编译原理大作业：C 到 LLVM 的编译器

小组成员：陈启乾、潘首安、谭弈凡

一、使用说明

1. 环境配置

- 系统：Windows 10 & Ubuntu 20.04
- 语言：Python 3.9
- 安装 ANTLR4: ANTLR4 Installation
- 安装 Python 依赖库: `pip install -r requirements.txt`
- （在 Ubuntu 上面）安装 LLVM： `sudo apt install clang-format clang-tidy clang-tools clang clangd libc++-dev libc++1 libc++abi-dev libc++abi1 libclang-dev libclang1 liblldb-11-dev libllvm-ocaml-dev libomp-dev libomp5 lld lldb llvm-dev llvm-runtime llvm python-clang`

2. 编译与运行

1. 由 ANTLR 生成代码文件: `make parser`[1][2]
2. 运行程序: `python main.py <inputfilename> <outputfilename>`
3. 运行 LLVM IR：`lli [targetfile.ll]`（交互式解释器）或 `llc [targetfile.ll]`（编译成汇编，接下来用 clang 或 gcc 编译为机器码；目前编译器指定的架构 Triple 为 `x86_64-pc-linux`，可在代码中修改）

3. 测试

在 `test` 文件夹下有许多测试样例，`test.py` 可以编译所有的这些样例，从而检测程序是否正确。

4. 支持的语法简略

- 变量
  - 局部变量、全局变量；可以初始化也可以不初始化
  - 类型：包括 `int`, `long long`, `double`, `bool` 等变量类型
  - 支持了数组和字符串（包括常量字符串）
- 表达式：
  - 支持了绝大多数的算数预算、逻辑运算
  - 支持数组按下标读取(`a[i]`)，和对左值取地址 `&a[j]`
- 语句
  - 分支：`if-else`, `switch`
  - 循环：`while`, `do while`, `for`(包括 `continue` 和 `break`)
- 函数
  - 定义：支持不同类型参数、返回值；支持递归
  - 声明：支持可变参数、指针参数，可以通过声明的方式引入外部函数（如 `scanf` 和 `printf`）

4. 完成样例情况

实现了样例1（回文检测），样例2（归并排序），样例3（KMP算法）

二、文件结构

```
.
├── Makefile
├── README.md
├── doc 【文档】
│   └── doc.md
├── grammar 【语法文件】
│   ├── cpp20Lexer.g4            【词法文件】
│   └── cpp20Parser.g4           【句法文件】
├── main.py                      【主程序】
├── requirements.txt             【python 依赖包】
├── sample 【样例程序】
│   ├── sample1_palindrome.cpp   【回文数程序】
│   ├── sample1_palindrome.ll    【编译出的 LLVM IR】
│   ├── sample2_mergesort.cpp    【归并排序】
│   ├── sample2_mergesort.ll     【编译出的 LLVM IR】
│   ├── sample3_KMP.cpp          【KMP程序】
│   └── sample3_KMP.ll           【编译出的 LLVM IR】
├── src 【ANTLR 自动生成的部分】
│   ├── cpp20Lexer.interp
│   ├── cpp20Lexer.py
│   ├── cpp20Lexer.tokens
│   ├── cpp20Parser.interp
│   ├── cpp20Parser.py
│   ├── cpp20Parser.tokens
│   ├── cpp20ParserListener.py
│   └── cpp20ParserVisitor.py
├── tables.py                    【符号表】
├── test【一些简单的测试程序】
│   ├── address.cpp
│   ├── branch.cpp
│   ├── empty.cpp
│   ├── func.cpp
│   ├── func_recursion.cpp
│   ├── globalvar.cpp
│   ├── loop.cpp
│   ├── main.cpp
│   ├── printf.cpp
│   ├── str.cpp
│   ├── va_args.cpp
│   └── var.cpp
└── test.py                      【自动化测试】
```

- main.py 是程序的主要代码，table.py 是符号表代码
- src 文件夹是 antlr 自动生成的编译器 python 代码
- grammar 文件夹存储了语法文件
- doc 文件夹存储了程序文档
- test 文件夹存储了测试样例（test.py可以自动编译他们）
- sample 文件夹存储了示例程序

三、实现语法及原理

1. 变量与数组

1.1 符号表

我们在 `tables.py` 中定义了符号属性类 `NameProperty` 和符号表类 `NameTable` 。`NameProperty` 类定义了一个符号的所有属性，包括类型 `type`，值/地址 `value`，是否为有符号数 `signed`。`NameTable` 类的实例保存程序中声明所有符号（包括变量名，数组名与函数名）。其中定义了列表 `table` 和当前作用域深度 `currunt_scope_level` ，以栈的形式逐层保存符号。

符号表的使用：当程序进入一段新的作用域时，调用符号表的类函数 `enterScope`，将当前作用域深度+1，同时在列表 `table` 中加入一段新的词典。如果有临时变量/数组符号被声明，那么将该符号的相关属性添加至 `table` 列表的最后一段词典中；如果有全局变量/函数被声明，那么将该符号的相关属性添加至 `table` 列表的第一段词典中。当变量离开一段作用域时，调用符号表的类函数 `exitScope`，将当前作用域-1，同时弹出 `table` 列表的最后一段词典，释放这个作用域中的所有临时变量。

符号的调用：这部分定义在类函数 `getProperty` 中。在程序中识别到一标识符后，在 `table` 列表中从后向前搜索各个词典，返回第一个命中符号的 `NameProperty`。如果没有找到该标识符，则报错。

1.2 变量的初始化与访问

变量声明时候，会加入给其分配对应的栈空间（全局变量为静态空间），然后如果有初值，则初始化；没有则用零初始化，然后判断当前作用域深度，选择以全局变量/临时变量的形式将变量的标识符及其属性加入到符号表。这里保存的变量值（ `value` 字段）实际为其值所在地址。

在访问和修改变量的值之前，都会通过符号表获取其存储的地址。变量的访问，会先构造一个 `load` 语句从地址读取，传给表达式求值等部分。修改变量的值，则会使用 `store` 语句把变量的值放置到对应地址。

1.3 数组的初始化与访问

数组的声明通过 `llvmlite` 模块中添加全局变量与临时变量的方式实现。首先判断当前变量作用域深度 `currunt_scope_level`，选择调用 `ir.GlobalVarible` 或 `Builder.alloca` 接口，再将该变量加入符号表。如果在声明的同时对数组进行了初始化，则通过如下数组访问的方式进行值的更新即可。

```
ir.GlobalVariable(Module, Type, name = CName)
IRBuilder.alloca(Type, name = CName)
```

数组的访问通过 `llvmlite` 模块中加载指针指向变量值和保存变量值到指针的方式实现。先调用 `IRBuilder.gep` 获取待访问数组元素的地址，再调用 `IRBuilder.load` 加载地址到本地，运算结束后调用 `IRBuilder.store` 保存新的值到地址。

```
IRBuilder.gep(ptr, indices, inbounds=False, name='')
IRBuilder.load(ptr, name='', align=None)
IRBuilder.store(value, ptr, align=None)
```

2. 表达式

表达式分为函数调用，标识符，立即数，判断式，运算式，赋值式等。我们重写了 `visitExpression` 类，访问表达式时先递归地处理每个子表达式，将他们的类型统一，然后调用 `llvmlite` 中的表达式处理接口，最后返回一个包括表达式结果类型 `type`，表达式结果符号标志 `signed`，表达式结果的llvm值 `value` 的词典，以便使用。

2.1 运算表达式

运算表达式支持的符号包括 `'+'|'-'|'*'|'/'|'%'|'>>'|'<<'` 。实现的方式为：先递归地处理左、右子表达式，将他们转换为同一类型，记录为 `exprType`,再调用 `llvmlite` 处理表达式的接口，保存值到LLVMValue。最终返回的结果为 `{exprType,True,LLVMValue}`

```
IRBuilder.add(lhs, rhs, name='', flags=())
IRBuilder.sub(lhs, rhs, name='', flags=())
IRBuilder.mul(lhs, rhs, name='', flags=())
IRBuilder.sdiv(lhs, rhs, name='', flags=())
IRBuilder.srem(lhs, rhs, name='', flags=())
IRBuilder.shl(lhs, rhs, name='', flags=())
IRBuilder.lshr(lhs, rhs, name='', flags=())
```

2.2 判断表达式

判断表达式支持的符号包括 '==' | '!=' | '<' | '<=' | '>' | '>=' 。实现的方式为：先递归地处理左、右子表达式，将之转换为同一类型，根据数据类型选择调用 `llvmlite` 的接口，保存值到LLVMValue。最终返回的结果为 `{ir.IntType(1),True,LLVMValue}` 。

```
IRBuilder.icmp_signed(cmpop, lhs, rhs, name='')
IRBuilder.icmp_unsigned(cmpop, lhs, rhs, name='')
IRBuilder.fcmp_ordered(cmpop, lhs, rhs, name='', flags=[])
```

2.3 赋值表达式

赋值表达式支持对数组和变量进行赋值，具体的方法与数组/变量初始化的方式一致。

稍微具体来说，我们把所有能赋值的东西定义成为了 `leftExpression` ，与 `expression` 同样返回一个字典，但是不同的是，字典的 `value` 字段返回的是存储变量的地址。

处理 `leftExpression '=' expression` 的时候，我们就会将右侧 `expression` 的值存入左侧 `leftExpression` 提供的地址中。

赋值表达式也会返回一个值，就是 `leftExpression` 被赋予的值。

3. 函数

3.1 函数的声明和定义

函数的声明在 `visitFunctionDecl` 函数中。

```
LLVMFuncType = ir.FunctionType(ReturnType,ParameterTypeTuple,var_arg=is_var_arg)
LLVMFunc = ir.Function(self.Module, LLVMFuncType, name=FunctionName)
self.symbolTable.addGlobal(FunctionName,NameProperty(type = LLVMFuncType,value =
LLVMFunc))
```

我们会先读取函数声明中的参数列表并构建函数实例，加入符号表。这里如果不给 ir.Function 增加任何的块就可以让函数成为声明。

值得一提的是，LLVM 的 Function 模块支持可变参数（va_args），这给我们调用标准库（如 `printf`）产生了极大的方便。

支持对函数的声明可以让我们有机会调用 C 标准库的函数，这可以以极小的成本扩展我们程序的功能。

函数的定义在 `visitFunctionDef` 函数中。

定义大部分与声明相同，只是在最后多新建一个 `Builder` 和 `Block` 并让函数体往这里面填写。

```
Block = LLVMFunc.append_basic_block(name="__"+FunctionName)
Builder = ir.IRBuilder(Block)
self.Builders.append(Builder)
# ...
ValueToReturn=self.visit(ctx.block())
```

值得提到的是，C++ 可能出现一个函数没有 `return` 语句的情况，而 LLVM 却要求每一个 block 都必须有终结符，因此在函数结尾如果发现生成的 LLVM 没有终结符，我们需要补上一个：

```
if(not self.Builders[-1].block.is_terminated):
    self.Builders[-1].ret_void()
```

3.2 函数的调用

函数的调用语句的处理在 `visitFunctionCall` 中的 call 函数。

```
ret_value = Builder.call(property.get_value(), paramList, name='', cconv=None,
tail=False, fastmath=())
```

4. 程序结构

程序结构主要通过`llvmlite`中的跳转和条件跳转语句接口实现：

```
IRBuilder.branch(target)
IRBuilder.cbranch(cond, truebr, falsebr)
```

4.1 选择结构

程序支持的选择结构有 if 和 switch 。

注意 switch 语句的 case 里若语句大于一条需要加一个大括号把语句框起来。

if 语句会视有没有 else 的情况而分类讨论，有 else 则分成三个块，其中分别是 if 对应的语句块，else 对应的语句块和最终的结尾块。程序会先判断 if 的表达式，之后调用 `IRBuilder.cbranch` 根据表达式结果选择进入 if 语句块还是 else 语句块。无论是 if 语句块还是else语句块都会在结尾跳转到结尾块，结尾块存入数组。若没有 else 则分为两个块， if 语句块和结尾块，`IRBuilder.cbranch` 根据表达式结果选择进入if语句块还是结尾块，if 语句块结束后也会进入结尾块。

switch 语句会生成两个语句块链——判断链和语句链。判断链为许多判断是否跳转到语句链的语句，语句链则为 case 下面的语句。程序会调用函数判断表达式是否命中 case ，仍然用 `IRBuilder.cbranch` ，若命中则跳转到语句链，然后顺着语句链依次执行，若没有命中则跳到下一个判断块继续判断。如果全部没有命中则直接跳到结尾块，语句链结束之后也是跳到结尾块。如果有 break 就直接在 case 的语句块末尾转而跳转到 switch 语句结束。

4.2 循环结构

程序支持的循环结构有 while , dowhile, for。

while 和 dowhile 循环都分为三个块，分别保存判断表达式，循环部分和循环结束部分的语句。声明这三个块后，程序首先在当前 `IRBuilder` 输出指令跳转到判断表达式，然后输出判断表达式对应的语句，再调用 `IRBuilder.cbranch` 选择进入循环部分或循环结束部分，在循环部分结束后，判断当前块没被 `break/continue` 语句插入br语句时，调用 `IRBuilder.branch` 进入表达式判断语句块。

for 循环结构为 `for(forExprSet;expression;forExpr){ }`，括号内的三个部分都可以为空，目前不支持在 `forExprSet` 中进行变量的声明，但可以对已有的变量进行赋值。具体的实现中，for循环分为四个块，相比 while 和 dowhile 循环，在循环语句块后增加了更新语句块，逻辑基本一致。

4.3 跳转结构

程序支持的跳转结构有 break 和 continue 。程序在 `visitor` 中定义列表 `blockToBreak` 和 `blockToContinue`，以栈的形式保存语句块。在进入一段循环结构时，将该循环结构中调用 break 和 continue 时跳转至的语句块加入对应列表中。识别到 break 和 continue 语句时，调用 LLVM 的接口，输出跳转语句。

break 语句的功能为跳转到当前所在循环体的结束块。 continue 语句的功能为跳转到当前所在循环体的表达式判断块。特别地，对于 for 循环，continue 语句会跳转到更新语句块，再继续进行表达式判断。

四、难点与创新点

1. 对一些需要嵌套的语法（表达式，循环结构）进行中间代码生成时，需要递归地进行处理。这个过程需要对返回值进行类型定义，重组等操作，确定输出跳转的语句块，逻辑相对复杂
2. 声明了 scanf 和 printf 函数，用较小的代价实现了输入输出。
3. 程序的跳转部分实现较为复杂，需要多方面照顾
4. 在实现新语法的时候，很容易改的让原来支持的语法出现错误；为此我们保存了我们写好一个语法之后留下的测试文件，并且编写了自动化测试脚本，这样可以比较容易地（靠谱地）进行增量开发

五、小组分工

陈启乾：语法文件编写，函数调用部分，变量 & 符号表部分

潘首安：表达式部分，数组部分，循环和跳转结构

谭弈凡：选择结构编写，样例程序编写

---

1. 如果没有 make，也可以使用以下命令：`antlr4 -Dlanguage=Python3 grammar/cpp20Parser.g4 grammar/cpp20Lexer.g4 -visitor -o src`

2. 或者其实 `src` 文件夹里面有已经生成好的程序，理论上可以直接运行。