



成绩

中国农业大学

课程论文

(2019 -2020 学年春季学期)

论文题目: 编译原理综合实验

课程名称: 《编译原理》

任课教师: 王耀君

班 级: 计算机 172

学 号: 201730401043

姓 名: 张靖祥

目录

1 绘制 PL/0 编译程序函数的流程图	1
1.1 PL/0 文法说明	1
1.2 PL/0 主函数	2
1.3 PL/0 词法分析	3
1.4 PL/0 语法语义分析	3
2 PY/0 语言设计	7
2.1 软件总体架构	8
2.2 PY/0 语言文法定义	10
3 PY/0 语言各模块设计与数据结构	11
3.1 基础符号模块	11
3.2 错误处理模块	12
3.3 词法分析模块	13
3.3.1 Public 类型	13
3.3.1 Private 类型	14
3.4 运行时基础结构	15
3.5 运行时管理	15
3.5.1 Private 成员变量	16
3.5.2 Public 成员函数	16
3.6 语法分析基础结构	17
3.7 函数初始化	17
3.8 语义及语法分析	17
3.9 主程序	19
4 运行设计与效果展示	19
4.1 简单语句	20
4.1.1 赋值类语句	20
4.1.2 控制类语句	21
4.1.3 删除变量类	23
4.1.4 import 类	23
4.2 复合语句	23
4.2.1 分支类语句	24
4.2.2 循环类语句	24
4.2.3 函数类语句	25
4.3 综合展示	25

1 绘制 PL/0 编译程序函数的流程图

PL/0 编译程序总体架构如下图（图 1.1）所示，该程序的主要语法分析过程采用了递归下降子程序法实现，通过调用词法分析程序，并在其中穿插虚拟机代码生成函数，为了将上下文有关文法转化为上下文无关文法，定义符号表进行符号管理，并对出错内容进行处理，最终转化为虚拟机代码进行解释执行。

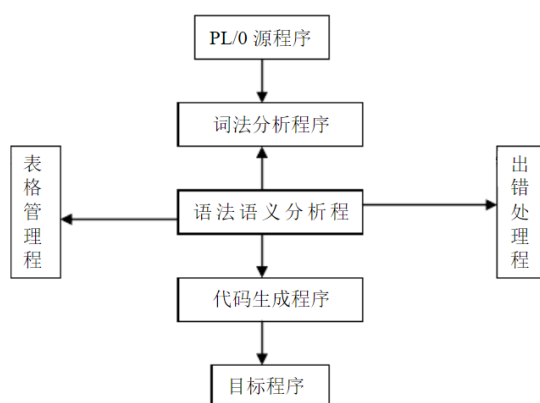


图 1.1

1.1 PL/0 文法说明

PL/0 的文法分析部分是在编译源程序的 `statement` 部分。这部分的代码可以总结出如下的结构。

<语句>::=<赋值语句> | <读语句> | <写语句> | <call 语句> | <if 语句> | <复合语句> |
<while 语句>); | .)

<赋值语句>::=<变量>:=<表达式>

<读语句>::=read(<变量>{,<变量>}) #注：这里的圆括号为句子的内部成分，不是 n 选一

<写语句>::=write(<表达式>{<表达式>}) #注：这里的圆括号，不是 n 选一

<call 语句>::=call<变量>

<if 语句>::=if<条件处理>(then | do)<语句> #注：没有 else

<while 语句>::=while<条件处理>do<语句>

<复合语句>::=begin<语句>{<语句>}end

<条件处理>::=<表达式><逻辑符号><表达式>

#注：源程序在条件处理的地方加入了无逻辑符号的判断，但是其词法分析并不能产生这种判断的代码，所以，无法进行是否为无逻辑符号的判断。并且，程序采用的分析方法中是不能有相同的前缀码的，如果加

入无逻辑符号的判断, 则 $\langle \text{条件处理} \rangle ::= \langle \text{表达式} \rangle$, 会产生相同前缀码。

$$\langle \text{表达式} \rangle ::= (+ \mid - \mid \varepsilon) \langle \text{处理项目} \rangle \{ (+ \mid -) \langle \text{处理项目} \rangle \}$$
$$\langle \text{处理项目} \rangle ::= \langle \text{处理因子} \rangle \{ (* | \backslash) \langle \text{处理因子} \rangle \}$$

<处理因子>::=<常量>|<变量>|<无符号常数>|(<表达式>)

$$\langle \text{逻辑符号} \rangle ::= = | > = | < = | > | < | \#$$
$$\langle \text{变量} \rangle ::= \langle \text{字母} \rangle \{ \langle \text{数字} \rangle \mid \langle \text{字母} \rangle \}$$
$$\langle \text{常量} \rangle ::= \langle \text{字母} \rangle \{ \langle \text{数字} \rangle \mid \langle \text{字母} \rangle \}$$
$$\langle \text{无符号常数} \rangle ::= \langle \text{数字} \rangle \{ \langle \text{数字} \rangle \}$$
$$\langle \text{字母} \rangle ::= a \mid b \mid \dots \mid y \mid z$$
$$\langle \text{数字} \rangle ::= 0 \mid 1 \mid \dots \mid 8 \mid 9$$

#注：其中<数字>一类的是词法分析已经完成的，直接调用即可

1.2 PL/O 主函数

PL/0 主函数执行流程如下图左（图 1.1.1），主函数初始化变量，并调用初始化变量的函数对符号表进行初始化，执行 **block** 函数（语法分析函数），如果语法分析成功产生虚拟机代码，则执行下图右（图 1.1.2）流程，关闭中间代码文件，调用解释器进行解释执行。

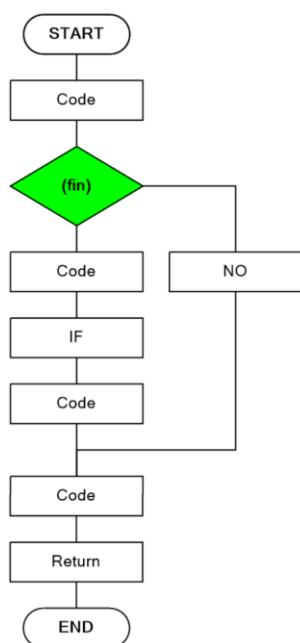


图 1.2.1

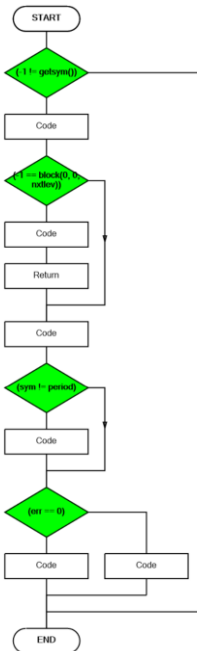


图 1.2.2

1.3 PL/0 词法分析

词法分析函数用于对词法进行分析，流程如下（图 1.2.1），将读入的文件按照词法进行最基础的分割，结果转化为保留符号、变量符号、数字、标点符号等。

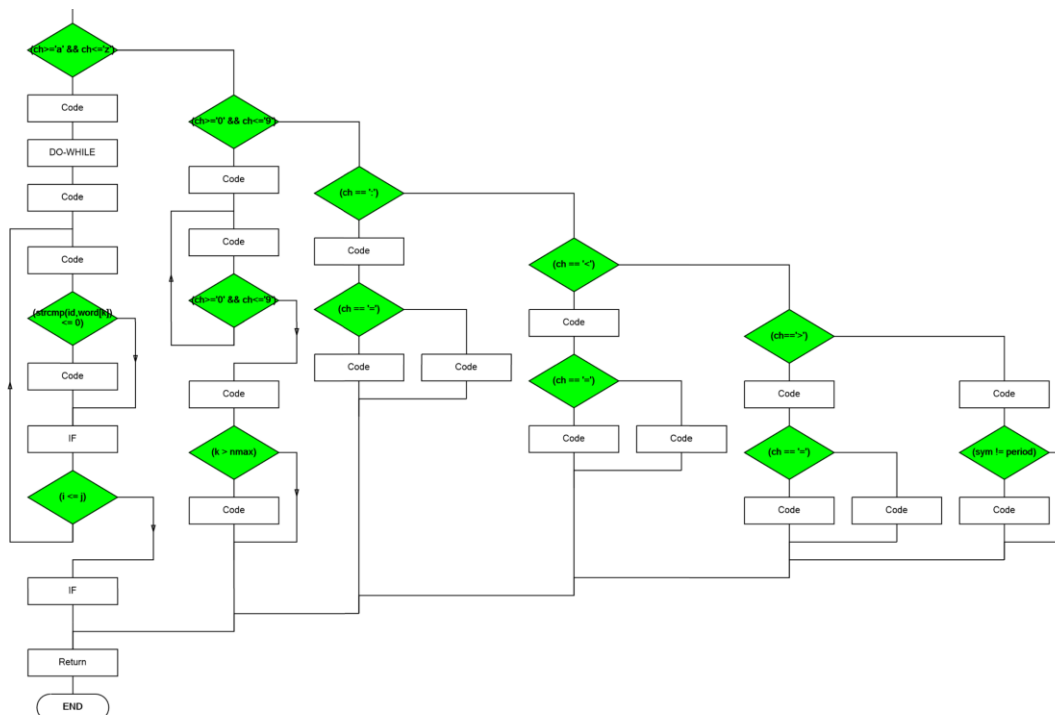


图 1.3.1

1.4 PL/0 语法规义分析

(1) 程序开始

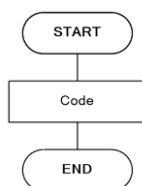


图 1.4.1

(2) procedure、var、const 语法

下图左 1 (图 1.4.2) $\langle \text{程序} \rangle ::= (\langle \text{常量声明} \rangle \mid \varepsilon) (\langle \text{变量声明} \rangle \mid \varepsilon) \{ \langle \text{函数过程} \rangle \} \langle \text{语句} \rangle$

下图左 2 (图 1.4.2) `<函数过程>::=procedure<变量>;<程序>`

下图左 3 (图 1.4.2) $\langle \text{变量声明} \rangle ::= \text{var } \langle \text{变量} \rangle \{, \langle \text{变量} \rangle \};$

下图左 4 (图 1.4.2) $\langle \text{常量声明} \rangle ::= \text{const} \langle \text{常量} \rangle = \langle \text{数字} \rangle \{, \langle \text{常量} \rangle = \langle \text{数字} \rangle \};$

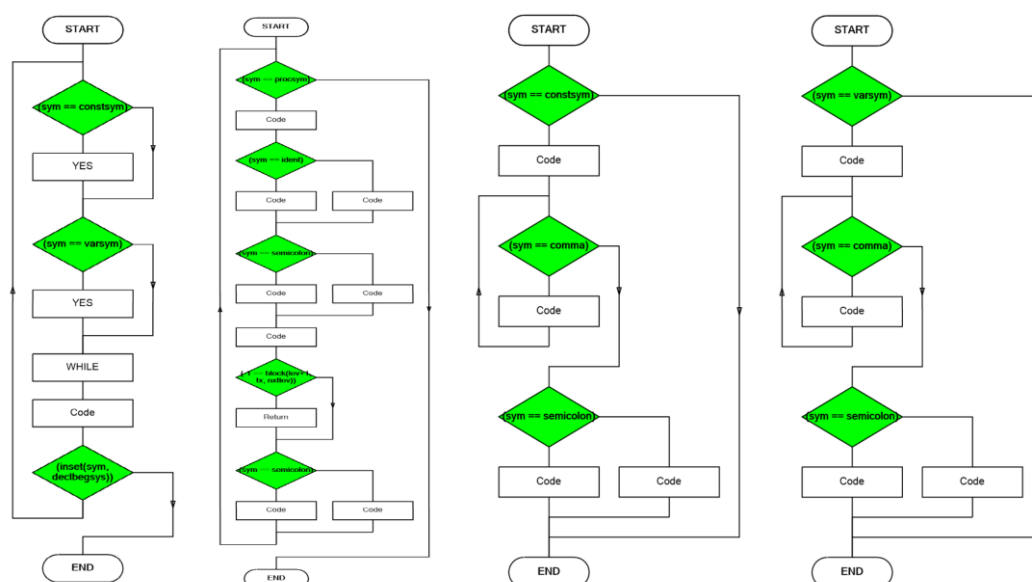


图 1.4.2

(3) 基本语句判断

如下图所示（图 1.4.3）

<语句>::=<赋值语句> | <读语句> | <写语句> | <call 语句> | <if 语句> | <复合语句> |
<while 语句>)(; | .)

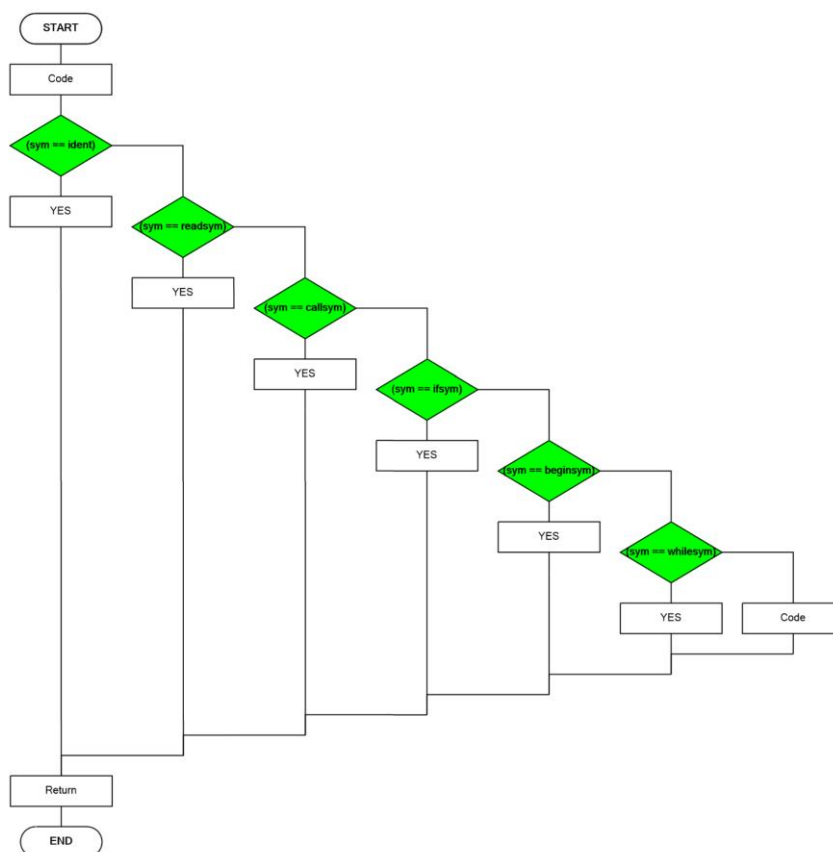


图 1.4.3

(4) 赋值语句、call 语句、if 语句

下图左（图 1.4.4） <赋值语句>::=<变量>:=<表达式>

下图右（图 1.4.4） <if 语句>::=if<条件处理>(then | do)<语句> #注：没有 else

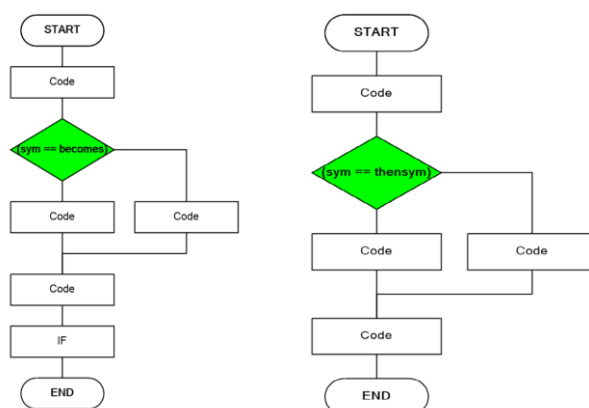


图 1.4.4

(5) read 语句、write 语句

下图左（图 1.4.5） <读语句>::=read(<变量>{,<变量>})

下图右（图 1.4.5） <写语句>::=write(<表达式>{<表达式>})

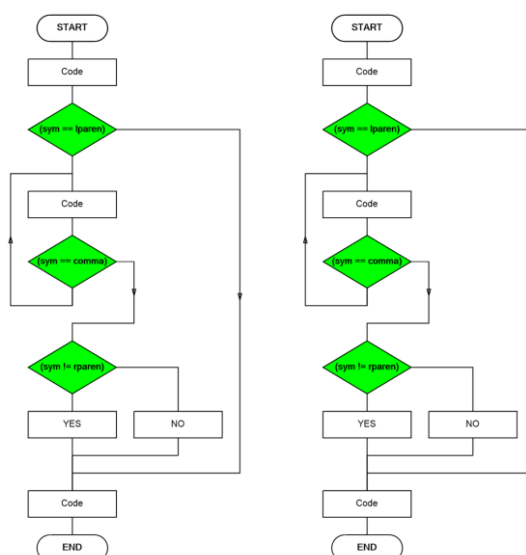


图 1.4.5

(6) while 语句、复合语句

下图左（图 1.4.6） <while 语句>::=while<条件处理>do<语句>

下图右（图 1.4.6） <复合语句>::=begin<语句>{<语句>}end

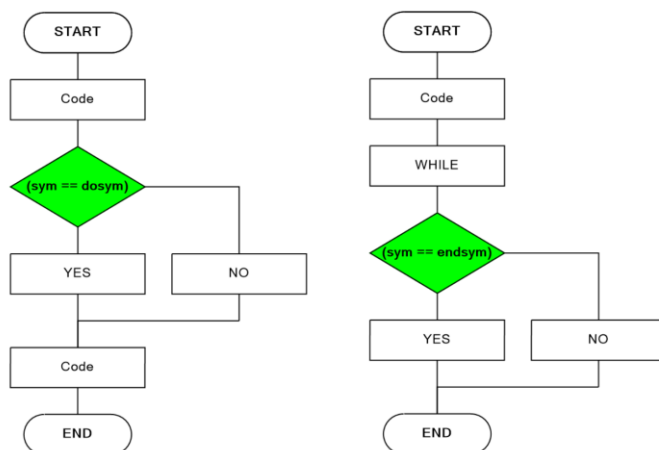


图 1.4.6

(7) 表达式处理

下图左（图 1.4.7） $\langle \text{表达式} \rangle ::= (+ | - | \varepsilon) \langle \text{处理项目} \rangle \{ (+ | -) \langle \text{处理项目} \rangle \}$

下图右（图 1.4.7） $\langle \text{处理项目} \rangle ::= \langle \text{处理因子} \rangle \{ (* | \backslash) \langle \text{处理因子} \rangle \}$

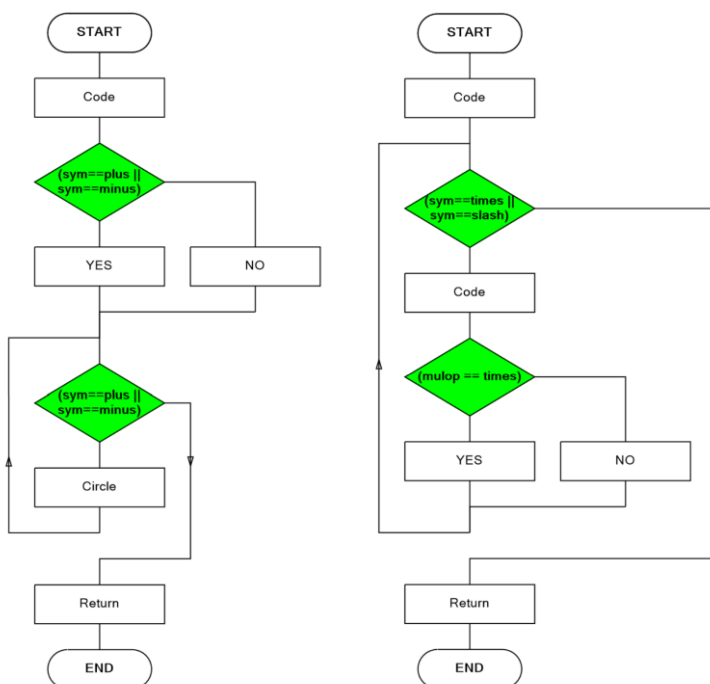


图 1.4.7

下图左（图 1.4.8） $\langle \text{处理因子} \rangle ::= \langle \text{常量} \rangle | \langle \text{变量} \rangle | \langle \text{无符号常数} \rangle | (\langle \text{表达式} \rangle)$

下图右（图 1.4.8） $\langle \text{条件处理} \rangle ::= \langle \text{表达式} \rangle \langle \text{逻辑符号} \rangle \langle \text{表达式} \rangle$

$\langle \text{逻辑符号} \rangle ::= = | > = | < = | > | < | \#$

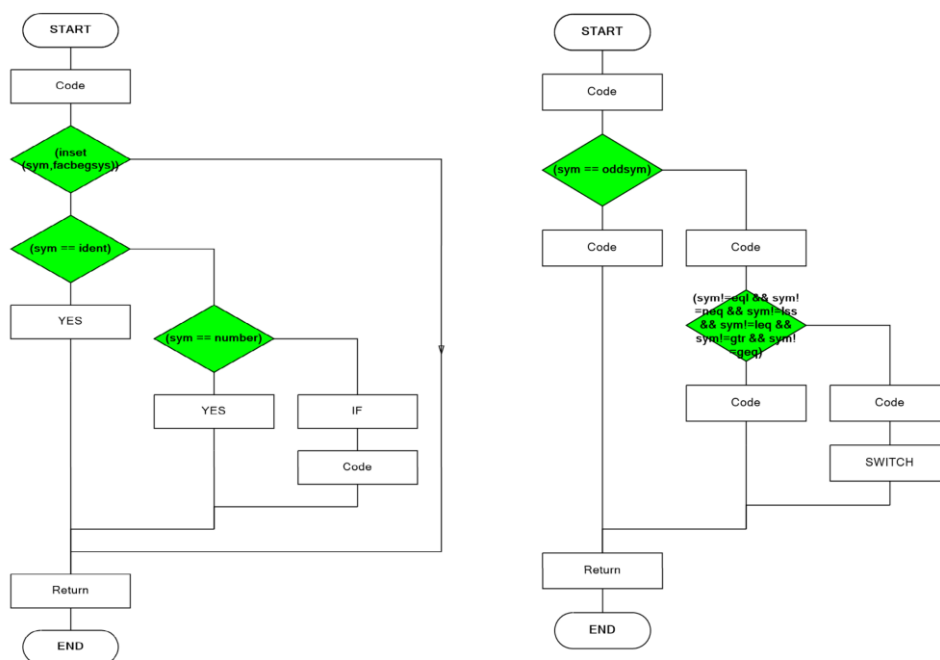


图 1.4.8

2 PY/0 语言设计

编写目的：PY/0 软件本着以个人兴趣爱好为基础，设计的编译原理实验课的大作业，在经过了程序框架的考虑后，希望以纯解释型语言的风格完成类似于 python 语法的编译器。包括了词法分析、语法分析、语义分析、出错处理、符号表管理这一系列模块的设计。

此报告附带了《PY/0 编译软件说明文档》，其中有更为详细的函数说明，本报告中只列举函数定义，不对函数具体实现过程进行分析。

其中包含了以下的功能设计：

1. 赋值类语句
2. and、or、not 关键字；
3. int、float、string 类型定义、引用、运算；
4. int、float、string 的向量类型定义、引用、运算；
5. break、continue、return 的支持；
6. del 删除变量；
7. import 引用语句；
8. if、elif、else 分支类结构；
9. while、for 循环；
10. def 函数声明；
11. 内联函数与用户自定义函数的调用；
12. print、input 格式化输入输出函数；

13. max、min 函数；
14. range 序列生成函数；
15. int、float、string 类型转换函数；
16. 出错处理机制。

参考资料：编译原理课件、PL/0 编译器源代码、python 语法说明文档。

2.1 软件总体架构

本软件所有模块的调用规则如下（图 2.1.1），为避免循环引用，我将部分模块进行了拆分处理。下图所示的所有模块均为头文件形式，并非类的继承关系。在后面中，我将详细的阐释各个模块的作用以及所有函数的设计说明。

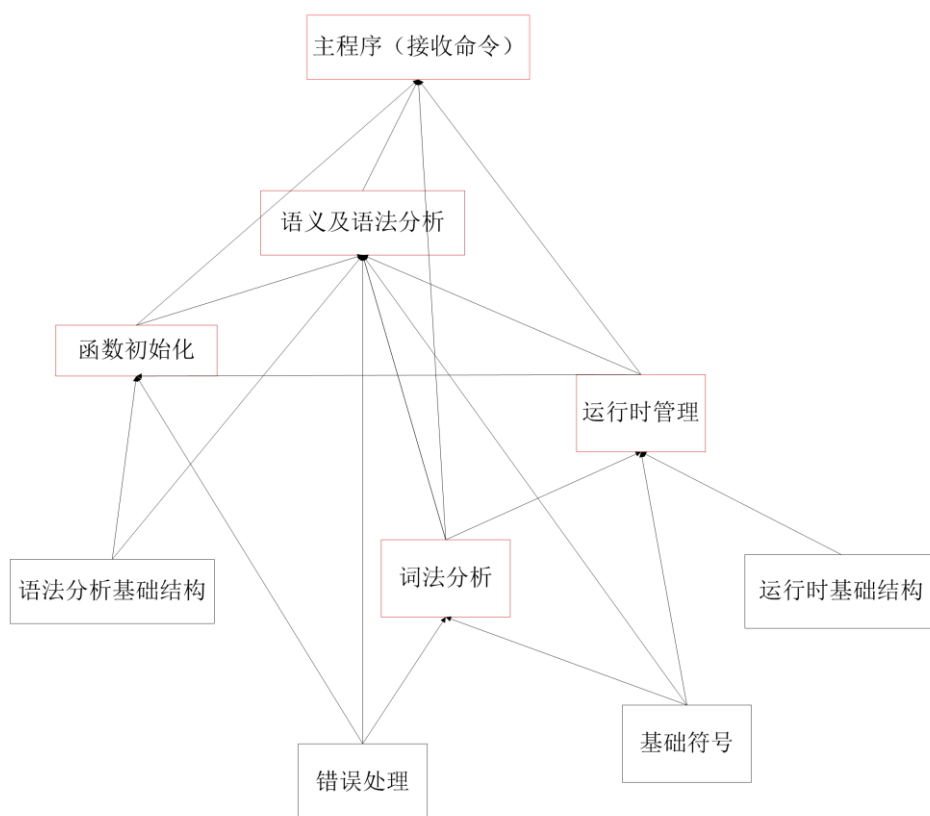


图 2.1.1

基础符号模块：为最基础模块之一，不调用任何其他模块，所有的模块几乎都调用此模块的内容，将抽象的数字转化为枚举类型，这也是我设计的第一个模块，且后续几乎无改动。

错误处理模块：为最基础模块之二，不调用任何其他模块，用于显示错误信息，并给出错误处理，需

要由其他模块调用该模块，选择适当的错误类型。由于时间有限，我只对关键的错误进行了十分简略的说明，报错内容较为简单。

词法分析模块：该模块用于分词，原本设计后来又进行了多次函数增加，为了满足更多的需求。提供的功能为：通过文件名对文件内容进行分词、获取下一个词、获取上一个词、跳过代码区、获取代码长度、获取当前代码位置、跳转代码。

运行时基础模块：此模块为运行时管理模块调用的子模块，内涵多种需要的结构。将其设计为一个子模块为了避免，循环引用的问题。

运行时管理：本文件管理所有的运行时状态，对 `phrase` 语法语义分析提供功能调用。所有函数原则上都是公开成员变量，只有变量为内部私有，提供的功能为添加变量到变量表、获取变量表中的变量对象、代码块层级提高、代码块层级降低、进入函数时保留暂时移出的变量、退出函数时将移出的变量放入变量表、删除变量、记录 `import` 的列表、展示整个变量表。

语法分析基础结构：此模块为语法与语义运行时管理模块调用的子模块，内涵多种需要的结构。本来并没有将其设计为一个子模块。但是在定义 `initfunction` 初始化函数模块的时候，出现了该函数必须要调用该头文件中的结构体，而运行时管理又必须要调用 `initfunction` 初始化函数，出现了循环引用的问题，无法进行编译。经过了长时间的查资料后，我发现我的整个程序框架还是不够完善，头文件的定义应当尽量的减少对于函数过程的描述，否则容易出现循环引用的问题。于是我将所有的结构类都提取了出来，放到了专门的头文件里，解决了循环引用的问题。

函数初始化模块：此模块为函数初始化，在主函数中调用。为了满足程序分层结构，我将函数分为两类：1. 内联函数，比如 `print` 函数和 `input` 函数。2. 用户自定义函数。这个函数用于初始化所有的内联函数。所有内联函数采用函数指针的形式将函数的指针保存到符号表中，以便进行调用。函数的指针类型如下定义：

```
typedef phrase::return_value(*funpointer)(phrase::infunction_type * infunction);
```

使用的方法为，定义上述类型的函数，对变量进行处理，如果没有返回值则返回整型 0。然后将函数指针添加到符号表中。

语法语义分析模块：此模块为程序最重要的模块，语法及语义分析模块。此模块的功能为主要的代码执行功能，对外提供一个 `run` 函数开始执行，提供 `function_return` 用于记录返回值。

主函数：用于界面的显示与基本功能的调用，程序分为两种打开方式，一是采用命令行的方式打开，或采用拖动方式打开，传入一个 `PY/0` 文件物理位置的命令行参数；二是不传入命令行参数直接打开，进入命令操作界面。

2.2 PY/0 语言文法定义

#1. the start of the input

$\langle \text{file_input} \rangle := (\text{NEWLINE} \mid \langle \text{stmt} \rangle)^* \text{ENDMARKER}$

#2. statement and simple statement

$\langle \text{stmt} \rangle := \langle \text{simple_stmt} \rangle \mid \langle \text{compound_stmt} \rangle$

$\langle \text{simple_stmt} \rangle := \langle \text{small_stmt} \rangle \text{NEWLINE}$

$\langle \text{small_stmt} \rangle := (\langle \text{del_stmt} \rangle \mid \langle \text{expr_stmt} \rangle \mid \langle \text{pass_stmt} \rangle \mid \langle \text{flow_stmt} \rangle \mid \langle \text{import_stmt} \rangle)$

$\langle \text{expr_stmt} \rangle := \langle \text{or_test} \rangle \mid (\text{NAME} ((\langle \text{augassign} \rangle \langle \text{or_test} \rangle) \mid ('=' \langle \text{or_test} \rangle)))$

$\langle \text{augassign} \rangle := ('+=' \mid '-=' \mid '*=' \mid '/=')$

$\langle \text{del_stmt} \rangle := \text{'del' NAME}$

$\langle \text{pass_stmt} \rangle := \text{'pass'}$

$\langle \text{flow_stmt} \rangle := \langle \text{break_stmt} \rangle \mid \langle \text{continue_stmt} \rangle \mid \langle \text{return_stmt} \rangle$

$\langle \text{break_stmt} \rangle := \text{'break'}$

$\langle \text{continue_stmt} \rangle := \text{'continue'}$

$\langle \text{return_stmt} \rangle := \text{'return' or_test}$

$\langle \text{import_stmt} \rangle := \text{'import' (STRING \mid NAME)}$

#3. compound statement

$\langle \text{compound_stmt} \rangle := \langle \text{if_stmt} \rangle \mid \langle \text{while_stmt} \rangle \mid \langle \text{for_stmt} \rangle \mid \langle \text{funcdef} \rangle$

$\langle \text{if_stmt} \rangle := \text{'if' } \langle \text{or_test} \rangle \text{' ' } \langle \text{suite} \rangle (\text{'elif' } \langle \text{or_test} \rangle \text{' ' } \langle \text{suite} \rangle)^* [\text{'else' ' ' } \langle \text{suite} \rangle]$

$\langle \text{while_stmt} \rangle := \text{'while' } \langle \text{or_test} \rangle \text{' ' } \langle \text{suite} \rangle$

$\langle \text{for_stmt} \rangle := \text{'for' NAME 'in' } \langle \text{or_test} \rangle \text{' ' } \langle \text{suite} \rangle$

$\langle \text{suite} \rangle := \text{NEWLINE INDENT } \langle \text{file_input} \rangle \text{DEDENT}$

$\langle \text{funcdef} \rangle := \text{'def' NAME } \langle \text{parameters} \rangle \text{' ' } \langle \text{suite} \rangle$

$\langle \text{parameters} \rangle := \text{'(' [typedarglist] ') '}$

$\langle \text{typedarglist} \rangle := \text{NAME (' ' NAME)}^*$

$\langle \text{or_test} \rangle := \langle \text{and_test} \rangle (\text{'or' } \langle \text{and_test} \rangle)^*$

$\langle \text{and_test} \rangle := \langle \text{not_test} \rangle (\text{'and' } \langle \text{not_test} \rangle)^*$

$\langle \text{not_test} \rangle := [\text{'not' }] \langle \text{comparison} \rangle$

$\langle \text{comparison} \rangle := \langle \text{expr} \rangle [\langle \text{comp_op} \rangle \langle \text{expr} \rangle]$

$\langle \text{comp_op} \rangle := \text{'<' \mid '>' \mid '==' \mid '>=' \mid '<='}$

```

<expr> := <term> (('+' | '-') <term>)*
<term> := <factor> (('*' | '/') <factor>)*
<factor> ::= <atom> <trailer>*
<atom> := <atom_base> | <array> | '(' or_test ')'
<atom_base> := NAME | NUMBER | STRING
<array> := '(' <or_test> (',' <or_test>)* ')'

<trailer> := '[' <arglist> ']' | '[' <subscript> ']'
<subscript> := <or_test>
<arglist> := <or_test> (',' <or_test>)*

```

3 PY/0 语言各模块设计与数据结构

3.1 基础符号模块

头文件名称: symboltable.h

名字空间命名: word

变量 (表 3.1):

表 3.1

名称	类型	含义
reserved	enum	枚举所有保留字
reserved_content	char **	所有保留字对应的字符
punctuation	enum	枚举所有标点符号
punctuation_content	char **	所有标点符号对应的字符
control	enum	枚举所有控制类符号
control_content	char **	所有控制类符号对应的字符
type	enum	枚举所有变量符号
escape_character_table_length	const int	转义字符表长度
escape_character_table	char *	转移字符表

其中, PY/0 语言保留字如下图 (图 3.1.1)

```
enum reserved {
    NULLWORD, AND, BREAK, CONTINUE, DEF,
    DEL, ELIF, ELSE, FOR, IF,
    IMPORT, IN, NOT, OR, PASS,
    RETURN, WHILE
};
```

图 3.1.1

其中，PY/O 语言标点如下图（图 3.1.2 和图 3.1.3）

```
enum punctuation {
    COMMA=20, ASSIGN, ADD_EQUAL, MINUS_EQUAL, MULTI_EQUAL,
    DIV_EQUAL, COLON, LESS, MORE, EQUAL,
    LESS_EQUAL, MORE_EQUAL, ADD, MINUS, MULTI,
    DIV, REMAINDER, DIV_CEIL, PERIOD, PAREN_L,
    PAREN_R, BRACKET_L, BRACKET_R, LOGICAL_OR, LOGICAL_AND
};
```

图 3.1.2

```
char punctuation_content[][5]{
    ",", "=", "+=", "-=", "*=",
    "/=", ":", "<", ">", "==",
    ">=", "<=", "+", "-", "*",
    "/", "%", "//", ".", "(",
    ")", "[", "]", "|", ""
};
```

图 3.1.3

3.2 错误处理模块

头文件名称：error.h

名字空间命名：error

函数及结构列表（表 3.2）：

表 3.2

名称	参数列表	返回值	含义
<code>syntax_error</code>	<code>int</code> location, <code>std::string</code> reason	<code>void</code>	报错处理
<code>syntax_error</code>	<code>std::string</code> reason, <code>int</code> loc	<code>void</code>	报错处理
<code>syntax_error</code>	<code>std::string</code> reason	<code>void</code>	报错处理

`error_reason` 所有的错误说明：

```
std::string space_error = "the Bad use of space";
std::string number_exceeded_error = "number exceeded error!";
std::string escape_character_error = "escape character error!";
std::string float_number_error = "float number error!";
std::string string_not_complete = "string not complete!";
std::string type_unrecognized = "type unrecognized!";
std::string code_level_exceed = "code level exceed, the max level of code is 9999!";
std::string function_level_exceed = "function level exceed, the max level of function is 999!";
std::string carry_return_error = "should be carry return";
```

```

std::string unrecognized_syntax_error = "unrecognized syntax error";
std::string del_type_error = "del type error";
std::string del_name_error = "no name is found in variable table";
std::string pass_used_error = "pass cannot used here";
std::string break_used_error = "break can only used in for and while loop";
std::string continue_used_error = "continue can only used in for and while loop";
std::string return_used_error = "return used error";
std::string value_not_found_error = "value not found error";
std::string return_error = "return error";
std::string return_function = "return value cannot be a function";
std::string value_not_found = "value not found";
std::string expression_error = "expression error";
std::string value_type_error = "value type error";
std::string operation_not_support = "operation not support";
std::string unsupport_type = "unsupport type";
std::string list_number_error = "list number error";
std::string bracket_error = "bracket error";
std::string not_function = "not a function";
std::string name_not_found = "name not found";
std::string array_created_error = "array created error";
std::string list_number_outof_range = "list number outof range";
std::string while_syntax_error = "while syntax error";
std::string for_syntax_error = "for syntax error";
std::string if_syntax_error = "if syntax error";
std::string array_operate_length_error = "array operate length error";
std::string function_parameter_num_error = "function parameter num error";
std::string function_parameter_not_support_error = "function parameter not support error";
std::string function_print_parameter_error = "function print parameter error";
std::string function_define_error = "function define error";
std::string import_error = "import error";

```

3.3 词法分析模块

头文件名称: symboltable.h

名字空间命名: 不采用名字空间

类名称: lexer

3.3.1 Public 类型

Public 成员变量 (表 3.3.1):

表 3.3.1

名称	类型	含义
word_code	int	读取的代码
word_int	int	读取的 int 型数据
word_double	double	读取的 float 型数据
word_string	std::string	读取的 string 型数据
word_name	std::string	读取的 string 型数据
lineloc	int	当前代码执行代码行

Public 成员函数 (表 3.3.1):

表 3.3.1

名称	参数列表	返回值	含义
lexer	<code>char*</code> filename, <code>bool</code> listlex=false		读取文件、分词
get_location	无	<code>int</code>	获取当前代码段执行位置
jump_location	<code>int</code> location	<code>void</code>	跳转代码段
skip_block	无	<code>void</code>	跳过代码区
get_next	无	<code>void</code>	获取下一个 word
get_back	无	<code>void</code>	获取上一个 word
code_length	无	<code>int</code>	返回代码长度
get_lexer_name	无	<code>const char*</code>	当前 lexer 的名称

3.3.1 Private 类型

Private 成员变量（表 3.3.3）:

表 3.3.3

名称	类型	含义
line_buf	<code>std::string</code>	读取行缓存区
char_buf	<code>char</code>	读取字符缓冲
next_char_buf	<code>char</code>	下一个字符的缓冲
char_loc	<code>char</code>	字符位置指针
line_length	<code>int</code>	行的长度
level	<code>int</code>	表示代码的层级
line_total	<code>int</code>	表示总的行号
line_num	<code>int</code>	表示当前读取行号
in_file	<code>std::ifstream*</code>	读入的文件
word_type	<code>int</code>	读入 word 的类型
word_numvalue	<code>int</code>	数字类型返回
word_floatvalue	<code>double</code>	数字类型返回
word_stringvalue	<code>std::string</code>	字符串型返回
word_namevlaue	<code>std::string</code>	名字类型返回
max_length_of_word	<code>const int</code>	变量名字的最长长度
max_number_length	<code>const int</code>	变量数字的最长长度
word_buffer	<code>char</code>	读取时的字符缓冲区
reserved_word_length	<code>const int</code>	保留字最大长度
is_reserved_word	<code>bool</code>	是否为保留字
symbol_code_list	<code>std::vector<int></code>	所有分割符号的记录
symbol_location_list	<code>std::vector<int></code>	分割符号对应的位置
symbol_code_length	<code>int</code>	所有分割符号总长度
symbol_int_list	<code>std::map<int, int></code>	当前为 <code>int</code> 型数据对应的值
symbol_double_list	<code>std::map<int, double></code>	当前为 <code>double</code> 型数据对应的值
symbol_name_list	<code>std::map<int, std::string></code>	当前为 <code>string</code> 型数据对应的值
symbol_string_list	<code>std::map<int, std::string></code>	当前为 <code>string</code> 型数据对应的值

<code>symbol_code_location</code>	<code>int</code>	当前读取位置
<code>lexer_name</code>	<code>std::string</code>	读取文件的物理位置

public 成员函数（表 3.3.4）:

表 3.3.4

名称	参数列表	返回值	含义
<code>getchar</code>	无	<code>void</code>	读取文件，获取下一个 <code>char</code> 字符
<code>getword</code>	无	<code>void</code>	获取下一个 <code>word</code> 字
<code>letter_start</code>	无	<code>void</code>	以字母开头的词分析子程序
<code>number_start</code>	无	<code>void</code>	以数字开头的词分析子程序
<code>get_number_from_word</code>	无	<code>int</code>	将字符转化为数字
<code>string_start</code>	无	<code>void</code>	以引号开头的字符串分析子程序
<code>punctuation_start</code>	无	<code>void</code>	以标点开头的词分析子程序

3.4 运行时基础结构

头文件名称: `runningbasic.h`

名字空间命名: `running`

变量（表 3.4）:

表 3.4

名称	类型	含义
<code>name_type</code>	枚举型	所有在 <code>PY/D</code> 语言出现的符号的变量类型
<code>function_table</code>	结构体	当 <code>name_type</code> 为函数时，调用此结构
<code>name_table</code>	结构体	符号表中存放的基本结构体

3.5 运行时管理

头文件名称: `running.h`

名字空间命名: 不采用名字空间

类名称: `running_table_class`

3.5.1 Private 成员变量

表 3.5.1

名称	类型	含义
running_level	int	当前变量所在层级
max_level	const int	层级上限
max_function_level	const int	函数嵌套上限
running_table	std::vector<running::name_table*>	运行时符号表
save_table	std::stack<std::stack<running::name_table*> >	函数调用时变量临时存储
import_list	std::vector<std::string>	Import 路径的记录列表

3.5.2 Public 成员函数

表 3.5.2

名称	参数列表	返回值	含义
running_table_class	无	无	构造函数
append_table	running::name_type type, std::string name, void* value	void	向符号表添加字段
get_items	std::string name	running::name_table*	查找符号表
get_items_location	std::string name	int	
get_table	无	std::vector<running:: name_table*>*	返回总符号表
running_levelup	无	int	层级提高
running_leveldown	无	int	层级降低
running_delete_from	int	void	从给定值删除符号表元素
in_function	std::string name	int	清除 name 位置下所有变量， 并将其存储到 save_table，返回 清除的位置
out_function	int location	void	与 in_function 为逆过程
del_function	std::string	void	删除变量
in_import	std::string	bool	给定 import 值，判断是否存在
out_import	std::string	void	退出 import，清除该内容
show_table	无	void	展示整个符号表

3.6 语法分析基础结构

头文件名称: phrasebasic.h

名字空间命名: phrase

变量 (表 3.6)

表 3.6

名称	类型	含义
<code>return_value</code>	结构体	所有带有返回值的子程序的返回结构
<code>bracket_type</code>	枚举型	用于记录括号类型
<code>infunction_type</code>	结构体	函数参数列表信心

3.7 函数初始化

头文件名称: initfunction.h

名字空间命名: init_name

类名称: init_function

函数名称 (表 3.7)

表 3.7

名称	参数列表	返回值	含义
<code>init_function</code>	<code>running_table_class*</code>	无	构造函数
<code>function_max</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联求最大值函数
<code>function_min</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联求最小值函数
<code>function_show_table</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联展示变量表函数
<code>function_print</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联格式化输出函数
<code>function_input</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联输入函数
<code>function_int</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联类型转换函数
<code>function_float</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联类型转换函数
<code>function_string</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联类型转换函数
<code>function_range</code>	<code>phrase::infunction_type*</code>	<code>phrase::return_value</code>	内联产生 <code>range</code> 序列函数

3.8 语义及语法分析

头文件名称: phraser.h

名字空间命名: 无名字空间

类名称: phraser

变量表（3.8.1）:

表 3.8.1

名称	类型	含义
function_return	phrase::return_value	记录函数返回值
code_lexer	lexer*	保存分词对象的指针
table	running_table_class*	保留符号表操作对象指针
bracket_balance	std::stack<phrase::bracket_type>	对方括号和圆括号进行记录
break_sign	bool	Break 标志
continue_sign	bool	Continue 标志
return_sign	bool	Return 标志
loop_conting	int	当前循环层数

函数表（表 3.8.2）:

表 3.8.2

名称	参数列表	返回值	含义
phraser	lexer* code_lexer, running_table_class* table	无	构造函数
run	无	void	开始执行
file_input	无	void	分割换行符与简单句
stmt	无	void	区分简单句与复合句
simple_stmt	无	void	分割简单句与换行符
small_stmt	无	void	区分简单句类型
flow_stmt	无	void	区分 break、continue 等
del_del_stmt	无	void	删除句
del_atom	std::string name	bool	删除字句
expr_stmt	无	void	表达式判断
break_stmt	无	void	break 句
continue_stmt	无	void	continue 句
return_stmt	无	void	return 句
import_stmt	无	void	import 句
compound_stmt	无	void	复合句判断
array_def	无	phrase::return_value	定义向量
or_test	无	phrase::return_value	分割 or 运算符
and_test	无	phrase::return_value	分割 and 运算符
not_test	无	phrase::return_value	分割 not 运算符
comparison	无	phrase::return_value	分割>、<、==等运算符
expr	无	phrase::return_value	分割+、-运算符
term	无	phrase::return_value	分割*、/运算符
factor	无	phrase::return_value	调用 atom 与 trailer
atom	无	phrase::return_value	判断原子变量
trailer	phrase::return_value*	void	分割下标访问与函数调

			用
subscript	phrase:: <code>return_value*</code>	<code>void</code>	下标访问
arglist	phrase:: <code>return_value*</code>	<code>void</code>	函数调用
argument	无	phrase:: <code>infunction_type*</code>	函数拆分参数列表
if_stmt	无	<code>void</code>	分支结构 if、elif、else
while_stmt	无	<code>void</code>	循环结构 while
for_stmt	无	<code>void</code>	循环结构 for
funcdef	无	<code>void</code>	函数定义
if_atom_format	无	<code>void</code>	分支结构字句
value_transmit	phrase:: <code>return_value*</code> , phrase:: <code>return_value*</code>	<code>bool</code>	不同类型值之间的自动 转化函数
basic_operate	running:: <code>name_type</code> type, <code>void*</code> val1, <code>void*</code> val2, word:: <code>punctuation</code> operator_	<code>void*</code>	+, -, *, /操作运算符运 算
switch_value	phrase:: <code>return_value*</code> name, running:: <code>name_type</code> type	<code>void</code>	值类型转换
debug	无	<code>void</code>	用于调试使用

3.9 主程序

变量表（表 3.9）：

表 3.9

名称	参数列表	返回值	含义
split	<code>char*</code> src, <code>const char*</code> separator, <code>char**</code> dest, <code>int*</code> num	<code>void</code>	命令行读入命令字符串分割
welcome_show	无	<code>void</code>	展示欢迎文字
main	<code>int</code> argc, <code>char**</code> argv	<code>void</code>	主函数

4 运行设计与效果展示

本程序的运行原理为：

- （1）主程序调用符号表，新建一个符号表对象的指针

- (2) 将自身路径添加到符号表指针的 `import` 列表
- (3) 创建分词程序对象，传入带打开的文件地址，分词对象对其进行分词操作
- (4) 创建初始化函数列表，将初始化内容添加到符号表
- (5) 创建语法分析解释器，传入分词指针和符号表指针。
- (6) 运行语法分析解释器程序。
- (7) 用 `catch` 接受错误，所有的错误均由 `error` 类抛出，并由主函数接收。

4.1 简单语句

简单语句的设计与 PL/0 相似，简单语句包含了赋值类语句、`import` 语句、控制类语句、删除变量语句、

4.1.1 赋值类语句

赋值类语句为整个语法设计的核心内容，用于表达式赋值，整体采用了返回值的形式传递参数，该语句的设计文法如下：

`<expr_stmt> := <or_test> | (NAME ((<augassign> <or_test>) | ('=' <or_test>)))`

`<augassign> := ('+=' | '-=' | '*=' | '/=')`

语句可以为一个单独的字句，可以使等于赋值，可以用 `+=`、`-=`、`*=`、`/=` 赋值

`<or_test> := <and_test> ('or' <and_test>)*`

其中 `or_test` 包含了 `or` 语句的判断，出现 `or` 的时候自动判断前后函数的值和类型，只要有一个不是 `int` 型的 0 或 `float` 型的 0 即为成功，返回 `int` 型的 1。

`<and_test> := <not_test> ('and' <not_test>)*`

其中 `and_test` 包含了 `and` 语句的判断，出现 `and` 的时候自动判断前后函数的值和类型，只要有一个是 `int` 型的 0 或 `float` 型的 0 即为失败，返回 `int` 型的 0。

`<not_test> := ['not'] <comparison>`

出现 `not` 的时候对后面的值去否定，原本是 `int` 型的 0 则取 `int` 型 1，否则取 `int` 型 0

`<comparison> := <expr> [<comp_op> <expr>]`

`<comp_op> := '<' | '>' | '==' | '>=' | '<='`

表达式判断，如果出现了判断类表达式，则返回的值一定为 `int` 型的 1 或 0

`<expr> := <term> (('+' | '-') <term>)*`

`<term> := <factor> (('*' | '/') <factor>)*`

`<factor> ::= <atom> <trailer>*`

此部分内容为基本表达式内容，与 PL/0 一致，但是原子类型加入了后缀，用于调用数组和函数。这部分的设计为指针传入参数，如果出现 trailer 则通过指针进行修改。

`<atom> ::= <atom_base> | <array> | '(' or_test ')'`

`<atom_base> ::= NAME | NUMBER | STRING`

`<array> ::= '(' <or_test> (',' <or_test>)* ')'`

这部分的 array 为我设计的向量

`<trailer> ::= '(' [<arglist>] ')' | '[' <subscript> ']'`

`<subscript> ::= <or_test>`

`<arglist> ::= <or_test> (',' <or_test>)*`

如果 trailer 值为函数，则调用函数子程序，此内容在 phrase 介绍中已经说过，不再赘述。

例：

```
a=1+2*5-5*10.5
b=5>2+6
c=0 and a
d=0 or a
e=not d
show_table()
```

```
name table:
  tpye  name  level  value
float  a     0     -41.500000
int     b     0      0
int     c     0      0
int     d     0      1
int     e     0      0
```

图 4.1.1

```
int_t = 10
array_a = (int_t, 20, 30)
array_b = (int_t, int_t + 0.5, 30)
array_c = array_a * array_b
array_d = (int_t)
print("array_f = {}", (0, 1, 2) + (2, 3, 4))
show_table()
```

```
array_f = (2.000000, 4.000000, 6.000000)
name table:
  int  int_t  0  10
  array int  array_a  0
           10    20    30
  array float array_b  0
           10.000000  10.500000  30.000000
  array float array_c  0
           100.000000  210.000000  900.000000
  array int  array_d  0
           10
```

图 4.1.2

4.1.2 控制类语句

控制类语句包括 break、continue、return

break 语句设计

设置 break_sign 标签，由 break 触发，在程序的 file_input 进行判断，如果 break_sign 标签为真，则退出执行。此标签只可以被调用 file_input 的 for 循环或 while 循环进行终止，终止后循环调用 skip_block 函数跳过代码段终止。

<break_stmt> := 'break'

例：

```
for i in (1,2,3):
    print("now i value: {} \n", i)
print("\n\n")
for j in (1,2,3):
    print("now j value: {} \n", i)
    break
```

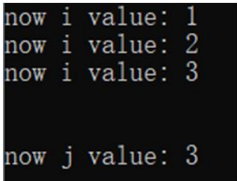


图 4.1.3

(1) continue 语句设计

设置 continue_sign 标签，由 continue 触发，在程序的 file_input 进行判断，如果 continue_sign 标签为真，则退出执行。此标签只可以被调用 file_input 的 for 循环或 while 循环进行终止，终止后跳转到循环开始代码段。

<continue_stmt> := 'continue'

例

```
for i in (1,2,3):
    print("now i value: {} \n", i)
    if i >= 2:
        continue
    print("now i + 1 value: {} \n", i + 1)
```

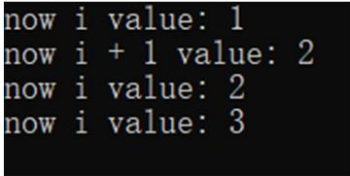


图 4.1.4

(2) return 语句

return 语句需要设置 return_sign，由 return 触发，值记录到 function_return 中，方便调用函数的函数接受。

<return_stmt> := 'return' or_test

例：

```
def add (val1, val2):
    return val1 + val2

print("add value is : {}", add(10, 20) )
```

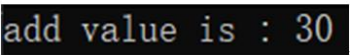


图 4.1.5

4.1.3 删除变量类

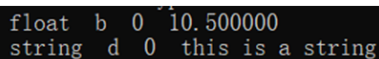
目前该内容只能对对象进行直接删除，无法删除 array 中的元素

`<del_stmt> ::= 'del' NAME`

例：

```
a = 10
b = 10.5
c = (10,20,20)
d = "this is a string"
```

```
del a
del c
show_table()
```



```
float b 0 10.500000
string d 0 this is a string
```

图 4.1.6

4.1.4 import 类

该内容我新建了一个 lexer 分词程序，启动了与主函数一样的流程进行程序执行。

`<import_stmt> ::= 'import' (STRING | NAME)`

例：



图 4.1.7

4.2 复合语句

复合语句包括了 if、elif、else 分支语句，while、for 循环语句、def 函数定义语句。我利用了代码块的跳转方式实现了这些功能。

`<compound_stmt> ::= <if_stmt> | <while_stmt> | <for_stmt> | <funcdef>`

4.2.1 分支类语句

通过 `or_test` 进行判断，如果结果为真则执行随后的语句，然后跳过 `elif` 和 `else` 语句。否则记录一个是否为真的 `bool` 变量，循环找到所有的 `elif`，若成功执行则将记录设置为真，最后判断是否有 `else`，如果有 `else` 且记录为假则进行执行。代码块内部提升层级，结束后降低层级。

```
<if_stmt> := 'if' <or_test> ':' <suite> ('elif' <or_test> ':' <suite>)*['else' ':' <suite>]
<suite> := NEWLINE INDENT <file_input> DEDENT
```

例：

```
a=(10, 20, 30, 40)
for i in a:
    if i <= 10:
        print("if is true, now i is: {} \n\n", i)
    elif i <= 20:
        print("elif is true, now i is: {} \n\n", i)
    else:
        print("else, now i is: {} \n\n", i)
```

```
if is true, now i is: 10
elif is true, now i is: 20
else, now i is: 30
else, now i is: 40
```

图 4.2.1

4.2.2 循环类语句

循环类语句的共同点为，进入循环前记录当前代码位置，在循环退出后判断退出原因，并选择是否跳转到循环开头或跳转到循环结束。

(1) for 循环

将循环变量的值计入到符号表，代码块内部提升层级，结束后降低层级。

```
<for_stmt> := 'for' NAME 'in' <or_test> ':' <suite>
<suite> := NEWLINE INDENT <file_input> DEDENT
```

例：

```
for i in (1, 2, 3, 4):
    print("for value: {} \n\n", i)
show_table()
```

```
for value: 1
for value: 2
for value: 3
for value: 4
```

```
int i 0 4
```

图 4.2.2

(2) while 循环

```
<while_stmt> := 'while' <or_test> ':' <suite>
<suite> := NEWLINE INDENT <file_input> DEDENT
```

例：

```

a=10
while a>5:
    print("now a value is : {} \n",a)
    a-=1

```

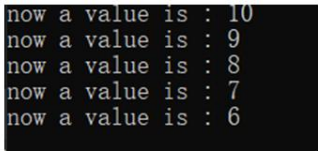


图 4.2.3

4.2.3 函数类语句

(1) 函数声明

函数声明仅仅读取变量列表，并记录参数到符号表，不对其进行执行，跳过代码区。

```

<funcdef> := 'def' NAME <parameters> ':' <suite>
<suite> := NEWLINE INDENT <file_input> DEDENT

```

例：

```

def fun(n):
    print("this is a function")

```

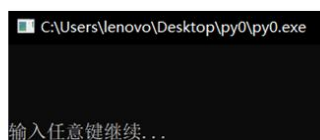


图 4.2.4

(2) 函数调用

对于用户类函数调用时，创建新的解释器，传入函数的分词程序 `lexer`，函数位置，提升符号表层级，并将所有的符号值填入符号表。

对于内联函数，通过查表找到函数指针，用函数指针调用。

例：

```

a=max( (1,2,5,6,4) )
print("a : {} \n", a)
val="1236.65"
b=int(val)
print("b : {} \n", b)
c=float(val)
print("c : {} \n", c)

m=input("please input a value: ")
print("m+1={} \n", float(m) + 1)

```

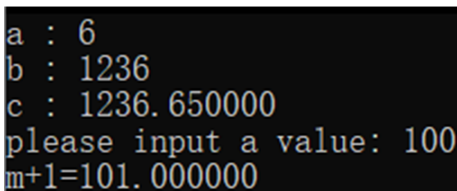


图 4.2.5

4.3 综合展示

(1) 求解 4 层汉诺塔问题

```
def move(n,a,b,c):
    if n == 1:
        print("{} --> {}".format(a,c))
    else:
        move(n-1, a, c, b)
        print("{} --> {}".format(a,c))
        move(n-1, b, a, c)
move(4, "A", "B", "C")
```

```
A --> B
A --> C
B --> C
A --> B
C --> A
C --> B
A --> B
A --> C
B --> C
B --> A
C --> A
B --> C
A --> B
A --> C
B --> C
```

图 4.3.1

(2) 101 到 200 之间的素数

```
def is_prime(n):
    for i in range(2,n):
        t = n / i
        if n == t * i:
            return 0
    return 1

sum=0
for i in range(101,200):
    if is_prime(i):
        print("{}\n".format(i))
        sum += i
print("The sum of the primes is: {}".format(sum))
```

```
101
103
107
109
113
127
131
137
139
149
151
157
163
167
173
179
181
191
193
197
199
The sum of the primes is: 3167
```

图 4.3.2

(3) 1000 以内的完全数

```
def perfect_fun(n):
    s = n
    for i in range(1, n):
        t = n / i
        if n == t * i:
            s -= i
    if s == 0:
        return 1
    else:
        return 0
```

```
import "perfect.txt"

for i in range(2,1000):
    if perfect_fun(i):
        print("perfect value: {}".format(i))

perfect value: 6
perfect value: 28
perfect value: 496
```

图 4.3.3

(4) 输出九九乘法表

```
i = 1
while i < 10:
    j = 1
    while j <= i:
        print("{}*{}={} ".format(i, j, i * j))
        j += 1
    i += 1
    print("\n")
```

```
for i in range(10):
    for j in range(i+1):
        print("{}*{}={} ".format(i, j, i * j))
    print("\n")
```

```
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7  7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

图 4.3.2