

程序设计语言原理

**语言设计规格说明书**

|  |  |
| --- | --- |
| 院（系）名称 | 计算机学院 |
| 学生姓名 | 黄俊鹏 |
| 学号 | ZY1906408 |

**2019**年**12**月

1. 任务分工与安排

**工作量：**我负责语言设计中大概全部(90%)语法的设计，主要是参考网络上的python的语法文档和C语言的语法文档，包括书上IMP语言和书本附录语言。

**主要设计内容有**：

1.设计的语言在命令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、同时我们考虑设计了IF-Then-Else和While-Do当下流行的C语言结构（同时包含了递归机制），并且考虑异常的错误捕捉设计了Try-Catch的语法结构。

2. 在名称方面，设计兼容了数组形式表达。

3. 在声明方面，我们的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过程抽象机制（包含了函数调用函数的迭代机制）。

4. 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）。

5. 在表达式方面我们的设计包含了完备的函数抽象机制、过程抽象机制（参考MATLAB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。

6. 在参数方面的设计，我们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。

**其次**：本语言的语法指称语义全部都是我设计的，主要是参考书本和附录。

**最后：**PPT语法部分和前面的总结和设计原则都是我写的。其次我还负责组织同学开会讨论，敲定设计目标，并且安排任务分工等等。

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

5、无副作用的简单命令式语言，语义指称和实现上都不设计存储改变的副作用。

1. 语言设计思路与原理

**设计目标：**从底层到上层完整实现一门语言。

**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，完美的纠错机制。

**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：**

**(1)精简性：**

此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“简单”。

**(2)完备性：**

虽然此语言较为精简，但是就词法来说，我们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，我们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：**

我们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，用户可直接从我们的产品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，我们通过语法树可以给出用户在哪一行，哪一部分代码可能发生了哪一类型错误。

1. 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 Dl 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 [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))

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)