



成绩

中国农业大学

课程论文

(2019 -2020 学年春季学期)

论文题目: py0 编译器设计

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 任务概述 | 1 |
| 2 总体设计 | 1 |
| 3 各模块设计与数据结构 | 3 |
| 3.1 基础符号模块 | 3 |
| 3.2 错误处理模块 | 5 |
| 3.3 词法分析模块 | 6 |
| 3.3.1 Public 类型 | 7 |
| 3.3.1 Private 类型 | 10 |
| 3.4 运行时基础结构 | 12 |
| 3.5 运行时管理 | 14 |
| 3.5.1 private 成员变量 | 14 |
| 3.5.2 public 成员函数 | 15 |
| 3.6 语法分析基础结构 | 19 |
| 3.7 函数初始化 | 20 |
| 3.8 语义及语法分析 | 26 |
| 3.8.1 简单句 | 28 |
| 3.8.2 表达式 | 31 |
| 3.8.3 复合语句 | 43 |
| 3.9 主程序 | 54 |
| 4 运行设计与效果展示 | 56 |
| 4.1 简单语句 | 56 |
| 4.1.1 赋值类语句 | 56 |
| 4.1.2 控制类语句 | 57 |
| 4.1.3 删 除 变量类 | 59 |
| 4.1.4 import 类 | 59 |
| 4.2 复合语句 | 59 |
| 4.2.1 分 支 类 语 句 | 60 |
| 4.2.2 循 环 类 语 句 | 60 |
| 4.2.3 函 数 类 语 句 | 61 |
| 4.3 综合展示 | 61 |

1 任务概述

编写目的：py0 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，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。

参考资料：编译原理课件、pl0 编译器源代码、python 语法说明文档。

2 总体设计

本软件所有模块的调用规则如下（图 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 3 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。

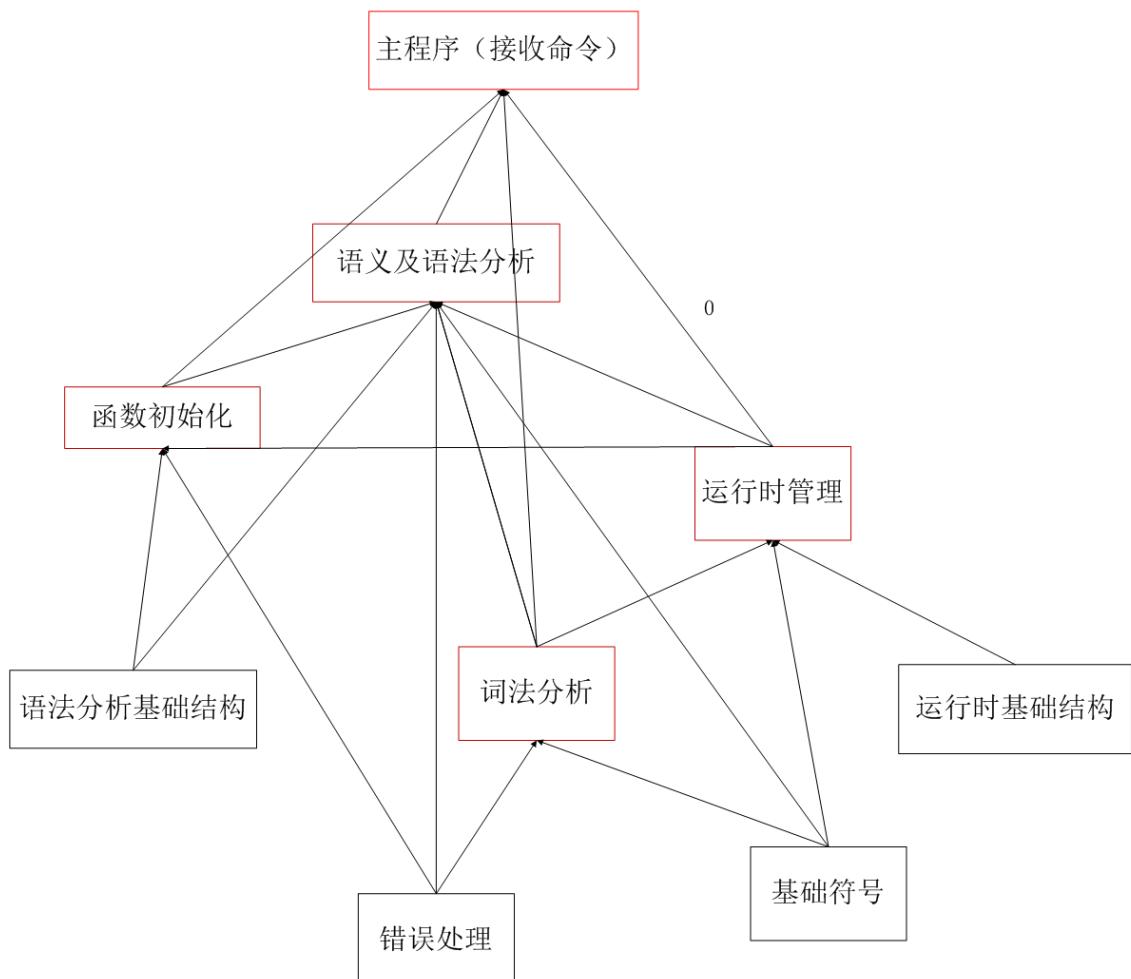


图 1

函数语法定义如下：

```
#Grammar for my programming language py0
#This content is consulted from python's grammar, due to the difficulties that compose a wholly new grammar from
the scratch without unexpected bug.
#The entire grammar is not object orientated.
```

#Start symbols for the grammar:

```
#     file_input: a sequence of commands read from a file
```

#NEWLINE is refer to line feed

#ENDMARKER is refer to the end of the file

#1. the start of the input

```
<file_input> := (NEWLINE | <stmt>)* ENDMARKER
```

#2. statement and simple statement

```
<stmt> := <simple_stmt> | <compound_stmt>
```

```
<simple_stmt> := <small_stmt> NEWLINE
```

```
<small_stmt> := (<del_stmt> | <expr_stmt> | <pass_stmt> | <flow_stmt> | <import_stmt>)
```

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

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

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

```
<pass_stmt> := 'pass'
```

```
<flow_stmt> := <break_stmt> | <continue_stmt> | <return_stmt>
```

```
<break_stmt> := 'break'
```

```
<continue_stmt> := 'continue'
```

```
<return_stmt> := 'return' or_test
```

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

#3. compound statement

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

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

```
<while_stmt> := 'while' <or_test> ':' <suite>
```

```
<for_stmt> := 'for' NAME 'in' <or_test> ':' <suite>
```

```
<suite> := NEWLINE INDENT <file_input> DEDENT
```

```

<funcdef> := 'def' NAME <parameters> ':' <suite>
<parameters> := '(' [typedargslist] ')'
<typedargslist> := NAME (, NAME )*

```

```

<or_test> := <and_test> ('or' <and_test>)*
<and_test> := <not_test> ('and' <not_test>)*
<not_test>:= ['not'] <comparison>
<comparison> := <expr> [<comp_op> <expr>]

<comp_op> := '<' | '>' | '==' | '>=' | '<='
<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 各模块设计与数据结构

3.1 基础符号模块

此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。

头文件名称： symboltable.h

名字空间命名： word

变量：

| 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 |

| | | |
|--|------------------------|--------------|
| <code>punctuation_content</code> | <code>char **</code> | 所有标点符号对应的字符 |
| <code>control</code> | <code>enum</code> | 枚举所有控制类符号 |
| <code>control_content</code> | <code>char **</code> | 所有控制类符号对应的字符 |
| <code>type</code> | <code>enum</code> | 枚举所有变量符号 |
| <code>escape_character_table_length</code> | <code>const int</code> | 转义字符表长度 |
| <code>escape_character_table</code> | <code>char *</code> | 转移字符表 |

(1) reserved 保留字

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

(2) reserved_content 保留字对应的字符

```
char reserved_content[][][15] = {
    "", "and", "break", "continue", "def",
    "del", "elif", "else", "for", "if",
    "import", "in", "not", "or", "pass",
    "return", "while"
};
```

(3) punctuation 标点符号

```
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
};
```

(4) punctuation_content 标点符号对应的字符

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

(5) control 控制类符号

```
enum control {
    NEWLINE=50, ENDMARKER, INDENT, DEDENT
    //INDENT表示tab, DEDENT表示退格
};
```

(6) control_content 所有控制类符号对应的字符

```
char control_content[] {
    '\n', '\0', '\t', '\b'
```

(7) type 枚举所有变量符号

```
enum type {
| NAME=60,      STRING,      NUMBER,      FLOAT
};
```

(8) escape_character_table_length 转义字符表长度

```
const int escape_character_table_length = 12;
```

(9) escape_character_table 转移字符表

```
char escape_character_table[] {
| 'a',     '\a',   'b',     '\b',   'f',     '\f',
| 'n',     '\n',   'r',     '\r',   't',     '\t',
| 'v',     '\v',   '\\',    '\\\\',  '\\',    '\\',
| '\"',    '\"',   '?',     '?',    '0',    '\0',
};
```

3.2 错误处理模块

此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。

头文件名称：error.h

名字空间命名：error

函数及结构列表：

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

(1) syntax_error 报错处理函数

```
void error::syntax_error(int location, std::string reason) {
| printf("\n\nsyntax error in line %d\n", location);
| printf("    %s\n\n", reason.c_str());
| throw("syntax error");
}
```

仅以这个一个函数举例，用于输出错误的位置，并向上抛出异常。

(2) 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

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。

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:

| 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:

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

(1) lexer 构造函数(`char *filename, bool listlex=false`)

参数: 文件名、是否显示读取内容

作用: 抛出文件错误异常、读取并分词整个文件

```
std::ifstream *myfile=new std::ifstream(filename);
if (!(*myfile).is_open())
{
    throw("未成功打开文件");
}
lexer::lexer_name = filename;

lexer::in_file = myfile;
lexer::symbol_code_location = 0;
```

保存所有读取的内容到 symbol_code_list 中, 所有代码的行数保留在 symbol_location_list 中, 用于出错处理。所有变量类型保留在各自的 map 结构中, 以便读取, 其中 map 的第一个参数为代码位置、第二个参数为值。调用了 getword 函数, 表示读取下一个 word

```

lexer::getchar();
while (true) {
    lexer::getword();
    lexer::symbol_code_list.push_back(lexer::word_type);
    lexer::symbol_location_list.push_back(lexer::line_num);
    if (lexer::word_type == word::control::ENDMARKER) {
        lexer::symbol_code_location++;
        break;
    }
    if (lexer::word_type >= word::type::NAME) {
        if (lexer::word_type == word::type::NAME)
            lexer::symbol_name_list[lexer::symbol_code_location] = lexer::word_namevalue;
        if (lexer::word_type == word::type::STRING)
            lexer::symbol_string_list[lexer::symbol_code_location] = lexer::word_stringvalue;
        if (lexer::word_type == word::type::NUMBER)
            lexer::symbol_int_list[lexer::symbol_code_location] = lexer::word_numvalue;
        if (lexer::word_type == word::type::FLOAT)
            lexer::symbol_double_list[lexer::symbol_code_location] = lexer::word_floatvalue;
    }
    lexer::symbol_code_location++;
}
lexer::symbol_code_length = lexer::symbol_code_location;
lexer::symbol_code_location = 0;

```

还有打开显示读取内容的功能，不再做赘述。

(2) get_location 函数

获取当前执行代码位置的值，用于控制跳转类语句（if、elif、else、for、while、def）获取当前位置，以便在某些结果之后跳转回当前位置。

```

int lexer::get_location() {
    return lexer::symbol_code_location;
}

```

(3) jump_location 函数

跳转到某一位置，用于控制跳转类语句（if、elif、else、for、while、def）的跳转，与 get_location 一起使用，注意：该函数跳转后不会读取字符，需要再次调用 get_next 函数以正确的获取字符。

```

void lexer::jump_location(int location) {
    if (location >= 0 && location < lexer::symbol_code_length)
        lexer::symbol_code_location = location;
}

```

(4) skip_block 函数

跳过整个代码区，遇到 dedend 后停止。该函数跳转后不会读取字符，需要再次调用 get_next 函数以正确的获取字符。用于控制跳转类语句（if、elif、else、for、while、def）的跳转。

```

void lexer::skip_block() {
    int i = symbol_code_location;
    int level_t=1;
    while(true) {
        symbol_code_location++;
        if (lexer::symbol_code_list[symbol_code_location] == word::control::DEDENT) {
            level_t--;
            if (level_t == 0) {
                break;
            }
        } else if (lexer::symbol_code_list[symbol_code_location] == word::control::INDENT) {
            level_t++;
        }
    }
    lexer::get_next();
}

```

(5) get_next 函数与、word_code、word_int、word_double、word_int、word_name 变量从 0 开始，依次获取 word 名称，并将其存储在 word_code 里面
 如果是 int 类型，则读取 symbol_int_list 中的 int 值，存储到 word_int 中；
 如果是 float 类型，则读取 symbol_double_list 中的 double 值，存储到 word_double 中；
 如果是 string 类型，则读取 symbol_string_list 中的 string 值，存储到 word_string 中；
 如果是 name 类型，则读取 symbol_name_list 中的 string 值，存储到 word_name 中；
 并将当前的位置指针加 1

```

void lexer::get_next() {
    if (lexer::symbol_code_location == lexer::symbol_code_length) {
        lexer::word_code = word::control::ENDMARKER;
        return;
    }
    int loc = lexer::symbol_code_location;
    lexer::word_code = lexer::symbol_code_list[loc];
    lexer::lineloc = lexer::symbol_location_list[loc];
    if (lexer::word_code >= word::type::NAME) {
        if (symbol_code_list[loc] == word::type::NAME)
            lexer::word_name = lexer::symbol_name_list[loc];
        if (symbol_code_list[loc] == word::type::STRING)
            lexer::word_string = lexer::symbol_string_list[loc];
        if (symbol_code_list[loc] == word::type::NUMBER)
            lexer::word_int = lexer::symbol_int_list[loc];
        if (symbol_code_list[loc] == word::type::FLOAT)
            lexer::word_double = lexer::symbol_double_list[loc];
    }
    lexer::symbol_code_location++;
}

```

(6) get_back 函数

与 get_next 函数相反，将 symbol_code_location-2，读取后再+1

(7) get_lexer_name 函数

获取文件读取的磁盘位置，由构造函数存入。在运行时符号表管理的时候函数表输出位置使用。

```

const char* lexer::get_lexer_name() {
    return lexer::lexer_name.c_str();
}

```

(8) code_length 函数

获取整个代码文件分割后的长度。

```

int lexer::code_length() {
    return lexer::symbol_code_length;
}

```

3.3.1 Private 类型

Private 成员变量：

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

public 成员函数：

| 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 char 字符 |
| <code>getword</code> | 无 | <code>void</code> | 获取下一个 word 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 |

(1) `getchar` 读取 char 字符

采用了与 `pl0` 相似的读取方法，定义行缓冲与字符缓冲，读取文件读取下一个字符。利用 `level` 判断函数当前的层次结构，与 `python` 相似的 `tab` 键划分结构。将所有的 `tab` 键层次结构进入时转化为 '`\t`'，将退格符转化为 "`\b`"。该部分代码片段如下：

```
if (i > lexer::level + 1) {
    error::syntax_error(lexer::line_total, error::space_error);
}
else if (i < lexer::level) {
    std::string temp = "";
    for (int j = i; j < lexer::level; j++) {
        temp.append("\b");
    }
    lexer::line_buf = temp + lexer::line_buf;
}
else if (i == lexer::level+1) {
    lexer::line_buf = '\t' + lexer::line_buf;
}
```

(2) `getword` 读取 word 字

调用 `getchar` 函数，根据不同的类型调用不同的分析子程序

```
void lexer::getword() {
    if (lexer::char_buf == 0) {
        //如果是0则直接返回结束标识符
        lexer::word_type = word::control::ENDMARKER;
        return;
    }
    while (lexer::char_buf == ' ') {
        lexer::getchar();
    }
    char ch = lexer::char_buf;
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || ch=='_') {
        //字母开头分析
        lexer::letter_start();
    }
    else if (ch >= '0' && ch <= '9') {
        //数字开头分析
        lexer::number_start();
    }
    else if (ch == '"') {
        lexer::string_start();
    }
    else {
        lexer::punctuation_start();
    }
}
```

(3) `get_number_from_word` 将字符转化为数字

由 `number_start` 函数调用，用于将字符转化为数字，不能判断是否为 `float` 浮点型，只能取出整型，具体的判断过程在 `number_start` 函数中去处理。

```

int lexer::get_number_from_word() {
    int k = 0;
    int num = 0;
    do {
        if (char_buf == '_')
            continue;
        num = 10 * num + lexer::char_buf - '0';
        k++;
        lexer::getchar();
    } while((lexer::char_buf >= '0' && lexer::char_buf <= '9') || lexer::char_buf == '_');
    // 获取数字的值
    k--;
    if (k > max_number_length) {
        error::syntax_error(lexer::line_total, error::number_exceeded_error);
    }
    return num;
}

```

(4) string_start、punctuation_start、number_start、letter_start

这 4 个函数都为 getword 调用，为词法分析程序的关键判断逻辑，此内容不进行赘述，产生的符号都在符号表中体现出来了。其中，对于 number_start 函数，如果其中有“.”，则转化为 float 浮点型，若没有，则转化为 int 整型。

3.4 运行时基础结构

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。

头文件名称：runningbasic.h

名字空间命名：running

变量：

| 名称 | 类型 | 含义 |
|----------------|-----|------------------------|
| name_type | 枚举型 | 所有在 py0 语言出现的符号的变量类型 |
| function_table | 结构体 | 当 name_type 为函数时，调用此结构 |
| name_table | 结构体 | 符号表中存放的基本结构体 |

(1) name_type 枚举型

整个 py0 所支持的全部类型如下，包括了 int 整型、float 浮点型、string 字符串、array_int 整型向量、array_float 浮点型向量、array_string 字符型向量、function 函数类型、list 列表（此内容已被删除，因为其结构和控制逻辑过于复杂，支持此类型将导致程序复杂度大大提高）。

```
enum name_type {
    INT,
    FLOAT,
    STRING,
    ARRAY_INT,
    ARRAY_FLOAT,
    ARRAY_STRING,
    FUNCTION,
    LIST
};
```

(2) function_table 结构体

记录函数结构体，结构中包括了待执行函数的分词对象指针、分词对象的起点位置、参数个数、是否有不定参数（被取消，因为结构控制复杂）、函数参数列表、是否为嵌入式函数、嵌入式函数指针（此内容见 initfunction 部分的详细介绍）。

```
struct function_table {
    lexer* lex_table; //执行参数表
    int start_position; //起始点位置
    int parameter_num; //参数个数
    bool non_def_num; //是否有不确定个数的参数
    std::vector<std::string> *name_list;

    bool insert_function;
    void* function_pointer;
    //phrase::return_value(init_function::* f_max)
};
```

(3) name_table 结构体

用于记录到符号表中的结构体类型，其中 ntype 表示变量类型，见 (1)，name 为变量名字，level 为变量所

在的层次，用于在退出 if、elif、else、for、while、def 等结构时清理变量

pointer 表示变量值的指针，voi*表示可以存储所有类型指针，其中：

```
int 型用 pointer = (void*)(new int(value))
float 型用 pointer = (void*)(new double(value))
string 型用 pointer = (void*)(new std::string(value))
array_int 型用 std::vector<int>* val = new std::vector<int>()
        pointer = (void*)(val)
array_float 型用 std::vector<double>* val = new std::vector<double>()
        pointer = (void*)(val)
array_string 型用 std::vector<std::string>* val = new std::vector<std::string>()
        pointer = (void*)(val)
function 型用 function_table* val = new function_table
        pointer = (void*)(val)
```

```
struct name_table {
    name_type ntype;
    std::string name;
    int level; //表示变量所处的层级
    void* pointer; //指向相关结构体的指针
    bool exist; //表示是否被删除
    int reftimes; //表示引用次数
};
```

3.5 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。

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

| 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 |

(1) running_level

当前符号表的层级，在对符号表进行添加操作的时候自动计入变量层级，用于退出 if、else、elif、def、while、for 时候的变量清除操作。

(2) max_level

代码块层级上限，定义为 9999，超出后会报错。

(3) max_function_level

函数嵌套层数上限，定义为 999，超出后会报错。

(4) running_table

运行符号表，采用 vector 操作，running::name_table 指针类型。

(5) save_table

符号表的保存，只有进入函数的时候，保存函数位置之下所有符号表表项，退出函数后自动将保存的内容退出一层，再次添加回符号表。

(6) import_list

为了防止重复 import 造成错误，定义该变量结构，用于记录所有的 import 的物理位置，启动程序前要先添加主程序的路径，防止自己引用自己。

3.5.2 public 成员函数

| 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, void std::string name, void* value | | 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_levetdown | 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 |

(1) running_table_class 构造函数

初始化构造函数，将代码层级设为 0

```
running_table_class::running_table_class() {
    running_level = 0;
}
```

(2) append_table 函数

添加一个对象到符号表，传入类型、名字、值三个变量，值为 void 指针类型，具体已经在符号表基础结构中详细说明，这里不再赘述。首先判断该名字是否在符号表中，如果没有则添加，如果在符号表中，则清除该内容，并对其进行替换。将当前的代码层级自动填入其中。

```

void running_table_class::append_table(running::name_type type, std::string name, void* value) {
    running::name_table *tab = running_table_class::get_items(name);
    if (tab == NULL) {
        running::name_table* table = new running::name_table();
        table->level = running_table_class::running_level;
        table->ntype = type;
        table->name = name;
        table->exist = true;
        table->pointer = value;
        table->reftimes = 1;
        running_table_class::running_table.push_back(table);
    }
    else {
        tab->ntype = type;
        tab->pointer = value;
    }
}

```

(3) get_items 函数与 get_items_location 函数

都用于通过名字找变量，一个是返回变量的内容，一个是返回变量的位置

```

running::name_table* running_table_class::get_items(std::string name) {
    for (int i = running_table.size() - 1; i >= 0; i--) {
        if (running_table[i]->name == name && running_table[i]->exist == true)
            return running_table[i];
    }
    return NULL;
}

int running_table_class::get_items_location(std::string name) {
    for (int i = running_table.size() - 1; i >= 0; i--) {
        if (running_table[i]->name == name && running_table[i]->exist == true)
            return i;
    }
    return -1;
}

```

(4) get_table 函数

返回总符号表

```

std::vector<running::name_table*>* running_table_class::get_table() {
    return &(running_table_class::running_table);
}

```

(5) running_levelup 函数

用于将代码的层级提高，检测是否超上限。

```

int running_table_class::running_levelup() {
    running_table_class::running_level++;
    if (running_table_class::running_level >= running_table_class::max_level) {
        error::syntax_error(error::code_level_exceed);
        return -1;
    }
    return (int)running_table.size();
}

```

(6) running_leveledown 函数

用于将代码的层级降低，同时清除所有层级降低时对应的变量。原本的设计是删除指针，但是在函数返回值处理的时候，如果某个函数 fun 返回了自身内部的变量值，则需要对该变量值进行重新复制保存，再进行返回，需要复杂的逻辑判断，限于时间关系，我直接在函数返回的时候返回了值的指针，因此这里我将删除值的指针的操作给去除了，事实上造成了大量的内存泄露。

```

int running_table_class::running_leveledown() {
    int nowlevel = running_table_class::running_level;
    if (nowlevel > 0) {
        running_table_class::running_level--;
    }
    else {
        return -1;
    }
    int del_num = 0;
    for (int i = running_table.size() - 1; i >= 0; i--) {
        if (running_table[i]->level == nowlevel) {
            if (running_table[i]->exist == true) {
                //delete running_table[i]->pointer;
            }
            //delete running_table[i];
            del_num++;
        }
        else {
            break;
        }
    }
    for (int i = 0; i < del_num; i++) {
        running_table.pop_back();
    }
    return running_table.size();
}

```

(7) running_delete_from 函数

从给定位置删除函数表，同（6），限于时间关系，为了避免过于复杂的逻辑判断，我将删除值的指针的操作给去除了，事实上造成了大量的内存泄露。

```

void running_table_class::running_delete_from(int loc) {
    if (loc < 0) {
        return;
    }
    int del_num = 0;
    for (int i = running_table.size() - 1; i >= loc; i--) {
        if (running_table[i]->exist == true) {
            //delete running_table[i]->pointer;
        }
        //delete running_table[i];
        del_num++;
    }
    for (int i = 0; i < del_num; i++) {
        running_table.pop_back();
    }
}

```

(8) in_function 函数

进入函数时调用，根据名字自动删除函数名之后的变量值，并将其保存到堆栈，在退出函数时将其返回到符号表。返回删除的位置，以便 out_function 的时候指定删除位置。

```

int running_table_class::in_function(std::string name) {
    int loc = running_table_class::get_items_location(name);
    if (loc <= -1) {
        return -1;
    }
    if ((int)save_table.size() > max_function_level) {
        error::syntax_error(error::function_level_exceed);
        return -1;
    }
    std::stack<running::name_table*> temp_save;
    int items_num = 0;
    for (int i = running_table.size() - 1; i >= loc; i--) {
        temp_save.push(running_table[i]);
        items_num++;
    }
    save_table.push(temp_save);
    for (int i = 0; i < items_num; i++) {
        running_table.pop_back();
    }
    return running_table.size();
}

```

(9) out_function 函数

与 in_function 互为逆过程，删掉从给定位置后面的符号，然后从堆栈恢复符号表。

```
void running_table_class::out_function(int location) {
    running_table_class::running_delete_from(location);
    std::stack<running::name_table*> temp_save = save_table.top();
    save_table.pop();
    while (temp_save.empty() != true) {
        running::name_table* temp = temp_save.top();
        temp_save.pop();
        temp_save.push_back(temp);
    }
}
```

(10) del_function 函数

给定名字删除变量，事实上这个删除不是真的删除，而是将变量的存在标记为 false，因为直接删除将会造成大量的符号表内容的移动，降低效率。

```
void running_table_class::del_function(std::string name) {
    for (int i = running_table.size() - 1; i >= 0; i--) {
        if (running_table[i]->name == name && running_table[i]->exist == true) {
            running_table[i]->exist = false;
            delete running_table[i]->pointer;
        }
    }
}
```

(11) in_import 函数

等级 import 路径，返回是否重复，如果重复则表明该内容已经被 import，防止重复 import 操作。

```
bool running_table_class::in_import(std::string name) {
    int i;
    for (i = 0; i < (int)import_list.size(); i++) {
        if (import_list[i] == name) {
            return false;
        }
    }
    import_list.push_back(name);
    return true;
}
```

(12) out_import 函数

与 in_import 操作相反，从列表删除该内容。但是由于此内容没有时间完成，import 一旦进行将不可逆，在语义分析中不会将 import 对象清除。

```
void running_table_class::out_import(std::string name) {
    std::vector<std::string>::iterator it;
    for (it=import_list.begin();it!=import_list.end();it++) {
        if ((*it) == name) {
            import_list.erase(it);
            break;
        }
    }
}
```

(13) show_table 函数

用于展示整个符号表，由于逻辑判断复杂我仅仅展示部分代码。

```

void running_table_class::show_table() {
    printf("\n\n");
    printf("name table: \n tpye name level value\n");

    for (int i = 0; i < (int)running_table.size(); i++) {
        running::name_table* temp = running_table[i];
        if (temp->exist == false) {
            continue;
        }

        std::vector<int>* nums = (std::vector<int>*)(temp->pointer);
        std::vector<double>* floats = (std::vector<double>*)(temp->pointer);
        std::vector<std::string>* strings = (std::vector<std::string>*)(temp->pointer);
        running::function_table* functions = (running::function_table*)(temp->pointer);

        switch (temp->ntype) {
        case running::name_type::INT:
            printf(" int %s %d %d\n", temp->name.c_str(), temp->level,
                   *(int*)(temp->pointer));
            break;
        }
    }
}

```

3.6 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。

头文件名称：phrasebasic.h

名字空间命名：phrase

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

(1) return_value

所有带有返回值的子程序的返回结构，包括了是否有返回值（这个内容被取消，即使是没有值的函数也强制返回 int 类型的 0），返回值类型（int、float、string、array_int、array_float、array_string、function），返回值指针，返回值名字（仅当返回值是函数的时候需要），是否发生错误，错误类型（这两个只有内联函数调用的使用，详见 initfunction）。

```

struct return_value {
    bool hasvalue = true;
    running::name_type type; //如果有返回值，则找到返回值类型
    void* value; //表示返回值的指针
    std::string name;

    bool error = false;
    std::string error_type;
};

```

(2) bracket_type

用于存入堆栈，记录当前上一次的括号类型。

```

enum bracket_type {
    bracket,
    para
};

```

(3) infunction_type

记录函数类型和参数的列表。

```

struct infunction_type {
    std::vector<running::name_type*>* type ;
    std::vector<void*>* parameter ;
};

```

3.7 函数初始化

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。

头文件名称： initfunction.h

名字空间命名： init_name

类名称： init_function

| 名称 | 参数列表 | 返回值 | 含义 |
|---------------------|--------------------------|----------------------|-----------|
| init_function | running_table_class* | 无 | 构造函数 |
| function_max | phrase::infunction_type* | phrase::return_value | 内联求最大值函数 |
| function_min | phrase::infunction_type* | phrase::return_value | 内联求最小值函数 |
| function_show_table | phrase::infunction_type* | phrase::return_value | 内联展示变量表函数 |
| function_print | phrase::infunction_type* | phrase::return_value | 内联格式化输出函数 |
| function_input | phrase::infunction_type* | phrase::return_value | 内联输入函数 |
| function_int | phrase::infunction_type* | phrase::return_value | 内联类型转换函数 |

| | | | |
|------------------------|--------------------------|----------------------|-----------------|
| function_float | phrase::infunction_type* | phrase::return_value | 内联类型转换函数 |
| function_string | phrase::infunction_type* | phrase::return_value | 内联类型转换函数 |
| function_range | phrase::infunction_type* | phrase::return_value | 内联产生 range 序列函数 |

(1) init_function 构造函数

构造函数，用于将所有的下列所有的函数进行初始化，利用函数指针将所有的函数指针传入到符号表中。

下面的代码为片段。

```
init_function::init_function(running_table_class* table) { //running_table_class* table
    this->table = table;
    init_name::table = table;
    typedef phrase::return_value(*funpointer) (phrase::infunction_type * infunction);

    //register the function "max"
    running::function_table *fun_max = new running::function_table;
    fun_max->insert_function = true;
    funpointer function_max = init_name::function_max;
    fun_max->function_pointer = (void*) (function_max);
    table->append_table(running::name_type::FUNCTION, "max", (void*) (fun_max));

    //register the function "min"
    running::function_table* fun_min = new running::function_table;
    fun_min->insert_function = true;
    funpointer function_min = init_name::function_min;
    fun_min->function_pointer = (void*) (function_min);
    table->append_table(running::name_type::FUNCTION, "min", (void*) (fun_min));

    //register the function "show_table"
    running::function_table* fun_show_table = new running::function_table;
    fun_show_table->insert_function = true;
    funpointer function_show_table = init_name::function_show_table;
    fun_show_table->function_pointer = (void*) (function_show_table);
    table->append_table(running::name_type::FUNCTION, "show_table", (void*) (fun_show_table));

    ...省略其他内容...
}

//register the function "string"
running::function_table* fun_string = new running::function_table;
fun_string->insert_function = true;
funpointer function_string = init_name::function_string;
fun_string->function_pointer = (void*) (function_string);
table->append_table(running::name_type::FUNCTION, "string", (void*) (fun_string));
```

(2) function_max 函数

用于求最大值，在 py0 语言中使用。函数列表只能为 1 个，且必须是 array_int 或 array_float 类型。

```
phrase::return_value init_name::function_max(phrase::infunction_type* infunction) {
    std::vector<running::name_type>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    //printf("%d", type[0]);
    phrase::return_value ret;

    if (type->size() == 1) {
        if ((*type)[0] == running::name_type::ARRAY_INT ||
            (*type)[1] == running::name_type::ARRAY_FLOAT) {
            if ((*type)[0] == running::name_type::ARRAY_INT) {
                std::vector<int>* vec = (std::vector<int>*) ((*parameter)[0]);
                int maxx = (*vec)[0];
                for (int i = 1; i < (int)vec->size(); i++) {
                    if ((*vec)[i] > maxx) {
                        maxx = (*vec)[i];
                    }
                }
                ret.type = running::name_type::INT;
                ret.value = (void*) (new int(maxx));
            }
        }
    }
}
```

```

    else {
        std::vector<double>* vec = (*std::vector<double>*)((*parameter)[0]);
        double maxx = (*vec)[0];
        for (int i = 1; i < (int)vec->size(); i++) {
            if ((*vec)[i] > maxx) {
                maxx = (*vec)[i];
            }
        }
        ret.type = running::name_type::FLOAT;
        ret.value = (void*)(new double(maxx));
    }
}
else {
    ret.error = true;
    ret.error_type = error::function_parameter_not_support_error;
}
else {
    ret.error = true;
    ret.error_type = error::function_parameter_num_error;
}
return ret;
}
}

```

(3) function_min 函数

与 function_max 相反。用于求最小值，在 py0 语言中使用。函数列表只能为 1 个，且必须是 array_int 或 array_float 类型。函数定义不再赘述。

(4) function_show_table 函数

用于展示所有变量列表，在 py0 语言中使用。函数为 0 个，调用运行时管理的 show_table 函数。

```

phrase::return_value init_name::function_show_table(phrase::infunction_type* infunction) {
    std::vector<running::name_type>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    phrase::return_value ret;

    if (type->size() != 0) {
        ret.error = true;
        ret.error_type = error::function_parameter_num_error;
    }
    ret.type = running::name_type::INT;
    ret.value = (void*)(new int(0));

    table->show_table();
    return ret;
}

```

(5) function_print 函数

用于格式化输出，在 py0 语言中使用。函数为最少 1 个，且第一个参数为 string 类型，表示待输出字符串。字符串中包含若干个花括号”{}”用于表示带填入参数。函数的参数个数必须大于花括号个数+1，不做超过的错误处理。输出 int、float、string 类型时直接输出，对于 array 类型变量输出圆括号形式，对于所有的函数均输出”function”。不进行格式化控制，因为这样将会导致控制复杂。

```

phrase::return_value init_name::function_print(phrase::infunction_type* infunction) {
    std::vector<running::name_type>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    phrase::return_value ret;

    if (type->size() == 0) {
        ret.error = true;
        ret.error_type = error::function_parameter_num_error;
        return ret;
    }
    if ((*type)[0] != running::STRING) {
        ret.error = true;
        ret.error_type = error::function_print_parameter_error;
        return ret;
    }

    std::string* print_std = (std::string*) (*parameter)[0];
    int para_loc = 1;
    for (int i = 0; i < (int)print_std->size(); i++) {
        if ((*print_std)[i] == '(') {
            i++;
            char next = (*print_std)[i];
            int leng = -1;
            if (next >= '0' && next <= '9') {
                leng = next - '0';
                i++;
                next = (*print_std)[i];
            }
            if (next != ')') {
                ret.error = true;
                ret.error_type = error::function_print_parameter_error;
                return ret;
            }
        }

        if ((int)type->size() <= para_loc) {
            ret.error = true;
            ret.error_type = error::function_print_parameter_error;
            return ret;
        }

        void *val = (*parameter)[para_loc];
        running::name_type type_val = (*type)[para_loc];
        std::vector<int>* vec_int;
        std::vector<double>* vec_double;
        std::vector<std::string>* vec_string;

        switch (type_val) {
        case running::name_type::INT:
            printf("%d", *(int*)val);
            break;
        case running::name_type::FLOAT:
            printf("%f", *(double*)val);
            break;

        case running::name_type::STRING:
            printf("%s", (*(std::string*)val).c_str());
            break;
        case running::name_type::FUNCTION:
            printf("function_type");
            break;
        case running::name_type::ARRAY_INT:
            printf("(");
            vec_int = (std::vector<int>*)val;
            for (int j = 0; j < (int)vec_int->size(); j++) {
                printf("%d", (*vec_int)[j]);
                if (j != (int)vec_int->size() - 1) {
                    printf(",");
                }
            }
            printf(")");
            break;
        }
    }
}

```

```

    case running::name_type::ARRAY_FLOAT:
        vec_double = (std::vector<double>*)val;
        printf("(");
        for (int j = 0; j < (int)vec_double->size(); j++) {
            printf("%f", (*vec_double)[j]);
            if (j != (int)vec_double->size() - 1) {
                printf(",");
            }
        }
        printf(")");
        break;
    case running::name_type::ARRAY_STRING:
        vec_string = (std::vector<std::string>*)val;
        printf("(");
        for (int j = 0; j < (int)vec_string->size(); j++) {
            printf("\n%s\n", ( *vec_string)[j]).c_str());
            if (j != (int)vec_string->size() - 1) {
                printf(",");
            }
        }
        printf(")");
        break;
    }
    para_loc++;
}

else {
    printf("%c", (*print_std)[i]);
}
}

ret.type = running::name_type::INT;
ret.value = (void*)(new int(0));
return ret;
}
}

```

(6) function_input 函数

用于函数输入，在 py0 语言中使用。可以带有 0 个参数或 1 个参数，如果带有 1 个参数，则必须为 string 类型，表示提示信息，此语法与 python 一致。

```

phrase::return_value init_name::function_input(phrase::infunction_type* infunction) {
    std::vector<running::name_type*>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    phrase::return_value ret;

    if (type->size() == 1) {
        printf("%s", (*std::string)((*parameter)[0]).c_str());
    }
    else if (type->size() >= 2) {
        ret.error = true;
        ret.error_type = error::function_parameter_num_error;
        return ret;
    }

    ret.type = running::name_type::STRING;
    char buf[255];
    scanf_s("%s", buf, 254);
    buf[254] = '\0';
    ret.value = (void*)(new std::string(buf));
    return ret;
}

```

(7) function_int 函数、function_float 函数、function_string 函数

用于变量类型转换，在 py0 语言中使用。参数只能为 1，传入参数只能为 int、float、string 类型。下面仅以 function_string 函数进行举例说明。

```

phrase::return_value init_name::function_string(phrase::infunction_type* infunction) {
    std::vector<running::name_type>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    phrase::return_value ret;

    if (type->size() != 1) {
        ret.error = true;
        ret.error_type = error::function_parameter_num_error;
        return ret;
    }

    int val_int;
    double val_double;
    std::string val_string;
    std::string temp1;
    std::string temp2 = "";
    int zeronum = 0;
    switch ((*type)[0]) {
        case running::name_type::INT:
            val_int = *(int*)((*parameter)[0]);
            ret.value = (void*)(new std::string(std::to_string(val_int)));
            break;
        case running::name_type::FLOAT:
            val_double = *(double*)((*parameter)[0]);
            temp1 = std::to_string(val_double);

            for (int i = temp1.length() - 1; i >= 0; i--) {
                if (temp1[i] == '0') {
                    zeronum++;
                }
                else {
                    break;
                }
            }
            for (int i = 0; i < (int)temp1.length() - zeronum; i++) {
                temp2 += temp1[i];
            }
            ret.value = (void*)(new std::string(temp2));
            break;
        case running::name_type::STRING:
            val_string = *((std::string*)((*parameter)[0]));

            ret.value = (void*)(new std::string(val_string));
            break;
        default:
            ret.error = true;
            ret.error_type = error::function_parameter_num_error;
            return ret;
            break;
    }

    ret.type = running::name_type::STRING;
    return ret;
}

```

(8) function_range 函数

产生类似于 python 中的 range 的列表，只能产生整数。可以传入 1 个、2 个或 3 个参数，如果传入 1 个参数则表示起点为 0，终点为参数，步长为 1；2 个参数确定起点和终点；3 个参数确定起点终点和步长。

```

phrase::return_value init_name::function_range(phrase::infunction_type* infunction) {
    std::vector<running::name_type>* type = infunction->type;
    std::vector<void*>* parameter = infunction->parameter;
    phrase::return_value ret;

    if (type->size() < 1 || type->size() > 3) {
        ret.error = true;
        ret.error_type = error::function_parameter_num_error;
        return ret;
    }

    int startpos = 0;
    int endpos = 0;
    int stride = 1;

```

1 个参数设定

```

if (type->size() >= 1) {
    if ((*type)[0] != running::name_type::INT) {
        ret.error = true;
        ret.error_type = error::function_parameter_not_support_error;
        return ret;
    }
    int val = *(int*)((*parameter)[0]);
    endpos = val;
}

```

2 个参数设定

```

if (type->size() >= 2) {
    if ((*type)[1] != running::name_type::INT) {
        ret.error = true;
        ret.error_type = error::function_parameter_not_support_error;
        return ret;
    }
    int val = *(int*)((*parameter)[1]);
    startpos = endpos;
    endpos = val;
}

```

3 个参数

```

if (type->size() >= 3) {
    if ((*type)[2] != running::name_type::INT) {
        ret.error = true;
        ret.error_type = error::function_parameter_not_support_error;
        return ret;
    }
    int val = *(int*)((*parameter)[2]);
    stride = val;
}

```

产生序列

```

std::vector<int>* vec = new std::vector<int>();
for (int i = startpos; i < endpos; i += stride) {
    vec->push_back(i);
}

ret.value = vec;
ret.type = running::name_type::ARRAY_INT;
return ret;
}

```

3.8 语义及语法分析

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。

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：

| 名称 | 类型 | 含义 |
|----|----|----|
|----|----|----|

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

(1) `function_return` 变量

调用函数的时候，创建一个新的 `phraser` 对象，设置该值为 `int` 型 0，然后调用 `run` 对象执行函数操作，如果遇到 `return` 则会将该值赋值为 `return` 的值，如果没有 `return` 则没有变化。在调用者接收的时候就可以通过该公有成员得到其值。

(2) `code_lexer`、`table` 变量

在构造函数中将分词和符号表对象的指针进行赋值。

(3) `bracket_balance` 变量

用于平衡圆括号和方括号。遇到圆括号或方括号都会入栈一个值，在遇到另一半括号的时候出栈一个值，以确定是否符合语法。

(4) `break_sign`、`continue_sign`、`return_sign` 变量

这 3 个变量一旦为 `true`，则表明代码段执行结束，会直接退出 `file_input` 函数，由调用该 `file_input` 函数的模块重设其为 `false`。

(5) `loop_counting` 变量

用于对循环层数进行计数，当 `loop_counting` 为 0 时无法进行 `break` 和 `continue` 操作。

函数表：

| 名称 | 参数列表 | 返回值 | 含义 |
|--------------------------|--|-------------------|---|
| <code>phraser</code> | <code>lexer* code_lexer,</code> <code>running_table_class* table</code> | 无 | 构造函数 |
| <code>run</code> | 无 | <code>void</code> | 开始执行 |
| <code>file_input</code> | 无 | <code>void</code> | 分割换行符与简单句 |
| <code>stmt</code> | 无 | <code>void</code> | 区分简单句与复合句 |
| <code>simple_stmt</code> | 无 | <code>void</code> | 分割简单句与换行符 |
| <code>small_stmt</code> | 无 | <code>void</code> | 区分简单句类型 |
| <code>flow_stmt</code> | 无 | <code>void</code> | 区分 <code>break</code> 、 <code>continue</code> 等 |

| | | | |
|-----------------------------|---|---------------------------------------|-------------------------|
| <code>del_del_stmt</code> | 无 | <code>void</code> | 删除句 |
| <code>del_atom</code> | <code>std::string name</code> | <code>bool</code> | 删除字句 |
| <code>expr_stmt</code> | 无 | <code>void</code> | 表达式判断 |
| <code>break_stmt</code> | 无 | <code>void</code> | <code>break</code> 句 |
| <code>continue_stmt</code> | 无 | <code>void</code> | <code>continue</code> 句 |
| <code>return_stmt</code> | 无 | <code>void</code> | <code>return</code> 句 |
| <code>import_stmt</code> | 无 | <code>void</code> | <code>import</code> 句 |
| <code>compound_stmt</code> | 无 | <code>void</code> | 复合句判断 |
| <code>array_def</code> | 无 | <code>phrase::return_value</code> | 定义向量 |
| <code>or_test</code> | 无 | <code>phrase::return_value</code> | 分割 or 运算符 |
| <code>and_test</code> | 无 | <code>phrase::return_value</code> | 分割 and 运算符 |
| <code>not_test</code> | 无 | <code>phrase::return_value</code> | 分割 not 运算符 |
| <code>comparison</code> | 无 | <code>phrase::return_value</code> | 分割>、<、==等运算符 |
| <code>expr</code> | 无 | <code>phrase::return_value</code> | 分割+、-运算符 |
| <code>term</code> | 无 | <code>phrase::return_value</code> | 分割*、/运算符 |
| <code>factor</code> | 无 | <code>phrase::return_value</code> | 调用 atom 与 trailer |
| <code>atom</code> | 无 | <code>phrase::return_value</code> | 判断原子变量 |
| <code>trailer</code> | <code>phrase::return_value*</code> | <code>void</code> | 分割下标访问与函数调用 |
| <code>subscript</code> | <code>phrase::return_value*</code> | <code>void</code> | 下标访问 |
| <code>arglist</code> | <code>phrase::return_value*</code> | <code>void</code> | 函数调用 |
| <code>argument</code> | 无 | <code>phrase::infunction_type*</code> | 函数拆分参数列表 |
| <code>if_stmt</code> | 无 | <code>void</code> | 分支结构 if、elif、else |
| <code>while_stmt</code> | 无 | <code>void</code> | 循环结构 while |
| <code>for_stmt</code> | 无 | <code>void</code> | 循环结构 for |
| <code>funcdef</code> | 无 | <code>void</code> | 函数定义 |
| <code>if_atom_format</code> | 无 | <code>void</code> | 分支结构字句 |
| <code>value_transmit</code> | <code>phrase::return_value*, phrase::return_value*</code> | <code>bool</code> | 不同类型值之间的自动转化函数 |
| <code>basic_operate</code> | <code>running::name_type type, void* val1, void* val2, word::punctuation operator_</code> | <code>void*</code> | +、-、*、/操作运算符运算 |
| <code>switch_value</code> | <code>phrase::return_value* name, running::name_type type</code> | <code>void</code> | 值类型转换 |
| <code>debug</code> | 无 | <code>void</code> | 用于调试使用 |

3.8.1 简单句

(1) phrase 函数

构造函数，用于保存分词对象指针和名字表

```
phraser::phraser(lexer* code_lexer, running_table_class* table) {
    phraser::code_lexer = code_lexer;
    phraser::table = table;
}
```

(2) run 函数

调用 file_input 开始程序。

```
void phraser::run() {
    code_lexer->get_next();
    file_input();
}
```

(3) file_input 函数

遇到 break、continue、return 标志则返回，遇到空行则跳过，否则调用 stmt 函数进行执行。

```
void phraser::file_input() {
    int word_code = code_lexer->word_code;
    while (word_code != word::control::ENDMARKER) {
        //执行到结束为止
        if (break_sign == true || continue_sign == true || return_sign==true
            || word_code == word::control::DEDENT) {
            break;
        }

        if (word_code == word::control::NEWLINE) {
            //如果是新的一行，则执行结束
            code_lexer->get_next();
            word_code = code_lexer->word_code;
        }
        else {
            stmt();
        }
        word_code = code_lexer->word_code;
    }
}
```

(4) stmt 函数

判断语句是简单句还是复合句，调用相应的处理函数。

```
void phraser::stmt() {
    int word_code = code_lexer->word_code;

    if (word_code == word::reserved::DEL ||
        word_code == word::type::NAME ||
        word_code == word::reserved::PASS ||
        word_code == word::reserved::IMPORT ||
        word_code == word::reserved::BREAK ||
        word_code == word::reserved::CONTINUE ||
        word_code == word::reserved::RETURN) {
        simple_stmt();
    }

    else if (word_code == word::reserved::IF ||
              word_code == word::reserved::WHILE ||
              word_code == word::reserved::FOR ||
              word_code == word::reserved::DEF) {
        compound_stmt();
    }

    else {
        error::syntax_error(error::carry_return_error, code_lexer->lineloc);
    }
}
```

(5) simple_stmt 函数

简单句处理结构，简单句为 small_stmt 加换行符

```
void phraser::simple_stmt() {
    small_stmt();
    int word_code = code_lexer->word_code;
    if (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
    }
    else {
        error::syntax_error(error::carry_return_error, code_lexer->lineloc);
    }
}
```

(6) small_stmt 函数

判断简单句类型：赋值语句、del 删除语句、pass 语句、程序控制语句、import 语句。

```
void phraser::small_stmt() {

    int word_code = code_lexer->word_code;
    if (word_code == word::reserved::DEL) {
        code_lexer->get_next();
        del_del_stmt();
    }
    else if (word_code == word::type::NAME) {
        expr_stmt();
    }
    else if (word_code == word::reserved::PASS) {
        code_lexer->get_next();
        pass();
    }

    else if (word_code == word::reserved::BREAK ||
             word_code == word::reserved::CONTINUE ||
             word_code == word::reserved::RETURN) {
        flow_stmt();
    }
    else if (word_code == word::reserved::IMPORT) {
        code_lexer->get_next();
        import_stmt();
    }
    else{
        error::syntax_error(error::unrecognized_syntax_error, code_lexer->lineloc);
    }
}
```

(7) flow_stmt 函数

判断程序控制语句 break、continue、return

```
void phraser::flow_stmt() {
    int word_code = code_lexer->word_code;
    if (word_code == word::reserved::BREAK) {
        code_lexer->get_next();
        break_stmt();
    }
    else if (word_code == word::reserved::CONTINUE) {
        code_lexer->get_next();
        continue_stmt();
    }
    else if (word_code == word::reserved::RETURN) {
        code_lexer->get_next();
        return_stmt();
    }
}
```

(8) del_del_stmt 函数

del 删除语句用于分割变量之间的逗号，并依次调用删除字句进行删除。

```

void phraser::del_del_stmt() {
    int word_code = code_lexer->word_code;
    if (word_code == word::type::NAME) {
        std::string name = code_lexer->word_name;
        if (!del_atom(name)) {
            error::syntax_error(error::del_name_error, code_lexer->lineloc);
        }
    }
    else {
        error::syntax_error(error::del_type_error, code_lexer->lineloc);
    }

    code_lexer->get_next();
    word_code = code_lexer->word_code;
    while (word_code == word::punctuation::COMMA) {
        code_lexer->get_next();
        if (word_code == word::type::NAME) {
            std::string name = code_lexer->word_name;
            if (!del_atom(name)) {
                error::syntax_error(error::del_name_error, code_lexer->lineloc);
            }
        }
        else {
            error::syntax_error(error::del_type_error, code_lexer->lineloc);
        }
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
}

```

(9) del_atom 函数

删除字句，调用符号表中的删除函数进行删除。

```

bool phraser::del_atom(std::string name) {
    if (table->get_items(name) == NULL) {
        return false;
    }
    table->del_function(name);
    return true;
}

```

3.8.2 表达式

(10) expr_stmt 函数

对基本表达式类语句进行执行

```

void phraser::expr_stmt() {
    std::string waiting_value = code_lexer->word_name;
    code_lexer->get_next();
    int word_code = code_lexer->word_code;
}

```

判断等号，对等号左边的变量进行赋值，采用覆盖原先变量的方式进行赋值。

```

if (word_code == word::punctuation::ASSIGN) {
    //等号，表示赋值语句
    code_lexer->get_next();
    word_code = code_lexer->word_code;
    phrase::return_value value = or_test();
    table->append_table(value.type, waiting_value, value.value);
}

```

否则判断是否为 += 、 -= 、 *= 、 /= 计算赋值类语句

```

else //加等于的计算赋值语句
    if (word_code == word::punctuation::ADD_EQUAL ||
        word_code == word::punctuation::MINUS_EQUAL ||
        word_code == word::punctuation::MULTI_EQUAL ||
        word_code == word::punctuation::DIV_EQUAL) {

        running::name_table* name = table->get_items(waiting_value);
        if (name == NULL) {
            error::syntax_error(error::value_not_found_error, code_lexer->lineloc);
        }
    }
}

```

对解释到的赋值方法进行计算赋值

```

else {
    phrase::return_value frontal_value;
    frontal_value.type = name->nype;
    frontal_value.hasvalue = true;
    frontal_value.value = name->pointer;
    code_lexer->get_next();

    phrase::return_value value = or_test();
    if (!value_transmit(&frontal_value, &value)) {
        error::syntax_error(error::value_type_error, code_lexer->lineloc);
    }
    void* result = basic_operate(frontal_value.type, (void*)(&frontal_value),
        (void*)(&value), (word::punctuation) word_code);
    if (frontal_value.type == running::name_type::ARRAY_INT) {
        frontal_value.type = running::name_type::ARRAY_FLOAT;
    }
    table->append_table(frontal_value.type, waiting_value, result);
}
}

```

否则为调用无返回值类函数直接执行。

```

else {
    code_lexer->get_back();
    phrase::return_value ret = or_test();
    //error::syntax_error(error::expression_error, code_lexer->lineloc);
}

}

```

(11) break_stmt 函数

当 loop 层级大于 0 的时候判断为真，将 break 标志设置为 true。

```

void phraser::break_stmt() {
    if (loop_conting >= 1) {
        break_sign = true;
        return;
    }
    else {
        error::syntax_error(error::break_used_error, code_lexer->lineloc);
    }
}

```

(12) continue_stmt 函数

与 break 类似，当 loop 层级大于 0 的时候判断为真，将 continue 标志设置为 true。

```

void phraser::continue_stmt() {
    if (loop_conting >= 1) {
        continue_sign = true;
        return;
    }
    else {
        error::syntax_error(error::continue_used_error, code_lexer->lineloc);
    }
}

```

(13) return_stmt 函数

目前 return 函数我只能让其返回 1 个值，记录到 function_return 这个共有变量中，以供调用者读取。

```
void phraser::return_stmt() {
    int word_code = code_lexer->word_code;
    phrase::return_value ret = or_test();

    function_return.type= ret.type;
    function_return.value = ret.value;

    return_sign = true;
}
```

(14) import_stmt 函数

为了满足 import 语句，我在设计之初就将程序文件的运行设计为调用方式，这里我只需要调用该方法就可以直接完成 import 操作了。

```
void phraser::import_stmt() {
    int word_code = code_lexer->word_code;
    std::string name;
    if (word_code == word::type::NAME) {
        name = code_lexer->word_name;
    }
    else if (word_code == word::type::STRING) {
        name = code_lexer->word_string;
    }
    else {
        error::syntax_error(error::import_error, code_lexer->lineloc);
    }
    code_lexer->get_next();

    if (table->in_import(name)) {
        char* name2 = (char*)name.c_str();
        lexer* lex = new lexer(name2);
        phraser phr(lex, table);
        phr.run();
    }
}
```

(15) compound_stmt 函数

复合句判断函数包括对于分支结构的 if，循环结构的 for 和 while，对于函数定义 def 的判断。

```
void phraser::compound_stmt() {
    int word_code = code_lexer->word_code;
    if (word_code == word::reserved::IF) {
        if_stmt();
    }
    else if (word_code == word::reserved::WHILE) {
        while_stmt();
    }
    else if (word_code == word::reserved::FOR) {
        for_stmt();
    }
    else if (word_code == word::reserved::DEF) {
        funcdef();
    }
    else {
        error::syntax_error(error::carry_return_error, code_lexer->lineloc);
    }
}
```

(16) array_def 函数

定义向量函数，我希望做到的是向量列表如果有一个 string，则整个列表必须都是 array_string。如果都是数字，只要有 1 个浮点型，则必须所有的值都是浮点型 array_float，否则如果都是整数，则 array 为 array_int

型。于是我将所有读到的结果存入一个 vector 中，等到读完整个列表再做类型转换判断。

首先我先读取了第一个变量，并将其变量的类型

```
phrase::return_value phraser::array_def() {
    bracket_balance.push(phraser::bracket_type::para);

    phrase::return_value ret;
    int word_code = code_lexer->word_code;
    phrase::return_value ret1 = or_test();

    running::name_type type = ret1.type;
    std::vector<phrase::return_value> vec;
    running::name_type nowtype = type;
```

读入的类型必须为 int、float、string 这三种类型。

```
if (type == running::name_type::INT ||
    type == running::name_type::FLOAT ||
    type == running::name_type::STRING) {

    vec.push_back(ret1);
    word_code = code_lexer->word_code;
    while (word_code == word::punctuation::COMMA) {
        code_lexer->get_next();
        phrase::return_value ret2;
        ret2 = or_test();
        word_code = code_lexer->word_code;
        type = ret2.type;
    }
}
```

记录一个当前类型的变量，与新读入的变量做类型比较，并给出错误信息。

```
if (nowtype == running::name_type::STRING) {
    if (type == running::name_type::INT || type == running::name_type::FLOAT) {
        error::syntax_error(error::array_created_error, code_lexer->lineloc);
    }
}
else if (nowtype == running::name_type::INT) {
    if (type == running::name_type::STRING) {
        error::syntax_error(error::array_created_error, code_lexer->lineloc);
    }
    else if (type == running::name_type::FLOAT) {
        nowtype = running::name_type::FLOAT;
    }
}
else if (nowtype == running::name_type::FLOAT) {
    if (type == running::name_type::STRING) {
        error::syntax_error(error::array_created_error, code_lexer->lineloc);
    }
}

if (type == running::name_type::INT ||
    type == running::name_type::FLOAT ||
    type == running::name_type::STRING) {
    vec.push_back(ret2);
}
else {
    error::syntax_error(error::array_created_error, code_lexer->lineloc);
}
}
else {
    error::syntax_error(error::array_created_error, code_lexer->lineloc);
}
```

最后根据得到的类型，在未出现错误的情况下，建立各自不同的 vector 结构存储，并返回，下面仅以 float 类型为例，int 和 string 类型不做赘述。

```

    else if (nowtype == running::name_type::FLOAT) {
        ret.type = running::name_type::ARRAY_FLOAT;
        std::vector<double>* tempvec = new std::vector<double>();
        for (int i = 0; i < (int)vec.size(); i++) {
            phrase::return_value teret = vec[i];
            if (teret.type == running::name_type::INT) {
                int val = *(int*)teret.value;
                tempvec->push_back((double)val);
            }
            else if (teret.type == running::name_type::FLOAT) {
                double val = *(double*)teret.value;
                tempvec->push_back(val);
            }
            else {
                error::syntax_error(error::array_created_error, code_lexer->lineloc);
            }
        }
        ret.value = (void*)tempvec;
    }
}

```

最后将该数组返回

```

    return ret;
}

```

(17) or_test 函数与 and_test 函数

这两个函数相似，都是调用下级子程序，并判断是否存在 or 或 and，若不存在则直接返回，若存在，则通过 while 循环读取所有的变量，并判断返回 int 型的 0 或 int 型的 1。

下面仅以 or_test 举例：

```

phrase::return_value phraser::or_test() {
    int word_code = code_lexer->word_code;

    phrase::return_value ret1;
    phrase::return_value ret2;
    ret1 = and_test();

    word_code = code_lexer->word_code;
}

```

判断是否存在 or，如果存在则读取下一个值，并且 or 和 and 只支持 int 和 float 类型，对于其他的返回类型给出错误信息。

```

while (word_code == word::reserved::OR) {
    code_lexer->get_next();
    ret2 = and_test();
    word_code = code_lexer->word_code;

    if (!(ret1.hasvalue && ret2.hasvalue)) {
        error::syntax_error(error::expression_error, code_lexer->lineloc);
    }
    bool b1 = true;
    bool b2 = true;
    if (ret1.type == running::name_type::INT) {
        if (*(int*)(ret1.value) == 0)
            b1 = false;
    }
    else if (ret1.type == running::name_type::FLOAT) {
        if (*(double*)(ret1.value) == 0)
            b1 = false;
    }
    else {
        error::syntax_error(error::unsupported_type, code_lexer->lineloc);
    }
}

```

```

if (ret2.type == running::name_type::INT) {
    if (*(int*)(ret2.value) == 0)
        b2 = false;
}
else if (ret2.type == running::name_type::FLOAT) {
    if (*(double*)(ret2.value) == 0)
        b2 = false;
}
else {
    error::syntax_error(error::unsupport_type, code_lexer->lineloc);
}

```

变量值将会被保存到 ret1 中，并且只能是 int 类型的 0 或 1

```

ret1.type = running::name_type::INT;
if (b1 || b2) {
    ret1.value = (void*)(new int(1));
}
else {
    ret1.value = (void*)(new int(0));
}

return ret1;
}

```

(18) not_test 函数

检测是否出现 not，并调用下一级的子程序，如果出现 not 则翻转之前得到的类型。凡是 int 型或 float 型 0 都为 false，其余都是 true。

```

phrase::return_value phaser::not_test() {
    int word_code = code_lexer->word_code;
    phrase::return_value ret;
    bool isnot = false;
    if (word_code == word::reserved::NOT) {
        isnot = true;
        code_lexer->get_next();
    }
    ret = comparison();
}

```

判断 not 存在后，则翻转之前得到的值

```

if (isnot) {
    switch (ret.type) {
        case running::name_type::INT:
            if (*(int*)(ret.value) == 0) {
                ret.value = (void*)(new int(1));
            }
            else {
                ret.value = (void*)(new int(0));
            }
            break;
        case running::name_type::FLOAT:
            if (*(double*)(ret.value) == 0) {
                ret.value = (void*)(new double(1));
            }
            else {
                ret.value = (void*)(new double(0));
            }
            break;
        default:
            error::syntax_error(error::unsupport_type, code_lexer->lineloc);
    }
}
return ret;
}

```

(19) comparison 函数

调用子程序，并判断是否出现 >、<、==、>=、<=，若出现则再次调用子程序读取下一个片段，并对两个进行比较。调用类型转换函数，都转换为 int 类型进行比较。

```

phrase::return_value phraser::comparison() {
    int word_code = code_lexer->word_code;
    phrase::return_value ret1;
    phrase::return_value ret2;
    ret1 = expr();
    word_code = code_lexer->word_code;
    if (word_code == word::punctuation::LESS ||
        word_code == word::punctuation::MORE ||
        word_code == word::punctuation::EQUAL ||
        word_code == word::punctuation::MORE_EQUAL ||
        word_code == word::punctuation::LESS_EQUAL) {
        code_lexer->get_next();
    }
}

```

比较的片段如下

```

ret2 = expr();
switch_value(&ret1, running::name_type::INT);
switch_value(&ret2, running::name_type::INT);
int val1 = *(int*)ret1.value;
int val2 = *(int*)ret2.value;
int result = 1;

switch (word_code) {
case word::punctuation::LESS:
    if (val1 < val2)
        result = 1;
    else result = 0;
    break;
}

case word::punctuation::LESS_EQUAL:
    if (val1 <= val2)
        result = 1;
    else result = 0;
    break;
}

ret1.value = (void*)(new int(result));
}

return ret1;
}

```

(20) expr 函数和 term 函数

这两个函数都是 pl0 中就存在的函数，一个是用来判断+、-，另一个是用来判断*、/。

这里我仅以 expr 函数举例

```

phrase::return_value phraser::expr() {
    int word_code = code_lexer->word_code;
    phrase::return_value ret1;
    phrase::return_value ret2;
    ret1 = term();
    word_code = code_lexer->word_code;
    while (word_code == word::punctuation::ADD ||
           word_code == word::punctuation::MINUS) {

        int temp_operator = word_code;
        code_lexer->get_next();
        ret2 = term();
        word_code = code_lexer->word_code;
        if (!(ret1.hasvalue && ret2.hasvalue)) {
            error::syntax_error(error::expression_error, code_lexer->lineloc);
        }
        value_transmit(&ret1, &ret2);

        void* res = basic_operate(ret1.type, &ret1, &ret2, (word::punctuation)temp_operator);
        if (ret1.type == running::name_type::ARRAY_INT) {
            ret1.type = running::name_type::ARRAY_FLOAT;
        }
        ret1.value = res;
    }
    return ret1;
}

```

(21) factor 函数

此函数用来调用 atom 函数和 trailer 函数。需要注意的是，atom 之前函数传递的都是综合属性，而之后的 trailer 都是继承属性，需要传递指针对之前的值记进行修改。

```
phrase::return_value phraser::factor() {
    phrase::return_value ret;
    ret = atom();
    trailer(&ret);
    return ret;
}
```

(22) atom 函数

原子类型有一下几种情况：

1. 圆括号，则表示是一个 array
2. 整型、浮点型、字符串、名字，则表示是一个原子值
3. 名字又包括整型、浮点型、字符串、array、函数

```
phrase::return_value phraser::atom() {

    int word_code = code_lexer->word_code;
    phrase::return_value ret;
    ret.hasvalue = false;
```

针对第一种情况，调用 array_def 生成函数

```
if (word_code == word::punctuation::BRACKET_L ||
    word_code == word::punctuation::PAREN_L) {
    //圆括号，生成向量
    if (word_code == word::punctuation::PAREN_L) {
        code_lexer->get_next();

        phrase::return_value ret = array_def();
        word_code = code_lexer->word_code;
        if (word_code == word::punctuation::PAREN_R) {
            code_lexer->get_next();
        }
        else {
            error::syntax_error(error::expression_error, code_lexer->lineloc);
        }
    }
    return ret;
}
```

针对第二种情况，要做出不同的判断，并开辟相应的空间

```
if (word_code == word::type::NAME ||
    word_code == word::type::NUMBER ||
    word_code == word::type::FLOAT ||
    word_code == word::type::STRING) {
    ret.hasvalue = true;
    if (word_code == word::type::NUMBER) {
        ret.type = running::name_type::INT;
        int val = code_lexer->word_int;
        ret.value = (void*)(new int(val));
    }
    else if (word_code == word::type::FLOAT) {
        ret.type = running::name_type::FLOAT;
        double val = code_lexer->word_double;
        ret.value = (void*)(new double(val));
    }
    else if (word_code == word::type::STRING) {
        ret.type = running::name_type::STRING;
        std::string val = code_lexer->word_string;
        ret.value = (void*)(new std::string(val));
    }
}
```

如果是符号表的名字类型，则需要判断，如果是整型、浮点型、字符串则直接复制值，否则需要传递指针，需要用到后续的 trailer 函数去具体识别。

```

    else if (word_code == word::type::NAME) {
        running::name_table* name = table->get_items(code_lexer->word_name);
        if (name == NULL) {
            error::syntax_error(error::name_not_found, code_lexer->lineloc);
        }
        ret.name = code_lexer->word_name;
        if (name->nype == running::name_type::INT) {
            ret.type = running::name_type::INT;
            int val = *(int*)(name->pointer);
            ret.value = (void*)(new int(val));
        }
        else if (name->nype == running::name_type::FLOAT) {
            ret.type = running::name_type::FLOAT;
            double val = *(double*)(name->pointer);
            ret.value = (void*)(new double(val));
        }
        else if (name->nype == running::name_type::STRING) {
            ret.type = running::name_type::STRING;
            std::string val = *(std::string*)(name->pointer);
            ret.value = (void*)(new std::string(val));
        }
        else {
            ret.type = name->nype;
            ret.value = name->pointer;
        }
    }
}

```

最后将 ret 的值返回

```

    code_lexer->get_next();
    return ret;
}

```

(23) trailer 函数

trailer 可能遇到圆括号或方括号，如果是方括号，则调用 subscript 下标访问，需要前者是一个向量。否则如果是圆括号，则前者应当是一个函数。

```

void phraser::trailer(phrase::return_value* ret) {
    int word_code = code_lexer->word_code;
    if (word_code == word::punctuation::BRACKET_L ||
        word_code == word::punctuation::PAREN_L) {
        if (word_code == word::punctuation::BRACKET_L) {
            code_lexer->get_next();
            subscript(ret);
            word_code = code_lexer->word_code;
            if (word_code == word::punctuation::BRACKET_R) {
                code_lexer->get_next();
            }
            else {
                error::syntax_error(error::bracket_error, code_lexer->lineloc);
            }
        }
    }
}

```

如果是圆括号则表示函数调用，调用 arglist 函数

```

    else {
        code_lexer->get_next();
        arglist(ret);
        word_code = code_lexer->word_code;
        if (word_code == word::punctuation::PAREN_R) {
            code_lexer->get_next();
        }
        else {
            error::syntax_error(error::bracket_error, code_lexer->lineloc);
        }
    }
}

```

如果下标检测到的是另一半括号，则从括号堆栈中取出 top，判断是否有与之对应的另一半括号。

```

    else
      if (word_code == word::punctuation::BRACKET_R ||
          word_code == word::punctuation::PAREN_R) {
        if (bracket_balance.size() == 0) {
          error::syntax_error(error::bracket_error, code_lexer->lineloc);
        }
        if (word_code == word::punctuation::BRACKET_R) {

          if (bracket_balance.top() == phrase::bracket_type::bracket) {
            bracket_balance.pop();
          }
          else {
            error::syntax_error(error::bracket_error, code_lexer->lineloc);
          }
        }
      }
    }
}

```

(24) subscript 函数

表示前者是一个 array 类型，

首先进行类型判断，ret 必须是 array 类型，或一个 string 类型。并将括号类型入栈。

```

void phraser::subscript(phrase::return_value* ret) {
  phrase::return_value* retsub=new phrase::return_value;
  retsub->type = ret->type;
  retsub->name = ret->name;

  if (!(ret->type == running::name_type::ARRAY_FLOAT ||
        ret->type == running::name_type::ARRAY_INT ||
        ret->type == running::name_type::ARRAY_STRING ||
        ret->type == running::name_type::STRING)) {
    error::syntax_error(error::list_number_error, code_lexer->lineloc);
  }

  bracket_balance.push(phrase::bracket_type::bracket);
  phrase::return_value ret2 = or_test();

  int listnumber = 0;
}

```

判断下标必须是一个整型

```

if (ret2.type == running::name_type::INT) {
  listnumber = *(int*)ret2.value;
}
else {
  error::syntax_error(error::list_number_error, code_lexer->lineloc);
}

```

对于 array_float 类型和 string 类型的操作，仅以这两个为例，最后保存指针。

```

if (ret->type == running::name_type::ARRAY_FLOAT) {
  retsub->type = running::name_type::FLOAT;
  std::vector<double>* temp = (std::vector<double>*)(ret->value);
  if ((int)temp->size() <= listnumber) {
    error::syntax_error(error::list_number_outof_range, code_lexer->lineloc);
  }
  double val = (*temp)[listnumber];

  retsub->value = (void*)(new double(val));
}

```

```

.....
else if (ret->type == running::name_type::STRING) {
    retsub->type = running::name_type::STRING;
    std::string* temp = (std::string*)(ret->value);
    if ((int)temp->size() <= listnumber) {
        error::syntax_error(error::list_number_outof_range, code_lexer->lineloc);
    }
    std::string val = "";
    val.push_back((*temp)[listnumber]);
    retsub->value = (void*)(new std::string(val));
}
*ret = *retsub;
}

```

(25) arglist 函数

函数调用操作，首先调用 argument 函数检测所有的参数列表。判断函数是内联函数还是用户自定义函数，如果是前者，则只需要指针调用即可，如果是后者则需要

1. 提高符号表层级；
2. 清除并入栈后续变量；
3. 获取并记录当前代码位置；
4. 定义 lexer 指针传入符号表中找到的函数所在文件的分词对象指针；
5. 获取函数位置并跳转代码段；
6. 添加所有变量到变量表；
7. 定义 phraser 执行函数；
8. 将其默认返回值设置为 int 型 0；
9. 执行操作；
10. 读取返回值；
11. 清除符号表并出栈保存变量；
12. 降低符号表层级；
13. 代码段跳转。

我的整个程序都是围绕着函数执行和 import 操作设计的，从一开始就是如此，而函数执行确实是逻辑最为繁复的部分，也是我最后完成编写和测试的部分。

执行内联函数：

```

void phraser::arglist(phrase::return_value* ret) {
    if (!(ret->type == running::name_type::FUNCTION)) {
        error::syntax_error(error::not_function, code_lexer->lineloc);
    }

    bracket_balance.push(phrase::bracket_type::para);

    phrase::infunction_type* retlist = argument();

    running::function_table* func = (running::function_table*)ret->value;
    if (func->insert_function) {
        //嵌入式函数，用c语言实现
        typedef phrase::return_value(*funpointer)(phrase::infunction_type * infunction);

        funpointer funp = (funpointer)func->function_pointer;
        phrase::return_value rettemp = funp( retlist );
        ret->type = rettemp.type;
        ret->value = rettemp.value;
        if (rettemp.error) {
            error::syntax_error(rettemp.error_type, code_lexer->lineloc);
        }
    }
}

```

执行用户自定义函数：

```

else {
    int length = retlist->parameter->size();
    if (func->parameter_num != length) {
        error::syntax_error(error::function_parameter_num_error, code_lexer->lineloc);
    }
    int function_start_table_loc = table->in_function(ret->name);
    if (function_start_table_loc == -1) {
        error::syntax_error(error::not_function, code_lexer->lineloc);
    }
    int now_loc = code_lexer->get_location();
    int fun_loc = func->start_position;
    lexer* lex = func->lex_table;
    lex->jump_location(fun_loc);
    for (int i = 0; i < func->parameter_num; i++) {
        running::name_type type = (*(retlist->type))[i];
        void* val = (*(retlist->parameter))[i];
        std::string name = (*(func->name_list))[i];
        table->append_table(type, name, val);
    }
    phraser phr(lex, table);
    //create the return val
    phrase::return_value rettemp;
    rettemp.type = running::name_type::INT;
    rettemp.value = (void*)(new int(0));
    phr.function_return = rettemp;
    phr.run();

    rettemp = phr.function_return;
    ret->type = rettemp.type;
    ret->value = rettemp.value;

    table->out_function(function_start_table_loc);
    code_lexer->jump_location(now_loc);
    code_lexer->get_back();
    code_lexer->get_next();
}
}
}

```

(26) argument 函数

被 arglist 函数调用，获取所有的变量列表

```

phrase::infunction_type* phraser::argument() {
    phrase::infunction_type* ret=new phrase::infunction_type;
    ret->type = new std::vector<running::name_type>();
    ret->parameter = new std::vector<void*>();

    int word_code = code_lexer->word_code;
    if (word_code == word::punctuation::PAREN_R) {
        return ret;
    }
    phrase::return_value temp = or_test();

    ret->type->push_back(temp.type);
    ret->parameter->push_back(temp.value);

    word_code = code_lexer->word_code;
    while (word_code == word::punctuation::COMMA) {
        code_lexer->get_next();
        temp = or_test();

        ret->type->push_back(temp.type);
        ret->parameter->push_back(temp.value);
        word_code = code_lexer->word_code;
    }
    return ret;
}

```

3.8.3 复合语句

(27) if_stmt 函数

if 分支结构我的设计初衷是设计一个和 python 一样强大的结构，支持 if...elif...else 结构
调用 if_atom_format 函数，做基础结构的判断，并进行 or_test 判断结果检测。

```
void phraser::if_stmt() {
    code_lexer->get_next();
    int word_code = code_lexer->word_code;
    phrase::return_value ret = or_test();

    if_atom_format();
    word_code = code_lexer->word_code;

    //judge the return value of the or test.
    bool judge = true;
    if (ret.type == running::name_type::INT) {
        if (*int*) (ret.value) == 0) {
            judge = false;
        }
    }
    else if (ret.type == running::name_type::FLOAT) {
        if (*double*) (ret.value) == 0) {
            judge = false;
        }
    }
    else {
        error::syntax_error(error::if_syntax_error, code_lexer->lineloc);
    }
}
```

判断结果为 true，则执行代码块

```
//judge is true
if (judge) {
    //run the if block
    table->running_levelup();
    file_input();
    table->running_leveledown();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
    while (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
}
```

在判断结果为 true 的情况下，跳过所有的 elif 代码段

```
//skip all the elif block
while(word_code == word::reserved::ELIF) {

    code_lexer->get_next();
    phrase::return_value temp_ret = or_test();
    if_atom_format();
    word_code = code_lexer->word_code;

    code_lexer->skip_block();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
```

在判断结果为 true 的情况下，跳过所有的 else 代码段

```

if (word_code == word::reserved::ELSE) {
    code_lexer->get_next();

    if_atom_format();
    word_code = code_lexer->word_code;

    code_lexer->skip_block();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
}

```

判断结果为 false，记录是否存在一个 elif 代码段结果为 true

```

//judge is false
else {
    code_lexer->skip_block();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
    while (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }

    int already_elif = false;
    //state of else if
}

```

elif 结构判断，并执行

```

while (word_code == word::reserved::ELIF) {

    code_lexer->get_next();
    ret = or_test();
    judge = true;
    if (ret.type == running::name_type::INT) {
        if (*(int*) (ret.value) == 0) {
            judge = false;
        }
    }
    else if (ret.type == running::name_type::FLOAT) {
        if (*(double*) (ret.value) == 0) {
            judge = false;
        }
    }
    else {
        error::syntax_error(error::if_syntax_error, code_lexer->lineloc);
    }

    if_atom_format();
    word_code = code_lexer->word_code;

    //else if state, the judge is true
    if (judge && !already_elif) {
        table->running_levelup();
        file_input();
        table->running_leveledown();
        code_lexer->get_next();
        word_code = code_lexer->word_code;
        while (word_code == word::control::NEWLINE) {
            code_lexer->get_next();
            word_code = code_lexer->word_code;
        }
        already_elif = true;
    }
}

```

如果执行过 elif 代码段，则跳过所有的其余 elif 代码

```

else {
    code_lexer->skip_block();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
}

```

如果其余判断结果都为 false，则调用 else 代码段

```
//all of the judge aformationed is false, the run the else block
word_code = code_lexer->word_code;
if (word_code == word::reserved::ELSE) {
    code_lexer->get_next();
    if_atom_format();
    word_code = code_lexer->word_code;
    if (judge == true) {
        code_lexer->skip_block();
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
}
else {
    table->running_levelup();
    file_input();
    table->running_leveardown();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
    while (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
}
}
```

(28) while_stmt 函数

while 循环子程序，记录进入 while 的代码位置，执行循环体，判断退出条件，如果为 break 则直接跳过代码段退出循环，如果为 continue 则将 continue sign 设置为 false，并跳转代码到开头，重新执行。

```
void phraser::while_stmt() {
    int while_start_loc = code_lexer->get_location();
    code_lexer->get_next();
    int word_code = code_lexer->word_code;
    phrase::return_value ret = or_test();
    word_code = code_lexer->word_code;
    //while head
    if (word_code == word::punctuation::COLON) {
        code_lexer->get_next();
    }
    else {
        error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
    }
    word_code = code_lexer->word_code;
```

格式判断

```
    while (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    if (word_code == word::control::INDENT) {
        code_lexer->get_next();
    }
    else {
        error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
    }
}
```

循环层级加 1，开始循环，并记录判断循环条件是否满足

```

loop_conting += 1;
//in while
table->running_levelup();
while (true) {

    if (ret.type == running::name_type::INT) {
        if (*int*(ret.value) == 0) {
            break;
        }
    }
    else if (ret.type == running::name_type::FLOAT) {
        if (*double*(ret.value) == 0) {
            break;
        }
    }
    else {
        error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
    }
}

```

开始执行循环体，结束后判断 break 和 continue 标志

```

file_input();

if (break_sign == true) {
    break_sign = false;
    break;
}
if (continue_sign == true) {
    continue_sign = false;
}

```

代码段跳转到 while 开头，重新判断

```

table->running_leveardown();
code_lexer->jump_location(while_start_loc);
code_lexer->get_next();
ret = or_test();
code_lexer->get_next();
code_lexer->get_next();
while (word_code == word::control::NEWLINE) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

table->running_levelup();
}
if (code_lexer->word_code != word::control::DEDENT) {
    code_lexer->skip_block();
}

```

退出循环，跳转代码段，循环层级-1

```

code_lexer->get_next();
table->running_leveardown();
loop_conting -= 1;
}

```

(29) for_stmt 函数

for 循环的基本结构与 while 类似，但是 for 循环会遍历列表，每次进入循环会执行如下操作：

1. 记录 for 循环变量到符号表
2. 代码层级加 1
3. 设置循环变量值，使其遍历 array 列表
4. 调用 file_input 函数执行代码块
5. 代码层级减 1
6. 判断 break 和 continue 标签
7. 如果遍历结束，则退出，否则跳转到步骤 2

下面仅对 array_int 类型的循环进行展示，对于 array_float 和 array_string 类型不做赘述

```

void phraser::for_stmt() {
    code_lexer->get_next();
    int word_code = code_lexer->word_code;
    std::string for_name;
    //find the name of the variable of the for loop.
    if (word_code == word::type::NAME) {
        for_name = code_lexer->word_name;
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    else {
        error::syntax_error(error::for_syntax_error, code_lexer->lineloc);
    }
}

```

代码结构判断

```

if (word_code == word::reserved::IN) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
else {
    error::syntax_error(error::for_syntax_error, code_lexer->lineloc);
}

```

待循环体的类型检测，与代码段位置记录

```

//for loop only support the array type of variable
phrase::return_value ret = or_test();
if (!(ret.type == running::name_type::ARRAY_INT ||
    ret.type == running::name_type::ARRAY_FLOAT ||
    ret.type == running::name_type::ARRAY_STRING)) {
    error::syntax_error(error::for_syntax_error, code_lexer->lineloc);
}
word_code = code_lexer->word_code;

//record the location of the beginning of the for loop.
int for_start_loc = code_lexer->get_location();
if (word_code == word::punctuation::COLON) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
else {
    error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
}

```

结构判断

```

while (word_code == word::control::NEWLINE) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
if (word_code == word::control::INDENT) {
    code_lexer->get_next();
}
else {
    error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
}
word_code = code_lexer->word_code;

```

代码层级加 1，设置循环变量值，使其遍历 array 列表，调用 file_input 函数执行代码块

```

loop_conting += 1;
//int type of for loop
if (ret.type == running::name_type::ARRAY_INT) {
    table->append_table(running::name_type::INT, for_name, new int(0));
    std::vector<int>* vec = (std::vector<int>*)ret.value;

    table->running_levelup();
    for (int i = 0; i < (int)vec->size(); i++) {
        int val = (*vec)[i];
        running::name_table* tab = table->get_items(for_name);
        tab->pointer = (void*)(new int(val));

        file_input();
    }
}

```

判断 break 和 continue 标签

```

if (break_sign == true) {
    break_sign = false;
    break;
}
if (continue_sign == true) {
    continue_sign = false;
}

```

代码层级减 1，并跳转代码

```

table->running_leveledown();
code_lexer->jump_location(for_start_loc);
code_lexer->get_next();
word_code = code_lexer->word_code;
while (word_code == word::control::NEWLINE) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

code_lexer->get_next();
table->running_levelup();
}

if (code_lexer->word_code != word::control::DEDENT) {
    code_lexer->skip_block();

}
code_lexer->get_next();
table->running_leveledown();
}

```

不对 array_float 和 array_string 类型进行赘述。最后，循环层级减 1

```

loop_conting -= 1;
}

```

(30) funcdef 函数

函数定义函数，用于检测函数变量列表和函数名、函数开始位置、函数对应的分词对象指针，并将其记录到符号表。

```

void phraser::funcdef() {
    code_lexer->get_next();
    int word_code = code_lexer->word_code;
    if (word_code != word::type::NAME) {
        error::syntax_error(error::function_define_error, code_lexer->lineloc);
    }
    std::string function_name = code_lexer->word_name;
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

```

基本结构判断

```

int para_num = 0;
std::vector<std::string> arg_list;
if (word_code != word::punctuation::PAREN_L) {
    error::syntax_error(error::function_define_error, code_lexer->lineloc);
}
else {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

```

开始记录函数参数列表，并将其保存到 arg_list 中

```

if (word_code != word::punctuation::PAREN_R) {
    if (word_code == word::type::NAME) {
        arg_list.push_back(code_lexer->word_name);
        para_num++;
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    else {
        error::syntax_error(error::function_define_error, code_lexer->lineloc);
    }
}

```

循环读取所有的变量名称，进行记录

```

while (word_code == word::punctuation::COMMA) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
    if (word_code == word::type::NAME) {
        arg_list.push_back(code_lexer->word_name);
        para_num++;
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    else {
        error::syntax_error(error::function_define_error, code_lexer->lineloc);
    }
}

```

右括号结构判断

```

if (word_code != word::punctuation::PAREN_R) {
    error::syntax_error(error::function_define_error, code_lexer->lineloc);
}
else {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}
else {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

```

冒号结构和换行符判断

```

if (word_code != word::punctuation::COLON) {
    error::syntax_error(error::function_define_error, code_lexer->lineloc);
}
else {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

while (word_code == word::control::NEWLINE) {
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

else {
    fun_start_loc = code_lexer->get_location();
    code_lexer->get_next();
    word_code = code_lexer->word_code;
}

```

函数信息记录到符号表

```

running::function_table* fun = new running::function_table;
fun->insert_function = false;

fun->parameter_num = para_num;
fun->name_list = new std::vector<std::string>();
for (int i = 0; i < (int)arg_list.size(); i++) {
    fun->name_list->push_back(arg_list[i]);
}
fun->lex_table = code_lexer;
fun->start_position = fun_start_loc;

table->append_table(running::name_type::FUNCTION, function_name, (void*)fun);
code_lexer->skip_block();
code_lexer->get_next();
}

```

(31) if_atom_format 函数

用于 if_stmt 子结构正确性检测

```

void phraser::if_atom_format() {
    int word_code = code_lexer->word_code;
    if (word_code == word::punctuation::COLON) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    else {
        error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
    }

    while (word_code == word::control::NEWLINE) {
        code_lexer->get_next();
        word_code = code_lexer->word_code;
    }
    if (word_code == word::control::INDENT) {
        code_lexer->get_next();
    }
    else {
        error::syntax_error(error::while_syntax_error, code_lexer->lineloc);
    }
}

```

(32) value_transmit 函数

用于将两个输入变量转化为统一类型。

```

bool phraser::value_transmit(phrase::return_value* name1, phrase::return_value* name2) {
    running::name_type type1 = name1->type;
    running::name_type type2 = name2->type;
    if (type1 == type2) {
        return true;
    }
    if (type1 == running::name_type::FLOAT && type2 == running::name_type::INT) {
        name2->type = running::name_type::FLOAT;
        int value = *(int*)(name2->value);
        name2->value = (void*)(new double(value));
        return true;
    }
    else if (type1 == running::name_type::INT && type2 == running::name_type::FLOAT) {
        name1->type = running::name_type::FLOAT;
        int value = *(int*)(name1->value);
        name1->value = (void*)(new double(value));
        return true;
    }
    return false;
}

```

(33) basic_operate 函数

基本运算操作，支持+、-、*、/ 四种操作，支持浮点型、整型、字符型、向量进行操作。

```

void* phraser::basic_operate(running::name_type type, void* val1, void* val2, word::punctuation operator_) {
    int val1_int = 0;
    int val2_int = 0;
    int val_int = 0;
    double val1_double = 0;
    double val2_double = 0;
    double val_double = 0;
    std::string val1_str;
    std::string val2_str;
    std::string val_str;

    running::name_type type1 = ((phrase::return_value*)(val1))->type;
    running::name_type type2 = ((phrase::return_value*)(val2))->type;
    val1 = ((phrase::return_value*)(val1))->value;
    val2 = ((phrase::return_value*)(val2))->value;
}

```

对于整型与浮点型，直接进行四则运算，这里仅以整型为例

```

switch (type) {
case running::name_type::INT:
    val1_int = *(int*)(val1);
    val2_int = *(int*)(val2);
    if (operator_ == word::punctuation::MULTI || operator_ == word::punctuation::MULTI_EQUAL) {
        val_int = val1_int * val2_int;
    }
    else if (operator_ == word::punctuation::DIV || operator_ == word::punctuation::DIV_EQUAL) {
        val_int = val1_int / val2_int;
    }
    else if (operator_ == word::punctuation::ADD || operator_ == word::punctuation::ADD_EQUAL) {
        val_int = val1_int + val2_int;
    }
    else if (operator_ == word::punctuation::MINUS || operator_ == word::punctuation::MINUS_EQUAL) {
        val_int = val1_int - val2_int;
    }
    else {
        error::syntax_error(error::operation_not_support, code_lexer->lineloc);
    }
    return (void*)(new int(val_int));
break;
}

```

字符串型只支持加法，用作字符串拼接操作

```

case running::name_type::STRING:
    val1_str = *(std::string*)(val1);
    val2_str = *(std::string*)(val2);
    if (operator_ == word::punctuation::ADD || operator_ == word::punctuation::MULTI_EQUAL) {
        val_str = val1_str + val2_str;
    }
    else {
        error::syntax_error(error::operation_not_support, code_lexer->lineloc);
    }
    return (void*)(new std::string(val_str));
break;

case running::name_type::ARRAY_STRING:
    error::syntax_error(error::operation_not_support, code_lexer->lineloc);
}

```

为了降低逻辑复杂性，只要是 array 类型加法，只支持 int 和 float，并且都转化为 float 进行运算。不支持广播机制。

```

if (type == running::name_type::ARRAY_INT || type == running::name_type::ARRAY_FLOAT)
    std::vector<int>* val_array1_int;
    std::vector<double>* val_array1_double;
    std::vector<int>* val_array2_int;
    std::vector<double>* val_array2_double;
    int length1 = 0;
    int length2 = 0;
    std::vector<double> templ;
    std::vector<double> temp2;
}

```

我将所有的值导入到统一的 double 结构，方便后续四则运算，前后变量做相同的转化，下面以 templ 变量为例

```

if (type1 == running::name_type::ARRAY_INT) {
    val_array1_int = (std::vector<int>*)(val1);
    length1 = val_array1_int->size();
    for (int i = 0; i < length1; i++) {
        temp1.push_back((double)((*val_array1_int)[i]));
    }
}
else if (type1 == running::name_type::ARRAY_FLOAT) {
    val_array1_double = (std::vector<double>*)(val1);
    length1 = val_array1_double->size();
    for (int i = 0; i < length1; i++) {
        temp1.push_back((double)((*val_array1_double)[i]));
    }
}
else {
    error::syntax_error(error::operation_not_support, code_lexer->lineloc);
}

```

判断两个变量是否等长

```

if (length1 != length2) {
    error::syntax_error(error::array_operate_length_error, code_lexer->lineloc);
}

```

array 类型四则运算

```

std::vector<double>* val_array = new std::vector<double>();
if (operator_ == word::punctuation::MULTI || operator_ == word::punctuation::MULTI_EQUAL) {
    for (int i = 0; i < length1; i++) {
        double val = temp1[i] * temp2[i];
        val_array->push_back(val);
    }
}
else if (operator_ == word::punctuation::DIV || operator_ == word::punctuation::DIV_EQUAL) {
    for (int i = 0; i < length1; i++) {
        double val = temp1[i] / temp2[i];
        val_array->push_back(val);
    }
}
else if (operator_ == word::punctuation::ADD || operator_ == word::punctuation::ADD_EQUAL) {
    for (int i = 0; i < length1; i++) {
        double val = temp1[i] + temp2[i];
        val_array->push_back(val);
    }
}
else if (operator_ == word::punctuation::MINUS || operator_ == word::punctuation::MINUS_EQUAL) {
    for (int i = 0; i < length1; i++) {
        double val = temp1[i] - temp2[i];
        val_array->push_back(val);
    }
}

```

返回运算结果

```

else {
    error::syntax_error(error::operation_not_support, code_lexer->lineloc
}
return val_array;
}

return NULL;
}

```

(34) switch_value 函数

转化类型为待转化类型

```

void phraser::switch_value(phrase::return_value* name, running::name_type type) {
    switch (name->type) {
        case running::name_type::INT:
            if (type == running::name_type::INT) {
                return;
            }
            else if (type == running::name_type::FLOAT) {
                name->type = running::name_type::FLOAT;
                int val = *(int*)name->value;
                name->value = (void*)(new double(val));
            }
            else {
                error::syntax_error(error::unsupport_type, code_lexer->lineloc);
            }
        case running::name_type::FLOAT:
            if (type == running::name_type::FLOAT) {
                return;
            }
            else if (type == running::name_type::INT) {
                name->type = running::name_type::INT;
                double val = *(int*)name->value;
                int temp = (int)val;
                name->value = (void*)(new int(temp));
            }
            else {
                error::syntax_error(error::unsupport_type, code_lexer->lineloc);
            }
    }
}

```

(35) debug 函数

打印输出当前位置 lexer 对应的字符，用于排错使用。在任意位置均可插入该函数对当前值进行输出。

```

void phraser::debug() {
    int word_code = code_lexer->word_code;
    if (word_code < 20) {
        printf("type: reserved word. name: %s\n", word::reserved_content[word_code]);
    }
    else if (word_code < 50) {
        printf("type: punctuation. value: %s\n", word::punctuation_content[word_code-20]);
    }
    else if (word_code == word::control::NEWLINE) {
        printf("type: control. value: new line\n");
    }
    else if (word_code == word::control::DEDENT) {
        printf("type: control. value: ded ent\n");
    }
    else if (word_code == word::control::INDENT) {
        printf("type: control. value: ind ent\n");
    }
    else if (word_code == word::control::ENDMARKER) {
        printf("type: control. value: end mark\n");
    }
    else if (word_code == word::type::NAME) {
        printf("type: name. value: %s\n", code_lexer->word_name.c_str());
    }
    else if (word_code == word::type::NUMBER) {
        printf("type: number. value: %d\n", code_lexer->word_int);
    }
    else if (word_code == word::type::FLOAT) {
        printf("type: float. value: %f\n", code_lexer->word_double);
    }
    else if (word_code == word::type::STRING) {
        printf("type: string. value: %s\n", code_lexer->word_string.c_str());
    }
}

```

3.9 主程序

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 py0 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。

| 名称 | 参数列表 | 返回值 | 含义 |
|---------------------|--|-------------------|--------------|
| split | <code>char* src,</code> <code>const char* separator,</code> <code>char** dest,</code> <code>int* num</code> | <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 argc,</code> <code>char** argv</code> | <code>void</code> | 主函数 |

(1) split 函数

命令行读入命令字符串分割，参数依次为：src 源字符串的首地址(buf 的地址)，separator 指定的分割字符，dest 接收子字符串的数组，num 分割后子字符串的个数。

```
void split(char* src, const char* separator, char** dest, int* num)
{
    char* pNext;
    int count = 0;

    if (src == NULL || strlen(src) == 0)
        return;

    if (separator == NULL || strlen(separator) == 0)
        return;
    char* buf;
    pNext = (char*)strtok_s(src, separator, &buf);
    while (pNext != NULL) {
        *dest++ = pNext;
        ++count;
        pNext = (char*)strtok_s(NULL, separator, &buf);
    }
    *num = count;
}
```

(2) welcome_show 函数

显示版本信息函数

```
void welcome_show() {
    printf("\nprogram language py0 [version: 1.0 beta]");
    printf("\nRelease day: 2020/6/7 ");
    printf("\nAuthor: CAU-CIEE Zhang Jingxiang - Ethan.");
    printf("\nTo score the source code please connect to 1967527237@qq.com\n\n");
}
```

(3) main 函数

采用命令行的方式打开，或采用拖动方式打开，传入一个 py0 文件物理位置的命令行参数，则程序执行完直接退出。否则进入命令行操作方式。

```

int main(int argc, char** argv)
{
    if (argc == 2) {
        try {
            running_table_class* table = new running_table_class();
            table->in_import(argv[1]);
            lexer* lex = new lexer(argv[1]);
            init_function initfun(table);
            phraser phr(lex, table);
            phr.run();
            printf("\n");
            printf("\n");
        }
        catch (const char* c) {
            printf("lethal error happend, type: %s.\n\n", c);
        }
        catch (...) {
            printf("lethal error happend, type: unkown.\n\n");
        }
        printf("\n\n输入任意键继续... ");
        getch();
        return 0;
    }

    printf("type '?' to see the command\n");
    welcome_show();
    while (true) {
        printf(">>");
        std::string name;

        std::getline(std::cin, name);
        char* buf = (char*)name.c_str();
        char* revbuf[8] = { 0 }; //存放分割后的子字符串
        int num = 0;
        split(buf, " ", revbuf, &num); //调用函数进行分割
        if (num == 0) {
            continue;
        }
        if (strcmp(revbuf[0], "quit") == 0) {
            if (num != 1) {
                printf("parameter error!\n\n");
            }
            else {
                break;
            }
        }
        else if (strcmp(revbuf[0], "?") == 0) {
            printf(" quit --quit the shell\n");
            printf(" run --run a py0 program\n");
            printf(" version --the version of the program\n");
            printf("\n\n");
        }
        else if (strcmp(revbuf[0], "version") == 0) {
            if (num != 1) {
                printf("parameter error!\n\n");
            }
            else {
                welcome_show();
            }
        }
        else if (strcmp(revbuf[0], "run") == 0) {
            if (num != 2) {
                printf("parameter error!\n\n");
            }
            else if (strcmp(revbuf[1], "?") == 0) {
                printf(" PATH --the path of the program, no space is permitted\n");
            }
            else {
                try {
                    running_table_class* table = new running_table_class();
                    table->in_import(revbuf[1]);
                    lexer* lex = new lexer(revbuf[1]);
                    init_function initfun(table);
                    phraser phr(lex, table);
                    phr.run();
                    printf("\n");
                    printf("\n");
                }
                catch (const char* c) {
                    printf("lethal error happend, type: %s.\n\n", c);
                }
                catch(...){
                    printf("lethal error happend, type: unkown.\n\n");
                }
            }
        }
    }
}

```

```
    }
}
else {
    printf("command error!\n\n");
}
```

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 pl0 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 pl0 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 |

```
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 |
| array | float | array_b | 0 |
| | | 10.000000 | 10.500000 |
| array | float | array_c | 0 |
| | | 100.000000 | 210.000000 |
| array | int | array_d | 0 |
| | | 10 | 900.000000 |

4.1.2 控制类语句

控制类语句包括 break、continue、return

(1) 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
```

```
now i value: 1
now i value: 2
now i value: 3
now j value: 3
```

(2) 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)
```

```
now i value: 1
now i + 1 value: 2
now i value: 2
now i value: 3
```

(3) 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))
```

```
add value is : 30
```

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 |

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：

```
test_a = 10
def test_fun(val1, val2):
    return val1 - val2

int test_a 0 10
function test_fun 0
    function type: user function      lexer file: C:\Users\lenovo\Desktop\test.txt
    start position: 15
    parameter nums: 2
int a 0 10
int b 0 20
int c 0 -10
```

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：

| | |
|---|--|
| <pre>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)</pre> | <pre>if is true, now i is: 10 elif is true, now i is: 20 else, now i is: 30 else, now i is: 40</pre> |
|---|--|

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：

| | | |
|---|--|----------------------|
| <pre>for i in (1, 2, 3, 4): print("for value: {} \n\n", i) show_table()</pre> | <pre>for value: 1 for value: 2 for value: 3 for value: 4</pre> | <pre>int i 0 4</pre> |
|---|--|----------------------|

(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
```

```
now a value is : 10
now a value is : 9
now a value is : 8
now a value is : 7
now a value is : 6
```

4.2.3 函数类语句

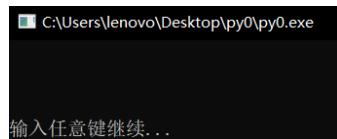
(1) 函数声明

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")
```



(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)
```

```
a : 6
b : 1236
c : 1236.650000
please input a value: 100
m+1=101.000000
```

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("{} --> {} \n".format(a,c))
    else:
        move(n-1, a, c, b)
        print("{} --> {} \n".format(b,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

```

(2) 输出九九乘法表

```

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

```

或

```

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