C2LLVM

项目简介

A simple compiler: from C to LLVM IR.

环境配置

- 系统：Ubuntu 22.04.3 LTS
- 语言：Python 3.11.7
- 安装antlr4，antlr4-python3-runtime，llvmlite

```
# 1.安装altlr4
$ pip install antlr-tools
$ antlr4    # 执行此命令会自动检查并安装antlr4运行所需要的Java环境
...
# 2.安装antlr4-python3-runtime
$ pip antlr4-python3-runtime
# 3.安装llvmlite
$ pip install llvmlite
```

- 安装clang/llvm

```
$ sudo apt-get install llvm
$ sudo apt-get install clang
```

使用说明

以下命令均运行在`src`目录下

- 生成某个源代码的IR

```
$ pwd
~/C2LLVM/src
$ python main.py test/xxx.c     # 在xxx.c同级目录下生成xxx.ll
```

- 执行IR

```
$ lli xxx.ll
```

语法支持

- 源代码组织顺序必须遵循**头文件引用→全局变量声明→函数定义**
- 数据类型：void，int，bool，double，char，string，一维数组
  - **不支持任何指针变量**
  - **不支持结构体**
  - 对于char来说，仅支持单个字符，不支持转义符如`\t \n \0`

```
char c = '0'; // 合法
char c = '\0'; // 不合法，抛出语法错误
```

- 对于string来说，仅支持的转移符为 `\n`

```
printf("hello world\n"); // 合法
printf("hello world\t"); // 不会抛出语法错误，但不会将\t识别为转义符
```

- 一维数组的声明和赋值(只能先声明，再逐个元素赋值)

```
int arr[3];
arr[0] = 1; // 合法
int a = arr[0]; // 合法
char str[3] = "kas"; // 不合法
```

- 单行注释，块注释**(仅支持英文注释)**
- 运算符
  - 算数运算符（+,-,*,/,%）
  - 逻辑运算符（&&,||）
  - 关系运算符（>,<,≥,≤,==,≠）
  - **不支持任何单目运算符（++,--）**
- 控制流语句 `if-elif-else`， `for`， `while` 以及任意层级的嵌套使用
  - 嵌套示例

```
int a = 6, b = 20;
if(a == 3) {
    b = b + 1;
    printf("%d\n", b);
} else if(a == 6) {
    if(b == 20) {
        b = b * 2;
        printf("%d\n", b);
    }
} else {
    ...
}
```

  - 不支持在for的表达式中定义变量

```
for(int i = 0; i < 3; i = i + 1) { } // 不合法

int i;
for(i = 0; i < 3; i = i + 1) { } // 合法
```

  - 不支持continue，break语句
- 函数定义，函数调用（**参数和返回值不能是数组类型**，只能是其他支持的数据类型）
  - 函数定义的最后一条语句必须是 `return`
- 标准库函数调用（`gets,strlen,printf,scanf,atoi`）

```
char str[100];
gets(str);
int len = strlen(str);
printf("%d\n", len);
```

实现原理

整体思路

首先定义语法规则，然后使用Antlr4生成对应的抽象语法分析树（AST）；

遍历AST的每一个结点，使用LLVM提供的接口生成中间代码（本项目中用到的llvmlite就是LLVM提供的接口）。

符号表

代码中定义了 SymbolTable 类，用来管理变量；核心的数据结构是一个列表，列表项是一个字典，存储变量名和存储地址之间的映射关系，列表的结尾项是当前作用域的变量；

每当进入一个新的作用域（比如函数、控制语句），就在列表结尾新增一项；

每当退出一个作用域时，就将列表的结尾删除掉；

查询变量时，优先在列表结尾查找，如果找不到，则在倒数第二项查找，如果直到列表的第1层都没有找到目标符号，那么便抛出错误。

变量定义方法如下

```
var = ir.GlobalVariable(Module, Type, name) # 全局变量
var = Builder.alloca(Type, name) # 局部变量
```

函数

使用字典存储函数名与函数定义之间的映射关系，方便查找

定义函数语句如下

```
funcType = ir.FunctionType(ReturnType, ParamList)   # 指定返回类型和参数列表
func = ir.Function(Molule, funcType, name)  # 函数实体
```

函数调用语句如下

```
retvar = Builder.call(func, args)   # args是实际参数
```

控制语句

将一个控制语句分割为多个基本块（basic block），通过设计基本块之间的条件跳转，实现控制逻辑

比如 if 语句就可以分为2个块，即True和False

编译原理课程上讲解中间代码生成时，涉及到不同控制语句如何分块

```
    condition = ...
    if (!condition) goto False
True:
    goto Next
False:
    ...
    goto Next
Next:
```

错误处理

定义了一个语义错误基类`SemanticError`

在语义分析的过程中遇到错误则执行下面代码

```
raise SemanticError(msg=reason, ctx) # msg是错误原因, ctx是上下文指针
```

目前支持的语义错误检测包括

- 变量重定义
- 引用未知变量
- 调用未知函数
- 函数参数不匹配
- 不支持的类型转换
- 不支持的运算（如浮点数不能取余）

难点

左值和右值

左值就是变量的地址，右值就是地址单元中存储的内容

对于id来说，符号表中存储的信息是它的地址（也就是指针）

1. 如果表达式是 `id = expr`，那么此时id返回左值刚好合适，然后调用`store`语句

   ```
   Builder.store(expr, id_ptr) # 将expr的值存入对应地址
   ```

2. 如果表达式是 `a = id`，那么此时id应该返回右值，即变量的内容；

   代码实现需要先调用`load`，再调用`store`

   ```
   newt = Builder.load(id_ptr)
   Builder.store(newt, a_ptr)
   ```

3. 调用函数时，比如 `int add(int a, int b) {...}` 这里的a和b也应该是右值

控制语句跳转

> 通过假链回填处理if语句的跳转

if语句的语法规则定义如下

```
# 语法规则 - if语句
ifBlocks : ifBlock (elseIfBlock)* (elseBlock)? ;
ifBlock : 'if' '(' condition ')' '{' body '}' ;
elseIfBlock : 'else' 'if' '(' condition ')' '{' body '}' ;
elseBlock : 'else' '{' body '}' ;
```

生成中间代码时，基本块跳转逻辑如下

```
        ifBlock : 'if' '(' condition ')' '{' body '}' ;
        ----------------------------------------
            ...          ; 紧跟着上一个block，不需要新建基本块
            %cond = ...
            br i1 %cond, label %true, label %false
        true:
            ...body...
            br label %next
        false:
            ...
        ----------------------------------------
```

```
        elseIfBlock : 'else' 'if' '(' condition ')' '{' body '}' ;
        ----------------------------------------
            ...            ; 紧跟着上一个 if 的 falseBlock
            %cond = ...
            br i1 %cond, label %true, label %false
        true:
            ...body...
            br label %next
        false:
            ...
        ----------------------------------------
```

```
        elseBlock : 'else' '{' body '}' ;
        ----------------------------------------
            ...       ; 紧跟着上一个 if/elif 的 falseBlock
            ...body...
            br label %next
        next:
        ----------------------------------------
```

下面的讨论我用 子block 来描述 ifBlock、elseIfBlock、elseBlock

子block 里面都有 `br label %next` 语句

但是%next是所有子block处理完之后才能确定，所以采取**回填假链**的方法；

即将%next定义在所有子block的结尾，然后遍历需要跳转的每一个block，添加一行代码`br label %next`

类型转换

为了保证程序的正常运行，很多地方需要隐式的类型转换

1. 赋值类型转换，比如`id = expr`，需要将expr的类型转换为id的类型才能完成赋值
2. 算数类型转换，比如`expr1 + expr2`，如果是`int8 + int32`，则需要转换为`int32+int32`，或者`int32 + double->double+double`

3. 函数参数类型转换，比如函数定义为`int add(int a)`，调用时`add(3.0)`，则需要将参数转换为int类型
4. 函数返回值类型转换，比如函数定义时`int add(){doule a; return a;}`，则需要将a转换为int类型再返回

转义字符识别

在读取string时，源代码中的`\n`在词法分析时返回的token其实是`\\n`，即转义字符表示单独的一个字符，但token是2个字符`\\`和`n`，需要将token中的`\\n`都替换为`\n`，才能正确处理

小组成员&分工

- 郭恩惠：ANTLR g4文件编写，语法分析：符号表，函数定义，变量声明，赋值语句，ret语句，if语句，while语句，expr表达式求值，库函数调用（printf）
- 彭宗睿：测试程序源语言，库函数调用（strlen，gets，scanf，atoi），自定义函数调用
- 郭宇飞：for语句，char变量读取，bool变量读取