语言设计规格说明书

1. 语言设计的背景及范型

　　Python 是解释执行的动态语言，具有丰富的语法糖和类库。缺点是执行效率低；并且动态语言由于没有构建的过程，因此很多错误只有等到运行时才会发现，代码检查工具的效率不高。

　　因此，我们想设计一种类Pascal的静态类型的编译型语言，既拥有高效的执行效率，又具有丰富的语言特性。

　　它的优点有：

1、严格的结构化形式，简明灵活的控制结构。
2、丰富完备的数据类型
3、运行效率高
4、查错能力强， 语言简单易学

2. 语言的语法、语义规格说明

2.1 词法EBNF

2.1.1 词法定义：

本语言的词法包括以下几大部分的类型定义：

2.1.2 关键字：

本语言有20个关键字，在此将其定义为keyWord类型，EBNF表达如下：
keyWord ::= begin | proc | while | var | func | is | do | array | in | record | if | let | then | of | type | end | else | const | try | catch

2.1.3 运算符：

本语言有36种运算符，同样地，将其定义为Calculation类型，EBNF表达如下：
Calculation ::= + | – | * | / | , | ; | > | >= | < | <= | = | == | != | (|) | { | } | [|] | ^ | && | || | ! | /* | */ | : | % | // | ++ | –– | >> | << | += | –= | *= | /+

2.1.4 数值类型：

本语言有7种数值类型，在此将其定义为Value类，EBNF表达如下：
valueType ::= Integer | Char | Real | Boolean
另外
digit ::= ”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”

Integer ::= digit，{digit}
Real ::= int，”.”，int
Char ::= ”a” | ”b” | ”c” | ”d” | ”e” | ”f” | ”g” | ”h” | ”i” | ”j” | ”k” | ”l” | ”m” | ”n” | ”o” | ”p” | ”q” | ”r” | ”s” | ”t” | ”u” | ”v” | ”w” | ”x” | ”y” | ”z”
Boolean ::= ”0” | ”1”

2.2 语法EBNF

2.2.1 命令

我们将Identifier，Integer_Literal，Character_Literal，Operator为原子操作，不再进一步定义。

```
Command：：=
                    | V_name：= Expression
                      | Identifier(Actual_Parameter_Sequence)
                    | Command；Command
                      | begin Command end
                    | let Declaration in Command
                    | if Expression then Command else Command
                    | while Expression do Command
                    | try Command catch( Identifier : Type_denoter) Command
```

指称语义：

```
execute：Commamd→(Environ→Store→Store)
execute〚 〛env sto =
          sto
execute〚V：= E〛env sto =
     let variable val = identify E env sto in
     let val = evaluate E env sto in
     update_variable(sto，var，val)
execute〚I(APS)〛env sto =
         let procedure proc = find(env,I) in
         let args = give_arguments APS env sto in
         proc args sto
execute〚C1; C2〛env sto =
         execute C2 env(execute C1 env sto)
execute〚begin C end〛=
         execute C
execute〚let D in C〛 env sto =
         let (env’，sto’) = elaborate D env sto in
         execute C(overlay(env’，env)) sto’
execute〚 if E then C1 else C2〛env sto =
         if evaluate E env sto = truth_value true
         then execute C1 env sto
         else execute C2 env sto
execute〚while E do C〛=
         let execute_while env sto =
            if evaluate E env sto = truth_value true
```

```
                then execute_while env (execute C env sto)
                else sto
            in
            execute_while
execute 〚try  C catch (I:T ) C〛 =
            let execute_catch  env sto =
                evaluate C env sto
                if evaluate I env sto = Type_denoter
                then execute C env sto
            in
            execute_catch
```

2.2.2 表达式

```
Expression ： ： =  Integer_Literal
                       |  Character_Literal
                          |  V_name
                          |  Identifier(Actual_Parameter_Sequence)
                          |  Operatpr Expression
                          |  Expression Operatpr Expression
                          |  (Expression)
                       |  let Declaration in Command
                          |  if Expression then Command else Command
                          |  {Record_Aggegate}
                          |  [Array_Aggregate]
                       | * Identifier
                       | & Identifier
Record_Aggegate ： ： =
                       |  Identifier = Expression
                        |  Identifier = Expression， Array_Aggregate
Array_Aggregate ： ： =  Expression
                        |  Expression， Array_Aggregate
```

指称语义：

```
evaluate : Expression → (Environ → Store → Value)
evaluate_record : Record_Aggregate→(Environ → Store → Record_value)
valuate_array : Array_Aggregate→(Environ→Store→Array_Value)
//evaluate E env sto 给出在环境env和存储sto之下执行E得到的值
//evaluaterecord RA env sto给出在环境env和存储sto之下对记录聚集RA求值得到的记录
//evaluate array AA env sto给出在环境env和存储sto之下对数组聚集AA求值得到的数组
evaluate [IL] env sto =
            integer(integer_valuation IL)
evaluate [CL] env sto =
            character(character_valuation CL)
evaluate[V] env sto =
            coerce(sto, identifyy env sto)
evaluate [I (APS)] env sto =
            let function func= find(env,I) in
            let args =give_arguments APS env sto in
```

```
evaluate[O E] env sto =
           let unary_operator unop=find (env, id O)in
           let vall=evaluate El env sto in
           unop val
evaluate[E1 O E2] env sto =
            let binary_operator binp=find(env, id O)in
           let val1=evaluate E1 eny sto in
           let val2=evaluate E2 eny sto in
           binp( vall, val2)
evaluate[(E)]=
          evaluate E
evaluate [let D in E] env sto=
          let(env', sto') = elaborate env sto in
          evaluate E(overlay(env', env)) sto'
evaluate [if E1 then E2 else E3] env sto=
            if evaluate E1 env sto= truthvalue_true
            then evaluate E2 env sto
            else evaluate E3 env sto
evaluate [{RA}] env sto=
          record_value(evaluate_record RA env sto)
evaluate [[AA]] env sto=
          array_value( evaluate_array AA env sto)
evaluate_record[I~E] env sto=
          let val = evaluate E env sto in
          unit_record val (1, val)
evaluate_record[I~E, RA] env sto=
          let val=evaluate E env sto in
          let ecval=evaluate_record RA eny sto in
          joined_record_val(I, val, reeval)
evaluate_array[ E] env sto=
          let val evaluate env sto in
          let arrval= evaluate_array AA env sto in
          abutted_array_val(val, arrval)
```

2.2.3 名字

```
V_name ::= Identifier
               | V_name, Identifier
               | V_name[Expression]
```

指称语义：

```
identify:V_name→(Environ→Store→Value_or_Variable)
//iddenify V eny sto 给出在环境env存储sto之下由V命名的值或变量
identify[1]env sto=
          find(env,  D)
identify[V,1]env sto=
          let field(id,value(record_value recval)=
                         value(field–val(id,recval))
```

```
                field(id,variable (record-variable recvar))=
                    variable(field-var(id,recvar))
        in
        field(I,identify V env sto)
identify [V[E]] env sto=
        let component(integer int,value (array-value arrval))=
                    value(component-val(int,arrval))
            commponent(integer int,variable(array-variable arrvar))=
                    variable(component-var(int,arrvar))
         in
         component(evaluate E env sto,identify V env sto)
```

2.2.4 声明

```
Declaration ::= const Identifier = Expression
                | var Identifier : Type_denoter
                | proc Identifier(Formal_Parameter_Sequence)is Command
                | func Identifier (Formal_Parameter_Sequence) : Type_denoter is
            Expression
                | type Identifier is Type_denoter
                | Declaration; Declaration
```

指称语义：

```
claborate:Declaration→(Environ→Store-→Environ XStore)
 //elaborate D env sto 给出在环境env存储sto之下确立声明D得到的束定和改变的存储
elaborate [const I~E] env sto=
            let val=evaluate E env sto in
            (bind(I,valuc val),sto)
elaborate [var I:T] env sto=
            let(sto',var)=allocate-variable T env sto in
            (bind(I,variable var),sto')
elaborate [proc I(FPS)~C]env sto=
        let proc args sto=
            let env=overlay(bind (I,procedure proc),env)in
            let parenv=bind-parameters FPS args in
            execute C(overlay(parenv,env))sto'
        in
        (bind(1,procedure proc),sto)
elaborate [func I(FPS):T~E]env sto=
        let func args sto=
              let env=overlay(bind(I,function func),env)in
          let parenv=bind-parameters FPS args in
          evaluale E(overlay (parenv,env))sto'
        in
        (bind(1,function func),sto)
elaborate [type I~T] env sto=
        let alloc sto=allocate-variable T env sto in
          (bind(I,allocator alloc),sto)
elaborate [D1； D2]env sto=
let(env',sto')=elaborate DI env sto in
let(env", sto")=elaborate D2(overlay (env',env))sto' in
(overlay(env",env'),sto")
```

2.2.5 参数

```
Formal_Parameter_Sequence ::=
                    | Formal_Parameter
                    | Formal_Parameter, Formal_Parameter_Sequence

Formal_Parameter ::= Identifier : Type_denoter
                    | var Identifier : Type_denoter
                    | proc Identifier(Formal_Parameter_Sequence)
                    | func Identifier(Formal_Parameter_Sequence):Type_denoter

Actual_Parameter_Sequence ::=
                    | Actual_Parameter
                    | Actual_Parameter, Actual_Parameter_Sequence

Actual_Parameter ::= Expression
                    | var V_name
                    | proc Identifier
                    | func Identifier
```

指称语义：

定义四个指称函数:

- bind_parameters : Formal_Parameter_Sequence → (Argument * → Environ)
- bind_parameter : Formal_Parameter → (Argument → Environ)
- give_arguments : Actual_Parameter_Sequence → (Environ → Store → Argument *)
- give_argument : Actual_Parameter → (Environ → Store → Argument)

//hind_parameters FPS args 给出形参序列FPS和变元表args，结合后的束定
//bind_parameter FParg给出形式参数FP和变元arg，结合后的束定
//give_arguments APS env sto 给出在环境env存储sto之下实参序列APS产生的变元表
//give–argument AP env sto给出在环境env存储sto之下实参数AP产生的变元

1.将形参数列与变元表arg结合后的束定。

```
   bind_parameters 〖FP, FPS〗(arg • args) =
      overlay(bind_parameters FPS args, bind_parameter FP arg)
```

2.将形参与变元arg结合后的束定。

```
   bind_parameter 〖I : T〗(value val) =
      bind(I, value val)
  bind_parameter 〖var I : t〗(variable var) =
     bind(I, value val)
  bind_parameter 〖proc I(FPS)〗(procedure proc) =
     bind(I, procedure proc)
  bind_parameter 〖func I(FPS): T〗(function func) =
     bind(I, function func)
```

3.给出在环境env存储sto之下实参序列APS产生的变元表。

```
   give_arguments 〖AP, APS〗 env sto =
        (give_argument AP env sto) •(give_arguments APS env sto)
```
4.给出在环境env存储sto之下实参AP产生的变元。
```
    give_argument 〖var V〗 env sto =
        let variable var=identify V env sto in
        variable var
    give_argument 〖proc I〗 env sto =
        let procedure proc=find(env, I) in
        procedure proc
    give_argument 〖func I〗 env sto =
        let function func=find(env, I) in
        function fun
```

2.2.6 类型指示符

```
Type_denoter ::=  Identifier
                | array Integer_Literal of Type_denoter
                | record Record_Type_denoter end
                | Char
                | Boolean
                | Integer
                | Real
                | * Type_denoter

Record_Type_denoter ::= Identifier : Type_denoter
                | Identifier : Type_denoter, Record_Type_denoter
```

指称语义：

Allocate_variable:Type-denoter →(Environ →Store→StoreX variable Allocate

record:Record_ Type denoter →(Environ→Store→StoreXRecord-Variable)

 //allocate_variable T env sto在环境env存储sto之下创建一个T类型的变量

//allocate_record RT env sto在环境env存储sto之下，创建一个具有RT指明的域的记录变量，并给出改变了的存储和此记录变量

```
Allocate_variable[I] eny sto=
        let allocator alloc=find(env,I)in
        alloc sto
allocate_variable [arrayIL of T]env sto=
        let(sta,arrvar)=
              allocate_array(sto,integer_valuation IL
              allocate_variable T env)in
          (sto’, array-variable arrvar)
allocate_ variable [record RT end]env sto=
        let(sto,recvar)=allocate_record RT env sto in
allacate_record[I:T] env sto=
        let(sto,var)=allocate_ variable T env sto in
        (sto’,unit_record_ var(I,var))
allocate_record[I:T,RT]env sto=
        let(sto,var)=allocate_variable T env sto in
        let(sto”,recvar)=allocate_ record RT env sto’ in
        (sto”, joined_record_var(I,var,recvar))
```

2.2.7 程序.

```
Program ::= Command
指称语义：
run:Program→(Text→Text)
standard_environ： Environ
chr_function:Function
ord_function :Function
end_of_file_function :Function
end_of_line_function :Function
get_ procedure:Procedure
put_procedure： Procedure
get_integer_procedure:Procedure
put_integer_procedure:Procedure
get_end_of_line_procedure:Procedure
put_end_ of _line_procedure:Procedure
//run Ptxt给出在给定的输入正文文件时执行程序P得到的输出正文文件
 //standard_environ由所有预定义实体的束定组成的标准环境
//以下从chr_function到put_end_of_line_procedure都是标准的函数和过程
run[C]input=
        let sto=update(empty store,input_loe,text input)in
        let sto=update(sto,ouput_loc,text empty_text)in
        let sto”=execute C stnadard_environ sto in
        let text output=fetch(sto”,output_loc)in
        output
standard_environ=
        {“Boolean”→allocator primitive_allocator,
        “false”→truth_value false,
        “true”→truth value true,
         id\→let notop(truth value tr)=truth_ value(not tr)in
                unary_operator notop,
        id”/\”→binary_ operator(logical both),
        id“\y”–binary_operator(logical either),
        “Integer”→allocator primitive_allocator,
        “maxint”→integer maximum–integer,
        id“+”→binary_operator(arithmetic sum),
        id“–”→binary_operator(arithmetic difference),
        id”x”→binary operator(arithmetic product),
        id”/”→binary_operator(arithmetic truncated–quotient),
        id“//”→binary_ operator(arithmetic modulo),
        id”<”→binary_operator(relational less),
        id“<=”→binary_operator(relational(not ”greater)),
        id“>”→binary_operator(relational greater),
        id“>=”→binary_operator (relational( not ”less)),
        “char”→allocator primitive_allocator,
        “chr”→function chr_function, “ord”→function ord_function
        “eof”→function end_ of_file_function,
         “eol”→function end_ of_ line_function,
              “get”→procedure get_procedure,
              “put”→procedure put_procedure,
            “getint”→procedure get integer_procedure,
            “putint”→procedure put_integer_procedure,
             “geteol”→procedure get_end_ of_line_procedure,
            “puteol”→procedure put end_of_line_procedure,
            id“=”→binary_operator(=),
            id “\=”→binary_operator(≠) ,
}
chr_function=
    let func(value(integer int)·nil)sto=
              character(decode(int))
        in
        func
    ord_function=
        let func(value(character char)·nil)sto=
              integer(code(char))
        in
        func
    end_of_file_funetion=
        let func nil sto=
              let text txt=fetch(sto.input_loc)in
                 truth–value(end_of_file(txt))
        in
        func
    end_of _line_function=
        let func nil sto=
              let text txt=fetch(sto,input–loc)in
              truth_ value(end_of_line(txt))
        in
        func
    end_of line_function=
        let func nil sto=
              let text txt=fetch(sto,input–loc)in
                 truth_value(end_of_line(txt))
        in
        func
get_procedure=
        let proc( variable var·nil)sto=
           let text txt=fetch(sto,input_ loc)in
           let(char,txt’)=get txt in
           let stor=update_variable (sto,var,character char)in
           update(sto,input_loc,text txt)
        in
        proc
    put_procedure=
             let proc(value(character char)·nil)sto=
                let text txt=fetch(sto,output–loc)in
                let txr=append(txt,char)in
```

```
                    update(sto,output_loc,text txt)
                in
                proc
      get_integer_procedure=
      let proc(variable var·nil)sto=
          let text txt=fetch(sto,input–loc)in
          let txr=skip_blanks txt in let(int,txt”)=get=signed_integer txt’in
          let sto=update_ variable(sto,var,integer int)in
          update(sto,input_loc,text txt”)
      in
      proc
        put–ineger_procedure=
let proc(value(integer int)·nil)sto=
          let text txt=fetch(sto,output_loc)in
          let txt=append_ signed_integer(txt,int)in
          update(sto,output_loc,text txt)
in
proc
       get_end_of_line_procedure=
       let proc nil sto= let text txt=fetch(sto,input_loc)in
       let txt=skip_ line txt in
       update(sto,input loc,text txr)
in
proc
       put_end_of_line_procedure=
let proc nil sto=
                let text txt=fetch(sto,output_loc)in
                let txt=append(txt,end_of _line_character)in
                update(sto,output_loc,text txt’)
in
proc
```

2.3 语言的代码范例

```
let
    type Line is record
        length:Integer,
        content: array 80 of Char
    end;

    proc getline(var l:Line) is
        begin
            l.length := 0;
            while !eol() do
                begin
                     get(var l.content[l.length]);
                     l.length := l.length + 1;
                end;
            geteol();
         end;

    proc putreversedline(l:Line) is
         let var i:Integer
         in
            begin
                i := l.length;
                while i>0 do
                    begin
                        i := i-1;
                        put(l.content[i])
                    end;
                puteol()
            end;
         var currentline:Line
in
    while !eof() do
        begin
            getline(var currentline);
            putreversedline(currentline)
```

3. 语法分析器的实现

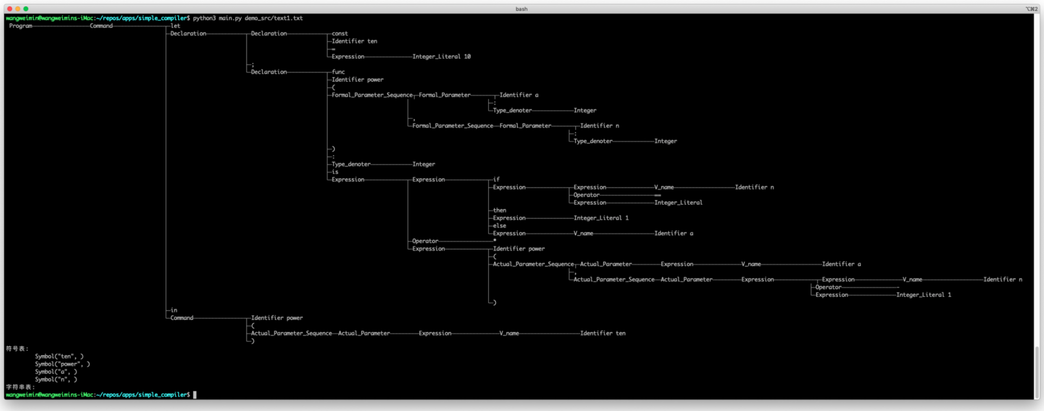
我们使用LR(1)文法构造了语言的语法分析器，可以语言源代码进行词法和语法分析并输出语法分析树和符号表。

运行 `python3 main.py make-parse-table` 程序会读取 parser/bnf.txt 中定义的语言范式构建LR(1)分析表；运行： `python3 main.py src_path` 语法分析器会对src_path指定的源代码文件进行语法分析。

下面为语法分析器对以下源程序进行语法分析的结果：

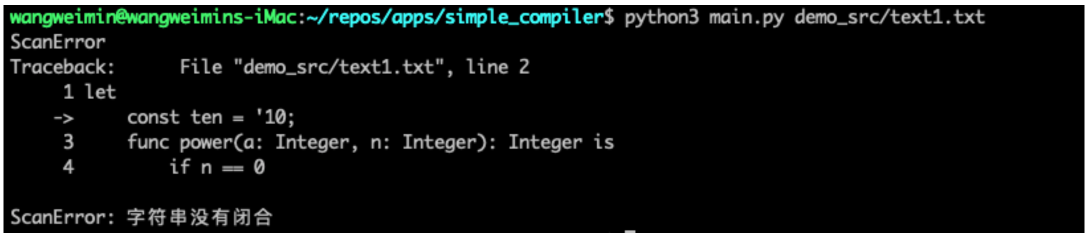
```
let
    const ten = 10;
    func power(a: Integer, n: Integer): Integer is
        if n == 0
        then 1
        else a * power(a, n-1)
in
    power(ten)
```

程序语法树构造结果：



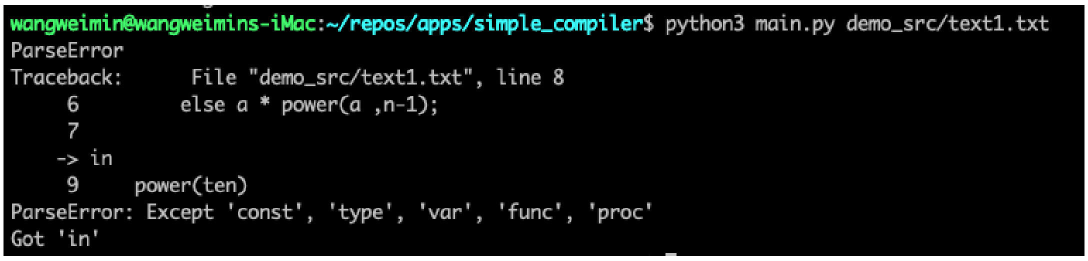
```
wangweimin@wangweimins-iMac:~/repos/apps/simple_compiler$ python3 main.py demo_src/text1.txt
 Program────Command────┬let
                       ├Declaration────┬Declaration────┬const
                       │               │               ├Identifier ten
                       │               │               ├=
                       │               │               └Expression────Integer_Literal 10
                       │               ├;
                       │               └Declaration────┬func
                       │                               ├Identifier power
                       │                               ├(
                       │                               ├Formal_Parameter_Sequence┬Formal_Parameter────┬Identifier a
                       │                               │                         │                    ├:
                       │                               │                         │                    └Type_denoter────Integer
                       │                               │                         ├,
                       │                               │                         └Formal_Parameter_Sequence──Formal_Parameter────┬Identifier n
                       │                               │                                                                         ├:
                       │                               │                                                                         └Type_denoter────Integer
                       │                               ├)
                       │                               ├:
                       │                               ├Type_denoter────Integer
                       │                               ├is
                       │                               └Expression────Expression────┬if
                       │                                              │             ├Expression────┬Expression────V_name────Identifier n
                       │                                              │             │              ├Operator──────==
                       │                                              │             │              └Expression────Integer_Literal
                       │                                              │             ├then
                       │                                              │             ├Expression────Integer_Literal 1
                       │                                              │             ├else
                       │                                              │             └Expression────V_name────Identifier a
                       │                                              ├Operator──────*
                       │                                              └Expression────┬Identifier power
                       │                                                             ├(
                       │                                                             ├Actual_Parameter_Sequence┬Actual_Parameter────Expression────V_name────Identifier a
                       │                                                             │                         ├,
                       │                                                             │                         └Actual_Parameter_Sequence──Actual_Parameter────Expression────┬Expression────V_name────Identifier n
                       │                                                             │                                                                                       ├Operator──────-
                       │                                                             │                                                                                       └Expression────Integer_Literal 1
                       │                                                             └)
                       ├in
                       └Command────────┬Identifier power
                                       ├(
                                       ├Actual_Parameter_Sequence──Actual_Parameter────Expression────V_name────Identifier ten
                                       └)
符号表:
        Symbol("ten", )
        Symbol("power", )
        Symbol("a", )
        Symbol("n", )
字符串表:
wangweimin@wangweimins-iMac:~/repos/apps/simple_compiler$
```

语法分析程序还能检测源文件中的词法、语法错误，并给出人性化的提示：
源文件存在词法错误时的输出：



```
wangweimin@wangweimins-iMac:~/repos/apps/simple_compiler$ python3 main.py demo_src/text1.txt
ScanError
Traceback:      File "demo_src/text1.txt", line 2
     1 let
    ->      const ten = '10;
     3      func power(a: Integer, n: Integer): Integer is
     4          if n == 0

ScanError: 字符串没有闭合
```

源文件存在语法错误时的输出：



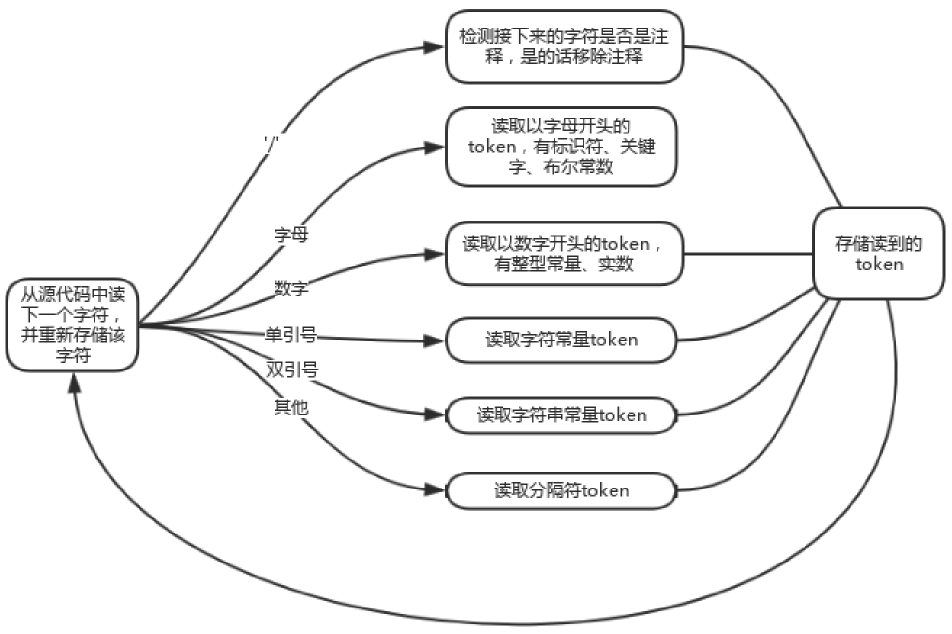
```
wangweimin@wangweimins-iMac:~/repos/apps/simple_compiler$ python3 main.py demo_src/text1.txt
ParseError
Traceback:      File "demo_src/text1.txt", line 8
     6          else a * power(a ,n-1);
     7
    -> in
     9      power(ten)
ParseError: Except 'const', 'type', 'var', 'func', 'proc'
Got 'in'
```

3.1 词法分析的实现

词法分析器对语言源代码进行词法分析，词法分析的结果为Token序列、符号表和字符串表

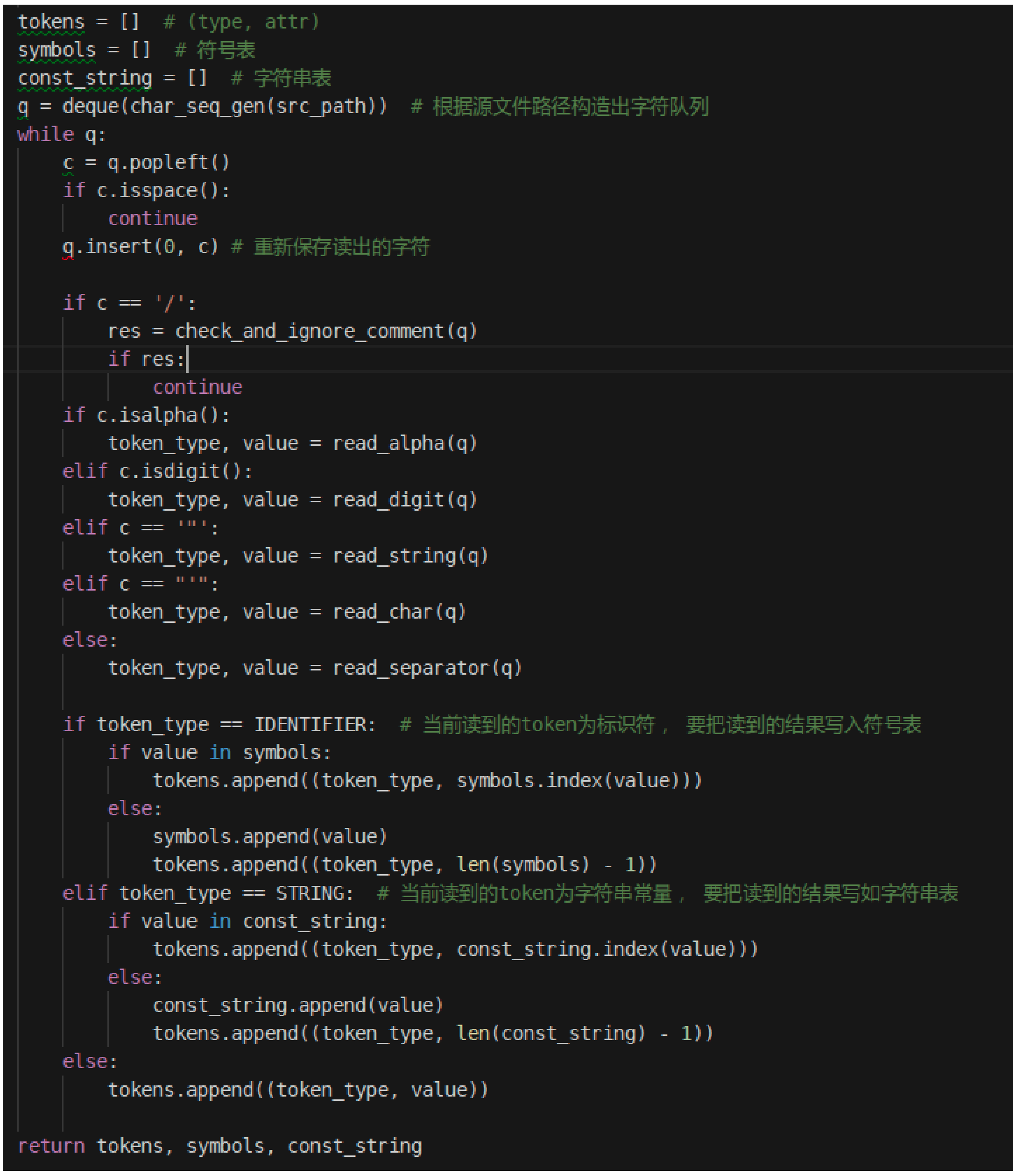
词法分析器使用了状态机进行实现，状态转移如下：

词法分析 状态机



检测接下来的字符是否是注释，是的话移除注释
读取以字母开头的token，有标识符、关键字、布尔常数
读取以数字开头的token，有整型常量、实数
读取字符常量token
读取字符串常量token
读取分隔符token
存储读到的token
从源代码中读下一个字符，并重新存储该字符
'/'
字母
数字
单引号
双引号
其他

对应代码为：

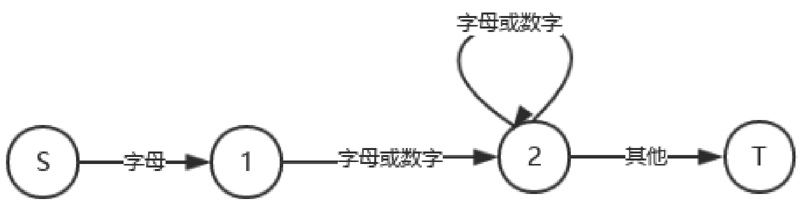


```
tokens = []  # (type, attr)
symbols = []  # 符号表
const_string = []  # 字符串表
q = deque(char_seq_gen(src_path))  # 根据源文件路径构造出字符队列
while q:
    c = q.popleft()
    if c.isspace():
        continue
    q.insert(0, c) # 重新保存读出的字符

    if c == '/':
        res = check_and_ignore_comment(q)
        if res:
            continue
    if c.isalpha():
        token_type, value = read_alpha(q)
    elif c.isdigit():
        token_type, value = read_digit(q)
    elif c == '"':
        token_type, value = read_string(q)
    elif c == "'":
        token_type, value = read_char(q)
    else:
        token_type, value = read_separator(q)

    if token_type == IDENTIFIER:  # 当前读到的token为标识符， 要把读到的结果写入符号表
        if value in symbols:
            tokens.append((token_type, symbols.index(value)))
        else:
            symbols.append(value)
            tokens.append((token_type, len(symbols) - 1))
    elif token_type == STRING:  # 当前读到的token为字符串常量， 要把读到的结果写如字符串表
        if value in const_string:
            tokens.append((token_type, const_string.index(value)))
        else:
            const_string.append(value)
            tokens.append((token_type, len(const_string) - 1))
    else:
        tokens.append((token_type, value))

return tokens, symbols, const_string
```

首先程序根据用户传入的源文件路径构造出一个字符队列，然后程序进入循环迭代，每循环一次，就从字 符队列中读出一个Token。 下面来详细分析循环中的逻辑： 每次循环开始后，从字符序列中读取一个字符，由于上文所述，可知该字符肯定为空格或者下一个Token 的开始字符。如果为空格则直接忽略，接下来则根据该字符的值缩小Token类型的范围，就比如若读到的 字符是字母，则接下来的Token肯定为标识符、关键字、布尔常数之一。然后调用相关函数读取接下来的 Token。 简单地说，程序把Token根据首字母进行分类识别，在循环中根据读到的Token首字符调用相关的识别函 数，同时存储识别到的Token。
下面是各类标识符的读取过程状态机：

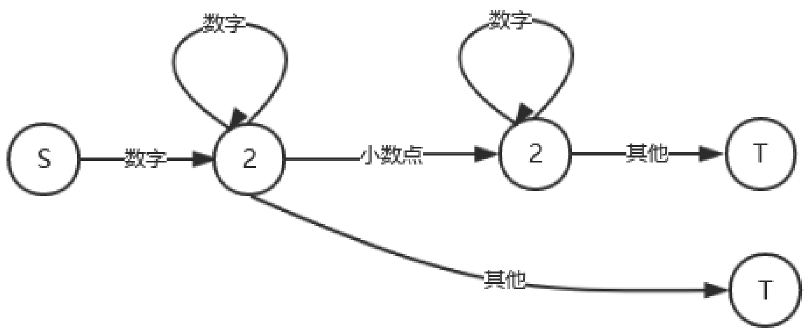
标识符、关键字、布尔常数识别状态机：

标识符、关键字、布尔常数 状态机



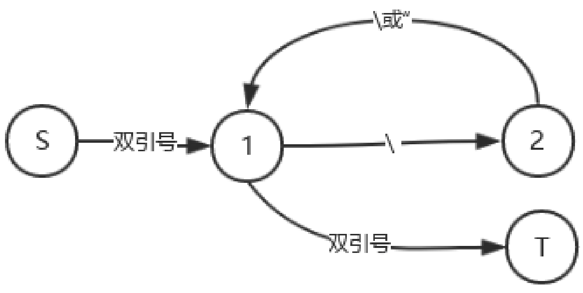
整形和实数识别状态机：

整形和实数 状态机



字符常数识别状态机：

字符常数 状态机



3.2 语法分析的实现

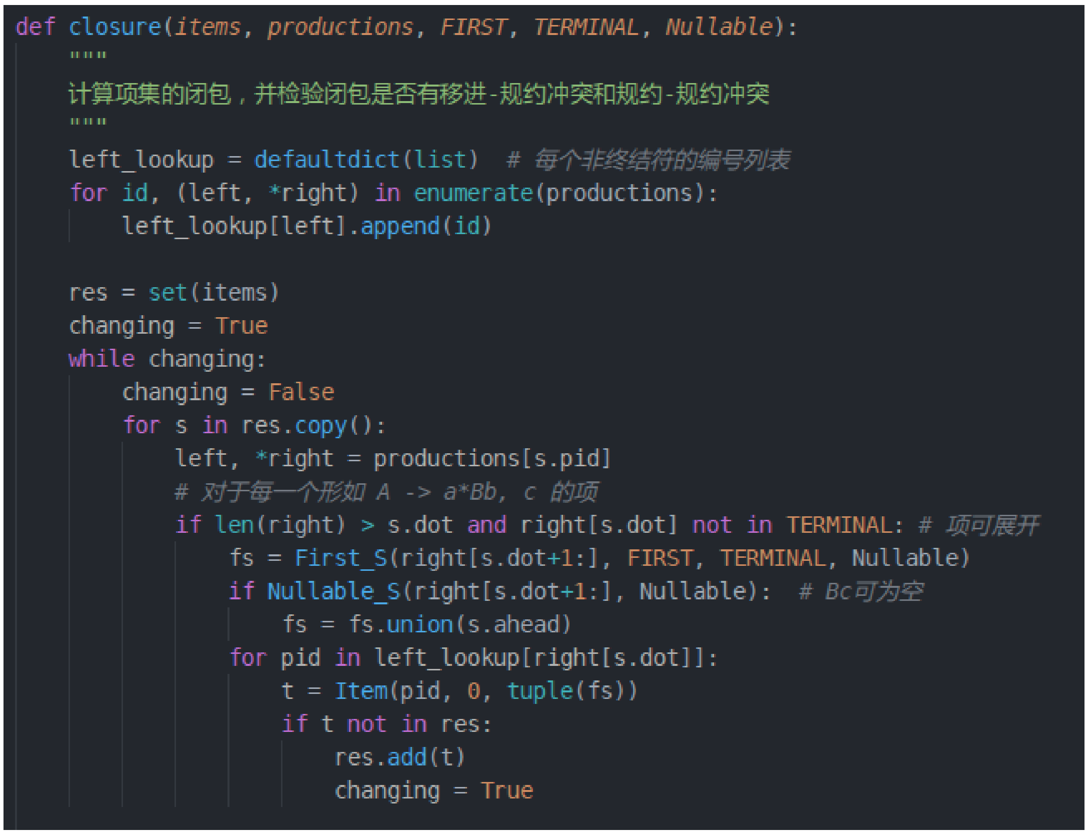
语法分析部分利用语言的文法推导式构造LR(1)分析表，并使用LR(1)分析表对语言源代码的Token序列进行语法分析。

语法分析表的构造

模块路径： parser/tools.py ，本模块可以根据输入的文法推导式列表计算其语法非终结符的FIRST集、NULLABLE集。可根据输入项集 计算项集的闭包、计算GOTO转移状态，求取语法分析表。

模块中项的由类 Item 表示，其成员函数有 pid dot ahead ， pid 为该项使用的推导式的编号， dot 为项中点的位置， ahead 为项的前看 符号集合，这里项的表示和课上的表示略有区别，这里 ahead 变成了集合，这样，多个相同推导式并且点的位置相同的项可以合并。

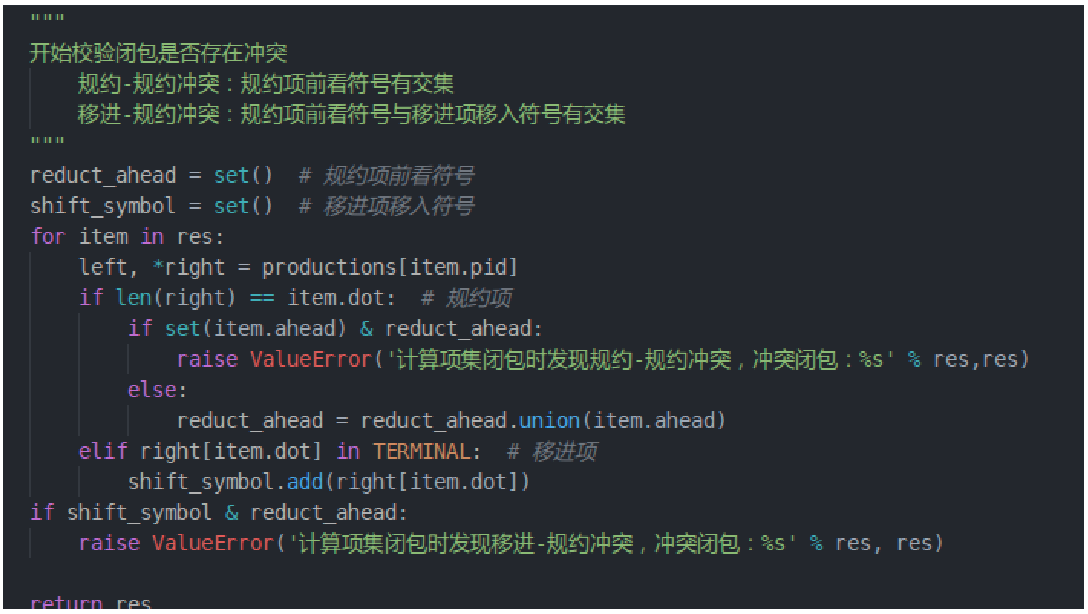
以下选取计算项集的闭包和求取语法分析表两个函数进行分析：



```python
def closure(items, productions, FIRST, TERMINAL, Nullable):
    """
    计算项集的闭包，并检验闭包是否有移进-规约冲突和规约-规约冲突
    """
    left_lookup = defaultdict(list)  # 每个非终结符的编号列表
    for id, (left, *right) in enumerate(productions):
        left_lookup[left].append(id)

    res = set(items)
    changing = True
    while changing:
        changing = False
        for s in res.copy():
            left, *right = productions[s.pid]
            # 对于每一个形如 A -> a*Bb, c 的项
            if len(right) > s.dot and right[s.dot] not in TERMINAL: # 项可展开
                fs = First_S(right[s.dot+1:], FIRST, TERMINAL, Nullable)
                if Nullable_S(right[s.dot+1:], Nullable):  # Bc可为空
                    fs = fs.union(s.ahead)
                for pid in left_lookup[right[s.dot]]:
                    t = Item(pid, 0, tuple(fs))
                    if t not in res:
                        res.add(t)
                        changing = True
```

函数先是计算出每个非终结符的推导式编号列表，用于在后面对待归约项目(形如 A–> α·Bβ)进行展开时求取新项的推导式标号。然后遍历 项集中每一项，如果项可展开(右部长度大于点的位置)且点后面为非终结符，这里以 A –> α·Bβ, (a,b,c,..) 为例，则开始求取展开项的 前看符号的集合，即 FIRST(βa)+FIRST(βb)+FIRST(βc)+... ，然后对于待规约符号 B 的每个推导式 B –> α , B –> β ...，生成相应的项 B –> ·α, ahead , B –> ·β, ahead 加入项集中。

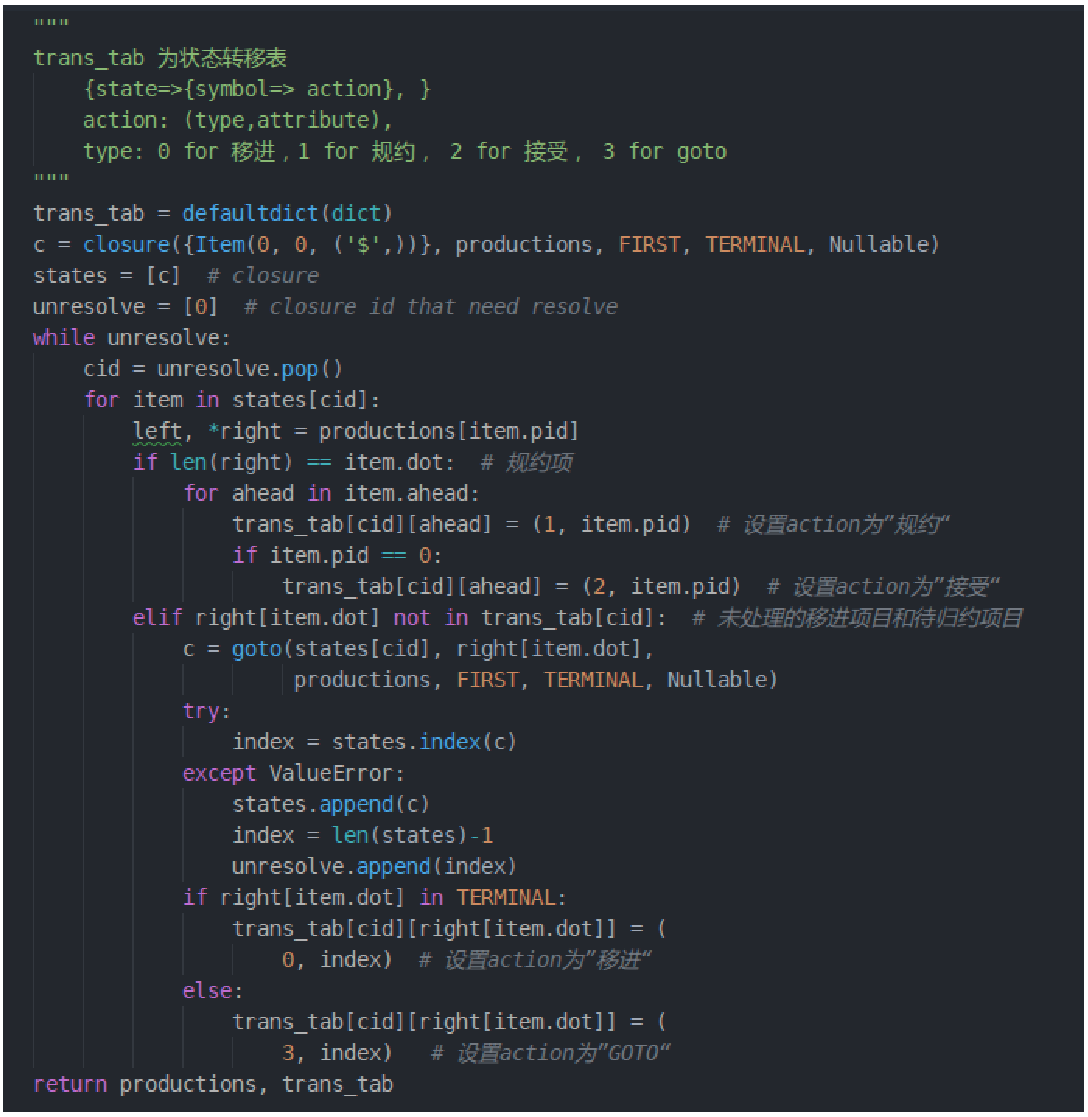
由于循环终止的条件是项集不发生改变，对此的实现为：设置一个标志变量 changing 表示本次迭代有没有对项集进行改变，在迭代中，对 项集进行了实际的修改后，就将此变量置为真。



```python
"""
开始校验闭包是否存在冲突
    规约-规约冲突：规约项前看符号有交集
    移进-规约冲突：规约项前看符号与移进项移入符号有交集
"""
reduct_ahead = set()  # 规约项前看符号
shift_symbol = set()  # 移进项移入符号
for item in res:
    left, *right = productions[item.pid]
    if len(right) == item.dot:  # 规约项
        if set(item.ahead) & reduct_ahead:
            raise ValueError('计算项集闭包时发现规约-规约冲突，冲突闭包：%s' % res,res)
        else:
            reduct_ahead = reduct_ahead.union(item.ahead)
    elif right[item.dot] in TERMINAL:  # 移进项
        shift_symbol.add(right[item.dot])
if shift_symbol & reduct_ahead:
    raise ValueError('计算项集闭包时发现移进-规约冲突，冲突闭包：%s' % res, res)

return res
```

计算完项集闭包后还要对计算出的闭包进行冲突检查，判断是否存在规约–规约冲突和移入–规约冲突。

构造语法分析表的过程如下：



```python
"""
trans_tab 为状态转移表
    {state=>{symbol=> action}, }
    action: (type,attribute),
    type: 0 for 移进，1 for 规约， 2 for 接受， 3 for goto
"""
trans_tab = defaultdict(dict)
c = closure({Item(0, 0, ('$',))}, productions, FIRST, TERMINAL, Nullable)
states = [c]  # closure
unresolve = [0]  # closure id that need resolve
while unresolve:
    cid = unresolve.pop()
    for item in states[cid]:
        left, *right = productions[item.pid]
        if len(right) == item.dot:  # 规约项
            for ahead in item.ahead:
                trans_tab[cid][ahead] = (1, item.pid)  # 设置action为"规约“
                if item.pid == 0:
                    trans_tab[cid][ahead] = (2, item.pid)  # 设置action为"接受“
        elif right[item.dot] not in trans_tab[cid]:  # 未处理的移进项目和待归约项目
            c = goto(states[cid], right[item.dot],
                      productions, FIRST, TERMINAL, Nullable)
            try:
                index = states.index(c)
            except ValueError:
                states.append(c)
                index = len(states)-1
                unresolve.append(index)
            if right[item.dot] in TERMINAL:
                trans_tab[cid][right[item.dot]] = (
                    0, index)  # 设置action为"移进“
            else:
                trans_tab[cid][right[item.dot]] = (
                    3, index)   # 设置action为"GOTO“
return productions, trans_tab
```

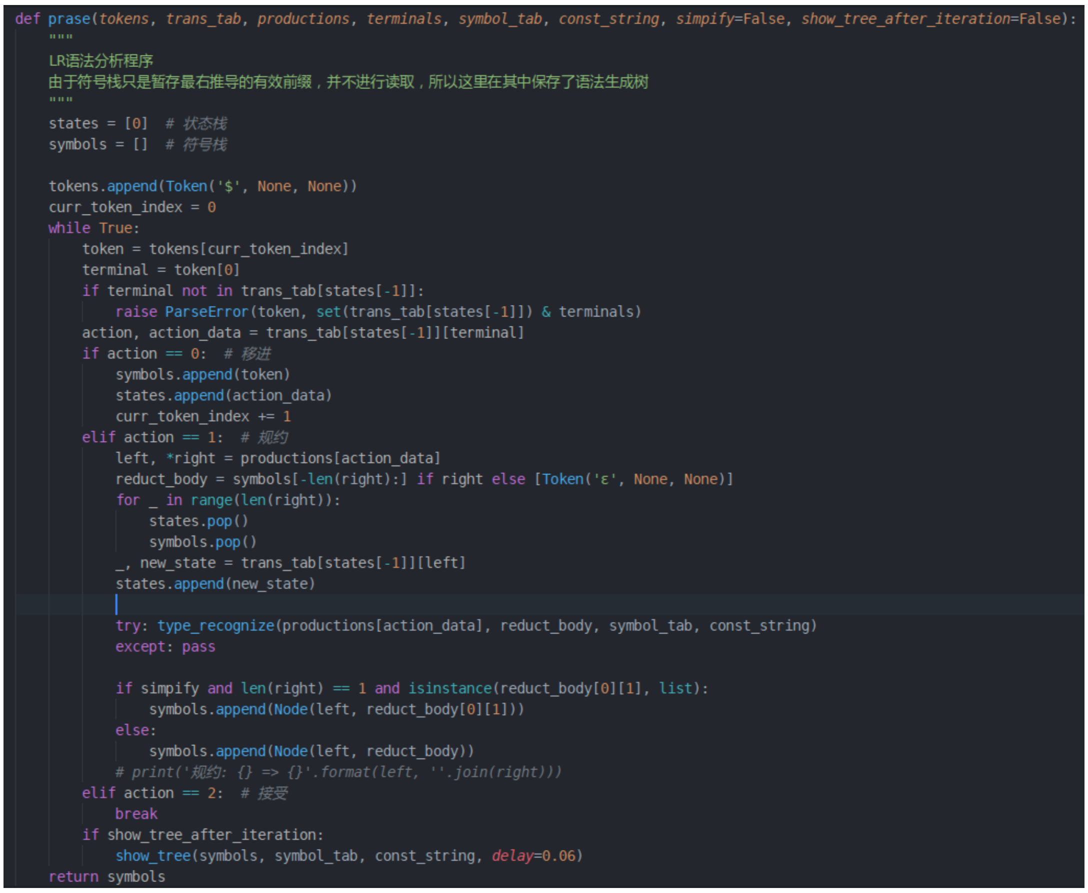
语法分析表既包含Action表又包含GOTO表，采用的数据结构为双层嵌套字典，外层字典的键为闭包编号(状态编号)，内层字典的键为终结 符和非终结符，终结符键对应的值为原Action表中的前看符号为终结符时进行的操作(移入并转移状态/使用推导式进行规约)，非终结符对应的值为GOTO表中当前状态输入非终结符后转移的状态。

函数维护了一个待处理闭包的栈，首先程序计算增广文法第一个项的闭包，将其加入待处理闭包栈，然后开始while循环，直到待处理闭包栈 为空。在每次循环体中，先取出栈顶未处理的闭包，遍历闭包的每一项：对于规约项，向语法分析表的相应位置填入规约动作和规约使用的推导表达式编号；对于移进项和待规约项，调用GOTO函数计算移进和规约之后项的闭包，判断新闭包是否已经存在，不存在则进行保存并 将新闭包加入待处理闭包栈，之后更新语法分析表。

由于在计算闭包的时候已经判断了冲突，所以这里无需再进行冲突判断。

语法分析模块

语法分析模块使用语法分析表进行语法分析，构造语法树。



```python
def prase(tokens, trans_tab, productions, terminals, symbol_tab, const_string, simpify=False, show_tree_after_iteration=False):
    """
    LR语法分析程序
    由于符号栈只是暂存最右推导的有效前缀，并不进行读取，所以这里在其中保存了语法生成树
    """
    states = [0]  # 状态栈
    symbols = []  # 符号栈

    tokens.append(Token('$', None, None))
    curr_token_index = 0
    while True:
        token = tokens[curr_token_index]
        terminal = token[0]
        if terminal not in trans_tab[states[-1]]:
            raise ParseError(token, set(trans_tab[states[-1]]) & terminals)
        action, action_data = trans_tab[states[-1]][terminal]
        if action == 0:  # 移进
            symbols.append(token)
            states.append(action_data)
            curr_token_index += 1
        elif action == 1:  # 规约
            left, *right = productions[action_data]
            reduct_body = symbols[-len(right):] if right else [Token('ε', None, None)]
            for _ in range(len(right)):
                states.pop()
                symbols.pop()
            _, new_state = trans_tab[states[-1]][left]
            states.append(new_state)

            try: type_recognize(productions[action_data], reduct_body, symbol_tab, const_string)
            except: pass

            if simpify and len(right) == 1 and isinstance(reduct_body[0][1], list):
                symbols.append(Node(left, reduct_body[0][1]))
            else:
                symbols.append(Node(left, reduct_body))
            # print('规约: {} => {}'.format(left, ''.join(right)))
        elif action == 2:  # 接受
            break
        if show_tree_after_iteration:
            show_tree(symbols, symbol_tab, const_string, delay=0.06)
    return symbols
```

程序采用双栈分保存状态和符号，并对符号栈进行特殊处理：每次弹栈时并不丢弃栈顶符号串，而是将他们作为新入栈符号的子节点，实质 上符号栈就保存了当前生成的语法森林(多个子语法树)，最后规约到开始符号后，符号栈栈顶元素即为语法树。