一款由C到LLVM的编译器

王世杰 沈冠霖 陈语凝

石墨文档链接：(https://shimo.im/docs/WkXRQHRhRRxHdWyP/)

一、使用说明

1. 环境配置

- 系统：Ubuntu16.04
- 语言：Python3.6
- 安装antlr4； 安装antlr4-python3-runtime、llvmlite；

```
pip install antlr4-python3-runtime
pip install llvmlite
```

2.运行说明

注意以下命令运行路径为/src目录下，即与main.py同级
生成某个源代码的LLVM IR代码

```
python main.py test/xxx.c    #该命令会在源代码同级目录下生成xxx.ll文件
```

快速编译test文件夹下所有源代码

```
python compile_all.py        #该命令会编译test文件夹下所有源代码
```

运行LLVM IR代码

```
lli xxx.ll
```

二、代码结构

Generator

> SymbolTable.py 符号表的实现
> Generator.py 语义分析实现，由C代码转到LLVM IR代码
> Constants.py 错误信息
> ErrorListener.py 语法、语义错误处理

Parser

> simpleC.g4 C语言部分语法实现，来自ANTLR官方对C语言的支持
> simpleC.interp 本项及以下5项由antlr4自动生成
> simpleC.tokens
> simpleCLexer.interp
> simpleCLexer.py 用于词法分析
> simpleCLexer.tokens
> simpleCListener.py
> simpleCParser.py 用于语法分析
> simpleCVisitor.py 语义分析基于ANTLR的Visitor模式进行(Listener模式与作业需求匹配不佳)

test

> palindrome.c 回文判断，可验证基本实现
> kmp.c KMP算法，可验证基本实现
> calc.c 四则运算，可验证复杂语义分析实现
> quickSort.c 快速排序，可验证复杂语义分析实现
> biTree.c 基于结构体的二叉树前序遍历，可验证结构体、结构体数组相关实现

三、实现方法与重难点简述

1.变量，结构体，函数的管理

1.1 符号表的实现

我们实现了SymbolTable类，也就是符号表类，用来管理变量（包括单一变量，数组，结构体变量）。
变量作用域是一个类似栈的结构------内层能访问外层变量，外层不能访问内层，一层结束后，其变量都没有意义了。因此我实现的符号表也是这种结构。
存储方式：一个数组，每个元素代表一层。数组每个元素是字典，其key是变量在C语言的变量名，value是变量在LLVM的类型，名称等，每个key-value键值对代表一个变量。
变量的各种操作：首先，进入和退出作用域（函数，分支，循环等）在符号表里体现为入栈和出栈---进入作用域，在数组后面append一个字典，代表新的一层的变量存储位置；退出作用域，删除数组最后一个元素，代表当前作用域的局部变量失效。
其次，在作用域内新增变量，则只需要访问数组最后一个字典元素，也就是当前局部变量。在这个字典里遍历key来判断是否重定义，如果没有重定义就新建一个键值对。
之后，查找变量的方法是由内到外，也就是数组由后到前访问。如果在内层找到，就不去外层找了，这就实现了变量的覆盖。如果找不到，就进行对应的异常处理。
最后，因为LLVM对于全局和局部变量处理策略不同，因此我们也要判断变量是全局还是局部。判断方法很简单：如果当前数组只有一个元素，代表当前在全局位置，否则是局部位置。

1.2 函数和结构体的管理

结构体我实现了Structure类。结构体整体是一个字典，每个key-value对代表一个结构体。key是结构体在C的名称，value里存储了其变量名和类型，暂时只支持结构体里面的变量是其他结构体或者单一变量，没有支持数组。读取结构体变量的时候，就直接遍历结构体的变量名数组进行匹配，提取对应的类型和顺序即可。
函数则是直接用一个字典存储。
通过遍历字典，可以很方便的进行结构体，函数的重定义和未定义错误处理。

1.3 变量，数组，结构体的LLVM实现

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

全局，局部变量在LLVM里分别用以上两种方法实现。其中LLVMName代表其在LLVM中的名称，Cname代表其在C中名称，Type是类型。如果是数组，结构体，也一样，只是类型有所不同。结构体变量类型依赖结构体名称，变量名称和类型，数组变量类型依赖数组类型和长度。

2. 函数

2.1 函数定义

函数包括函数定义，参数，代码块三大部分。
定义函数需要以下语句

```
LLVMFunctionType = ir.FunctionType(ReturnType, ParameterTypeList)
LLVMFunction = ir.Function(Module, LLVMFunctionType, name = CFunctionName)
```

也就是，函数类型由其返回值和变量数目和类型决定; 函数由其C语言命名和类型定义; 函数参数遍历获取即可，但是要注意，函数参数存储的时候需要符号表进一层，因为函数参数和函数内直接定义的局部变量是等价的。
代码块较为容易解决，和其他的代码块一样。

2.2 函数调用

函数调用使用以下语句

```
ReturnVariable = Builder.call(LLVMFunction, ParameterList)
```

也就是先读取LLVM函数本身和其参数列表，然后返回值相当于一个当前位置局部变量。
我们还实现了调用printf，scanf等库函数，调用方法和调用自己的函数大致一致。

3. 程序结构

3.1 选择结构

选择结构支持if-else if-else 语句，此处生成LLVM IR的思路和汇编中跳转的思路类似，通过创建不同的代码块，跳转或条件跳转到不同的label处。

3.2 循环结构

循环结构支持for循环和while循环。
对于for循环，其基本语法是：

```
for(init; condition ; step)
```

其中init初始化语句可以为空，也可以是多条用,分割的语句。其逻辑顺序为，在循环开始前先执行init语句，然后判定condition，若满足则执行循环的主题body，随后step，一次循环后再次判定condition，进行下一次循环。
对于while循环，基本语法为：

```
while(condition)
```

先判断condition，如果condition为真则执行body，执行完成后再次跳转到condition判断，进行下一次循环，直到不满足condition跳出循环语句。

4. 错误处理

4.1 语法错误

我们使用ErrorListener.py中的syntaxErrorListener()统一监听SimpleCParser在进行语法分析时产生的语法错误，获取其详细信息后反馈。

4.2 语义错误

我们使用ErrorListener.py中的SemanticError()统一监听Generator()各语义分析函数返回的错误，相关语义分析函数遇到语义错误时执行如下：

```
raise SemanticError(ctx=ctx,msg=TheResult["reason"])
```

- 目前支持被检测的语义错误有：
- 变量名重定义、未定义问题
- 结构体定义时格式错误
- 结构体中变量无法检索
- 函数重定义、未定义问题
- 不支持的类型转换
- 全局变量无法检索
- 变量定义和声明的类型不匹配
- 函数定义和声明的类型（考虑参数类型）不匹配
- 数组声明范围不合法

三、团队分工

王世杰	条件，循环等分支，2个样例代码编写，代码运行框架的搭建，llvm测

	试，异常处理和对应测试
沈冠霖	Antlr4处理g4的框架搭建，符号表，简单结构体，函数，变量赋值和初始化，数组
陈语凝	表达式求值，类型转换等，3个样例代码编写，异常处理