基于PLY的Python解析-2

实验内容

1. 利用PLY实现的Python程序的解析

本次学习的语法是选择语句和循环语句，需要注意的是本次使用的语法做了一些改进，不是纯粹的python2语法。

需要结合上次课四则运算的解析程序

（1）示例程序位于example3/

（2）需要进行解析的文件为**binary_search.py**和**select_sort.py**，分别对应二分查找和选择排序。

（3）需要完成以下内容的解析

```
--->   if
--->   while
--->   for
```

（4）解析结果以语法树的形式呈现

2. 编程实现语法制导翻译

（1）语法树上每个节点有一个属性value保存节点的值

（2）设置一个变量表保存每个变量的值

（3）基于深度优先遍历获取整个语法树的分析结果

在进行翻译条件语句和循环语句时，不能简单的进行深度优先遍历，要对于某些条件节点进行优先翻译

环境

- ***windows***
- ***python3.9***
- ***ply包***

运行

```
python main.py
```

实验步骤

1. 编写py_lex.py进行序列标记

通过观察**binary_search.py**和**select_sort.py**两份py代码，我们可以得出需要解析的tokens有以下这些：

```
tokens = ('DIV', 'LE', 'GE', 'INC', 'LEN', 'VARIABLE', 'NUMBER', 'IF', 'ELIF',
'ELSE', 'WHILE', 'FOR', 'PRINT', 'BREAK')
literals = ['(', ')', '{', '}', '<', '>', '[', ']', ',', ';', '=', '+', '-',
'*']
```

随后，对tokens中的每个标记定义

定义匹配规则时，需要以**t_**为前缀，紧跟在t_后面的单词，必须跟标记列表中的某个标记**名称**对应，同时使用**正则表达式**来进行匹配

同时定义句子之间的**间隔**和**错误处理**。

ply**使用"t_"开头**的变量来表示规则。

- 如果变量是一个**字符串**，那么它被解释为一个正则表达式，匹配值是标记的值。
- 如果变量是**函数**，则其文档字符串包含模式，并使用匹配的标记调用该函数。

该函数可以自由地修改序列或返回一个新的序列来代替它的位置。 如果没有返回任何内容，则忽略匹配。

通常该函数只更改“value”属性，它最初是匹配的文本。

代码：

```
# Define of tokens
def t_PRINT(t):
    r'print'
    return t
def t_NUMBER(t):
    r'[0-9]+'
    return t
def t_IF(t):
    r'if'
    return t
def t_ELIF(t):
    r'elif'
    return t
def t_ELSE(t):
    r'else'
    return t
def t_DIV(t):
    r'//'
    return t
def t_LE(t):
    r'<='
    return t
def t_GE(t):
    r'>='
    return t
def t_WHILE(t):
    r'while'
    return t
def t_FOR(t):
    r'for'
    return t
def t_BREAK(t):
    r'break'
    return t
def t_INC(t):
    r'\+\+'
    return t
def t_LEN(t):
    r'len'
    return t
```

```
def t_VARIABLE(t):
    r'[a-zA-Z\$_][a-zA-Z\d_]*'
    return t
# Ignored
t_ignore = " \t"
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

2. 编写py_pacc.py进行语法分析

观察需要解析的两个py文件，可以得到如下文法：

```
program->statements
statement->sstatements statement
          | statement
statement -> assignment
          | operation
          | print
          | if_elif_else
          | while
          | for
          | BREAK
assignment -> VARIABLE = NUMBER
           | VARIABLE = VARIABLE
           | VARIABLE = num_list
           | VARIABLE = VARIABLE [ expr ]
           | VARIABLE [ expr ] = NUMBER
numbers -> NUMBER
        | numbers , NUMBER
num_list -> [ numbers ]
operation -> VARIABLE = expr
          | VARIABLE [ expr ] = expr
expr -> term
     | expr + term
     | expr - term
     | expr DIV expr
     | LEN ( factor )
term -> factor
     | term * factor
     | term / factor
factor -> NUMBER
       | VARIABLE
       | ( expr )
       | VARIABLE [ expr ]
print -> PRINT ( VARIABLE )
if_elif_else -> if
             | if elif else
if -> IF ( condition ) { statements }
elif -> ELIF ( condition ) { statements }
else -> ELSE { statements }
condition -> factor > factor
          | factor '<' factor
          | factor LE factor
          | factor GE factor
for -> FOR ( conditions ) { statements }
while -> WHILE ( condition ) { statements }
```

```
conditions -> assignment ; condition ; increment
increment -> VARIABLE INC
```

又因为本次实验在进行翻译条件语句和循环语句时，不能简单的进行深度优先遍历，要对于某些条件节点进行优先翻译。

为了简化步骤，这里将所有类型的括号省略掉，不再对括号进行解析，也不需要将括号翻译到语法树中。

py_yacc.py代码：

```
#! /usr/bin/env python
# coding=utf-8
import ply.yacc as yacc
from py_lex import *
from node import node, num_node

# YACC for parsing Python

def p_program(t):
    '''program : statements'''
    if len(t) == 2:
        t[0] = node('[PROGRAM]')
        t[0].add(t[1])

def p_statements(t):
    '''statements : statements statement
                  | statement'''
    if len(t) == 3:
        t[0] = node('[STATEMENTS]')
        t[0].add(t[1])
        t[0].add(t[2])
    elif len(t) == 2:
        t[0] = node('[STATEMENTS]')
        t[0].add(t[1])

def p_statement(t):
    ''' statement : assignment
                  | operation
                  | print
                  | if_elif_else
                  | while
                  | for
                  | BREAK'''
    if len(t) == 2:
        if not isinstance(t[1], str):
            t[0] = node('[STATEMENT]')
            t[0].add(t[1])
        else:
            t[0] = node('[STATEMENT]')
            t[0].add(node('[BREAK]'))

def p_assignment(t):
    '''assignment : VARIABLE '=' NUMBER
                  | VARIABLE '=' VARIABLE
                  | VARIABLE '=' num_list
                  | VARIABLE '=' VARIABLE '[' expr ']'
```

```
                | VARIABLE '[' expr ']' '=' NUMBER '''
    if len(t) == 4:
        if not isinstance(t[3], str):
            '''assignment : VARIABLE '=' num_list'''
            t[0] = node('[ASSIGNMENT]')
            t[0].add(node(t[1]))
            t[0].add(node(t[2]))
            t[0].add(t[3])
        elif t[3][0].isdigit():
            '''assignment : VARIABLE '=' NUMBER'''
            t[0] = node('[ASSIGNMENT]')
            t[0].add(node(t[1]))
            t[0].add(node(t[2]))
            t[0].add(num_node(t[3]))
        else:
            '''assignment : VARIABLE '=' VARIABLE'''
            t[0] = node('[ASSIGNMENT]')
            t[0].add(node(t[1]))
            t[0].add(node(t[2]))
            t[0].add(node(t[3]))
    elif len(t) == 7:
        if t[2] == '=':
            '''assignment : VARIABLE '=' VARIABLE '[' expr ']' '''
            t[0] = node('[ASSIGNMENT]')
            t[0].add(node(t[1]))
            t[0].add(node(t[2]))
            t[0].add(node(t[3]))
            t[0].add(t[5])
        elif t[2] == '[':
            '''assignment : VARIABLE '[' expr ']' '=' NUMBER'''
            t[0] = node('[ASSIGNMENT]')
            t[0].add(node(t[1]))
            t[0].add(t[3])
            t[0].add(node(t[5]))
            t[0].add(num_node(t[6]))

def p_numbers(t):
    '''numbers : NUMBER
               | numbers ',' NUMBER'''
    if len(t) == 2:
        t[0] = node('[NUMBERS]')
        t[0].add(num_node(t[1]))
    elif len(t) == 4:
        t[0] = node('[NUMBERS]')
        t[0].add(t[1])
        t[0].add(num_node(t[3]))

def p_num_list(t):
    '''num_list : '[' numbers ']' '''
    if len(t) == 4:
        t[0] = node('[NUM_LIST]')
        t[0].add(t[2])

def p_operation(t):
    '''operation : VARIABLE '=' expr
                 | VARIABLE '[' expr ']' '=' expr '''
    if len(t) == 7:
        t[0] = node('[OPERATION]')
```

```
        t[0].add(node(t[1]))
        t[0].add(t[3])
        t[0].add(node(t[5]))
        t[0].add(t[6])
    elif len(t) == 4:
        t[0] = node('[OPERATION]')
        t[0].add(node(t[1]))
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_expr(t):
    '''expr : term
            | expr '+' term
            | expr '-' term
            | expr DIV expr
            | LEN '(' factor ')' '''
    if len(t) == 2:
        t[0] = node('[EXPRESSION]')
        t[0].add(t[1])
    elif len(t) == 4:
        t[0] = node('[EXPRESSION]')
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(t[3])
    elif len(t) == 5:
        t[0] = node('[EXPRESSION]')
        t[0].add(node('[LEN]'))
        t[0].add(t[3])

def p_term(t):
    '''term : factor
            | term '*' factor
            | term '/' factor '''
    if len(t) == 2:
        t[0] = node('[TERM]')
        t[0].add(t[1])
    elif len(t) == 4:
        t[0] = node('[TERM]')
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_factor(t):
    '''factor : NUMBER
              | VARIABLE
              | '(' expr ')'
              | VARIABLE '[' expr ']' '''
    if len(t) == 2:
        if t[1].isdigit():
            '''factor : NUMBER'''
            t[0] = node('[FACTOR]')
            t[0].add(num_node(t[1]))
        else:
            '''factor : VARIABLE'''
            t[0] = node('[FACTOR]')
            t[0].add(node(t[1]))
    elif len(t) == 4:
        '''factor : '(' expr ')' '''
```

```
        t[0] = node('[FACTOR]')
        t[0].add(t[2])
    elif len(t) == 5:
        '''factor : VARIABLE '[' expr ']' '''
        t[0] = node('[FACTOR]')
        t[0].add(node(t[1]))
        t[0].add(t[3])

def p_print(t):
    '''print : PRINT '(' VARIABLE ')' '''
    if len(t) == 5:
        t[0] = node('[PRINT]')
        t[0].add(node(t[3]))

def p_if_elif_else(t):
    '''if_elif_else : if
                    | if elif else'''
    if len(t) == 2:
        t[0] = node('[IF_ELIF_ELSE]')
        t[0].add(t[1])
    elif len(t) == 4:
        t[0] = node('[IF_ELIF_ELSE]')
        t[0].add(t[1])
        t[0].add(t[2])
        t[0].add(t[3])

def p_if(t):
    '''if : IF '(' condition ')' '{' statements '}' '''
    if len(t) == 8:
        t[0] = node('[IF]')
        t[0].add(t[3])
        t[0].add(t[6])

def p_elif(t):
    '''elif : ELIF '(' condition ')' '{' statements '}' '''
    if len(t) == 8:
        t[0] = node('[ELIF]')
        t[0].add(t[3])
        t[0].add(t[6])

def p_else(t):
    '''else : ELSE '{' statements '}' '''
    if len(t) == 5:
        t[0] = node('[ELSE]')
        t[0].add(t[3])

def p_condition(t):
    '''condition : factor '>' factor
                 | factor '<' factor
                 | factor LE factor
                 | factor GE factor '''
    if len(t) == 4:
        t[0] = node('[CONDITION]')
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_for(t):
```

```
    '''for : FOR '(' conditions ')' '{' statements '}' '''
    if len(t) == 8:
        t[0] = node('[FOR]')
        t[0].add(t[3])
        t[0].add(t[6])

def p_while(t):
    '''while : WHILE '(' condition ')' '{' statements '}' '''
    if len(t) == 8:
        t[0] = node('[WHILE]')
        t[0].add(t[3])
        t[0].add(t[6])

def p_conditions(t):
    '''conditions : assignment ';' condition ';' increment'''
    if len(t) == 6:
        t[0] = node('[CONDITIONS]')
        t[0].add(t[1])
        t[0].add(t[3])
        t[0].add(t[5])

def p_increment(t):
    '''increment : VARIABLE INC'''
    if len(t) == 3:
        t[0] = node('[INCREMENT]')
        t[0].add(node(t[1]))
        t[0].add(node(t[2]))

def p_error(t):
    print("Syntax error at '%s'" % t.value)

yacc.yacc()
```

3. 完善translation.py进行语法制导翻译

由于本次实验文法比较复杂，所以在进行翻译条件语句和循环语句时，不能简单的进行深度优先遍历，要对于某些条件节点进行优先翻译

根据2.中的文法对程序进行完善。

translation.py代码：

```
#! /usr/bin/env python
# coding=utf-8
from __future__ import division
v_table = {}  # variable table
def update_v_table(name, value):
    v_table[name] = value

def trans(node):
    # Translation
    # Assignment
    if node.getdata() == '[ASSIGNMENT]':
        '''assignment : VARIABLE '=' NUMBER
                      | VARIABLE '=' VARIABLE
                      | VARIABLE '=' num_list
                      | VARIABLE '=' VARIABLE '[' expr ']'
```

```
                    | VARIABLE '[' expr ']' '=' NUMBER'''
        if len(node.getchildren()) == 3:
            if node.getchild(2).getdata()[0].isdigit():
                value = node.getchild(2).getvalue()
                update_v_table(node.getchild(0).getdata(), value)
                node.getchild(0).setvalue(value)
            elif node.getchild(2).getdata() == '[NUM_LIST]':
                trans(node.getchild(2))
                value = node.getchild(2).getvalue()
                update_v_table(node.getchild(0).getdata(), value)
            else:
                value = v_table[node.getchild(2).getdata()]
                update_v_table(node.getchild(0).getdata(), value)
                node.getchild(0).setvalue(value)
        elif len(node.getchildren()) == 4:
            if node.getchild(2).getdata() == '=':
                trans(node.getchild(1))
                lst=node.getchild(0).getdata()
                index = int(node.getchild(1).getvalue())
                value = node.getchild(3).getvalue()
                v_table[lst][index] = value

            elif node.getchild(1).getdata() == '=':
                trans(node.getchild(3))

                lst=node.getchild(2).getdata()
                index = int(node.getchild(3).getvalue())
                value = v_table[lst][index]
                update_v_table(node.getchild(0).getdata(), value)

    # Operation
    elif node.getdata() == '[OPERATION]':
        '''operation : VARIABLE '=' expr
                    | VARIABLE '[' expr ']' '=' expr'''
        if len(node.getchildren()) == 3:
            trans(node.getchild(2))
            value = node.getchild(2).getvalue()
            node.getchild(0).setvalue(value)
            update_v_table(node.getchild(0).getdata(), value)
        elif len(node.getchildren()) == 4:
            trans(node.getchild(1))
            trans(node.getchild(3))
            lst = v_table[node.getchild(0).getdata()]
            index = int(node.getchild(1).getvalue())
            value = node.getchild(3).getvalue()
            lst[index] = value

    # Num_list
    elif node.getdata() == '[NUM_LIST]':
        '''num_list : '[' numbers ']' '''
        if len(node.getchildren()) == 1:
            trans(node.getchild(0))
            value=[]
            for i in node.getchild(0).getvalue().split():
                value.append(float(i))
            node.setvalue(value)

    # Numbers
```

```
    elif node.getdata() == '[NUMBERS]':
        '''numbers : NUMBER
                   | numbers ',' NUMBER'''
        if len(node.getchildren()) == 1:
            value = str(node.getchild(0).getvalue())
            node.setvalue(value)
        elif len(node.getchildren()) == 2:
            trans(node.getchild(0))

            value0 = node.getchild(0).getvalue()
            value1 = str(node.getchild(1).getvalue())
            node.setvalue(value0 + ' ' + value1)

    # Print
    elif node.getdata() == '[PRINT]':
        '''print : PRINT '(' VARIABLE ')' '''
        arg0 = v_table[node.getchild(0).getdata()]
        print(arg0)

    # Expr
    elif node.getdata() == '[EXPRESSION]':
        '''expr : term
                | LEN '(' factor ')'
                | expr '+' term
                | expr '-' term
                | expr DIV factor '''
        if len(node.getchildren()) == 1:
            trans(node.getchild(0))
            value = node.getchild(0).getvalue()
            node.setvalue(value)

        elif len(node.getchildren()) == 2:
            trans(node.getchild(1))
            value = len(node.getchild(1).getvalue())
            node.setvalue(value)

        elif len(node.getchildren()) == 3:
            trans(node.getchild(0))
            trans(node.getchild(2))
            arg0 = node.getchild(0).getvalue()
            arg1 = node.getchild(2).getvalue()
            op = node.getchild(1).getdata()
            if op == '+':
                value = arg0 + arg1
            elif op == '-':
                value = arg0 - arg1
            elif op == '//':
                value = arg0 // arg1
            node.setvalue(value)

    # Term
    elif node.getdata() == '[TERM]':
        '''term : factor
                | term '*' factor
                | term '/' factor '''
        if len(node.getchildren()) == 1:
            trans(node.getchild(0))
            value = node.getchild(0).getvalue()
```

```
            node.setvalue(value)

        elif len(node.getchildren()) == 3:
            trans(node.getchild(0))
            trans(node.getchild(2))
            arg0 = node.getchild(0).getvalue()
            arg1 = node.getchild(2).getvalue()
            op = node.getchild(1).getdata()
            if op == '*':
                value = arg0 * arg1
            elif op == '/':
                value = arg0 / arg1
            node.setvalue(value)

    # Factor
    elif node.getdata() == '[FACTOR]':
        '''factor : NUMBER
                  | VARIABLE
                  | '(' expr ')'
                  | VARIABLE '[' expr ']' '''
        if len(node.getchildren()) == 1:
            if node.getchild(0).getdata() == '[EXPRESSION]':
                trans(node.getchild(0))
                value = node.getchild(0).getvalue()
                node.setvalue(value)
            elif node.getchild(0).getdata()[0].isdigit():
                value = node.getchild(0).getvalue()
                node.setvalue(value)
            else:
                value = v_table[node.getchild(0).getdata()]
                node.setvalue(value)

        elif len(node.getchildren()) == 2:
            lst = v_table[node.getchild(0).getdata()]
            trans(node.getchild(1))
            index = int(node.getchild(1).getvalue())
            value = lst[index]
            node.setvalue(value)

    # Increment
    elif node.getdata() == '[INCREMENT]':
        update_v_table(node.getchild(0).getdata(),
v_table[node.getchild(0).getdata()] + 1)

    # Break
    elif node.getdata() == '[BREAK]':
        return '[BREAK]'

    # If_elif_else
    elif node.getdata() == '[IF_ELIF_ELSE]':
        '''if_elif_else : if
                        | if elif else'''
        if len(node.getchildren()) == 1:
            if trans(node.getchild(0)) == '[BREAK]':
                return '[BREAK]'
        elif len(node.getchildren()) == 3:
            trans(node.getchild(0).getchild(0))
            condition = node.getchild(0).getchild(0).getvalue()
```

```
            if condition:
                if trans(node.getchild(0)) == '[BREAK]':
                    return '[BREAK]'
            else:
                trans(node.getchild(1).getchild(0))
                condition = node.getchild(1).getchild(0).getvalue()
                if condition:
                    if trans(node.getchild(1)) == '[BREAK]':
                        return '[BREAK]'
                else:
                    if trans(node.getchild(2)) == '[BREAK]':
                        return '[BREAK]'
    # If
    elif node.getdata() == '[IF]':
        r'''if : IF '(' condition ')' '{' statements '}' '''
        children = node.getchildren()
        trans(children[0])
        condition = children[0].getvalue()
        if condition:
            for c in children[1:]:
                trans(c)

    # Elif
    elif node.getdata() == '[ELIF]':
        '''elif : ELIF '(' condition ')' '{' statements '}' '''
        children = node.getchildren()
        trans(children[0])
        condition = children[0].getvalue()
        if condition:
            for c in children[1:]:
                trans(c)

    # Else
    elif node.getdata() == '[ELSE]':
        if trans(node.getchild(0)) == '[BREAK]':
            return '[BREAK]'

    # While
    elif node.getdata() == '[WHILE]':
        r'''while : WHILE '(' condition ')' '{' statements '}' '''
        children = node.getchildren()
        flag = False
        while trans(children[0]):
            for c in children[1:]:
                if trans(c) == '[BREAK]':
                    flag = True
                    break
            if flag:
                break
    # For
    elif node.getdata() == '[FOR]':
        '''for : FOR '(' conditions ')' '{' statements '}' '''
        children = node.getchildren()
        conditions = children[0]
        trans(conditions.getchild(0))
        flag = False
        while trans(conditions.getchild(1)):
            for c in children[1:]:
```

```
                if trans(c) == '[BREAK]':
                    flag = True
                    break
            if flag:
                break
            trans(conditions.getchild(2))

    # Condition
    elif node.getdata() == '[CONDITION]':
        '''condition : factor '>' factor
                     | factor '<' factor
                     | factor LE factor
                     | factor GE factor '''
        trans(node.getchild(0))
        trans(node.getchild(2))
        num1 = node.getchild(0).getvalue()
        num2 = node.getchild(2).getvalue()
        op = node.getchild(1).getdata()
        if op == '>':
            node.setvalue(num1 > num2)
        elif op == '<':
            node.setvalue(num1 < num2)
        elif op == '<=':
            node.setvalue(num1 <= num2)
        elif op == '>=':
            node.setvalue(num1 >= num2)
    else:
        for c in node.getchildren():
            if trans(c) == '[BREAK]':
                return '[BREAK]'
    return node.getvalue()
```

4. 调用编写好的语法制导翻译系统解析目标程序

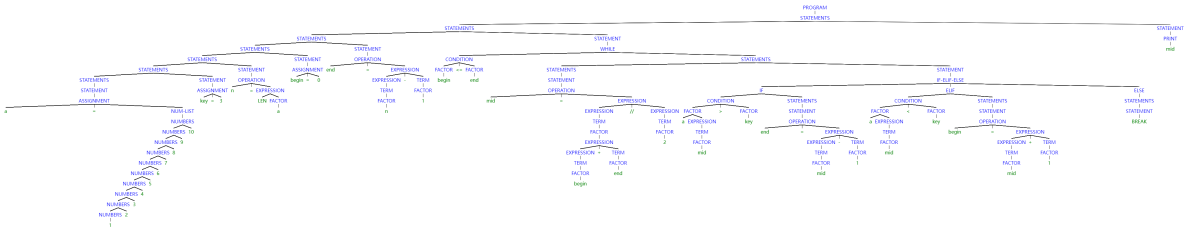
由于本次实验需要输出指定格式的语法树，所以编写一个get_tree(node)函数来递归获取语法树字符串

```
#! /usr/bin/env python
#coding=utf-8
from py_yacc import yacc
from util import clear_text
from translation import trans,v_table

file_name="select_sort.py"

def get_tree(node):
    global ans
    if node:
        ans+=str(node.getdata()).replace("[", "").replace("]", "").replace("/'",
"").replace("_","-")
    children=node.getchildren()
    if children:
        for child in children:
            ans+='['
            get_tree(child)
            ans+=']'

text=clear_text(open(file_name,'r').read())
```

```
# syntax parse
root=yacc.parse(text)
root.print_node(0)
# get tree string
ans=""
get_tree(root)
print("["+ans+"]")
# translation
trans(root)
print(v_table)
```

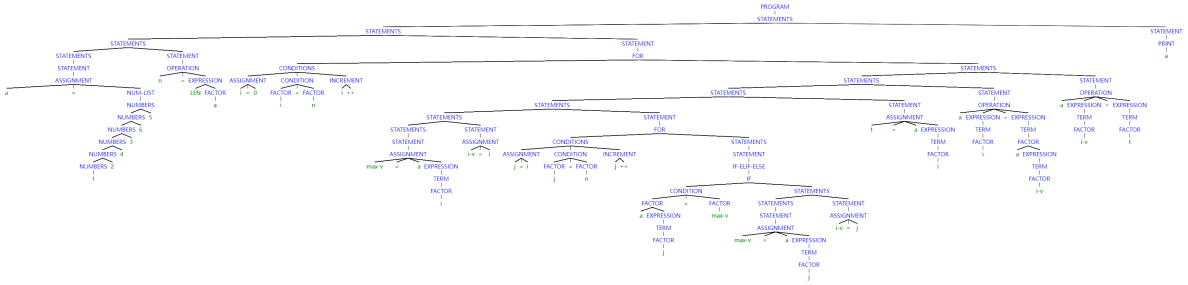
运行结果

(1) 解析binary_search.py



```
2.0
{'a': [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0], 'key': 3.0, 'n': 10, 'begin': 2.0, 'end': 3.0, 'mid': 2.0}
```

(2) 解析select_sort.py



```
[6.0, 5.0, 4.0, 3.0, 2.0, 1.0]
{'a': [6.0, 5.0, 4.0, 3.0, 2.0, 1.0], 'n': 6, 'i': 6.0, 'max_v': 1.0, 'i_v': 5.0, 'j': 6.0, 't': 1.0}
```