西北師範大學

学 年 论 文

题　　目：一个完整的编译器前端 EasyFront 的设计与开发

学　　院：计算机科学与工程学院

专　　业：计算机科学与技术

班　　级：09 级非师范 2 班

学生姓名：张彦升

学　　号：200971040231

指导教师：赵学峰

二零一二年六月二十日

目录

一个完整的动态语言前端——EasyFront

摘要

随着科学技术的迅猛发展，计算机已毋庸置疑的成为了我们身边必不可少的工具，人们可以随处享用各种程序带来的乐趣，而计算机软件的开发是个繁琐而耗时的工作，使用一门高效的程序设计语言往往会使开发过程事半功倍，本文以简单为基本原则，运用编译原理中学到的各种算法，以学习为目的，尝试设计实现一个动态语言的前端——EasyFront，其以 Python 为参考语言，并不涉及代码优化，仅用本科阶段所学的知识，包括词法分析，语法分析及语义分析和字节码生成。通过设计，掌握了前端的实现对不同 IR 表示的影响，及对整个编译系统框架的思考。

关键字 编译器 前端 IR 编译系统框架

第一章　前言

1.1 编译器设计背景

简单的说一个编译器就是一个程序，它可以阅读以某一种语言（源语言）编写的程序，并把该程序翻译成为一个等价的、用另一种语言（目标语言）编写的程序，编译器的重要任务之一就是报告它在翻译过程中发现的源程序中的错误。

如果目标程序是一个可执行的机器语言程序，那么它就可以被用户调用，处理输入并产生输出。解释器是另一种常见的语言处理器，它并不是通过翻译的方法生成目标程序，从用户角度看，解释器直接利用用户提供的输入执行源程序中指定的操作。在把用户输入映射成为输出的过程中，由一个编译器产生的机器语言目标程序通常比一个解释器快的多。然而，解释器的错误诊断效果通常比编译器更好，因为它逐句执行源程序。

一个编译系统往往包括语言预处理，词法分析，语法分析，语义分析，字节码生成，及代码优化部分。在 c 语言中预处理器负责把那些称为宏的缩写转换成源语言的语句，然后，将经过预处理的源程序作为输入传递给一个编译器，编译器经过词法分析，语法分析，及语义分析，字节码生成之后便可产生一个汇编语言程序作为其输出，因为汇编语言比较容易输出和调试。接着，这个汇编语言程序由一个成为汇编器的程序进行处理，并生成可重定向的机器代码。GCC 是最典型的编译器，它能完成大多程序设计语言的编译工作，它将所有的语言编译成同一 IR 表示，然后对其进行优化，最终形成相应字节码。GCC 复杂庞大，阅读源代码若没有帮助文档则举步维艰，相比而言 Python 的源代码较为简单，结构清晰，是学习的好入口。

Python 的源代码以 C 语言实现，其词法分析是一个独立的模块，从而方便为语言添加新的特性，其 grammar.h 中完整的定义了语言 DFA 结构，在 grammar 文件夹下完整的定义了语言文法规则，全部使用 BNF 表示法表示。

Python 的语法分析是一个运行有穷状态自动机的过程，它不断的从词法分析模块得到程序的词素，然后判断有穷状态自动机的各个状态是否合法，在运行有穷状态自动机的过程中构造 CST，CST 存有大量的冗余，从 CST 到 AST 转换之后使语法树节点大量减少，再从 AST 生成字节码并进行代码优化，生成字节码后由 Python 虚拟机执行。

想要设计一个完整的编译系统，凭一人之力，几乎不可能完成，而现代编译系统的重点都在代码优化部分，对于与机器无关的优化，通过对数据流分析，常量传播，部分冗余的优化便能取得很好的效果，然而现在编译系统的发展方向却是指令级并行优化，这需要深度了解计算机体系结构，要对流水线有一定的了解，能够避免数据冲突及控制冲突，不使流水线停顿。

总的来说，开发一门新的语言是十分具有挑战性的工作，这其中不仅仅是技术上的挑战，更多的是对编译理论的挑战，当然，这毫无疑问是一项庞杂而耗时的工程。

1.2 当今流行语言的比较

多半程序设计语言如同昙花一样，仅仅流行很短的时间就被人们抛弃，只有像 C/C++这样与底层联系较密的语言才会被人们长期的使用。人们总结 C/C++的缺点，以及现代大型工程开发所需求的语言特性，抛弃指针，省去 C++很少用的语言特性，以虚拟机为运行环境，增加各种常用数据结构及技巧，设计了很多实用的程序设计语言。这些新型的语言大多是动态类型语言，虽然许多大型

系统追求速度及效率还是选择了 C 语言，但是以一门类似 python 这样简单的语言处理身边的杂事，不得不说，那是将会对整个工程的开发带来极大的方便。

在 JAVA 再次升级后，还有人不满意其虚拟机的执行效率，除了在追求开发速度及开发人员水平等方面有诸多的限制因素时，C++依然是许多项目负责人的青睐语言，它们两者都是静态类型的语言，其中 JAVA 是纯面向对象编程语言，而 C++兼容面向过程编程，C++博大精深，JAVA 简单方便，两者都适合开发大型项目，而且配合设计模式，很容易使系统的耦合性升高，聚合性降低，极大的提高了复用性。

脚本语言的出现使程序员处在欢乐而轻松的环境中编程，这其中不乏像 Perl，Ruby，Python 这样使人为之一惊的语言。Perl 是魔鬼语言，它的语法诡异，而且配有强大的正则表达式库，简直是那些不喜欢走平常路的程序员的救星，它提供的某些语法拯救了一些人饥渴的肚肠。还有 Ruby，也足够使我们愉悦，它处理身边的琐事时是那么的方便，聪明的人绝不会使用 C 语言编译，链接产生可执行程序并处理那些本可以使用 Ruby 不超过 10 行代码就可解决的问题。最后，很庆幸，有 Python 这样一门简单的语言，它真的很简单，而且经过了时间的考验，越来越多的人开始使用 Python，它的语法简单的超乎你的想象。

就各种语言的特性，以学习为目的，坚持简单为原则，本系统选择 Python 为参考语言，通过对其设计思路的了解开发一门新型语言 EasyFront。

第二章　前端设计基础

2.1 预处理

对 c 语言而言，预处理是十分重要的一个环节，在预处理阶段要对宏定义、文件包含、条件编译等进行处理，预处理阶段一般只是简单的进行替换，一般从主文件切入，碰到所有#include 语句将所对应的文件包含入当前编译的文件中，并在内存中记录所有#define 语句，凡是在以后的过程中碰到在记录中的词素便全部替换，在此过程中对条件编译不符合部分进行删除，对符合部分进行保留，在这一步骤中，将所有注释全部删除，其实不是删除，而是使用空格或空行替换，只保留有用代码，处理完这一步之后，将处理后的数据交给词法分析处理。

2.2 词法分析

词法分析是编译的第一阶段。词法分析器的主要任务是读入源程序的输入字符，将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素。这个词法单元序列被输出到语法分析器进行语法分析。词法分析器通常还要和符号表进行互交。当词法分析器发现了一个标识符的词素时，它要将这个词素添加到符号表中。在某些情况下，词法分析器会从符号表中读取有关标识符种类的信息，以确定向语法分析器传送哪个词法单元。

词法分析器在编译器中负责读取源程序，因此它还会完成一些识别词素之外的其他任务。任务之一是过滤源程序中的注释和空白，另一个任务是将编译器生成的错误消息与源程序的位置联系起来。例如，词法分析器可以负责记录遇到的换行符的个数，以便给每个错误消息赋予一个行号。

词法分析过程中使用输入缓冲往往能够提高程序执行效率，有时候还需要一个双缓冲，有些语言的词法分析器还要能够回退以处理特殊情况的发生。在词法分析过程中大多数编译器使用有穷状态机完成对词素的识别，一般会将词素文法规则定义在代码中，使用控制语句完成自动机的互相转换，也有一部分编译器使用外部定义方式，使用程序解析文法规则，保存 NFA 的每个状态，将其转换成 DFA，并对其压缩，最终实现自动化的处理，当然也可以使用 lex 系统简化词法分析工作，lex

系统和 yacc 系统的配合使用可以方便的生成编译器前端，smalltalk 和 php 语言都是通过这种方式实现的。

2.3 语法分析

词法分析得到词素之后将其传递给语法分析器，语法分析器负责对每个词素进行上下文匹配，查看其是否符合语言规则，编译器中常用的方法可以分为自顶向下和自底向上的语法分析技术，顾名思义，自顶向下的方法从语法分析树的顶部（根节点）看是向底部（叶子节点）构造语法分析树，而自底向上的方法则从叶子节点开始，逐渐向根节点方向构造。语法分析时总是按照自左向右的方向来扫描分析的。

在语法分析过程中若遇到错误可以采取两种策略来处理，分别称为恐慌模式和短语层次模式，恐慌模式一旦发现错误就不断的丢弃输入中的符号，一次丢弃一个符号，直到找到同步词法单元集合中的某个元素为止。而短语层次模式则在遇到错误时，语法分析器可以在余下的输入上进行局部性纠正。

对程序设计语言，大部分都属于上下文无关文法，即可以使用自顶向下的分析技术也可以使用自底向上的分析技术。一般的在自顶向下的语法分析时，对文法消除左递归后，求出其 FIRST 和 FOLLOW 集合，采用预测分析技术便可完成语法分析，但是，采用表驱动的预测语法分析技术时会占用交大的空间，一般的将其改为链表方式的较为合适。对于自底向上的语法分析技术，其实质是一个规约的过程，采用一个语法分析栈，对规约过程中的每一步进行记录，并对栈顶元素进行查表规约，在特殊情况下，很可能会出现移入规约冲突或规约规约冲突，这要求在进行语法分析的过程之前要对其二义性进行处理，一般情况下使用 LR 语法分析器效果较好，但其相对庞大，人工实现非常困难，可以借助 yacc 系统完成。

2.4 语义分析和中间代码生成

语义分析一般是分析源程序的含义，并作相应的语义处理。程序的含义涉及两个方面，即数据结构的含义与控制结构的含义。数据结构的含义主要指与标识符相关联的数据对象，也即量的含义，不言而喻，量涉及类型与值，值在运行时刻确定，而类型则由程序中的说明部分来规定。不同的数据对象，有不同的机器内部表示，因此有不同的取值范围，对它们所能进行的运算也将是不同的，显然只有具有相同类型，因此具有相同机内表示的数据对象，或符合特定要求的数据对象才能进行相应的运算。确定标识符所关联的类型等属性信息，进行类型正确性的检查成为语义分析的基本工作之一。概括起来语义分析的基本功能如下：

1) 确定类型
2) 类型检查
3) 识别含义，并作相应的语义处理
4) 其他一些静态语义检查

语义分析部分以语法分析部分的输出（语法分析树或其它等价的内部中间表示）为输入，输出是中间表示代码，甚至是目标代码。一般情况下，语义分析部分仅产生中间表示代码，即语义分析与目标代码生成分为两遍来进行。尽管语义分析与目标代码的分开，使编译程序开发难度降低，但对于语义分析，它不像词法分析与语法分析分别基于正则文法与上下文无关文法，可以形式地描述，已形成系统的形式化的算法，可以按照机械，甚至自动的方式构造词法分析程序与语法分析程序。

第三章　原型系统的设计与实现

3.1 EasyFront 语言设计基础

本系统描述的编译器前端是在 Python 语言的基础上诞生的，Python 作为动态语言的一员，其简单的语法最为人们喜爱，而在这简单的语法规则背后蕴藏了多少秘密呢（或者更确切的说是宝藏），为了能够完整详细的掌握其内幕，本系统首先改进其文法，删除其中比较复杂的语法规则，取其子集，来完成一套完整的编译器前端。

3.1.1 数据类型

EasyFront 的数据类型主要包括三种，基本类型，元组类型和映射类型。

基本类型主要包括：整型，实型，字符串型，布尔型

元组类型：tuple[1]

映射类型：map[2]

上面三种类型在设计系统的时候就开始考虑了，其中加入 tuple 有些牵强，在最出设计的时候总是围绕着词法分析与语法分析转，从没有考虑过语言成型后的功能影响，但经过经验的积累，现在已确定 tuple 是多余的，取而代之的应该是 list 类型，相对 tuple，list 有很高的灵活性，是设计一门语言时选取容器的首选。不得不说的时，后面的工作阻碍了对 tuple 和 map 的设计，其最终没有被实现，因为对动态语言而言，不得不需要一个强有力的 object 基类，这个基类是表现动态特性的基石，然而在设计的时候遇到了种种问题，这个 object 类设计的并不是很合理，许多功能暂时无法实现，所以，不得不承认此系统真正的工作并没有完成。做为动态语言，所有的符号类型都是在系统运行时刻判别的，所以，并不需要静态语言那些前端复杂的实现，在 sourceforge 上有一个为 pascal 语言打造的轻量级编译系统 Neo pascal[1]，作者相当详细的介绍了静态语言类型的设计[2]，给本系统的设计带来了很大的启发。

3.1.2 保留关键字

EsayFront 共保留 26 个关键字，其中关键字全部首字母小写，这与 Python 不同，另外去掉了 Python 中的 with，yield 等难以上手的关键字，另外值得注意的是 Python 中的函数定义使用的是 def 关键字，而这里使用的是 fun 关键字，异常接受由原来的 except 改为了 C++的 catch，还保留了 Python 中的 lambda 表达式，这是一种十分有用的表达式。

表 3-1-1 EasyFront 保留关键字

ture	false	fun	class	if	elif
else	for	while	each	in	catch
as	finally	break	continue	and	or
xor	not	lambda	import	del	try
null					

[1] 做为动态语言，列表要比元组表现出更好的特性，但其相对难实现，如果感兴趣，读者可尝试实现 Python 中的 list。

[2] 这里的映射其实就是 Python 中的字典（dict），由于笔者使用最多的语言是 C++，所以所有的关键字都以 C++保留名字命名。这里的 map 正式 STL 中的一员，而 tuple 日常编程用的也不是太多，它属于 boost 库中一员。

3.1.3 运算符

表达式是描述计算规则的一种语言结构，它由操作数、操作符及圆括号组成。操作符主要包括算术运算符、逻辑运算符和关系元算符。与 Python 一样 EasyFront 同样没有三目运算符，只有单目和双目运算符。其中值得注意的是**，//，//=这三个运算符，对 C++程序员来说这三个运算符应该很陌生。

表 3-1-2 EasyFront 的运算符

分类	运算符	目数	优先级	功能说明
算术	+	单目	2	取正
	-			取负
	~			去反
	**	双目	1	取平方
	*		3	乘法
	/			除法（保留余数）
	%			取余
	//			除法（四舍五入余数）
	+		4	加法
	-			减法
	>>		5	右移
	<<			左移
	&		6	按位与
	^		7	按位异或
	|		8	按位或
关系	<		9	小于
	>			大于
	==			等于
	<=			小于等于
	>=			大于等于
	!=			不等
	in			存在
逻辑	not	单目	10	非
	and	双目	11	逻辑与
	or		12	逻辑或
	lambda		13	lambda 表达式
关系	=		14	等于
	+=			自加
	-=			自减
	*=			自乘
	/=			自除
	%=			自除（四舍五入余数）
	&=			自余
	|=			自按位与
	^=			自按位或

	<<=			自按位异或
	>>=			自左移
	**=			自右移
	//=			自取指

EasyFront 的优先级控制的比较严格，从表 3-1-2 中可以看出共有 14 个优先级，数字越小的优先级越高，这也同时意味着，它处在语法树的最下层，也就是说对赋值运算符，全部在语法树的顶部。

3.1.4 EasyFront 文法定义

一个优秀的文法规则很难凭一个人的想象力凭空产生，当下流行的诸多程序设计语言多是在前人的基础上发展而来的，JAVA 就是在 C 的基础上添加了面向对象特性，形成的一种新的语言。对于 python 的文法规则可以从其官网上找得到，下载的源码包中就含有其文法定义文件。表 3-1-3 列出了一些简单的例子，从这些例子中我们可以看出 EasyFront 的语言文法规则基本上与 python 的相同，拿变量定义来说，使用即定义，是动态语言最大的特性，从表 3-1-3 例子中可以看出，对于布尔表达式，我们可以从表 3-1-2 中知道 EasyFront 的优先级控制的很严格，所以我们在程序中很少使用括号来明确不同表达式的优先级。

表 3-1-3 EasyFront 的语句例子

语句	例子
注释	# hello ,this is a notation

变量定义	a = 3 #定义一个整型变量，值为 3
	b = 3.3 #定义一个实型变量，值为 3.3
	c = 'I am a string' #定义一个字符串变量，值为 I am a string
	d = " I am a string too" #定义一个字符串变量，同上

布尔表达式	a > b && b >=c || not d #a > b 为真且 b >= c 与 not d 有一为真返回真，否则返回假，或的优先级大于与的优先级
tuple 语句定义	tuple_val = ('a','b','c') #定义一个由字符组成的三元组
map 语句定义	map_val = ['a' = 1,'b' = 2,'c' = 3] #顶一个由字符到标志号的映射

if 语句	if a > b:
	  cout ('a > b')
	elif a > c:
	  cout ('a <= b')
	else:
	  cout ('at else scope')

while 语句	while a > b:
	  a ++
	  if a == 0:
	  break
	else:
	  cout ('here is the tail of while statement,you can do something easy if you want catch some signal')

for 语句	for a in tuple_list_map:
	  # for 循环体
	  cout (a)

	cout (‘ ‘)
import 语句	import module_indentifier #引用某个模块 import module_indentifier.concerete_class #引用某个模块的一部分

函数定义	fun fun_identifier (parameter1,marameter2):
	    # 函数体，函数参数以逗号分割，以冒号结束，无括号
	    cout (‘here is function body’)

类定义	class class_identifier base_calss:
	    # 类定义体
	    inner_data1 = 1
	    inner_data2 = class_identifier2 para
	    fun __init__(para1,para2):
	        #构造函数，如果需要
	        cout (‘here is the constructor’)
	    fun method_1 (para1,para2,para3):
	        #方法一，有三个参数
	        cout (‘there is something done at method_1’)
	    fun method_2 ():
	        # 方法二，无参
	        cout(‘method_2 has no parameter at all’)

函数调用	return_value = fun_identifier(parameter1,parameter2) #函数调用将返回值赋予 return_value 变量

类调用	class_instance = class_indentifier parameter
	# 构造类
	return_value = class_instance.method_1 (para1,para2,para3)
	# 访问内部方法
	value = class_instance::inner_data
	# 访问内部数据

异常捕捉	try :
	    f = fopen (‘blash.txt’,file::F_READ_ONLY)
	catch IOError as e:
	    cout (e.what)

EasyFront 的文法设计相对简单，最初设计的时候企图将整个语言的文法规则全部设置为正则文法，根据 smalltalk 语言的文法启示，如果存在一种语言完全不用括号，而其全部以文法上下规则中的缩进来判断文件结束与否，这样可以简化编程人员在编程时频繁使用上档键而减慢程序的书写速度问题，然而在实现的过程中，我们不得不佩服经典的语言文法规则，若没有括号则会使函数调用与普通表达式混淆，加大了程序语言的实现困难性，在后期经过多次尝试，最终将语言的文法规则定义如下，如表 3-1-4 所示，其全部以扩展的正则表达式表示，文法的根节点为 file_input。

表 3-1-4 EasyFront 文法定义

序号	产生式左部		产生式右部
001	file_intput	→	(NEWLINE | stmt)* ENDMARKER
002	stmt	→	simple_stmt | compound_stmt | suite
003	simple_stmt	→	small_stmt (';' small_stmt)*[';'] NEWLINE

004	compound_stmt	→	if_stmt | while_stmt | for_stmt | try_stmt | fundef_stmt | classdef_stmt
005	suite	→	simple_stmt | NEWLINE INDENT stmt+ DEDENT
006	small_stmt	→	assign | augassign | del_stmt | import_stmt
007	assign	→	expr ('=' expr)*
008	augassign	→	expr augmentassign expr
009	augmentassign	→	'+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//='
010	del_stmt	→	'del' NAME
011	import_stmt	→	'import' NAME
012	if_stmt	→	'if' expr ':' suite ('elif' expr ':' suite)* ['else' ':' suite]
013	while_stmt	→	'while' expr ':' suite ['else' ':' suite]
014	for_stmt	→	'for' expr 'in' expr ':' suite ['else' ':' suite]
015	try_stmt	→	'try' ':' suite ((catch_clase ':' suite) + ['finally' ':' suite] | 'finally' ':' suite)
016	catch_clase	→	'catch' [expr ['as' NAME]]
017	fundef_stmt	→	'fun' NAME ([parameters]) ':' suite
018	classdef_stmt	→	'class' NAME ':' suite
019	expr	→	con_expr | parameters |

			small_expr
020	parameters	→	parameter_atom (',' parameter_atom)*
021	parameter_atom	→	nostar_parameter | tuple_parameter | map_parameter
022	nostar_parameter	→	NAME ['=' expr]
023	tuple_parameter	→	'*' NAME ['=' expr]
024	map_parameter	→	'**' NAME ['=' expr]
025	con_expr	→	logical_or_expr | lambda_expr
026	lambda_expr	→	'lambda' [parameters] ':' con_expr
027	logical_or_expr	→	logical_and_expr ('or' logical_and_expr)*
028	logical_and_expr	→	not_expr ('and' not_expr)*
029	not_expr	→	'not' not_expr | comparison
030	comparison	→	small_expr (comp_op small_expr)*
031	comp_op	→	'<' | '>' | '==' | '<=' | '>=' | '!=' | 'in' | 'not'
032	small_expr	→	xor_expr ('|' xor_expr)*
033	xor_expr	→	and_expr ('^' and_expr)*
034	and_expr	→	shift_expr ('&' shift_expr)*
035	shift_expr	→	arith_expr (('<<' | '>>') arith_expr)*
036	arith_expr	→	term (('+' | '-') term)*
037	term	→	factor (('*' | '/' | '%' | '//') factor)*
038	factor	→	('+' | '-' | '~') factor | power
039	power	→	atom ['**' factor]
040	atom	→	(tuple_expr | map_expr | number_expr | keyword_expr | string_expr | name_expr)
041	tuple_expr	→	'(' tuple_variables ')'
042	map_expr	→	'[' map_variables ']'
043	tuple_variables	→	expr(',' expr)* [',']
044	map_variables	→	(expr '=' expr)(',' expr '=' expr)*
046	number_expr	→	NUMBER
047	string_expr	→	STRING+

048	keyword_expr	→	'null' | 'false' | 'true'
049	name_expr	→	NAME (fun_invoke | module_invoke)
050	module_invoke	→	('.' NAME)*
051	fun_invoke	→	'(' parameters ')'

3.2 词法分析 Scanner 设计

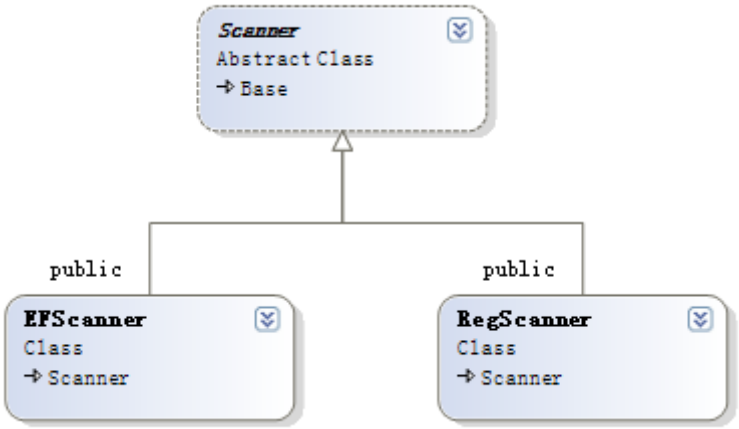

图 3-2-1 Scanner 类图

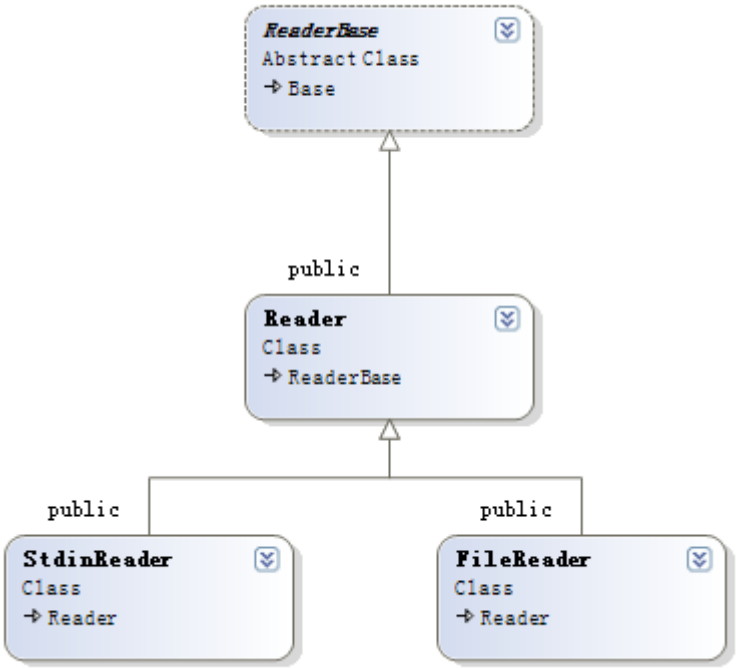

图 3-2-2 Reader 类图

词法分析主要完成词素的提取，它也同时对可知的词素拼写错误进行检测，其类设计结构图如图 3-2-1 所示，Base 是核心基类，虽然其没有任何实现，但是对整个工程是有意义的，Scanner 定义了一些抽象方法，并对其中一些提供了默认的实现，其拥有一个 Reader 成员，该 Reader 负责从缓

存或文件中读取程序源文件，每读取一个字符，将其行号及列号信息存在 Token 中的各个状态中，并在词法分析程序索取时返回一个包含有这所有信息的 Token，方便程序错误处理和打印。Scanner 有两个子类，对本系统只用到了 EFScanner，RegScanner 在最初设计时使用，后来丢弃，只使用 EFScanner。Scanner 中的主要方法为 nextToken，它负责每次从给定流中得到下一词素。在 nextToken 方法中分别调用其它方法，包括 isidentifierstart 判断是否是标识符开始（不包括数字），isidentifier 判断是否是标识符，isdigit 判断是否是数字，procspaces 处理空白等方法。

Reader 类是另一个重要的底层工具，其主要提供读取方法，其类图如 3-2-2 所示，它的实现对上层是不可见的，Scanner 只是负责向 Reader 索取，为提高程序读取速度，有必要为其设置缓冲区，这对处理大程序是至关重要的，另外如果有必要，还需考虑双缓冲，以便处理程序回读文件。在本系统中 Scanner 每次调用 Reader 的 getch 方法得到一个字符，Reader 类中的 getch 方法并不判断一个字符的类型，只负责检查是否到文件末尾，另外 readLine 和 read 方法给系统扩展带来了极大的便利。

因为 Python 是一个严格控制缩进的语言，所以在词法分析的时候使用了一些技巧，其使用 pendin 变量来记录是正缩进还是反缩进。每次在构造一个新的 token 并返回后，对 pendin 重新递减或递增，其恰好处在平衡状态，从而其一直是正确的。

另外在系统实现时，并没有添加预处理阶段，所以对于注释的删除一并都留给了词法分析，从而对系统实现带来了一点麻烦，当碰到新行符时并不能很好的处理注释与新行符的关系，最后，系统维护了一个字符缓存队列，保存了上次未使用的字符，方便下一次读取，从而保证了程序的正确性。

另外对由多个字符组成的运算符增大了我们的实现工作，其在维护了一个字符队列后也被顺利解决。

3.3 语法树构造及 visitor 类设计

语法分析由 parser 类来完成，parser 的类图如图 3-3-1 所示，虽然本系统只使用一个 parser，但在设计之初还是为其留下了接口。其构造函数传入 Scanner，parser 的入口函数为 module，其实质是每个入口是一个模块，当调用一个模块时，从语法树的根节点 file_input 出发匹配所有的词素，所有的过程都是一个递归的过程，当到达语法树的叶子节点时，仍为匹配到则提示错误。parser 类中的所有方法都是一个节点，所有节点之间嵌套调用，严格按照文法定义中的文法规则书写代码，通过这种方式构造的语法分析树存有大量的冗余，每次遍历语法树时几乎都要到达叶子节点才可知遍历结果，所以若系统需要改进的话，则需要对冗余语法树进行剪枝。

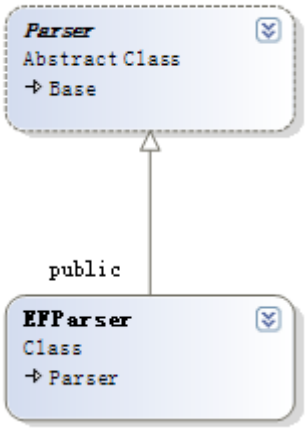

图 3-3-1 parser 类结构图

parser 的另一个有用的方法是 move 方法，move 只有简单的一句索取语句，若系统需要对每个 token 做特殊处理的话，可以在这个方法体里面添加更多的功能。

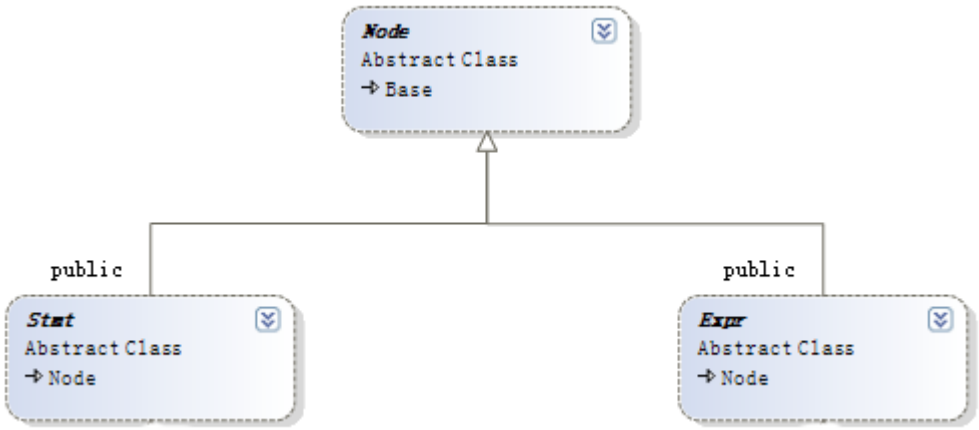

图 3-3-2 抽象语法树

在 EasyFront 中使用抽象语法树作为中间表达方式，该类根据 ASDL（The Zephyr Abstract Syntax Description Language）[3]被设计，所以其显得十分庞大，总体其按照两部分实现，如图 3-3-2 所示

语法分析树的节点信息全部定义在 node.h 中，根节点为 Node 节点，所有的节点分为语句节点和表达式节点，其全部派生自 Node 节点，如图 3-2-4 所示，语句节点包含了语言所持有的所有语句信息，包括 DelStmt，ImportStmt，AssignStmt，IfStmt，ElifStmt，ElseStmt，ForStmt，WhileStmt，TryStmt，CatchStmt，FunStmt，表达式节点包含了基本的表达式信息，包括 ParametersExpr，ConExpr，LambdaExpr，LogicalExpr，OrExpr，XorExpr，AndExpr，ShiftExpr，TermExpr，PowerExpr，FactorExpr，TupleExpr，MapExpr，NumberExpr，StringExpr，KeyWordExpr，NameExpr 等节点信息。另外一个很有用的节点结构为 Suite 节点，在系统中将所有以一个正缩进起始的区间称为一个 Suite，在遍历语法分析树时，将通过 Suite 来判断一个区块的开始与结束。

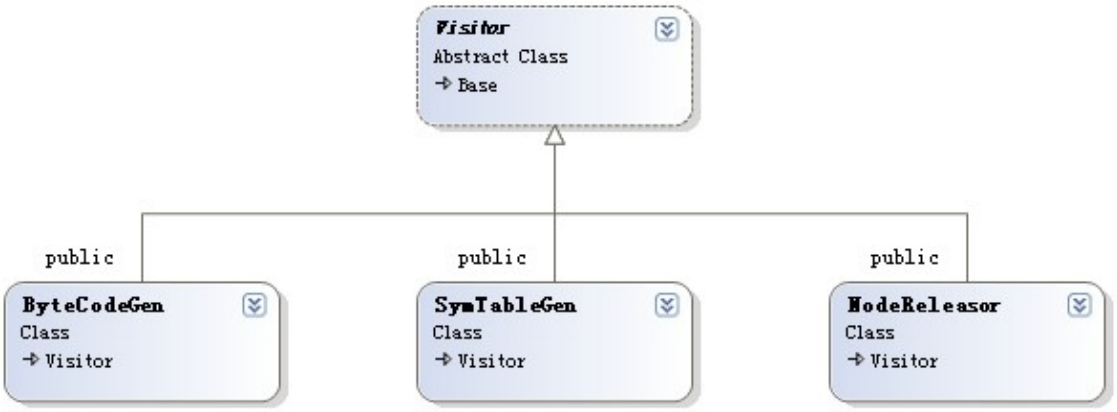

图 3-3-3 Visitor 类的设计

对语法树节点的设计全部按照文法定义来实现，其类继承及实现方式稍有所变化，整体结果可在本系统的源码中查看到，其有将强的层次结构，较容易理解。parser 的遍历其实就是一个构造节点的过程。

与语法树有密切关系的另一个类是 Visitor 类，该设计模式在《设计模式--可复用面向对象软件的基础》[4]这一本书中有很详细的介绍，运用在这里来遍历语法分析树恰到好处。其设计实现如图 3-3-3 所示，其有三个子类，ByteCodeGen 负责生成字节码，SymTableGen 负责生成符号表，NodeReleaser 负责释放语法分析树的节点，在这三个子类中遍历语法树的顺序有所不同，根据节点的类型会调正选择前序遍历还是后序遍历，语法分析树的每个节点对应一个方法体，其有各自的实现。该类略显庞大，代码冗

余量很多，但似乎只有这样才可以使系统更容易被扩展，更易读懂。

3.4 符号表结构

Python 的符号信息包括 FREE，GLOBAL，LOCAL，CELL 四种，其在不同作用域表现出不同的特性，从而其实现较为复杂。本系统将其简单化，只是简单的保存每个节点信息到该符号名的映射，另外对符号表进行分类，包括 CLASS，FUN，MODULE，OTHER 四种符号表结构，其主要方法对应于 enter_scope 和 out_scope 方法，这两个方法在遍历符号表时分别进入可离开不同符号表。

生成符号表的工作由 SymTableGen 类来完成，其负责遍历语法树的每个节点，碰到新的符号信息时加入符号表，并记录其节点所对应的位置信息，该类的缺陷是，符号表中的符号与节点的地址对应，当节点被释放时，该符号表不能再使用。

3.5 字节码定义及其生成器

在系统运行时维护一个执行栈 ExeStack，其记录程序运行时的结果，程序执行的过程是一个入栈与出栈的过程，栈中并不真正的保存数据，而是保存数据的 ID，根据 ID 在相应表中查找，然后将其调出，根据指令的功能作出相应的回馈。其信息表如表 3-5-1 所示。

表 3-5-1 字节码信息表

StrTable	存所有字符串	存由值到 ID 之间的映射，由于每份值在内存中只保留一份，所以不存在被覆盖的问题
IntTable	信息存所有整型信息	
FloatTable	存所有浮点数信息	
NameEntry	存所有的名字信息	存由 ID 到值之间的映射，方便查找，打印
ValueEntry	存所有的值信息	
ConstStringEntry	存所有的常量字符串信息	
IntEntry	存所有的整型信息	
DoubleEntry	存所有的浮点数信息	
CodeObjectEntry	存所有的字节码信息（函数）	

所有字节码以枚举类型定义，在 Opcode.h 中可查找到，如表 3-5-2 所示。EasyFront 的字节码完全使用 Python 的字节码，没有做任何更改，在这里只使用了其的子集。整个字节码执行时是一个压栈与出栈的过程，部分操作码含有参数。

字节码生成使用 ByteCodeGen 对象来完成，其是 Visitor 的子类，也是一个遍历语法树的过程，相对而言其代码比较庞杂，在其过程中使用另外一种设计模式 Builder，其负责在 ByteCodeGen 遍历语法树的过程时将字节码记录然后一并返回，其主要方法包括：new_block 进入一个新块，addop 添加一个新的指令，addop_i 添加一个含有一个参数的指令，addop_j 添加跳转指令。对于 CodeBuilder，为了能够生成不同的字节码表示方案，系统内也为其预留了接口，方便扩展。

由 CodeBuilder 生成的字节码恰好构成了一个 CFG（控制流图），该 CFG 的跳转位置并没有填充，在遍历完语法树之后，调用 back_pach 方法对 CFG 进行填充，最后将所有指令反向排列，方便执行，最后将其整体以 CodeObject 对象返回。

在对 CFG 回填后，可以在这里对代码进行优化，但是系统这里设计的并不是很好，许多优化技术并不实用，GCC 使用 SSA（静态单一赋值）进行优化很值得借鉴，系统的后期工作应该在这里。

在产生字节码之后，语法树便不再有用，调用 NodeReleasor 对语法树所有节点空间进行释放，系统前期工作基本完成。

表 3-5-2 主要字节码说明

POP_TOP	弹出栈顶元素
DUP_TOP	复制栈顶元素
DUP_TOP_TWO	赋值栈顶两个元素
NOP	空操作
UNARY_NOT	一元否定操作，弹出栈顶元素，求反，然后重新压入栈中
BINARY_POWER	二元操作，求平方，弹出栈顶两个元素，以第一个元素做第二个元素的幂运算
BINARY_MULTIPLY	二元操作，乘法，弹出栈顶两个元素，将其相乘之后其结果压入栈中
STORE_NAME	将栈顶元素的值存入其参数ID所对应的信息表中
BINARY_ADD	二元操作，加法，弹出栈顶两个元素，将其相加后其结果压入栈中
BINARY_SUBTRACT	二元操作，减法，弹出栈顶两个元素，将其相减后其结果压入栈中
BINARY_FLOOR_DIVIDE	二元操作，地板除法，弹出栈顶两个元素，将其相除后其结果下去整压入栈中
BINARY_TRUE_DIVIDE	二元操作，除法，弹出栈顶两个元素，将其相除后其结果整压入栈中
STORE_MAP	根据其后参数n，弹出n个元素，然后以字典形式保存与信息表中
LOAD_INT	根据其后参数ID，在IntEntry表中查找该int对象，然后将其压入栈中
BINARY_LSHIFT	二元操作，左移，弹出栈顶两个元素，将第二个元素左移第一个元素值的位数
BINARY_RSHIFT	二元操作，右移，弹出栈顶两个元素，将第二个元素右移第一个元素值的位数
BINARY_AND	二元操作，按位与，弹出栈顶两个元素，将两个对象的值做按位与操作然后重新压入栈中
BINARY_XOR	二元操作，按位异或，弹出栈顶两个元素，将两个对象的值做按位异或操作然后重新压入栈中
BINARY_OR	二元操作，按位或，弹出栈顶两个元素，将两个对象的值做按位或操作然后重新压入栈中
BREAK_LOOP	中断循环语句
RETURN_VALUE	将栈顶元素弹出，作为函数返回值传入另一个模块中
DELETE_NAME	根据其参数ID查找信息表中的对象，然后将其从信息表中移除
LOAD_CONST	根据其参数ID从常量表中读取常量对象，然后将其压入栈顶
LOAD_NAME	根据其参数ID从名字表中读取该对象的值，然后将其压入栈中
COMPARE_OP	根据其参数op来确定使用哪个二元操作，并将栈顶两个元素弹出，求值后，将bool对象压入栈中
JUMP_FORWARD	根据其后参数pos直接跳转到代码块的任意位置
JUMP_IF_FALSE_OR_POP	弹出栈顶元素，判断其是否为假，若为假，则跳转到期参数pos的位置
JUMP_IF_TRUE_OR_POP	弹出栈顶元素，判断其是否为真，若为真，则跳转到期参数pos的位置
CALL_FUNCTION	弹出栈顶元素，作为函数对象来调用，并使用其参数argc的值弹出相应个数的元素，以参数形式传递给该函数对象
MAKE_FUNCTION	将当前模块以函数对象封装
LOAD_FLOAT	根据其参数ID在float信息表中读取该对象，并将其压入栈中

3.6 错误处理

对程序错误处理并没有做可恢复接口，采用直接抛出异常方法(这种方案严格意义上并不可取)，但在有些情况下还是可以通过捕捉某个异常来达到恢复，开始时本系统将异常处理按照图 3-6-1 所示类继承方式设计，其中包括拼写异常，语法异常，语意异常和内存管理异常，对不同模块使用不同异常类，然而在后面的实现过程中将重点放在了语言实现上，对错误回复花了很少的精力，所以其大部分功能处于相同的状态，最终在实现系统时将所有的异常处理都重新整合到了一个类中，相对

简单了很多，这个类可以在 Exception.h 中查看到，但是图 3-6-1 中的设计方式是值得我们思考的，或许配合某种设计模式会有更好的效果。

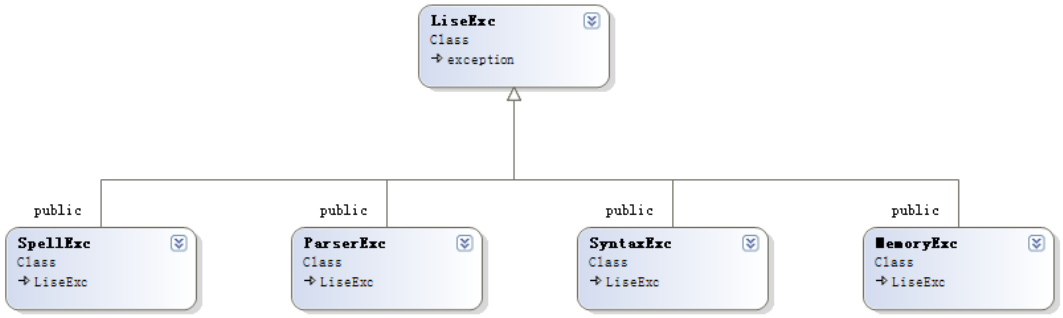

图 3-6-1 异常类

3.7 程序执行过程

字节码生成之后，便可以执行了。执行代码在 Executor.h 中定义，执行过程是顺序的，直到遇到跳转代码之后跳转到相应代码块继续执行，其每次索取一条指令，并判断该指令是否为空，然后根据字节码类型调用相应方法，执行过程相对简单，唯有函数调用设计欠缺考虑，以及未能实现 Object 基类功能，使得函数调用实现的有些迁强人意。

在执行过程中不断的申请新对象，并没有对其释放，整体内存泄漏严重，当稍微大的程序使之完全不能运行。系统整体性能有待提升。

第四章　系统运行结果测评与分析

4.1 系统测评

本系统含有两个版本，一个为 debug 版本一个为 release 版本。debug 版本将生成更多的调试信息，帮助我们完成代码调试工作。在 doc/test/目录下存有一部分测试代码，包括 bool 表达式的测试，函数测试，if 语句测试，while 语句测试，浮点数测试，字符串测试，还有一个完整的测试用例，该测试打印三位数的水仙花数，下面就以该用例来测试系统效果，该用例的完整代码如下：

```
1 # 打印 3 位数的水仙花数
2
3 a = 1
4
5 fun print_water_num(n1,n2,n3):
6       val = n1 * 100 + n2 * 10 + n3
7       result = n1 * n1 * n1 + n2 * n2 * n2 + n3 * n3 * n3
8       #cout("val:",val,"result:",result)
9       if val == result:
10              cout(val)
11
12 while a < 10:
13      b = 0
14      while b < 10:
15              c = 0
16              while c < 10:
17                      print_water_num(a,b,c)
18                      c += 1
19                      #cout("c:",c)
```

```
20          b += 1
21          #cout("b:",b)
22      a += 1
23      #cout("a:",a)
```

使用 TestLexical.exe 便可查看该代码被词法分析后的所有词素，运行方式为

TestLexical.exe water_num.ef

运行之后得到 output.txt，该文本文档中存有所有的词素信息，表 4-1 是一部分：

表 4-1 词法分析结果

TOKENFLAG	字符串	row	col
NEWLINE		2	0
NAME	a	3	1
EQUAL	=	3	3
NUMBER	1	3	5
NEWLINE		4	0
FUN_MARKER	fun	5	1
NAME	print_water_num	5	5
LPAR	(	5	20
NAME	n1	5	21
COMMA	,	5	23
NAME	n2	5	24
COMMA	,	5	26
NAME	n3	5	27
RPAR	)	5	29
COLON	:	5	30
NEWLINE		6	0
INDENT		6	4
NAME	val	6	5
……			

TOKENFLAG 的所有名称都可在 token.h 中找到，并有完整的注释，字符串为其在程序中出现的词素，在其后并有其出现的位置，程序在语法分析的过程中通过 TOKENFLAG 来构造语法树，并判断其正确性。

也可以使用 TestParser.exe 程序来查看语法分析时语法分析树构造过程，该程序结果并不是很清晰，这里不做演示。使用 release 版本的 EasyFront.exe 得到程序运行结果，生成字节码信息，运行方式为：

EasyFront.txt　water_num.ef

运行结果为：

```
153
370
371
407
```

字节码信息可在 debug.txt 文件中找到，从字节码信息中可以清楚的看到程序以块的方式组织运行，表 4-2 是一部分结果：

表 4-2 生成字节码结果

块号	字节码编号	字节码名称	字节码参数	字节码对象
0	0	LOAD_NAME	0	(n1)
	1	LOAD_INT	0	(100)
	2	BINARY_MULTIPLY	0	
	3	LOAD_NAME	1	(n2)
	4	LOAD_INT	1	(10)
	5	BINARY_MULTIPLY	0	

	6	BINARY_ADD	0	
	7	LOAD_NAME	2	(n3)
	8	BINARY_ADD	0	
	……			
1	0	LOAD_NAME	5	(cout)
	1	LOAD_NAME	3	(val)
	2	CALL_FUNCTION	1	
	3	JUMP_FORWARD	2	
	……			

从文件中可以看出函数定义与主函数是两个不同的模块，用横线分割，它们有各自的入口，主程序入口调用函数时将参数替换，并执行函数模块中的字节码，函数暂时不能返回任何值。

4.2 系统分析

从运行结果来看，系统的总体目标已达到，运行速度并不是很差劲，能够处理部分数据任务，包括：

1) 能够对字符串，浮点数进行处理
2) 能够支持基本一元及二元操作
3) 能够支持基本控制语句
4) 支持布尔表达式操作
5) 支持函数调用
6) 支持函数嵌套
7) 支持位运算

系统能够明确给出词法分析过程及字节码信息，达到了学习的目的。但是从系统内部分析，还存在很多问题，包括：

1) 未能实现 Object 基类，未能表现出真正的动态特性
2) 内存泄漏严重
3) 语法分析树冗余太多
4) 未能实现 list，map 等有用数据类型
5) 未做任何代码优化
6) 函数调用没有返回值
7) 模块之间切换问题较多

相比于 python 的代码，该系统全部使用 C++语言实现，其实是对 C 语言版的一个改装，运用了许多常用设计模式，虽然整体与 python 没有可比性，但其仍具有学习价值，系统整体架构较为整齐，若需扩展十分方便，总之，整个系统的性能还是很可观的。

总结与展望

本文对编译系统前端的设计作出了较全的分析，对常用数据结构及算法作出了讨论，并通过对原型系统的设计，详细讨论了在设计一个编译系统过程中遇到的种种问题。在词法分析过程中对 Linux 与 Windows 兼容做了简短的考虑，虽不支持 UTF8 编码，然其在 Windows 上的运行效果还算可以，总体达到了最初的目的。

通过对 Python 及 Gcc 的部分代码阅读，对编译原理中的知识有了更进一步的认识，以及对 ICC

和 Gcc 之间的比较，从一定程度上掌握了编译器实现词法分析，语法分析及语义分析的不同方案，以及不同 IR 表示对后期代码优化的影响。

另外对 C，C++，Java，Smalltalk，Ruby，PHP 和 Python 之间的文法规则进行对比，总结了不同文法规则在程序设计语言中的影响，实现类似 Python 这样有严格缩进要求的语言要比其它以括号约束代码开始与结束的语言要难，但是其仍然还是十分有意义的，对整个编译原理知识有很好的巩固。

总体上实现一个编译系统需要大量的人力，本系统只是一个小小的开始，若让整个系统可以真正运行，需要更大的功夫，其中的发展空间非常大，再者，加之代码优化部分需要更多的知识，这要求笔者不仅要有大量的编译原理知识，还要有很好的语言组织能力，以及要能很好的利用设计模式和面向对象设计基础，本系统存在的问题非常多，需要更多的思考和完善才可以。总之，若真正有需求，该系统需要更大的人力投入。

参考文献

[1] sourceforge. neopascal[EB/OL]. http://sourceforge.net/projects/neopascal/.

[2] 裘魏.编译器设计之路[M]. 机械工业出版社,2010.

[3] D.C.Wang, A.W.Appel, J.L.Korn, etal. The Zephyr Abstract Syntax Description Language[D]. Domain-Specific Languages Santa Barbara, California, October 1997.

[4] Gamma E, Heim R, Johnson R, etal. Design Patterns Elements of reusable Object-Oriented Software[M]. 1. 机械工业出版社, 2005-6-1.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers:Principles, Techniques, and Tools. Addison-Wesley, 1988.

[6] C. Allan, P. Augustinov, A. S. Christensen, L. Hendren,S. Kuzins, O. Lhot´ak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In OOPSLA ’05: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005.

[7] R. Alur, P. ˇCern´y, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In POPL ’05:Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 98–109, 2005.

[8] G. Ammons, R. Bodik, and J. Larus. Mining Specifications. In Proceedings of the 29th ACM Symposium on Principles ofProgramming Languages, pages 4–16, 2002.

[9] C. Anley. Advanced SQL Injection in SQL Server Applications, 2002.

[10] B. S. Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. In Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms,pages 541–550, 1995.

[11] J. Baker and W. C. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In Proceedings of the First International Conference on Aspect-Oriented Software Development,2002.

[12] T. Ball and S. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report.