**面向类Pascal的静态编译语言设计**

**及语言的语法与指称语义分析设计**

姓名：黄俊鹏 学号：ZY1906408

**1. 语言设计的背景与范型**

Python是一种解释型脚本语言，也是一种面向对象的动态类型语言，具有丰富的语法和类库。但是python的缺点有执行效率低，运行速度慢，很多情况下无法将程序连写成一行；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。

Pascal语言是最早出现的结构化编程语言，具有丰富的数据类型和简洁灵活的操作语句与控制结构，并且语法严谨，层次分明，程序易写，可读性强，编译运行效率高，查错能力强，有严格的结构化形式。

因此，本语言的设计目标是一种类Pascal静态类型的编译型语言，拥有高效的执行效率，又具有丰富的语言特性，并能够针对Python的部分缺点进行改进。所以本语言借鉴参考Python和C语言的众多优秀特性，同时以Pascal语言的静态编译特点为框架结构，设计出一个完备完整的，并且具有高效执行效率，严格词法语法错误反馈机制的结构化程序语言。

**2. 语言设计思路与特点**

设计目标：从底层到上层完整实现一门完备性良好的结构化的静态编译语言。

设计原则：严格的结构化形式，完备的数据类型，高效率的编写运行，严格的纠错机制。

设计思路：早期过程式语言调研，词法语法设计，指称语义设计与说明，词法分析，语法分析，纠错设计。

设计特色与优点：

(1)精简性：本语言作为一个轻量级静态编译语言，舍弃了不常用的词法和语法形式，使得用户在学习，编写，编译，查错过程中都能体会到简洁方便。

(2)完备性：虽然本语言较为精简，但是就词法来说，本语言几乎囊括了所有常用关键字，运算符和数据类型。就语法来说，本语言借鉴参考了Python、C、Pascal语言的语法结构，实现了从命令到程序的所有常见功能，完全满足轻量编程实践的需要。

(3)完整性：本语言完整实现了从语言设计到语法分析器的步骤，并且考虑到了用户体验，用户可直接从本语言的输出提示中获知编写的语言在编译层面的错误。

(4)词法语法错误反馈机制：借鉴了市场上绝大部分编译器的特点，本语言通过语法树可以给用户指出在哪一行，哪一部分代码可能发生了哪一类型错误。

**3. 语言词法设计与EBNF**

**3.1 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

**3.2 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 ::= + | - | \* | / | , | ; | > | >= | < | <= | = | == | != | ( | ) | { | } | [ | ] | ^ | && | || | ! | /\* | \*/ | : | % | // | ++ | -- | >> | << | += | -= | \*= | /+

**3.3 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

另外有：

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"

**4. 语言语法设计与EBNF**

**4.1 命令语法**

本语言在命令语法中包含完整的变量声明、参数传递、兼容原Pascal的begin-end与Let-in设计结构，同时本语言考虑设计了IF-Then-Else和While-Do当下流行的C语言结构（包含了递归机制的实现），并且考虑异常的错误捕捉设计了Try-Catch的语法结构。

本语言设计的语法结构相比Python会更加的简洁，并且设计的语句Command；Command解决了Python无法一行执行多个语句的缺点，设计重点在于静态编译特性能够高效率的编译执行代码。

本语言将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

**语义介绍：**

·空命令执行时无作用。

·赋值命令‘Vi=E’中V和E的类型必须等价。执行过程是：先对E求值，标识符V代表的变量被产生的值更新。

·过程调用命令‘I（APS)’。执行过程是：实参序列APS先求值，产生一变元素。在此变元表调用名为I的过程。变元表必须和I过程的形参序列匹配（个数、类型、次序）。

·顺序命令‘C1；C2’执行过程是：先执行C1，然后执行C2。

·括在语句括号中的命令‘begin C end'，仅执行C（C可按上述产生式展开）。

·块命令‘let D in C'的执行是：先确立声明，然后在所产生的束定环境下执行C。D所产生的束定对块外不起作用。

·if命令‘if E then cl elsec2'的执行是：先对表达式E求值。若为真执行C1后，跳出本命令；否则执行C2。E的类型必须是Boolean类型。

·while命令‘while E do C'的执行是：先对表达式E求值，若为‘真’，执行C；若while命令再次执行直至对E求值为‘假’，跳过C结束while命令。其中E必须是Boolean类型。

·try命令‘try C1 catch(I: T) C2’的执行是：先执行C，然后假如在C执行过程中出现错误I：T类型，则捕捉错误并且执行C2。

**4.2 表达式语法**

在表达式方面本语言设计包含了完备的函数抽象机制、过程抽象机制（参考Pascal的过程语言设计）、参数抽象机制，并且完备的兼容了数组机制与指针机制(\* Identifier,& Identifier)。

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

**语义介绍：**

·表达式‘IL’产生整数字面量的值，表达式必须是lnteger类型。

·表达式‘CL'产生字符字面量的值，表达式必须是Char类型。

·表达式‘V'产生由V标识的值，V是值或变量的名字，表达式的类型即V的类型。

·函数调用表达式‘I（APS）’的求值是：实参序列求值产生一个变元素，束定于L的函数以此变元表调用。APS必须和函数形参序列匹配。I必须束定于函数。该表达式类型即函数结果值类型。

·表达式‘OE'等价于函数调用‘O（E）'，O为运算符，E为基本表达式。

·表达式‘(E)’得到表达式E的结果值。

·块表达式‘let D in E'的求值是：先确立声明D，然后在此声明产生的束定覆盖的环境中对E求值。D产生束定对块外不起作用。表达式的类型即E的类型。

·if表达式‘if E1 then E2 else E3'的求值是：先对表达式E1求值，若为真，对E2求值；否则对E3求值。E1的类型必须是Boolean，表达式的类型用E2或E3。

·表达式‘{RA}’产生的值即记录聚集RA产生的值。‘{11=E1，……En)'的类型是‘record I1:T1…,In:Tn end',其中每个Ei的类型Ti。

·表达式‘[AA]’产生的值即数组聚集AA产生的值。‘[E1，…En]’的类型是‘array n of T’类型。每个表达式Ei的类型均为T。

·表达式‘\*I'产生一个标识符并且这个标识符类型是指针形式的\*，这里只有类型为地址的参数或者变量才能赋值给改标识符。

·表达式‘&I'产生一个标识符，并且这个标识符是对参数取其地址。

·记录聚集‘I=E'产生一个记录成分的值，仅有一个标识符和由表达式产生的值域。

·记录聚集‘I=E，RA'产生一个记录值。其第一域有标识符I和E产生的值。

·数组聚集‘E，AA'产生一个数组值。其第一成分（下标为0)是E产生的值，其余成分值由AA定（下标按整数递增）。

**4.3 名字语法**

在名称方面，设计兼容了数组形式表达，能完备的表示数组类型的计算。

V\_name ::= Identifier

    | V\_name, Identifier

| V\_name[Expression]

**语义介绍：**

·标识符‘I’是简单变量值的名字，即I必须束定于变量或值。其类型与变量或值同。

·限定标识符‘V.I’是由V标识的记录中的I域。V必须是记录类型。限定标识符的类型同V中I域的类型。

·索引名‘V[E]’是由V标识的数组中的由表达式E产生的值索引的V中成分，如数组中无此成分程序失败。V为数组类型，索引名的类型即数组成分的类型。

**4.4 声明语法**

在声明方面，本语言设计包含了常量和变量参数的机制(const，var)，proc过程抽象机制、func函数抽象机制（包含了函数调用函数的递归机制），本语言设计的函数和过程抽象机制是充分考虑Python与Pascal的语言特性，所以这两种机制是充分兼容的，能够保证语法上的完整性。

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

**语义介绍：**

·常量声明‘const I=E'的确立是：将I束定于由E产生的值，I和E同一类型。

·变量声明‘var I：T’的确立是：将2束定于新创建的类型为T的变量。变量当前值为初始化的未定义。变量的生存期限于确立本变量的块的生存期。

·过程声明‘proc I（FPS)is C'的确立是；将I束定于一个过程。该过程的形参序列是FPS，过程体是C。当以变元素调用过程时，首先将变元素和FPS匹配，然后执行C，被束定的形式参数覆盖过程声明的环境。

·函数声明‘func I（FPS)：T is E'的确立是；将1束定于一个函数，其形参序列是FPS，体是E，函数返回值类型是T。当以变元素调用函数时，首先将变元素和FPS匹配，然后对E求值，得返回值。被束定的形式参数覆盖函数声明的环境。

·类型声明‘type1=T'的确立是：将I束定于类型指明符T。

·顺序声明‘D1；D2’的确立是：先确立D1再确立D2，产生它们组合的束定环境，即D2的环境可覆盖D1的环境，D1，D2不应对同一标识符束定。

**4.5 参数语法**

在参数方面的设计，本语言包含了复数参数机制、多参数机制，并且兼容了过程抽象和函数抽象，总体上形成了一个完备的编程语言。多参数机制与复数参数机制可以使得函数的调用中多实参的传入，和多个实参与形参的结合，让函数抽象机制更加便捷迅速。

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

**语义介绍：**

·形参序列‘Fp1，…，Fpn'中的每一个形参Fpi和实参序列‘Ap1，…，Apn'产生的变元相匹配（形、实参的个数、类型、次序均一致，且均可为空）。

·形式参数‘I：T’与变元的结合是将I束定于变元。对应的实参数必须形如‘E’，对E求值得变元值。E的类型与指明符T类型同。

·形式参数‘var1：T’与变元变量相结合是将I束定于该变元。对应的实在参数必须形如‘var V'，变元变量由V标识。V的类型与指明符T的类型相同。

·形式参数可为‘proc I（FPS）’与过程变元相结合，将I束定于该变元。对应的实在参数必须形如‘proc I’，过程变元由I标识，该过程必须有与FPS等价的形参序列。

·形式参数可为‘func1（FPS)：T’与函数变元相结合，将I束定于该变元。对应的实在参数必须形如‘func I’，函数变元由I标识。该函数必须有与FPS等价的形参序列，且结果值类型与类型指明符T同。

**4.6 类型指示符语法**

在类型指示符方面，语法设计包含编程语言常用的类型声明（包含指针设计）。并且在类型指示符里面本语言的指针机制得到充分体现：\* Type\_denoter的形式，结合前面的&操作，本语言设计可以得到完备的指针系统，

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

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

**语义介绍：**

·类型指明符‘I’将指定的类型束定于I。

·类型指明符‘array IL of T'指明其值为数组的类型。该类型的每个数组值有一索引范围，下界为零，上界不大于整数字面量IL。对每个索引范围内的值都对应一个类型为T的数组成分。

·类型指明符‘record RT end'指明其值为记录的类型。该类型的每个记录值其记录结构由RT指明。

·记录类型指明符‘1：T'指明的记录结构仅有标识符为I，类型为T的一个域。

·记录类型指明符‘1：T，RT'指明的记录结构。其第一域有标识符I和类型T，其余域是由RT指明的记录结构，I不得为RT中域的标识符。

·记录类型指明符‘\*T'指明T其值为指针的类型

**4.7 程序语法**

Program ::= Command

**5. 语言指称语义设计说明**

**5.1 命令指称语义**

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

**5.2 表达式指称语义**

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)

**5.3 名字指称语义**

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)

**5.4 声明指称语义**

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 Dl env sto **in**

**let**(env"，sto")=elaborate D2(overlay (env',env))sto' **in**

(overlay(env",env'),sto")

**5.5 参数指称语义**

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

**5.6 类型指示符指称语义**

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 [arraylL 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))

**5.7 程序指称语义**

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

**6. 语言代码范例**

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)

**7. 语言具体设计与实现**

本语言使用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。

并且本语言是在Windows与Ubantu16.04的环境进行开发设计，设计过程充分考虑两个平台的兼容性。

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

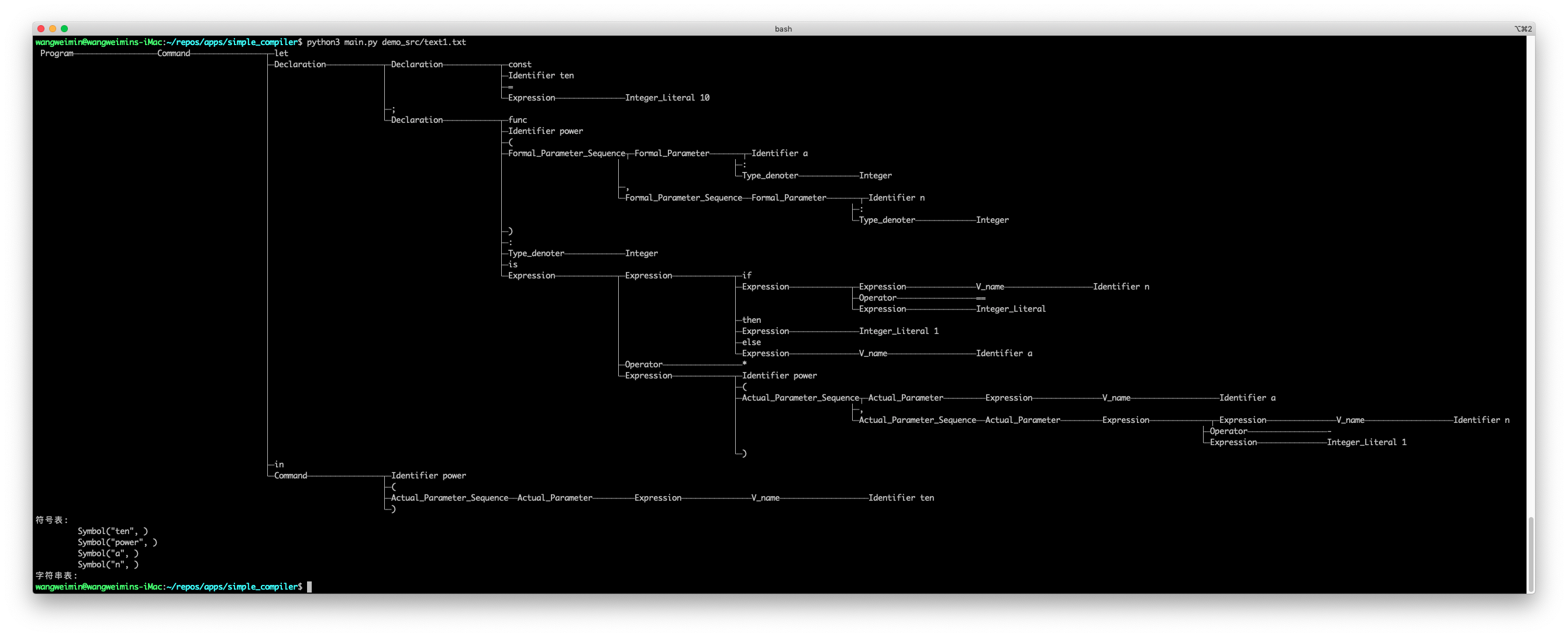
then 1

else a \* power(a, n-1)

in

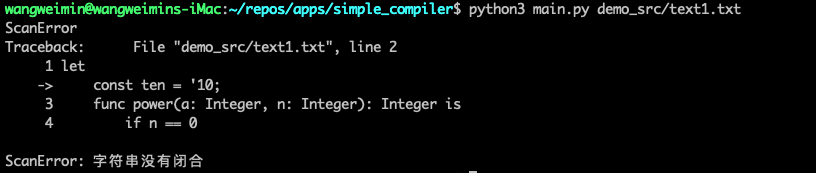
power(ten)

程序语法树构造结果：



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：



源文件存在语法错误时的输出：

