2014-6-9

PYWW -- 基于 LLVM 的 PYTHON 编译器

编译大作业说明文档

2011013239 文庆福 13681332621 thssvince@gmail.com
2011013248 王 需 18810456160 xu-wang11@mails.tsinghua.edu.cn
2011013256 杨 磊 18810305382 yl93528@gmail.com
清华大学 软件学院

目录

1. 开发环境与工具

操作系统： **Mint15 & Ubuntu13.04**

环境依赖： **LLVM 3.2**

Clang 3.2

LLVMPY 0.12.0

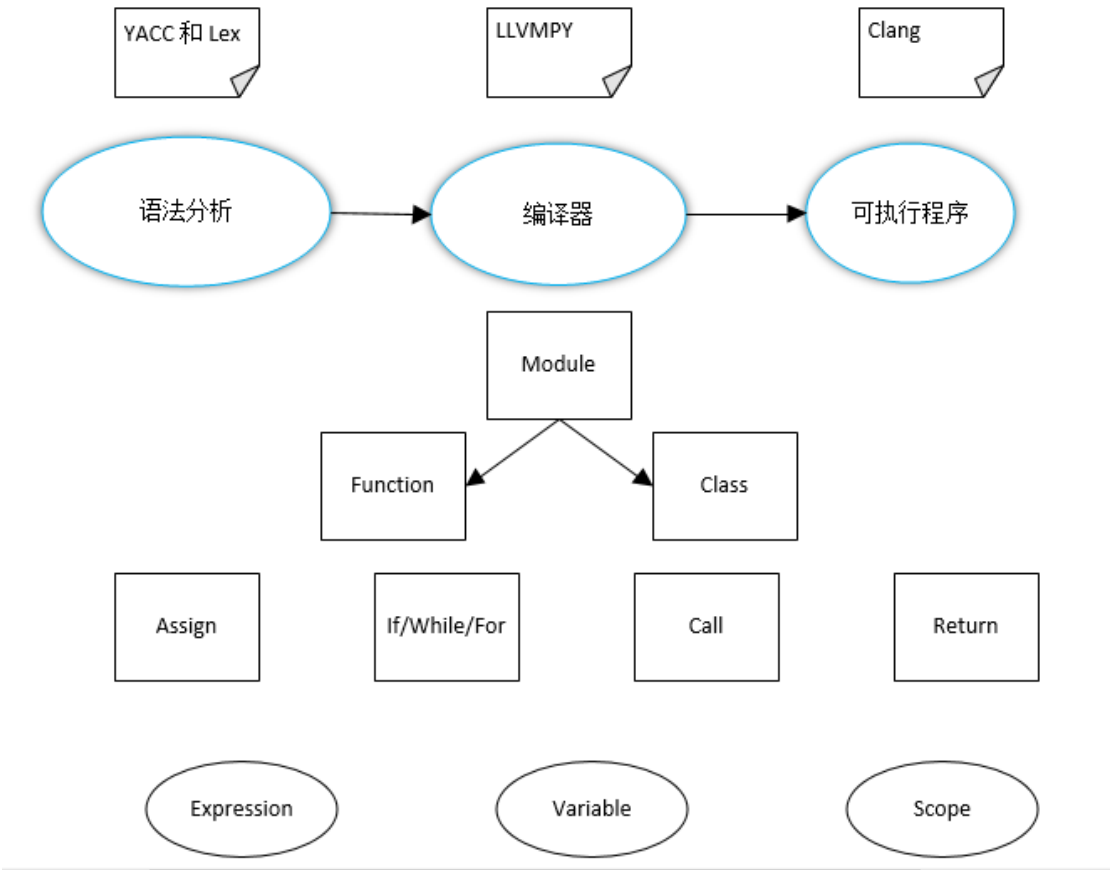
PLY 3.4

编辑器： **PyCharm**

代码管理工具： **Git**

2. 设计与实现

2.1 系统结构



在本次实验中，我们实现了 python 到 IR 再到可执行程序的转换。其中 IR 是 LLVM 的中间代码，类似于 MS 汇编。在我们的程序中，程序首先读入.py 文件，对 python 文件进行词法分析和语法分析（这部分分别由 lex 和 yacc 完成）生成一颗语法树 AST，然后对 AST 树进行编译得到.ll 文件。.ll 文件是 LLVM 的中间代码，使用 Clang 对该文件进行编译就可以得到相应的可执行程序。如上图所示，本次实验的重心集中在如何对 Python 进行编译这块。其基本思路是把一个 Module 编译为 main 函数，将其中的函数声明编译为普通的函数，将类编译为结构体加函数，具体细节会在 2.4 部分详述。

2.2 语法分析

Python 的词法分析与语法分析的工作我们是采用开源工具 PLY 来实现的。PLY 是 lex 和 yacc 的 python 实现，包含了它们的大部分特性。PLY 采用 COC（Convention Over Configuration，惯例优于配置）的方式实现各种配置的组织，比如：强制词法单元的类型列表的名字为 tokens，强制描述词法单元的规则的变量名为 t_TOKENNAME 等。我们只需要定义好 token 列表、token 模式、token 的取值、行号与位置信息以及错误处理等，就可以调用 lexr.lex()来进行词法分析。具体代码见 python_token.py, python_lex.py。

2.3 文法分析

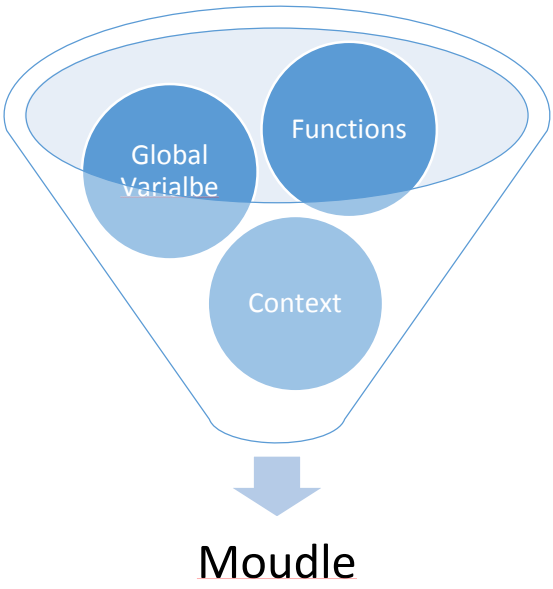
语法分析部分采用 yacc 实现，Yacc 用的分析技术是著名的 LR 分析法或者叫移进-归约分析法。LR 分析法是一种自下而上的技术：首先尝试识别右部的语法规则，每当右部得到满足，相应的行为代码将被触发执行，当前右边的语法符号将被替换为左边的语法号。LR 分析法一般这样实现：将下一个符号进栈，然后结合栈顶的符号和后继符号，与文法中的某种规则相比较。

在调用 yacc.parse() 函数解析完成后，我们可以得到一课抽象语法树 AST，然后基于 AST 做语义分析以及编译相关的工作。目录下的 AST.txt 就是我们快速排序程序 python 代码转化后得到的 AST。

对于词法分析分析和语法分析，这部分代码并不是我们写的，我们直接使用了 python4ply[3]这个开源项目中的 python_lex.py, python_token.py, python_yacc.py 三个文件。更多详细内容可以阅读参考资料中的[2][3][4][5]。

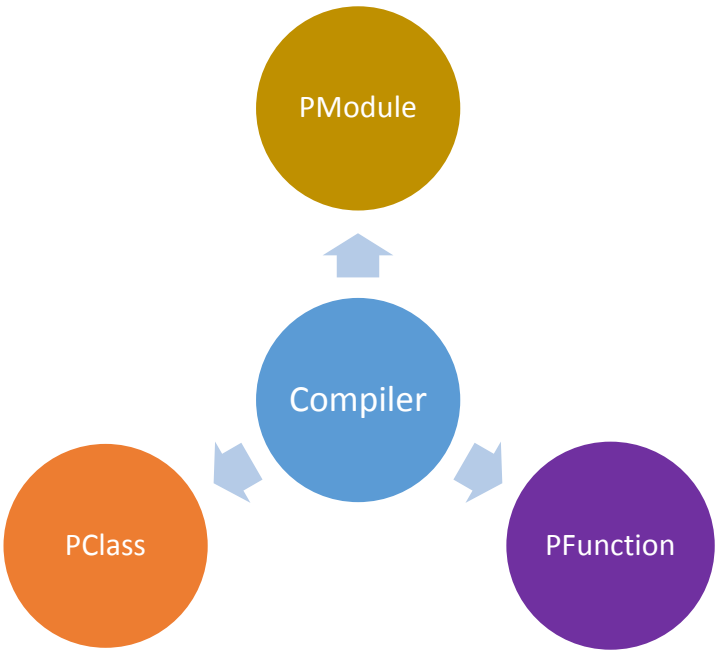
2.4 编译

在 LLVM 一个 Module 抽象为程序的主体，其中包含着执行环境的信息，全局变量和声明的函数。



每一个函数对应有参数、返回值和函数体，函数体由 Builder 对象管理，包括栈和堆的内存分配、读、写，变量之间的运算，添加代码块进行代码跳转等等。

在程序中共有四个主要的类执行编译的工作，如下图所示：



主要的实质性的编译工作在 Compiler 类中完成，而在 PModule,PFunction,PClass 分别代表着 python 中的三种作用域（Module, Function, Class）。这三部分主要负责对编译工作进行管理，包括变量管理，函数定义的管理，在 python 中这三个作用域都可以嵌套定义函数。

在编译的过程中，我们将一个 Python 中的 Module，也就是一个.py 文件编译为一个 Main 函数，每一个函数有自己相应的变量表对变量进行管理。一个.py 文件由语句和函数定义组成，在编译时，因为 python 是没有类型的，我们在遇到一个函数定义的时候

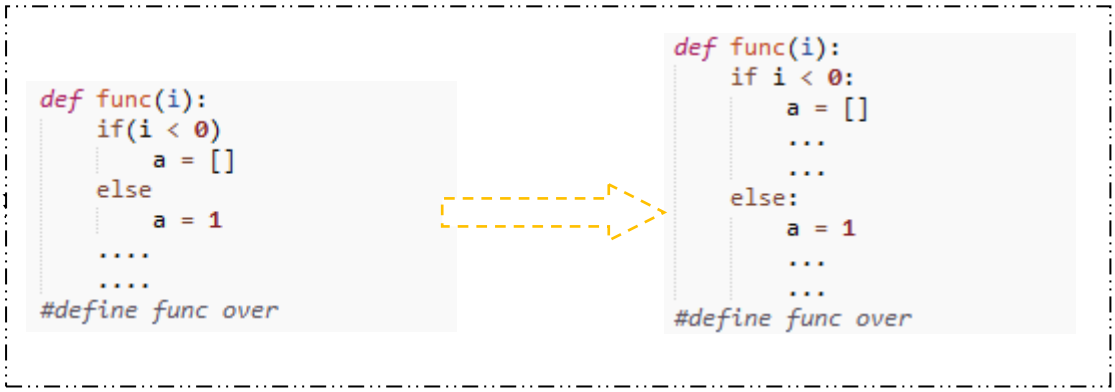
并不会立即的将函数编译，而是只是生成其对应的对象 PFunction 保存下来，当遇到函数调用时，则根据对函数传入的参数分析出函数的参数类型，在对函数进行编译，但是在对函数编译的时候我们并不知道函数的实际返回值得类型是什么，而函数编译必须在其声明前完成，所以在编译的时候我们首先假设函数的返回值为空，在对函数体编译后推断出函数的返回值类型，如果恰好为空，那么可以很 nice 的继续编译，如果函数的返回值不是空，那么我们将已经编译好的函数删除，并对该函数重新编译。

因为 Python 是一种解释型的语言，变量的类型只有在运行时在能确定，如果我们将其编译为汇编的话，对每一个变量的类型都必须要了如指掌，这样才能合理进行变量的内存分配。在我们的实现中，我们对程序做了如下两个假设：

A. 变量在相同的作用域内不能出现类型变化，即如果一个变量在某个阶段为 int 类型，在另一个阶段为 list 类型，这种情况是不允许。

B. 一个函数只能有一种类型的返回值而不能有两种类型。这个要求一方面跟编译语言相适应，两一方面，也是对 A 条件的保证，如果一个函数有两种类型的返回值，那么当进行函数调用时，我们就无法确定一个变量的类型了。

之所以做出这样的考虑，我们觉得模糊的类型对编译型语言是灾难性的。如下函数所示：



如果 a 拥有两种类型，ifelse 块下面的语句就要分两种情况进行编译，也就是右图所示的情景。

类的编译过程分为两个部分，一个是将类组合为结构体，这个任务是分析出其成员，二是将类的成员函数编译为普通的函数，这个过程中，只要我们将类的对象的指针作为 self 参数传递进去就可以访问类的成员。在类中有一个比较特殊的函数需要处理，那就是构造函数，在 python 中即__init__(self)函数。在调用__init__(self)函数的时候需要完成对类的数据进行初始化。同时__init__(self)函数并没有返回值，我们在调用__init__(self)函数来构造一个对象的时候，需要三个步骤：

（1） 在函数外部分配内存空间。

（2） 在函数内部调用对类的成员初始化。

（3） 编译__init__(self)函数。

（4） 将 self 对象赋值给变量。

在对类的成员进行访问时，其算法如下：

（1） 由属性名查找其成员在结构体中的声明顺序。

（2） 计算属性在结构体中的指针偏移。

（3） 通过指针从内存中加载出属性的值。

2.5 类型分析

类型分析在编译的过程中完成，当对对象赋值的时候，我们根据值的类型可以得道变量的类型。

2.6 分支循环语句块

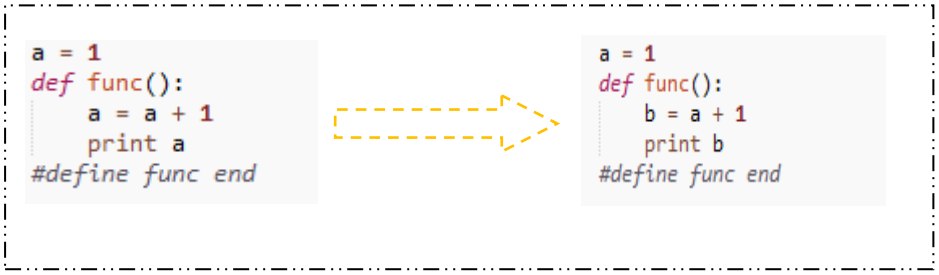
通过调用 LLVM 中的 BRANCH 来生成分支语句，为不同的分支语句设置不同的标签名字，通过标签间的跳转来实现分支语句的效果。WHILE 和 FOR 都依照 IF 的思想来构建，将 WHILE 和 IF 语句块分成循环条件和循环主体，循环条件根据 IF 来判断循环是否继续进行。

值得一提的是对循环嵌套情况的处理，当解析到 WHILE 或者 FOR 时，都会当做语句块来嵌套处理，每一个语句块都会被解析成 before, body 和 after 三个标签（通过命名区分），分别用来指向循环初始值，循环主题和循环结束后的跳转。

2.7 变量作用域

在我们的程序中，我们支持 Module，Function, Class 三种作用域。每一个作用域内都有自己的参数表，当对变量进行访问的时候，只在对应的作用域内寻找变量的定义。但这三种作用域如何跨越并没有处理。实际上，Python 在对变量的作用域处理的时候并没有清晰的要求，而是考虑的是赋值的顺序，即如果变量在函数中被赋值，那么这个变量就是一个局部变量，任何在赋值前的取值操作都是错误的，如果该变量没有没赋值，那么这个变量就是外部变量，可以访问。

如



两段程序的执行结果并不相同，其中左边将显示 a 未定义的错误，而右边将显示 2.

3. 功能与测试

3.1 实现功能

1. 支持+、-、*、/、>>、<<、and、or、not 算术运算和逻辑运算
2. 支持 print 函数,print 可以有多个参数。
3. 支持 If， for, while 等语句。
4. 支持函数嵌套编译
5. 支持类的编译

3.2 编译快排

快排函数主体

```
a = [1, 2, 3, 9, 3, 4, 7, 10, 90, 20, 11, 10, 20]
qsort(a, 0, 12)
i = 0
while i < 13:
    print a[i]
    print '\n'
    i = i + 1
```

编译后.ll 文件（部分）

```
beforewhile:                                      ; preds = %while, %entry
  %12 = load i32* %i
  %15 = icmp slt i32 %12, 13
  br i1 %15, label %while, label %afterwhile

while:                                            ; preds = %beforewhile
  %a3 = load [13 x i32]** %a
  %16 = load i32* %i
  %17 = getelementptr [13 x i32]* %a3, i32 0, i32 %16
  %18 = load i32* %17
  %calltmp = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @printd, i32 0, i32 0), i32 %18)
  %calltmp4 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([2 x i8]* @str1, i32 0, i32 0))
  %15 = load i32* %i
  %19 = add i32 %15, 1
  store i32 %19, i32* %i
  br label %beforewhile

afterwhile:                                       ; preds = %beforewhile
  ret i32 0
}

define void @"qsort_i32*_i32_i32"(i32*, i32, i32) {
entry:
  %array = alloca i32*
  store i32* %0, i32** %array
  %head = alloca i32
  store i32 %1, i32* %head
  %tail = alloca i32
  store i32 %2, i32* %tail
  %3 = load i32* %head
  %l = alloca i32
  store i32 %3, i32* %l
  %4 = load i32* %tail
  %r = alloca i32
  store i32 %4, i32* %r
  %array1 = load i32** %array
  %l2 = load i32* %l
  %r3 = load i32* %r
  %5 = add i32 %l2, %r3
```

编译并输出结果

```
yl@ubuntu:~/Desktop/Pyww-master$ clang test2.ll
yl@ubuntu:~/Desktop/Pyww-master$ ./a.out
1
2
3
3
4
7
9
10
10
11
20
20
90
```

3.3 编译二分查找

二分查找函数主体

```
array = [1, 2, 3, 4, 7, 10, 20, 20, 90]
num = 10
binary_search(array, 8, num)
```

编译后.ll 文件（部分）

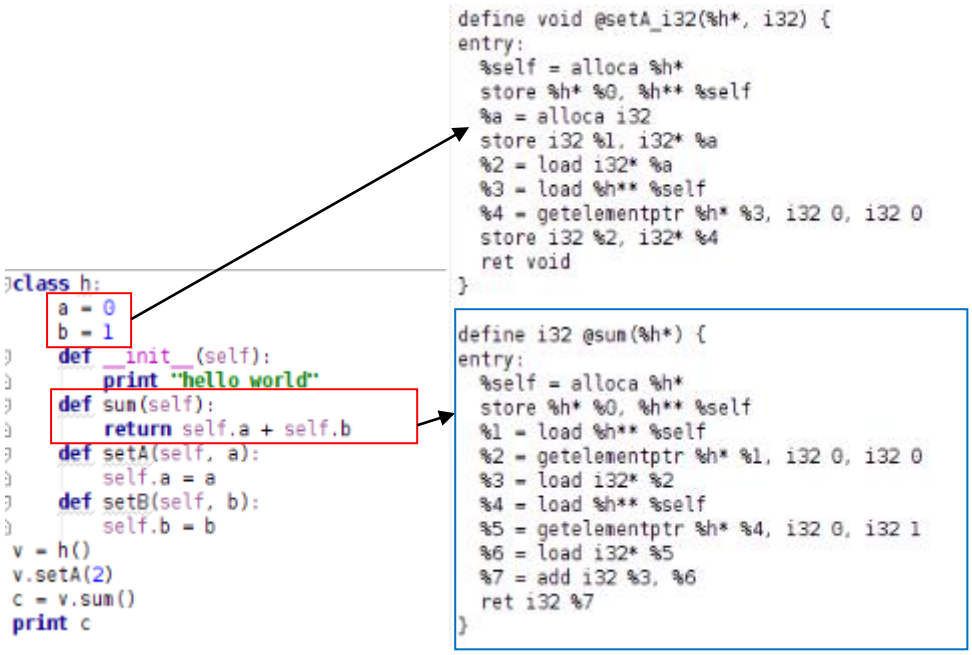
```
define void @"binary_search_i32*_i32_i32"(i32*, i32, i32) {
entry:
  %a = alloca i32*
  store i32* %0, i32** %a
  %n = alloca i32
  store i32 %1, i32* %n
  %m = alloca i32
  store i32 %2, i32* %m
  %low = alloca i32
  store i32 0, i32* %low
  %3 = load i32* %n
  %high = alloca i32
  store i32 %3, i32* %high
  br label %beforewhile

beforewhile:                                      ; preds = %ifend, %entry
  %high1 = load i32* %high
  %low2 = load i32* %low
  %4 = icmp sle i32 %low2, %high1
  br i1 %4, label %while, label %afterwhile

while:                                            ; preds = %beforewhile
  %low3 = load i32* %low
  %high4 = load i32* %high
  %5 = add i32 %low3, %high4
  %6 = sdiv i32 %5, 2
  %mid = alloca i32
  store i32 %6, i32* %mid
  %a5 = load i32** %a
  %7 = load i32* %mid
  %8 = getelementptr i32* %a5, i32 %7
  %9 = load i32* %8
  %midval = alloca i32
  store i32 %9, i32* %midval
  br label %beforeif0

afterwhile:                                       ; preds = %beforewhile
  ret void
```

编译并输出结果

```
5yl@ubuntu:~/Desktop/Pyww-master$ clang test3.ll
yl@ubuntu:~/Desktop/Pyww-master$ ./a.out
5
```

3.4 编译类

类成员访问、类函数访问



编译并运行结果：

```
yl@ubuntu:~/Desktop/Pyww-master$ clang test8.ll
yl@ubuntu:~/Desktop/Pyww-master$ ./a.out
hello world
3
```

4. 致谢

非常感谢王朝坤老师孜孜不倦地传授我们编译原理的知识，感谢陈俊助教呕心沥血辛苦批阅我们的作业！

感谢这一路相伴一同奋斗的战友们！

5. 参考资料

1. www.**llvmpy**.org/
2. http://www.dabeaz.com/ply/
3. http://dalkescientific.com/Python/python4ply-tutorial.html#pattern
4. http://blog.csdn.net/chosen0ne/article/details/8077880
5. http://www.cnblogs.com/P_Chou/p/python-lex-yacc.html#5

6. 附录

6.1 文档编写修订日志

版本	时间	参与人员	主要工作
Version 1.0	2014 年 6 月 9 日	文庆福、王需、杨磊	完成开发环境与工具、代码设计与实现、技术难点和功能测试的编写