中读取 Cact 程序。

2. 使用 ANTLR 进行词法和语法分析。
3. 进行语义检查，确保程序的合法性。
4. 处理常量表达式并生成相应的表达式树。
5. 生成 LLVM IR 代码并输出到文件。

其中main函数代码如下（注：为方便查看部分代码解释来自AI）：

```
int main(int argc, char *argv[]) {
  auto options = configOptions(argv);
  auto args = options.parse(argc, argv);

  // 如果用户请求帮助信息（--help），则打印帮助并退出
  if (args.count("help")) {
    std::cout << options.help() << std::endl;
    return 0;
  }

  // 获取未匹配的参数（非选项参数，通常是源文件）
  const auto &unmatched = args.unmatched();

  // 如果未提供任何源文件，输出错误提示并退出
  if (unmatched.empty()) {
    std::cerr << "Please provide a source file." << std::endl;
    return 1;
  }

  // 如果提供了多个源文件，输出错误提示并退出
  if (unmatched.size() > 1) {
    std::cerr << "Please provide only one source file." << std::endl;
    return 1;
  }

  // 如果没有指定 emit-llvm 选项（即不生成LLVM IR），则直接退出
  if (!args["emit-llvm"].as<bool>())
    return 0;

  // 打印要编译的源文件名
  std::cout << "Compiling " << unmatched[0] << std::endl;
```

```
  // 获取源文件路径
  std::filesystem::path sourceFilePath(unmatched[0]);

  // 检查源文件是否存在，不存在则输出错误提示并退出
  if (!std::filesystem::exists(sourceFilePath)) {
    std::cerr << "Source file does not exist." << std::endl;
    return 1;
  }

  // 获取源文件名（去除扩展名）作为输出文件名基础
  auto srcFileName = sourceFilePath.stem().string();

  // 获取输出路径参数（如果有），否则使用默认名（此处代码不完整，结尾被截断了）
  std::filesystem::path output = args.count("output") ?
args["output"].as<std::string>() : srcFi; // 此行未完成

  // 编译源文件为LLVM IR，并返回编译结果（假设 compileToLLVM 是已有的函数）
  return compileToLLVM(sourceFilePath, output);
}
```

其中compileToLLVM函数是main调用的最重要的函数，负责将 Cact 源代码编译为 LLVM IR。它的主要功能：

```
// compileToLLVM 是将 Cact 源代码编译为 LLVM IR 的核心函数
int compileToLLVM(const std::filesystem::path &file, const
std::filesystem::path &output) {

    // 打开源代码文件（file 是 Cact 源文件路径）
    std::ifstream stream(file);

    // 如果文件打开失败，输出错误信息并返回错误码
    if (!stream) {
        std::cerr << "Failed to open file: " << file << std::endl;
        return 1;
    }

    // 创建语法树指针（后续将用于保存解析结果）
    antlr4::tree::ParseTree *tree;
    // 将输入文件转换为 ANTLR 可处理的输入流
    antlr4::ANTLRInputStream input(stream);
```

```
    // 创建词法分析器（Lexer），用于将输入流分解为词法单元（Token）
    cactfrontend::CactLexer lexer(&input);
    // 构建词法单元流（TokenStream），供语法分析器使用
    antlr4::CommonTokenStream tokens(&lexer);
    // 创建语法分析器（Parser），用于生成语法树
    cactfrontend::CactParser parser(&tokens);
    // 移除默认的词法错误监听器（避免默认错误信息输出）
    lexer.removeErrorListeners();
    // 移除默认的语法错误监听器
    parser.removeErrorListeners();
    // 创建自定义错误监听器，用于收集编译错误信息
    cactfrontend::CactSyntaxErrorListener cact_error_listener(file);
    // 为词法分析器和语法分析器添加自定义错误监听器
    lexer.addErrorListener(&cact_error_listener);
    parser.addErrorListener(&cact_error_listener);
```

接着尝试解析整个文件为语法树。compilationUnit()是我们语法分析树的根节点，是这个上下文无关文法的起始变元。如果存在语法错误，通过自定义的错误监听器进行检查，如果有错误，输出信息并返回1。

```
try {
    // 使用语法分析器解析整份源码文件，返回语法树的根节点
    tree = parser.compilationUnit();

    // 调用自定义错误监听器检查是否有语法错误
    if (cact_error_listener.hasSyntaxError()) {
        std::cerr << "Syntax error(s) found in the source file. Compilation
failed." << std::endl;
        return 1;  // 若有语法错误，终止编译流程
    }
}
catch (const std::exception &ex) {
    // 捕获异常，如语法分析过程中的崩溃、非法输入等
    std::cerr << "Parsing failed: " << ex.what() << std::endl;
    return 1;
}
```

再进行本次实验需要完成的语义分析：

```
// 创建语义检查器，用于符号定义与引用等分析
cactfrontend::SymbolRegistrationErrorCheckVisitor symbolRegistrationVisitor;

try {
    // 访问语法树，执行符号注册及语义检查
    symbolRegistrationVisitor.visit(tree);

    // 若无异常，输出语义检查通过
    std::cout << "Semantic check completed." << std::endl;
}
catch (const std::exception &ex) {
    // 捕获语义错误，例如重复定义、未定义变量等
    std::cerr << "Semantic error(s) found in the source file. Compilation
failed." << std::endl;
    std::cerr << ex.what() << std::endl;
    return 1;
}
```

语义分析流程

经过与组员和学长的多次讨论，我们最终决定沿用与往届学长讨论过后的思路。整个语义分析流程分为三次**pass**。

第一个pass

创建一个符号注册访问器，用于检查语义错误，例如，变量是否声明、类型匹配等。而后调用 visit(tree) 方法进行语义检查。如果存在问题，将抛出异常并输出错误信息。

```
  // check if there is any syntax error
  cactfrontend::SymbolRegistrationErrorCheckVisitor symbolRegistrationVisitor;

  // check if there is any semantic error
  try {
    symbolRegistrationVisitor.visit(tree);
    std::cout << "Semantic check completed." << std::endl;
  }
  catch (const std::exception &ex) {
    std::cerr << "Semantic error(s) found in the source file. Compilation
failed." << std::endl;
```

```
    std::cerr << ex.what() << std::endl;
    return 1;
  }
```

第二个pass和第三个pass

第二个pass，创建一个常量表达式evaluation访问器，传入符号注册的结果。调用visit(tree) 以处理常量表达式并生成表达式树。

第三个pass，创建 LLVM IR 生成器，将解析树转换为 LLVM IR 代码，并输出到指定文件。如果出现错误，捕获异常并输出错误信息。

```
  auto constEvalVisitor =
cactfrontend::ConstEvalVisitor(symbolRegistrationVisitor.registry);

  // evaluate constant expression and generate expression tree in the tree
nodes
  try {
    constEvalVisitor.visit(tree);
    std::cout << "Expression generation completed." << std::endl;
  }
  catch (const std::exception &ex) {
    std::cerr << "Semantic error(s) found in the source file. Compilation
failed." << std::endl;
    std::cerr << ex.what() << std::endl;
    return 1;
  }

  auto srcFileName = file.stem().string();
  auto formattedIRCodeStream = std::ofstream(output);
  auto irFormatter = cactfrontend::LLVMIRFormatter();
  irFormatter.setOutputStream(formattedIRCodeStream);
  auto irCodeStream = std::stringstream();
  auto llvmIRGenerator = cactfrontend::LLVMIRGenerator(irCodeStream,
srcFileName, symbolRegistrationVisitor.registry);
  try {
    llvmIRGenerator.visit(tree);
    irFormatter.format(irCodeStream);
    formattedIRCodeStream.close();
```

```
    std::cout << "IR generation completed." << std::endl;
  }
  catch (const std::exception &ex) {
    std::cerr << "IR generation failed." << std::endl;
    std::cerr << ex.what() << std::endl;
    return 1;
  }
```

至此通过一共3个pass，完成了对 Cact 源代码的语义分析和 LLVM IR 生成。

节点属性

我们在CactParser.g4文件中定义了抽象语法树的节点属性。所有的规则和属性名及其定义如下：

```
ompilationUnit - 编译单元
declaration - 声明
constantDeclaration - 常量声明
dataType - 数据类型
constantDefinition - 常量定义
constantInitialValue - 常量初始值
variableDeclaration - 变量声明
variableDefinition - 变量定义
functionDefinition - 函数定义
functionType - 函数类型
functionParameters - 函数参数
functionParameter - 函数参数定义
block - 代码块
blockItem - 块项
statement - 语句
assignStatement - 赋值语句
expressionStatement - 表达式语句
returnStatement - 返回语句
ifStatement - 条件语句
whileStatement - 循环语句
breakStatement - 跳出语句
continueStatement - 继续语句
```

```
expression - 表达式
constantExpression - 常量表达式
condition - 条件
leftValue - 左值
primaryExpression - 基本表达式
```

下面对一些重要的进行说明

compilationUnit

```
compilationUnit: (declaration | functionDefinition)*;
```

功能：定义一个编译单元，可以包含多个声明或函数定义，表示整个源文件的逻辑结构。常量

constantDeclaration

```
constantDeclaration: Const dataType constantDefinition (Comma
constantDefinition)* Semicolon;
```

功能：常量声明，包括数据类型、常量标识符和初始值，允许多个常量用逗号分隔，最后以分号结束。

constantDefinition

```
constantDefinition
  locals[
    CactBasicType need_type,
    std::shared_ptr<CactConstVar> constant,
    std::vector<std::variant<int32_t, float, double, bool>> value,
  ]: Identifier (LeftBracket IntegerConstant RightBracket)* Equal
constantInitialValue;
```

功能：定义常量，包括常量标识符和初始值，允许多维数组。其中对于locals指令， need_type 表示常

量的预期数据类型，确保在解析和使用时类型匹配，避免错误。 constant 是一个智能指针，指向常量

变量对象，管理其生命周期，确保安全使用并避免内存泄漏。 value 用于存储常量的实际值，支持多种

类型（整数、浮点数、布尔值），允许灵活初始化。

constantInitialValue

```
constantInitialValue
  locals[
    uint32_t current_dim,
    CactType type,
    bool flat_flag,
    std::vector<std::variant<int32_t, float, double, bool>> value,
  ]: constantExpression | LeftBrace (constantInitialValue (Comma
constantInitialValue)*)? RightBrace;
```

功能：常量初始值，可以是常量表达式或者常量数组。具体地，在locals中： current_dim 用于跟踪当前数组维度，以便解析多维数组的初始值。 type 存储常量的类型信息，确保初始值的类型匹配。 flat_flag 指示当前值是否为扁平结构，从而决定如何处理嵌套的初始值。最后， value 用于存储

实际的初始值数据，支持多种类型。变量

variableDeclaration

```
variableDeclaration
  locals[
    CactBasicType need_type,
    std::vector<std::variant<int32_t, float, double, bool>> value,
  ]: dataType variableDefinition (Comma variableDefinition)* Semicolon;
```

功能：变量声明，包括数据类型和变量定义，允许多个变量用逗号分隔，最后以分号结束。

variableDefinition

```
variableDefinition
  locals[
    CactBasicType need_type,
    std::shared_ptr<CactConstVar> variable,
    std::vector<std::variant<int32_t, float, double, bool>> value,
  ]: Identifier (LeftBracket IntegerConstant RightBracket)* (Equal
constantInitialValue)?;
```

功能：定义变量，包括变量标识符和初始值，允许多维数组。

函数

functionDefinition

```
functionDefinition
  locals[
    observer_ptr<Scope> scope,
    observer_ptr<CactFunction> function,
    ]: functionType Identifier LeftParenthesis (functionParameters)?
RightParenthesis block;
```

功能：函数定义，包括函数类型、函数标识符、参数列表和函数体。scope 是一个指向当前作用域的智能指针，用于管理和访问变量和常量的上下文信息，确保在解析过程中能够正确处理作用域内的标识符。function 则是指向当前函数的智能指针，用于管理函数的上下文，确保在解析和执行时能够正确访问函数的属性和状态。这两个指针共同帮助编译器有效地管理作用域和函数信息，确保正确的语义分析和作用域解析。

functionType

```
functionType: Void | Int32 | Float | Double | Bool;
```

功能：定义函数的返回类型，支持多种基本类型。

functionParameters

```
functionParameters: functionParameter (Comma functionParameter)*;

functionParameter
  locals[
    std::shared_ptr<CactConstVar> parameter,
  ]: dataType Identifier (LeftBracket IntegerConstant? RightBracket
(LeftBracket IntegerConstant RightBracket)*)?;
```

功能:定义函数参数的类型和名称，支持可选的数组维度

第一个pass: 符号注册

符号表的结构

符号表是编译器和解释器中至关重要的数据结构，负责管理程序中所有标识符的信息。在一个新的标识符被声明时，其应当被注册到符号表中。符号表是一个作用域树，记录每个作用域中标识符与其信息的映射。

本实验中实现的符号表在symbol-registry.h 文件中。符号表依靠两个结构体实现：Scope和SymbolRegistry。

Scope

Scope结构用于管理符号表中的作用域。以下是对其成员的说明:

- `scopeName` : 存储当前作用域的名称
- `parent` : 指向父作用域的指针
- `variableMap` : 一个映射，存储变量名称到其在 variableVec 中索引的映射关系。
- `variableVec` : 存储当前作用域中注册的变量的向量
- `registerVariable` 在当前作用域中注册一个新变量，如果已经存在同名变量则抛出异常。否则则将变量添加到 variableVec 中，并在 variableMap 中记录其索引。
- `setParent` 用于设置当前域的父作用域
- `getParent` 用于获取当前域的父作用域
- `findVarLocal` 用于检查当前作用域是否包含指定名称的变量
- `getVariable` 用于根据变量名查找变量，若当前作用域没有则会去父作用域查找，直到找到或到达全局作用域。若未找到则抛出异常

SymbolRegistry

SymbolRegistry 用于管理符号表中的作用域和函数。以下是对其成员的说明：

- globalScope: 是指向全局作用域的指针
- scopeID :存储函数名称到其作用域索引的映射
- scopes:一个向量，存储所有作用域的指针
- functionID:存储函数名称到其索引的映射
- functions:一个向量，存储所有函数的指针
- newScope:创建一个新的作用域，并返回该作用域的指针
- createGlobalScope:创建并返回全局作用域的指针
- getGlobalScope:返回全局作用域的指针
- getScopeByFunc:根据名称获取函数的指针，若不存在则返回空指针
- func2String:返回函数的描述字符串，包括返回类型、名称和参数列表
- isGlobal:检查给定变量是否为全局变量

符号注册访问器

符号注册访问器会创建符号表并将声明的变量注册到符号表中，其在symbol-registration-visitor.h中实现。

符号注册访问器的主要作用有：对于函数声明，循环等会产生新作用域的规则，创建不同的作用域；当变量被声明时，将变量注册到当前作用域中；当变量被使用时，在符号表中进行查找，找不到则报错。

在代码中由结构体SymbolRegistrationErrorCheckVisitor来遍历抽象语法树，实现符号注册和语义检查。构造函数会遍历bulit_in_functions来注册内置函数。该结构体方法有：

- `startSemanticCheck` 和 `completeSemanticCheck` 方法用于标记语义检查的开始和结束，并可选择地打印日志信息
- `visitCompilationUnit` 方法处理编译单元，检查 `main` 函数是否存在，且其返回类型和参数符合要求
- `visitConstantDeclaration` 和 `visitVariableDeclaration` 方法分别处理常量和变量的声明，记录其类型并注册
- `visitDataType` 方法根据上下文返回相应的数据类型
- `visitConstantDefinition` 和 `visitVariableDefinition` 方法记录常量和变量的名称、类型，并检查初始化值
- `visitFunctionDefinition` 方法处理函数的定义，检查返回类型和参数

- `enterScope` 和 `leaveScope` 方法用于管理符号的作用域，确保变量和函数在正确的作用域中注册
  以上方法并不是全部，还有很多处理各种语句和表达式的方法，并在其中进行类型检查。这里以方法visitConstantInitialValue为例，该方法用于处理常量初值。

```
std::any visitConstantInitialValue(ConstantInitialValueCtx *ctx) override {
    startSemanticCheck("ConstantInitialValue");
    // if it is a constant expression, check the value's type
    const uint32_t current_dim = ctx->current_dim;
    const CactType &type = ctx->type;
    const uint32_t type_dim = type.dim();
```

开始时调用startSemanticCheck记录当前正在进行的语义检查，便于调试和日志记录。然后从上下文中获取当前维度、常量类型和类型的维度，用于后续的检查。

```
  auto const_expr_ctx = ctx->constantExpression();
  auto const_init_val_ctx = ctx->constantInitialValue();

  if (const_expr_ctx) {
    if (current_dim != type_dim)
      throw std::runtime_error("expression expecting an array, but got a
scalar");
    visit(const_expr_ctx);
    if (type.basic_type != const_expr_ctx->basic_type)
      throw std::runtime_error("a value of type \"" +
                               type2String(const_expr_ctx->basic_type) +
                               "\" cannot be used to initialize an entity of
type \"" +
                               type.toString() + "\"");
  }
```

如果常量表达式存在，首先检查当前维度与类型维度是否匹配，若不匹配则抛出错误。然后调用visit进行进一步语义分析，最后检查量表达式的类型与预期类型是否一致，不一致则抛出错误。

```
  // if it is an array
  else {
    assert(current_dim >= 0);
    if (current_dim >= type_dim)
```

```
      throw std::runtime_error("expression expecting a scalar, but got an
array");

    // count the number of child in constantInitialValue() array
    uint32_t init_val_count = const_init_val_ctx.size();

    // set default attributes of children -- constant initial values
    for (auto &child : const_init_val_ctx) {
      child->current_dim = current_dim + 1;
      child->type = type;
    }

    // check if the initial value could be a flat array
    // it would happen if all children are constant expressions
    ctx->flat_flag = false;
    if (current_dim == 0) {
      ctx->flat_flag = true;
      for (auto &child : const_init_val_ctx) {
        if (!child->constantExpression()) {
          ctx->flat_flag = false;
          break;
        }
      }
    }
```

如果常量表达式不存在，首先检查当前维度是否大于等于类型维度，若是，抛出错误，表明期望的是标量而不是数组。然后获取初始值上下文的大小，遍历每个子节点并设置其当前维度及类型。如果当前维度为0，检查所有子节点是否都是常量表达式。如果是，则标记为平面数组

```
    // case (1): if this initial value is a flat array, reset current_dim of
children and visit them
    if (ctx->flat_flag) {
      for (auto &child : const_init_val_ctx) {
        child->current_dim = type_dim;
      }

      // count the maximum number of elements the flattened array could have
      uint32_t max_count = 1;
      for (uint32_t i = 0; i < type_dim; i++) {
        max_count *= type.array_dims[i];
      }
```

```
      // check if the number of elements is valid
      if (init_val_count > max_count)
        throw std::runtime_error("too many initializer values");
    }
```

如果是平面数组，重置子节点的维度，并计算最大初始值数量。如果初始值数量超过限制，抛出错误。

```
    // case (2): count result is no more than array_dims[current_dim]
    else if (0 <= current_dim && current_dim < type_dim) { // normal case
      if (init_val_count > type.array_dims[current_dim])
        throw std::runtime_error("too many initializer values");
    }
```

检查当前维度下的初始值数量是否超过数组的维度限制，如超过则抛出错误

```
    // visit all children
    for (auto &child : const_init_val_ctx) {
      visit(child);
    }

  }

  completeSemanticCheck("ConstantInitialValue");
  return {};
  }
```

递归地对子节点调用visit进行检查。最后调用 completeSemanticCheck ，记录检查完成的状态。

还有一些处理其他语句和表达式的方法，就不一一列举了

第二个pass：类型检查常量求值

表达式树

在语义分析中，要为常量表达式构建表达式树表达式树是由表达式节点组成的树，每个节点表示一个操作或一个值。

实现

本实验中实现的表达式树在const-eval-and-expression-generation.h文件中，主要由结构体ConstEvalVisitor进行解析期间解析期间进行常量表达式的初始话和求值。

访问器的构造函数接受一个指向符号注册表的共享指针，以管理变量和常量的作用域。主要方法有：

- `visitConstantDefinition` 方法用于处理常量初始化
- `visitConstantInitialValue` 方法用于访问常量表达式或常量列表初始值
- `visitVariableDefinition` 方法处理变量初始化
- `visitAssignStatement` 、 `visitReturnStatement` 、 `visitIfStatement` 和 `visitWhileStatement` 方法分别处理赋值、返回、条件和循环语句，获取表达式并设置相应的上下文
- `visitExpression` 、 `visitConstantExpression` 方法处理算术表达式
- `visitLeftValue` 方法用于处理左值
- `visitNumber` 方法处理数字类型
- `visitUnaryExpression` 、 `visitMulExpression` 、 `visitAddExpression` 、 `visitRelationalExpression` 等方法处理各种运算符，根据上下文生成相应的表达式树。

ConstEvalVisitor的各个方法使其能够处理常量和变量的定义与初始化、解析表达式并生成相应的表达式树、处理控制流语句（如赋值、返回、条件和循环。以方法visitConstantInitialValue为例，它负责理常量的初始值。

```
std::any visitConstantInitialValue(ConstantInitialValueCtx *ctx) override {
    // If it is a constant expression, return the value's type by visiting
constant expression and return the result
    if (ctx->constantExpression()) {
      std::vector<ConstEvalResult> return_vec;
      auto res = getConstExpr(ctx->constantExpression());
      return_vec.emplace_back(res);
      return return_vec;
    }
```

首先检查ctx中是否存在常量表达式。如果存在，则获取常量表达式的只，存入向量中并返回向量。

```
    // If it is a list of constant values, return a vector of constant values.
    // The return value is a vector of constant values with the same type.
    // The width of vector is (size(current_dim) / size(basic_type))
    else {
      std::vector<ConstEvalResult> values;

      uint32_t width = ctx->type.width(ctx->current_dim); // width of the
current dimension
      assert(width != 0);
      uint32_t size_of_basic_type = sizeOf(ctx->type.basic_type); // size of
the basic type
      uint32_t child_num = ctx->constantInitialValue().size(); // number of
children in the list
      uint32_t expected_child_num; // expected number of children in the list
      uint32_t child_width; // width of the children
```

如果ctx中没有常量表达式，先初始化values向量用于存储常量，接着在确保当前维度的宽度不为0后获取基本类型的大小，计算常量初始化值列表中的子项数量。

```
      // if flat_flag is set, the values are stored in a flat array at dim-0
      if (ctx->flat_flag) {
        assert(ctx->current_dim == 0);
        expected_child_num = ctx->type.size() / size_of_basic_type;
        child_width = 1u;
      }
      else {
       expected_child_num = width;
       child_width = ctx->type.size(ctx->current_dim + 1) /
size_of_basic_type;
      }
```

如果是扁平数组，则在确保当前维度为 0后计算预期子项数量和宽度；否则计算当前维度的预期子项数量和宽度。

```
      assert(child_num <= expected_child_num); // the number of children is no
more than expected

      for (auto &child : ctx->constantInitialValue()) {
        auto value = getConstInitVal(child);
```

```
          values.insert(values.end(), value.begin(), value.end());
        }

        assert(values.size() == child_width * child_num); // check if the number
of values inputed is correct

        // pad the values with 0 if the number of values is less than expected
        values.resize(child_width * expected_child_num, 0);

        return values;
      }
    }
```

确保实际子项数量不超过预期数量后，遍历常量初始化值并放入向量values中。检查values的大小是否与预期的相等，如偏小则填充0。最后返回包含着常量初始化值的values。

第三个pass：IR生成

IR的功能

IR是一种应用于抽象机器的编程语言，主要用于帮助分析计算机程序。编译器在转换源代码时，通常先将代码转换为一个或多个中间表述，以便进行最佳化并生成机器语言。我们的实验中采用的IR是LLVM IR。

解析

本实验中实现的IR在ir-generator.h 文件中

首先实现一个用于变量重命名的工具LocalIdentifierMangler

```
struct LocalIdentifierMangler {
  std::unordered_map<observer_ptr<Scope>, size_t> scopeIDMap{};
  std::string rename(const std::shared_ptr<CactConstVar>& var) {
    const auto &name = var->name;
    auto scope = var->scope;
    if (!scopeIDMap.contains(scope))
      scopeIDMap[scope] = scopeIDMap.size();

    size_t scopeID = scopeIDMap[scope];
```

```
    return name + ".b." + std::to_string(scopeID);
  }
  void clear() {
    scopeIDMap.clear();
  }
};
```

主要目的是为局部变量生成唯一的名字，避免作用域冲突。它内部维护了一个 scopeIDMap，这是一个以作用域指针为键、唯一编号为值的哈希表。这样可以为每个作用域分配一个唯一的 ID。

```
struct LLVMIRGenerator final : public CactBaseVisitor
```

LLVMIRGenerator 是一个继承自 CactBaseVisitor 的最终类（final 关键字表明不能被进一步继承）。
它的主要功能是以 LLVM IR 的形式生成中间代码。同时,它会遍历语法树（CactParser::...Context），将抽
象语法树（AST）中的节点转换为对应的 LLVM IR。

接下来对这个类中的一些重要方法进行一些解析

```
  std::any visitCompilationUnit(CactParser::CompilationUnitContext *ctx)
override {
    irCodeStream << std::format("; ModuleID = '{}'\nsource_filename = \"
{}.cact\"\n", moduleName, moduleName);
    declareExternalFunctions();
    for (auto &child : ctx->children)
      visit(child);
    return {};
  }
```

它负责处理编译单元（即整个源文件）的节点。它调用 `declareExternalFunctions()`，用于声明所有外部函数（如标准库函数），确保后续 IR 代码能正确引用这些函数。然后，方法遍历整个编译单元，`ctx->children`，对每个子节点递归调用 `visit(child)`，这样可以依次处理源文件中的所有声明和语句，生成对应的 IR 代码。

```
  std::string shortCircuitConditionalJumpIRGen(const std::string &labelPrefix,
const std::shared_ptr<CactExpr>& cond, const std::string &thenBranchLabel,
const std::string& elseBranchLabel);

  std::string whileStatementIRGen(const std::string &labelPrefix,
                                  const std::shared_ptr<CactExpr> &cond,
                                  CactParser::StatementContext *loopBodyCtx);

  std::string statementIRGen(const std::string &labelPrefix,
CactParser::StatementContext *ctx);

  std::any visitFunctionDefinition(CactParser::FunctionDefinitionContext *ctx)
override;

  std::any visitDeclaration(CactParser::DeclarationContext *ctx) override;
```

这几行代码声明了 `LLVMIRGenerator` 类中与 IR 代码生成相关的几个核心成员函数。

- `shortCircuitConditionalJumpIRGen` 用于生成带有短路求值的条件跳转 IR 代码。它接收一个标签前缀、一个条件表达式指针、以及 then/else 分支的标签名，通常用于实现逻辑与（&&）或逻辑或（||）等短路条件判断。
- `whileStatementIRGen` 用于生成 while 循环的 IR 代码。它需要一个标签前缀、循环条件表达式指针和循环体的语法树上下文指针。该方法会负责生成循环的条件判断、跳转和循环体相关的 IR 代码。
- `statementIRGen` 用于生成单条语句的 IR 代码。它接收标签前缀和语句上下文指针，适用于通用语句的 IR 生成。
- `visitFunctionDefinition` 是访问函数定义节点的方法，重写自基类。它会处理函数的声明、参数、局部变量和函数体，生成完整的函数 IR。
- `visitDeclaration` 是访问变量声明节点的方法，同样是重写自基类。它负责处理变量或常量的声明，并生成相应的 IR 代码。

这些方法共同构成了将语法树各部分转换为 LLVM IR 代码的主要接口，是编译器后端代码生成阶段的关键组成部分。

```
  void allocateVariable(const std::shared_ptr<CactConstVar> &var, const
std::string &newName);;
```

`allocateVariable` 用于为变量分配存储空间。`EvaluationCodegenResult` 包含两个字符串字段：`code`（生成的 IR 代码片段）和 `result`（计算结果），便于表达式求值时同时返回代码和结果。

```
  EvaluationCodegenResult fetchAddressCodeGen(const std::shared_ptr<CactExpr>
&expr);
  EvaluationCodegenResult evaluationCodeGen(const std::shared_ptr<CactExpr>
&expr);
  EvaluationCodegenResult arithmeticBinaryOpCodeGen(const
std::shared_ptr<CactExpr> &expr);
  EvaluationCodegenResult predicateBinaryOpCodeGen(const
std::shared_ptr<CactExpr> &expr);
  EvaluationCodegenResult unaryOpCodegen(const std::shared_ptr<CactExpr>
&unary);
  EvaluationCodegenResult functionCallCodegen(const std::shared_ptr<CactExpr>
&expr);
```

用于生成不同类型表达式的 IR 代码。这些函数都返回 `EvaluationCodegenResult`，以便在代码生成过程中灵活组合。

```
  static std::string basicTypeString(const CactBasicType &type) {
    static std::map<CactBasicType, std::string> typeMap = {
        {CactBasicType::Int32, "i32"},
        {CactBasicType::Float, "float"},
        {CactBasicType::Double, "double"},
        {CactBasicType::Bool, "i1"},
        {CactBasicType::Void, "void"}
    };
    return typeMap.at(type);
  }
```

`basicTypeString` 是一个静态函数，用于将内部的基本类型（如 `Int32`、`Float` 等）转换为 LLVM IR 所需的类型字符串（如 `"i32"`、`"float"` 等），方便类型映射。

```
    std::optional<CactParser::StatementContext *>
  reduceStatement(CactParser::StatementContext *ctx);

    static std::optional<CactParser::StatementContext *>
  reduceWhileLoop(CactParser::WhileStatementContext *ctx);

    std::optional<CactParser::StatementContext *>
  reduceIfBranch(CactParser::IfStatementContext *ctx);
```

`reduceStatement`、`reduceWhileLoop` 和 `reduceIfBranch` 用于语法树的简化，尝试将循环或条件分支简化为更简单的形式。

```
    std::string rename(std::shared_ptr<CactConstVar> var) {
      return localIdentifierMangler.rename(var);
    }
    std::string addressOf(std::shared_ptr<CactConstVar> var);
    std::string temporaryName(int id) {
      return std::to_string(id);
    }

    std::string assignReg() {
      return "%" + temporaryName(temporaryID++);
    }
```

`rename` 方法通过 `LocalIdentifierMangler` 生成变量的唯一名，防止作用域冲突。`addressOf` 用于获取变量的地址名。`temporaryName` 和 `assignReg` 用于生成临时寄存器名，`assignReg` 会自增 `temporaryID`，确保每个寄存器名唯一。

此外，还有一系列用于生成特定 IR 代码的辅助函数，这些成员共同构成了 IR 代码生成器的核心功能。

任务分工与总结

王琦琪，李彦辰： 负责前两个pass的工作，包括语义分析、符号注册和表达式求值等工作，分别负责代码框架设计和代码撰写，以及对应报告部分的撰写。

胡贤哲：负责第三个pass IR生成的工作，包括IR中间代码生成，负责代码设计和编写，以及对应实验报告的撰写。