中国科学院大学《编译原理》
CACT 词法语法分析实验报告

小组编号 15 小组成员姓名 李紫嫣 梁婧琪 邹为 实验编号 1

实验名称 PR002-CACT 日期 2025.5.29

1 实验要求

完成中间代码的生成,中间代码可以考虑生成 LLVM IR (可以通过 llc 编译,lli 解释执行),如果为自己设计的 IR,则需要在代码中加入一个 IR 的解释器。

评测标准

运行输出结果与 gcc 11.4 编译运行结果逐字符比对相同

2 实验设计与思路

2.1 整体框架设计与修改

基于 LLVM 的 CACT 语言编译器,采用 ANTLR 进行词法和语法分析,通过 visit 模式实现语义分析和 IR 生成。

主要组件包括: ANTLR 生成的 CACTLexer/CACTParser 作为词法/语法分析器; 构建一个用于管理变量/函数声明和作用域的符号表系统,同时可以打印调试信息;类型系统用于处各种理基本类型和复杂数组类型;使用 LLVM API 生成中间代码,并输出为文件存储;集成语法和语义错误收集,并输出报错信息。

工作流程

源文件 –> ANTLR 词法分析 –> 生成 Token 流 –> ANTLR 语法分析 –> 构建语法树 –> 遍历语法树 –> 符号表管理 –> 类型检查–> IR 生成 –> 作用域处理 –> 类型推断 –> LLVM 指令生成

错误收集与结果生成

有错误打印错误信息,否则控制台打印最外层符号表和 IR,同时以文件输出 IR。

2.2 适用于 LLVM IR 的符号表设计

基本结构:SymbolInfo 结构体

```
struct SymbolInfo {
       Kind kind;
       std::string type;
       llvm::Value* value;
       std::vector<int> dims;
   };
```

kind,区分一个符号是变量还是函数;type,存储变量的数据类型和函数的函数签名;value 存储对应的 LLVM 值指针;dims 对变量存储其数组维度信息。符号表类中提供了符号添加方法、查询方法和更新方法。

2.3 访问器主要内容(CACTVisitorImpl 类)

2.3.1 成员变量介绍

2.3.2 常量合法性验证辅助函数

基础字面量常量检测(isConstLiteralExp 函数):

表 1: 成员变量及其作用

变量	作用
scopeStack	作用域栈(支持嵌套作用域)
context	LLVM 上下文
builder	IR 构建器
functionMap	函数映射表

- 进行空值检查,检查上下文节点和 addExp 子节点是否存在
- 验证 parser tree 的结构严格满足 ConstExp->AddExp->MulExp->UnaryExp->PrimaryExp->Number
- 确认最底层的 primaryExp 是数字节点

基础常量初始化验证:

- 如果是空节点,返回 false
- 如果是空列表,则是合法空初始化,返回 true
- 如果是一个基础字面量常量,则去调用 isConstLiteralExp 检查是否是合法字面量常量
- 如果是嵌套列表递归检查每一个子元素是否合法

判断是否是数组的平铺初始化:

- 递归结构中,如果当前是 constExp,就是基本情况(一个值),返回 true
- 如果当前是初始化列表(如 1, 2, 3, 4),那么每个子项都必须是 constExp,不能嵌套
- 一旦出现嵌套,返回 false

2.3.3 递归处理多维数组初始化

递归函数 processArrayInitVal 用于多维常量数组的初始化处理,将一个多维数组的初始化表达式展开成一个一维的 LLVM IR 常量列表。对于嵌套数组结构的处理过程如下。对于外层的数组结构,检查数组的实际维度是否满足维度要求,如果满足,则对每个子项递归处理,进入下一维度。

```
size_t expectedElements = dims[currentDim];
size_t actualElements = ctx->constInitVal().size();

if (actualElements > expectedElements) {
    std::cerr << "[Error] Too many initializers at dimension " << currentDim << std::endl;
    return false;
}
for (auto child : ctx->constInitVal()) {
    // 递归调用, 进入下一维度
    if (!processArrayInitVal(child, baseType, dims, elements, currentDim + 1)) {
        //std::cerr << "[Error] Initialization failed at dimension " << currentDim + 1 << std::endl;
        return false;
    }
}
```

在若干次递归处理之后，进入到最内层维度。如果当前是最内层维度，但 ctx 不是标量，允许递归遍历该维度所有子节点。如果 ctx->constExp() 不为空，说明当前是一个标量表达式，就使用 visitConstExp() 访问这个表达式，生成 LLVM 常量值，加入 elements 中。

```
// 如果当前是最内层维度，但ctx不是标量，允许递归遍历该维度所有子节点（因为最内层可能多层花括号嵌套）
      if (currentDim == dims.size() - 1) {
          for (auto child : ctx->constInitVal()) {
              if (!processArrayInitVal(child, baseType, dims, elements, currentDim)) {
                  std::cerr << "[Error] Initialization failed at innermost dimension " << currentDim
                      << std::endl;
                  return false;
              }
          }
          return true;
      }
```

对于平铺式初始化的特殊情况，如果当前维度是最外层，(即 currentDim == 0)，并且所有元素都是常量表达式，不含嵌套括号，那么使用上面所述 isFlatInitForArray 辅助函数判断平铺式初始化的合法性。如果合法，遍历所有子元素，将每个 constExp 结果转为 llvm::Constant* 并加入到 elements 中。

2.3.4 递归构造 LLVM IR 中的多维数组常量(用到前一个函数构造出的一维常量列表)

createArrayConstant 函数负责构造 LLVM IR 中的多维数组常量(ConstantArray)，通过递归方式将一维元素 elements 填入多维数组的 LLVM ConstantArray 中。

对于非最内层的维度，先对当前层计算当前这一层中的每个元素由多少个扁平元素组成。然后对于当前层的每一个索引(dims[currentDim] 个)，递归调用自己，生成 dims[currentDim] 个子数组，并在调用后增加索引。

```
size_t currentIndex = startIndex;
for (int i = 0; i < dims[currentDim]; ++i) {
    Constant* subArray = createArrayConstant(baseType, dims, elements,
                                    currentIndex, currentDim + 1);
    if (!subArray) {
        return nullptr;
    }
    subArrays.push_back(subArray);
    currentIndex += elementsPerSubArray;
}
```

最后，创建当前层的 LLVM 数组类型并构造 ConstantArray。函数最终返回一个 ConstantArray*，其每个元素是一个子数组 ConstantArray*，从而在递归中层层嵌套，形成多维常量结构。

如果循环到了最内层维度(currentDim == dims.size() - 1)，创建一个一维数组 ConstantArray，从 startIndex 开始取 dims[currentDim] 个元素，组装出一个对应类型的常量数组。

```
currentElements.insert(currentElements.end(),
    elements.begin() + startIndex,
    elements.begin() + endIndex);
ArrayType::get(getLLVMType(baseType), dims[currentDim])
```

2.3.5 函数定义节点处理

visitFuncDef 函数用于访问和处理函数定义节点 FuncDefContext。此函数主要实现以下功能：解析函数返回类型和参数类型(为了支持数组，需要记录下数组维度)；使用 LLVM API 创建函数签名及函数对象；管理符号表和作用域；在 LLVM IR 中生成函数入口块和参数分配；访问函数体代码块；函数结束时清理作用域和验证函数。

函数通过语法树节点获取函数返回类型字符串，将返回类型压入栈，方便后续访问返回类型信息，然后利用 getLLVMType 函数将字符串类型转换为 LLVM Type*，并且记录数组的维度。

对于函数参数，函数遍历所有函数参数，取它的类型名，通过“[”字符计数统计参数文本中数组维度数量以及显式维度大小。如果是数组参数，从最内层维度开始向外层构造多维数组类型，维度信息倒序后调整为从外到内，最后将数组类型退化为指针类型。在函数入口块、符号表等处理结束之后，遍历函数参数并存入符号表。

2.3.6 常量声明节点处理

visitConstDecl 函数将解析出的常量声明转换成 LLVM IR 中的全局常量定义，并维护符号表。函数从语法树上下文中获取常量声明的基础类型，保存在 baseType。由于一个 const 声明可能包含多个常量定义 constDef，所以要遍历所有常量定义，取出常量名。对于数组常量，通过 IntConst() 获取这些维度的具体大小，存入 dims 向量。

调用 getLLVMType，根据基础类型和维度信息构造对应的 LLVM 类型，保存在 varType。接下来，对标量常量和数组常量分情况初始化。如果 dims 为空，认为这是一个标量常量。通过初始化值是否是一个常量表达式 constExp() 来确认其合法性。若合法，就根据基础类型将初始化表达式字符串转为对应的 LLVM Constant 类型。

```
Constant* initVal = nullptr;
    if (dims.empty()) {
        // 标量常量
        if (!def->constInitVal()->constExp()) {
            errors.push_back("Invalid scalar const initialization: " + name);
            continue;
        }
        auto constExp = def->constInitVal()->constExp();
        if (baseType == "int") {
            initVal = ConstantInt::get(context, APInt(32, std::stoi(constExp->getText())));
        } else if (baseType == "float") {
            initVal = ConstantFP::get(context, APFloat(std::stof(constExp->getText())));
        }
    }
```

如果是数组常量，就调用辅助函数 processArrayInitVal 递归处理数组初始化值，生成所有元素的 LLVM 常量集合 elements，此函数已经介绍过。将 elements 提供给 createArrayConstant 函数，生成多维数组的 LLVM Constant 表示。

```
    else {
        // 数组常量
        std::vector<Constant*> elements;
        if (!processArrayInitVal(def->constInitVal(), baseType, dims, elements)) {
            errors.push_back("Invalid array const initialization: " + name);
```

```
            continue;
        }

        // 创建多维数组常量
        initVal = createArrayConstant(baseType, dims, elements);
    }
```

最后,创建全局常量变量并添加到符号表。此时要处理重定义错误,如果符号表添加失败,说明重复,报错。

2.3.7 变量声明节点处理

我们重写语义分析器中的 visitVarDecl 实现变量声明节点处理功能。此代码功能包括: 对变量进行类型解析、维度分析、初始化检查,并根据变量作用域生成 LLVM IR。

2.3.8 语句处理

visitStmt 函数用于处理赋值语句、返回语句。

处理赋值语句时,赋值语句的特征是 ctx->lVal() && ctx->exp(),即有一个左值 lVal 和一个右值 exp。函数获取左值的变量名 name,遍历作用域栈,从内到外查找 name,找到该变量的符号信息 info。若变量未声明或未分配 LLVM 值则报错。对于右值表达式,调用 visit(ctx->exp()) 访问右值表达式 AST,返回一个 std::any,然后转换为 Value*。

```
    const SymbolTable::SymbolInfo* info = nullptr;
    for (auto it = scopeStack.rbegin(); it != scopeStack.rend(); ++it) {
        if (it->contains(name)) {
            info = it->lookup(name);
            break;
        }
    }
```

特别地,如果是一个数组变量,则不能直接用 store 来赋值。此时需要先得到左值数组元素的实际类型,然后比较确认左右值类型是否一致。如果没有问题,就调用辅助函数生成 memcpy 指令,把右值的内存内容复制到左值内存区域。如果不是数组,就直接生成 store 指令,把右值存储到左值地址。

如果是返回语句,当前不在函数中则报错,如果在函数中,要获取函数返回值类型并检查类型是否匹配,如果合法,生成 ret 指令,返回表达式的结果值。

2.3.9 一元表达式

visitUnaryExp 处理 UnaryExp 类型的语法节点。

如果 UnaryExp 中有 Ident,说明这个一元表达式是一个函数调用。那么提取函数名,在作用域栈中查找函数信息。如果没有找到,报错。对于每个函数参数,调用 visit,获取对应的 Value*,如果参数类型不对或者数量不对,也需要报错。最后构造一个 call 指令并返回函数调用的结果。

对于普通一元表达式,则交给子节点处理。

2.4 主程序

主程序控制读取 cact 文件，通过 ANTLR 生成语法树，调用语义分析与 LLVM IR 生成，将错误收集到错误列表中，若有错误则报错，没有错误则将 IR 保存到文件。

2.5 辅助与测试脚本

```
# 生成所有样例的中间代码
./cmptest.sh

# 手动编译连接
# libcact/libcact.a
llc llvm_ir/file.ll -filetype=obj -o o/file.o
clang++ o/file.o libcact/libcact.a -o out/file
# 或，项目根目录下执行
./genrun.sh

# 测试运行输出
./run.sh

# 使用clang或gcc转为C对象验证结果
./clangtest.sh
./gccrun.sh
```

2.5.1 静态库函数

使用到的静态库链接是使用 c++ 编写后生成的

```
#include <iostream>
extern "C" {

void print_int(int x) {
std::cout << x << std::endl;
}
void print_float(float x) {
    std::cout << x << std::endl;
}
void print_char(char c) {
    std::cout << c << std::endl;
}
int get_int() {
    int x;
    std::cin >> x;
    return x;
}
float get_float() {
    float f;
    std::cin >> f;
```

```
    return f;
}
char get_char() {
    char c;
    std::cin >> c;
    return c;
}
void print_bool(bool b) {
    std::cout << (b ? "true" : "false") << std::endl;
}

}
```

2.5.2 对比验证

Clang

```
# 使用 clang 将 .cact 转换为 LLVM IR
if clang -x c -S -emit-llvm "$src" -o "$ir_file" 2>/dev/null; then
  echo ">>> IR: $ir_file"
else
  echo ">>> Failed IR : $src"
  echo
  continue
fi

# 使用 llc 生成 .o
if llc "$ir_file" -filetype=obj -o "$obj_file"; then
  echo ">>> llc obj: $obj_file"
else
  echo ">>> Failed llc obj: $ir_file"
  echo
  continue
fi

# 使用 clang++ 进行链接
if clang++ "$obj_file" "$LIB_A" -o "$exe_file"; then
  echo ">>> link $exe_file"
  echo "Success: $src"
else
  echo ">>> Failed link $obj_file"
  echo
  continue
fi
echo
```

gcc

```
#!/usr/bin/env bash
set -u
```

```
# 1. 根目录，脚本假设放在 /cact 下执行
ROOT_DIR="$(cd "$(dirname "$0")" && pwd)"
SRC_DIR="$ROOT_DIR/test/samples_lex_and_syntax"
LIB="$ROOT_DIR/libcact/libcact.a"
OUT_DIR="$ROOT_DIR/out_exec"

mkdir -p "$OUT_DIR"

echo "输出目录：$OUT_DIR"
echo "使用静态库：$LIB"
echo

# 2. 遍历所有 .cact 源文件
for src in "$SRC_DIR"/*.cact; do
  # 如果目录里没有 .cact 文件，glob 会展开成字面 "*.cact"
  [[ -e "$src" ]] || { echo "没有找到任何 .cact 文件，退出."; exit 0; }

  base="$(basename "$src" .cact)"
  obj="$ROOT_DIR/$base.o"
  exe="$OUT_DIR/$base"

  echo "=== 处理 $src ==="

  # 3. 编译成目标文件
  gcc -x c -c "$src" -o "$obj"
  if [[ $? -ne 0 ]]; then
    echo " [跳过] 编译失败：$src"
    rm -f "$obj"
    echo
    continue
  fi

  # 4. 链接生成可执行文件
  g++ "$obj" "$LIB" -o "$exe"
  if [[ $? -ne 0 ]]; then
    echo " [跳过] 链接失败：$base.o"
    rm -f "$obj"
    echo
    continue
  fi

  # 5. 清理并报告
  rm -f "$obj"
  echo " [成功] 生成：$exe"
  echo
done

echo "全部处理完毕。可执行文件在 $OUT_DIR 下。"
```