基于 Perl 语言的改进与代码实现

邓家豪 ZY2006003
孙书玮 ZY2006324

目录

一、设计语言的背景驱动

1. Perl 介绍

Perl 是一种高级、通用、直译式、动态的程序语言，由 Larry Wall 设计，发表于 1987 年 12 月 18 日。Perl 一般被认为是 Practical Extraction and Report Language(实用获取与报表语言)的缩写，是由 C 以及 sed、awk、Unix shell 及其它语言演化而来的一种语言。Perl 具有高级语言的强大灵活能力和灵活性，又像脚本描述语言一样方便。Perl 不需要编译器和链接器来运行代码，你要做的只是写出程序并告诉 Perl 来运行而已，这意味着 Perl 对于小的编程问题的快速解决方案和为大型事件创建原型来测试潜在的解决方案是十分理想的，Perl 被称为脚本语言中的瑞士军刀。

2. Perl 的不足与改进

正是由于 Perl 的灵活性和“过多的”冗余语法，它才获得了 write-only 的批评，因为许多 Perl 程序的代码难以阅读，并且达到相同目的的程序的代码长度功能差异可以是十到一百倍。Perl 引入了上下文的概念，根据上下文解释编程者之意，可能产生臆断，也因此产生种种编程陷阱。 下面列举出 Perl 的具体不足点以及我们的改进方式。

2.1 类型推导不彻底

Perl 作为一个动态语言，有一定的类型推导功能，但是又不够彻底。它分了三个数据类型，分别是标量、数组、哈希，定义时在变量名前分别加$、@、% 三个符号，这导致程序可读性低，他人在阅读代码时短时间内难以区分这三个字符，编写代码时所有的变量前都要相应的加上这三个符号，增加编写难度。针对这一问题，我们不使用这三个符号定义变量，而是通过关键字“var”来统一定义所有变量。例如：

定义整型：var i=1;
定义字符串：var s = “str”;
定义数组：var arr = (1,2,3);
定义哈希：var h = ('a'=>1,'b'=>2);

这样 Perl 的使用者定义变量时更加方便，不易出错，将类型的判断完全寄予类型推导来实现，在后续变量大大降低了开发难度，使得程序可读性也更高。

2.2 数值和数组取等号不合适，容易引发歧义

Perl 是一种弱类型语言，某些代码非常容易引起歧义，例如：$a = @list; 其中 a 是标量，list 是数组，按照常规逻辑来说，当我们赋值的时候，我们隐含地就认定两者相等了。也就是说，执行完$a = @list 之后，$a 应该就是@list。而相等又意味着自反和传递。也就是说如果 a==b，那么 b==a；同时如果 a==b，b==c，那么 a==c。赋值就意味着相等，相等就意味着自反和传递，这两个规则 Perl 也是承认的，但是，当我们考察$a=@list 的时候，它

崩溃了。所以这条语句 a 实际上是等于 list 的长度。

针对这个问题，我们将 Perl 语言改成强类型语言，不同类型变量完全不能相互赋值，并且我们引入 length 函数，让 a = length(list); 来返回 list 的长度。这样代码可读性大大增强，改成强类型语言之后，使得语言更加严谨规范。

2.3 标量之间运算不安全

Perl 语言通过$符号来定义整型和字符型，两者难以区分。例如

```
#!/usr/bin/perl
$a = 1;
print $a . 1;
```

```
11
```

```
#!/usr/bin/perl
$a = 1;
print $a + 1;
```

```
2
```

第一个示例 a 是字符串，．运算是一个连接字符串的运算符输出了一个“11”的字符串。第一个示例 a 是整型数字，输出 2。

针对这个问题，我们去除 ．运算，在定义字符串时必须加上引号，通过+运算来连接两个字符串，这样更加合理更加易读易懂。在+运算时，需要进行类型判断，若+前后量是字符串就连接两个字符串，若是整型数字，就相加得到新的数值。

2.4 合并数组的问题

Perl 支持泛型，也就是数组中可以嵌套数组，数组可以作为数组的一个元素。但是 Perl 中却不是真正的支持嵌套，而是隐形得将数组进行合并。例如

```
#!/usr/bin/perl
@a = (1,2,(3,4));
$l = @a;
print "@a[2]\n";
print $l;
```

```
3
4
```

a 是定义得一个嵌套数组，按照常规逻辑，数组得长度应该为 3，最后一个元素是(3,4)这个数组，但是实际我们输出 a 得第三个元素时输出 3，长度为 4。也就是将数组(3,4)中得元素加入到了数组 a 中，而不是我们认为得嵌套。

针对这一问题我们做了修改，最后返回得长度应该为 3，第三个元素是(3,4)这一个数组。

2.5 增加向量类型

在当今的各种应用环境中，向量都会大量的使用，我们在 Perl 语言中增加向量的使用，可以对向量直接进行加减乘的运算，让 Perl 有更广阔的应用厂家，更方便的解决各种向量问题。增加关键字 vec 来定义向量，定义方式很简单，如 vec a=(1,2,3); 这样 a 就是一个向量类型的变量了。

2.6 特殊变量的改写

Perl 语言中定义了一些特殊的变量，通常以 $, @, 或 % 作为前缀，例如$_, @_等等。定义这些特殊变量本意是为了代码更加简洁优美, 但是如果大量的使用这些特殊变量会导致代码很难读懂，得不偿失。

对此我们希望对这些特殊变量做一些更改，采用关键字来代替特殊变量使用，利用$_是表示传给子函数的参数列表，我们用关键字 PARAM 来代替$_，这样这些特殊变量更加容易记住，容易读懂，容易使用。

二、 语言的文法和语义

1. EBNF 语法

关键字：

Key_word ::= ‘var’ | ‘vec’ | ‘sub’ | ‘my’ | ‘local’ | ‘state’ | ‘if’ | ‘else’ | ‘while’ | ‘for’ | ‘foreach’ | ‘true’ | ‘false’ | ‘break’ | ‘continue’ | ‘return’ | ‘print’ | ‘function’ | ‘length’ | ‘SUM’ | ‘PARAM’

运算符：

运算符: Operator ::= <Arithmetic-op> | <Comparison -op> | <Assigning -op> | <Bit -op> | <Logical-op>
算术运算符: Arithmetic-op ::= ‘+’ | ’-‘ | ’*’ | ’/’ | ’%’ | ’**’
比较运算符: comparison-op ::= ‘==’ | ’!=’ | ‘<=>’ | ‘>’ | ’<’ | ‘>=’ | ‘<=’
赋值运算符: Assigning-op ::= ‘=’ | ‘+=’ | ’-=’ | ‘*=’ | ’/=’ | ’%=’ | ‘**=’
位运算符: Bit-op ::= ‘&’ | ‘|’ | ‘^’ | ‘~’ | ‘<<’ | ‘>>’
逻辑运算符: Logical-op ::= ‘and’ | ‘&&’ | ‘or’ | ‘||’ | ‘not’

实数类型的 EBNF：

实数分为整数型和浮点型,
Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Int_value ::= <Digit> { <Digit> }
Float_value ::= <Int_value> '.' | <Int_value> '.' <Int_value>
| [<Float_value> | <Int_value>] 'e' ['+' | '-'] [<Float_value> | <Int_value>]

向量类型的 EBNF：

Vector_ele ::= <Int_value> | <Float_value>
Vector_value ::= <Vector_ele>{ ‘,’ <Vector_ele> }

Vector ::= ‘(‘<Vector_value>‘)’

字符串类型的 EBNF：

String_value ::= ‘ ” ’ { <unicode_char> } ‘ “ ‘
其中 unicode_char 表示所有字符，在此就不一一列举。

布尔类型的 EBNF：

Bool_value ::= ‘true’ | ‘false’

数组类型的 EBNF:

Array_ele ::= <Int_value> | <Float_value> | <Vector> | <String_value> | <Bool_value>
Array_value ::= <Array_ele> { ‘,’ [<Array _ele> | <Array_value>] }
Array ::= ‘(’<Array_value>’)’

哈希类型 EBNF：

Common_value ::= <Array> | <Bool_value> | <String_value> | <Vector> | <Float_value>
| < Int_value >
Key_value ::= <Common_value> ‘=>’ < Common_value >
Hash_ele ::= <Key_value>{‘,’< Key_value >}
Hash ::= ‘(‘ <Hash_ele> ‘)’

标识符：

Letter ::= '_' | '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' | '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'
Identifier ::= <Letter> { <Letter> | <Digit> }

程序 EBNF：

Program ::= <Program_head><Program_body>
Program_head ::= ‘#!/usr/bin/perl’
Program_body ::= < Program_body_part >{< Program_body_part >}
Program_body_part ::= <Declaration_part><Sub_Program_part><Command_part>

声明 EBNF：

Declaration_part ::= [<Declaration>{<Declaration>}]
Declaration ::= <scope_lable><Declaration_type>
Declaration_type ::= <Var_ Declaration_type> | <Vec_ Declaration_type>
| <Sub_Func_ Declaration >

Var_ Declaration_type ::= ‘var ’ <Variable_name_part>’=’<Variable _value_part>’;’
Vec_ Declaration_type ::= ‘vec ’ <Variable_name_part>’=’<Variable _value_part>’;’
Sub_Func_ Declaration ::= ‘function’ <Variable_name_part>’(’ [<Param_list>] ’);’
Scope_lable ::= ‘local ’ | ’my ’ | ’ state ’
Variable_name_part ::= <Identifier>{‘,’<Identifier>}
Variable _value_part ::= < Variable _value >{‘,’<Variable _value>}
Variable_value ::= <Array> | <Bool_value> | <String_value> | <Vector> | <Float_value>
| < Int_value > | <Hash>

子程序 EBNF：

Sub_Program_part ::= ‘sub ’ <Sub_Program_name> ‘{’ < Sub_Program _body> ‘}’
Sub_Program_name ::= <Identifier>
Sub_Program _body ::= <Program_body>

命令 EBNF：

Command_part ::= <Command>{<Command>}
Command ::=
| <Identifier> ‘=’ <Expression>’;’ //赋值语句
| <Identifier> ‘(’<Param_list>’)’ ‘;’ //调用子程序
| <Declaration_part> //声明语句
| ‘if(’ <Expression> ‘){’ <Command > ‘}’ [{‘elsif(’ <Expression> ‘){’
<Command> ‘}’ } ‘else’ ‘{’ <Command > ‘}’] //条件语句
| ‘while(’ <Expression> ’){’ <Command > ‘}’ //while 循环语句
//for 循环语句
| ‘for(’ <Declaration>’;’ <Expression>’;’<Command> ‘){‘ < Command >’}’
| ‘foreach’ Identifier ‘(’ <Array>‘)’ ‘{’ <Command> ‘}’ //foreach
循环
| <Jump_Command> //跳转指令

Jump_Command ::= ‘break;’ | ‘continue;’ | ‘return ’ <Expression>’;’
Param_list ::= <Param_value> | <Param_name>
Param_value ::= <Variable _value>{‘,’<Variable _value>}
Param_name ::= <Identifier>{‘,’<Identifier>}

表达式 EBNF：

Expression ::= <Expression> <Operator> <Expression>
| [<Operator>] < Expression >
| <Int_value>
| <Float_value>
| <Bool_value>

| FuncExpr //函数返回值
| ‘(‘<Expression>’)’
| <Identifier> //已定义的变量
| < Expression ><Index>
| <PrimaryExpr> <Slice>

FuncExpr ::= <Identifier> ‘(’<Param_list>’)’
Index ::= ‘[‘ <Expression> ‘]’
Slice ::= ‘[‘ [<Expression>] ‘:’ [<Expression>] [‘:’ [<Expression>]] ‘]’

2. 语义

2.1 辅助函数

empty_store : Store
将所有单元映射为 unused（表示该存储空间未使用）的 Store 的元素。
形式化定义：empty_store = λloc.unused // 对存储空间的任何位置定义为 unused

allocate : Store → Store × Location
allocate 是某个存储快照 sto 映射为 sto'其中某些单元 loc 由未使用变成未定义(即分配了)。
形式化定义：allocate sto =
let loc = any_unused_location (sto) **in** **//** 将 unused 变成 undefined
(sto [loc→ undefined]，loc)

deallocate : Store × Location → Store
deallocate 是将存储快照 sto 中某些单元 loc 变为未使用，从而映射出新的 sto'。
形式化定义：deallocate (sto，loc) = sto [loc → unused]

update : Store × Location × Storable→Store
update 是在 sto 中以 stble 值将 loc 单元更新，从而映射出新的 sto'。
形式化定义：update (sto，loc，stble) = sto [loc→stored stble]

fetch : Store × Location → Storable
fetch 是从某 sto 的 loc 单元上取出 stble 值。
形式化定义：fetch (sto，loc) =
let stored_value (stored stble) = stble //如果是可存储值，就返回该值
stored_value (undefined) = fail //如果是 unused/undefined 返回 fail
stored_value (unused) = fail
in
stored_value (sto(loc))

empty_environ : Environ

empty_environ 是所有标识符均处于未束定状态的空环境。
形式化定义：enpty_environ = λI. unbound　　//　清空环境

bind : Identifier×Bindable → Environ
bind 是只要有了标识符束定于可束定体,则环境状态就改变了。
形式化定义：bind(I,bdble) = λI'.if I'=I then bound bdble else unbound　　//只要标识符等于 I，返回 bdble，否则返回 unbound

overlay : Environ×Environ → Environ
overlay 是两束定环境组成新环境 evn+evn'，若某标识符在两环境中均有束定，则新环境中以后束定复盖先束定。
形式化定义：overlay (env'，env) =λI. if env'(I)/=unbound then env'(I) else env(I)　　//如果是束定的，返回 env'　否则返回 env 中的

find : Environ×Identifier→Bindable
find 是在环境 env 中找出标识符 I 所对应的可束定体 bdble，若未束定则返回⊥。
形式化定义：find(env,I) =

```
let bound_value(bound bdble) = bdble
    bound_value(unbound) =  ⊥
in
    bound_value(env(I))
```

2.2　抽象语法与指称语义

Command 抽象语法:

```
Command   ::= Skip
            | Identifier := Expression
            | let Declaration in Command
            | Command; Command
            | if Expression then Command else Command
            | while Expression do Command
            | Identifier(Actual_Parametor)
            |  return Expression
```

指称语义：

execute：Commamd→(Environ→Store→Store)
//任何命令的执行导致当前存储快照下产生新的存储快照。
可以写作：execute〖C〗 env sto = sto'
下面为具体 Comand 的详细指称语义：
空语句:
execute〖Skip〗env sto = sto
//在环境 env 和存储 sto 下执行 Skip 的结果是不变的存储 sto。

赋值语句：

```
execute〖I:= E〗env sto =
        let val = evaluate E env sto in
        let variable loc = find(env，I) in
            update(sto，loc，val)
//先在 env sto 中对 E 求值，并放入临时变元 val 中；再在 env 找出 I 束定的单元放于临时变
//元 loc 中；最后以 sto，loc，val 作参数调用辅助函数 update。loc 中的值被修改，I 因而得
//赋值。
```

声明语句：

```
execute〖let D in C〗 env sto =
        let (env'，sto') = elaborate D env sto in
            execute C(overlay(env'，env)) sto'
//先在 env sto 中确立声明 D。由于确立 D 既改变了 env 又改变了 sto，则将它们作为序偶取
//出。然后在复盖了的新环境 env'' 和存储 sto'下执行 C。而 env''=overlay(env'，env)。
```

顺序语句：

```
execute〖C1; C2〗env sto =
                    execute C2 env(execute C1 env sto)
//连续执行两命令的语义是:第一命令 C1 在 env sto 下执行，不会改变束定的环境只改变存
//储。因此 C2 在改变了的存储，即括号内执行结果，中执行。
```

条件语句：

```
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
//在 env sto 下执行条件语句，首先在此 env sto 中对 E 求值，若为真值域中的 true，则在同
//一 env sto 下执行 C1，否则 C2。E 为布尔型。
```

循环语句：

```
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
//执行 while 命令的语义是 execute_while 函数的在 evn sto 之下执行结果。该函数是递归定
//义的，每次执行都会改变 sto，否则，sto 不变。E 为布尔型。
```

Expression 抽象语法：

```
Expression ::=      Real_value
                  | Bool_value
                  | Identifier
                  | unary_operator Expression
                  | Expression   binary_operator Expression
```

| Expression **?** Expression **:** Expression
| Identifier(Actual_Parmenter)
| (Expression)
| Expression[Index]

指称语义：
evaluate: Expression → (Environ → Store → Value)
每个表达式的求值是将 Expression 域中的元素 E 映射为从环境中取出改变了状态值，即在某一环境 env 的存储快照 sto 下求值 value。
下面是具体表达式的形式化语义：

evaluate〖R〗env sto =
Real_value (valuation R)
valuation 是将一个数对应一个实数。
Real_value 为辅助函数，计算实数的值。

Bool_value 的值只有两种：true 和 false
evaluate〖false〗env sto = truth_value false
evaluate〖true〗env sto = truth_value true

evaluate[I] env sto = coerce(sto, identifier env sto)
coerce 为辅助函数，在当前存储快照下找到一个标识符，得到其值。形式化语义为：
coerce(sto，find(env，I))
=val //若 I 束定于 val （常量）
|fetch(sto，loc) //若 I 束定于 loc （变量）

一元运算符的表达式以 ！运算符和&运算符为例说明。
evaluate〖！E〗env sto =
let truth_value tr = evaluate E env sto **in**
truth_value(not (tr))
not 为辅助函数。

evaluete〖&I〗env sto =
find_address(sto, find(env，I)) //返回当前快照下 I 对应的可束定体的地址
find_address 为辅助函数
find_address：Store × Bindable → Address

二元运算符以 ‘+’算术运算符和‘<’比较运算符为例说明。
evaluate〖E1 + E2〗env sto =
let Real_value R1 = evaluate E1 env sto **in**
let Real_value R2 = evaluate E2 env sto **in**
Real_value (sum(R1，R2))
Sum 为辅助函数，将两个实数值求和。
evaluate〖E1 < E2〗env sto =

let Real_value R1 = evaluate E1 env sto **in**
let inte Real_value R2 = evaluate E2 env sto **in**
truth_value (less(int1，int2))

less 为辅助函数。

三元运算符 ? :

evaluate〖E1 ? E2 : E3〗env sto =
if evaluate E1 env sto = truth_value true
then //若 E1 为 true
let Real_value R1 = evaluate E2 env sto **in**
Real_value(R1)
else //若 E1 为 false
let Real_value R2 = evaluate E3 env sto **in**
Real_value(R2)

函数调用:

evaluate〖I(AP)〗env =
let function func = find(env,I) **in**
let arg = give_argument AP env **in**
func arg

数组索引:

evaluate [A[index]] env sto=
array_value(evaluate_array A index env sto)

evaluate_array 为辅助函数，根据数组和索引找到对应存储空间的值。

哈希表达式和数组类似。

声明的抽象语法：

Declaration ::= **vec** Identifier = Expression //声明一个向量
| **var** Identifier = Expression //声明一个其他变量

声明确立的语义函数是:

elaborate: Declaration→ (Environ → Store → Environ×store)

每个声明的确立是将 Declaration 域中的元素 D 映射为将环境 env 下存储快照 sto 变为有了新束定的环境 env'×sto'的快照。具体可写作:

elaborate〖D〗 env sto =(env',sto')

elaborate〖**var** I = E 〗env sto =
let val = evaluate E env sto **in**
let (sto'，loc)= allocate sto **in**
let update(sto'，loc，val) **in**
(bind(I，variable loc)，sto'')

先在 env sto 下对表达式 E 求值，放入临时值变元 val，用辅助函数 allocate 在 sto 中分配一

个单元名为 loc，得到改变了的存储和该单元的对偶(sto'，loc)，然后更新 loc 区域的值得到 sto''，将 l 与 loc 束定返回(env'，sto'')新环境与存储。

三、设计以及结果验证

1. 主要功能以及示例代码结果展示

示例代码说明，在 src 文件下，有四个代码文件，分别为我们设计的代码的定义语句、向量使用语句、函数语句以及最后集成的一个应用实例。其中的"//"代表注释。其中的代码可以任意更改。在 Java 代码中的 Test 类中的 filePath 可以更改为不同的文件名，方便老师查看。

Talk is cheap. Show you the code.

代码放在了 GitHub 上，https://github.com/Asswei7/perl_improve

方便老师查看使用。

1.1 类型推导和类型匹配

我们在定义变量时，可以使用 var 和 vec 两种方式。对于整形、字符串、列表的变量使用 var 方式定义，对于由整数组成的向量类型，必须使用 vec 来定义。

```
var a= 5;        var b = “str”;        var c = (1,2,(1,3,4,5),”qwe”);
vec v = (1,2,3);
```

由于 Perl 语言的标量和向量类型使用很不方便，我们采取了强制类型的方式，但是定义时不需要显式地指定变量的类型，我们可以根据定义的变量值推导出其变量类型。

举例而言： （以下代码在文件 code.txt 下，可以更改使用）

```
var a= 5;   //自动识别出变量 a 为整形
var b = “qwe”;  //自动识别变量为字符串型
a = b;  //输出错误提示类型不匹配
a = “asd”;   // 输出错误提示，因为字符串类型不能传递给整形变量
a = 12;
b = “asd”;  //以上两句均能正常解析
```

1.2 向量类型的使用

我们内嵌了一个向量类型的变量，方便一些特定的处理。

举例而言:

```
var muti = 2;
vec r = (1,2,3,4);
//定义了两个向量类型的变量
vec s = (3,4,5,6);
//向量的加法即对应元素相加，得到的也是向量，必须使用 vec 声明
```

```
vec a = r+s;
//向量的乘法得到其内积，是一个整数，需要使用 var 定义得到
var b = r*s;
//以下两个语句分别是向量的加法和乘法，即向量的每个元素相加或相乘
vec c = r*muti;
vec d =r+muti;
//SUM 是向量的内置的函数，得到向量内所有元素相加之和
var c = SUM(s);
```

1.3 内置的 length 函数

针对 Perl 语言求列表类型的元素个数的不足，我们设计了内嵌的 length 函数，可以求出列表中的元素个数。

```
var ls =(1,2,(2,2),2,3,(2,3),"qwe");
var num = length(ls);
print(num);                    //输出为 7，即可以很好地识别正确的元素的个数。
```

1.4 内置的 PARAM 变量和函数

针对 Perl 语言许多$_类似的符号给人们的可读性带来很大的困扰，我们内置了一个 PARAM 变量作为函数的参数，在函数内部可以直接使用。

```
    function f(int);
    var res = f(12);
    sub f{
        var temp = PARAM;
        print(temp);
        return temp;
}
```

在 print 语句中就可以正确地打印出，12.正确地获取了函数传递给的参数值。

1.5 示例代码整合完成功能

我们这里举一个现实中最常见的例子，对一个向量进行加权求和，得到其结果。这种 应用场景在生活中十分常见，比如在计算学术的加权平均分、员工的绩效工资等等。

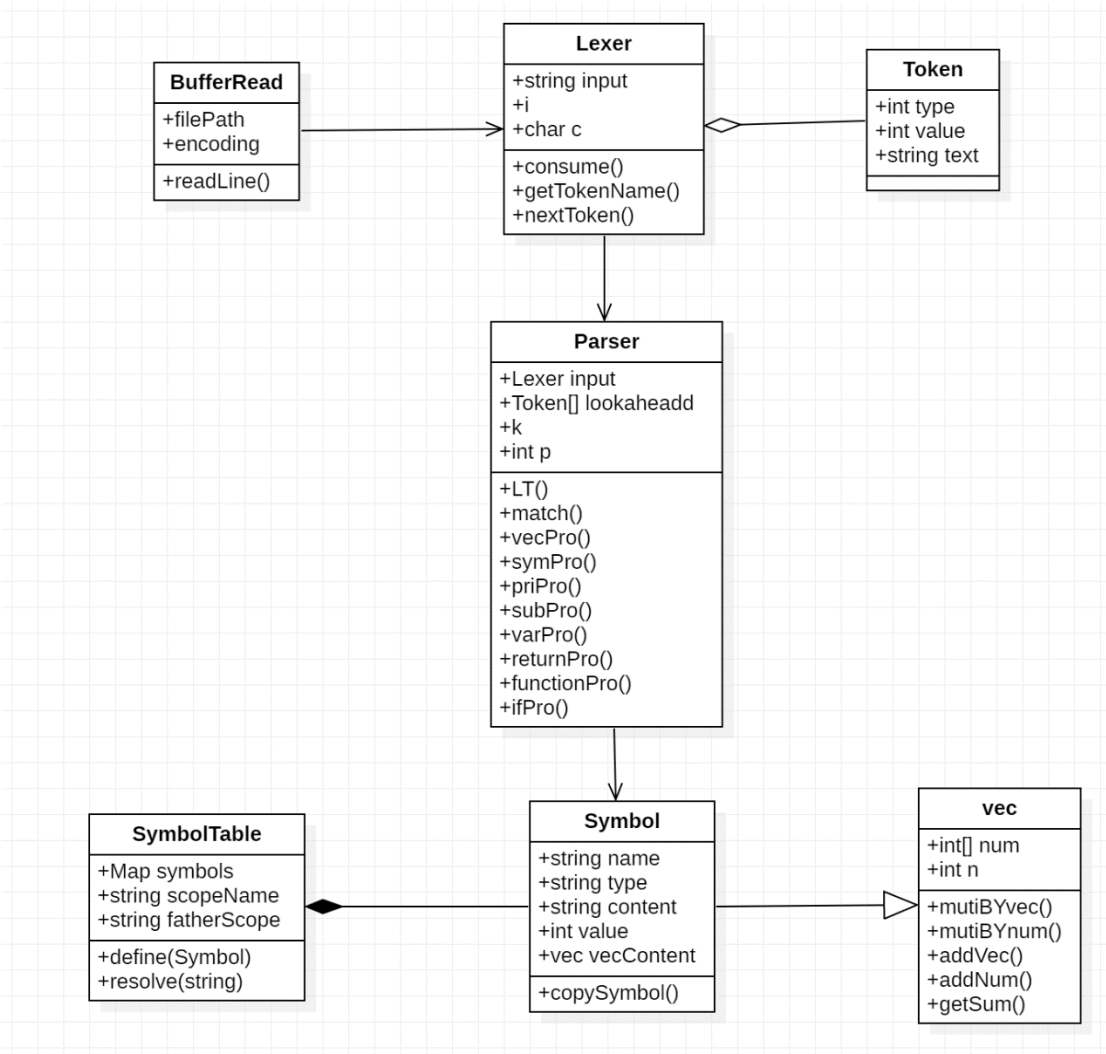
```
    vec grades = (82,71,92,98);
    //每门课的学分，也可以理解为权重
    vec credits = (3,2,4,1);
    //计算总成绩，每门课成绩乘其学分再求和
    var totalGrades = grades*credits;
    //计算学术的总学分
    var totalCredits = SUM(credits);
    //声明函数
```

```
function getCredit(int);
var a = 12;
//调用函数，得到绩点
var JIDIAN = getCredit(89);
//函数体
sub getCredit{
 var temp = PARAM;
 //测试能否读到全局作用域中的变量
 var b = a;
 print(b);
 var credit = 0;
//成绩在 90 以上，绩点为 50。低于 90，少一分绩点减 1.
 if temp > 89:   credit = 50;
 if temp < 90:   credit = 50-(90-temp);
 return credit;
}
print(JIDIAN);
```

最后可以正确地输出结果。

2.主体架构介绍

2.1 系统的类图



我们主要是使用上图中的七个类进行实现。

BufferReader 类主要是根据文件路径名读取文件的每行代码。

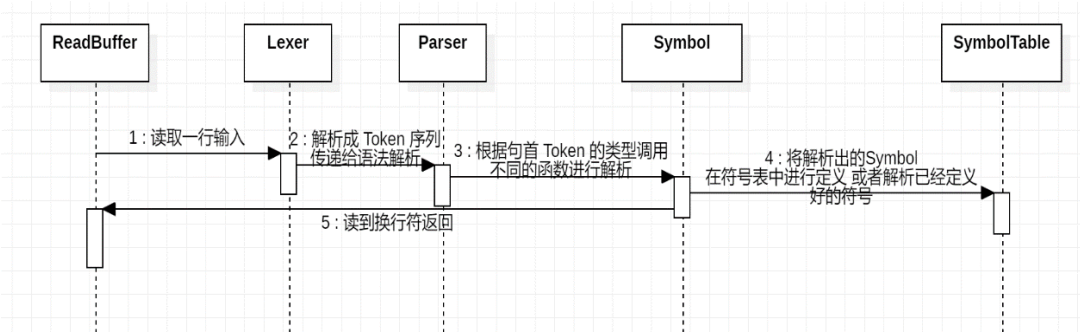
Lexer 类将输入的 String 类型的代码解析成为 Token[ ]序列，并标识出 Token 的类型。

其中 Token 有这三个属性，记录其类型，如果是 string 类型在 content 中记录，如果是 int 类型在 value 中记录。

Parser 类读入的是 Token 序列，根据第一个 Token 可以识别出这行代码的作用，然后调用不同的处理函数，对这行代码进行解析。

在 Parser 解析后，对新定义的变量生成一个 Symbol 类，存入 SymbolTable 中，下次更改这个变量值时可以解析。还有一个比较特殊的 vector 类，需要我们自己实现。

2.2 系统的时序图



首先是 BufferReader 读取文件中每一行的输入，以 string 类型传递给词法分析器，词法分析器将输入的字符串解析成为一个个的 Token，以 Token[ ]数组的形式传递给语法解析器。语法解析器根据句首的 Token 的类型，调用不同的函数进行处理。举例来说，句首的 Token 是 var 类型，说明为一个定义变量的语句，使用 varPro()函数进行处理，如果是 NAME 类型，即以变量名打头的语句，即为赋值语句，使用 symPro()函数进行处理，以此类推。

然后每一行根据其类型可以得到不同的新的符号。

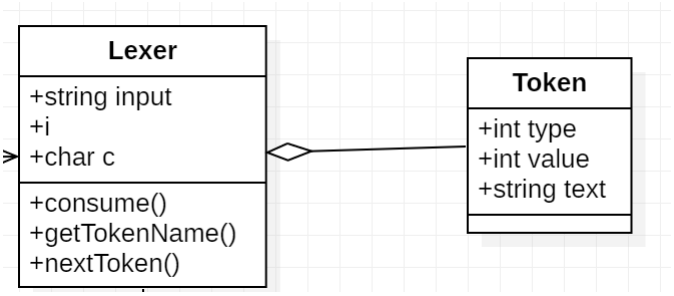
如果是 var/vec 语句，是定义语句，会生成一个新的符号并赋值，生成一个新的 Symbol，并放入 SymbolTable 当中定义。

如果是 NAME 语句，即以变量名开头的语句，为赋值语句，首先在符号表中解析该变量，如果找不到提示变量未定义。如果找到了进行类型匹配，看是否可以赋值，如果满足条件，更改这个 Symbol，并在 SymbolTable 中进行更新。

至于函数语句和 return 语句，会在下面具体类中详细介绍。

3 .主要的几个类介绍

3.1 Token 以及词法分析器



读入输入的字符串，然后根据下标 i，得到当前的字符 c，如果 c 是类似于＂[＂、＂，＂这种符号，则根据不同的符号返回不同的 Token 类型，并且将其的 type 字段和 text 字段补充完整。Token 的 value 字段主要是针对整数。

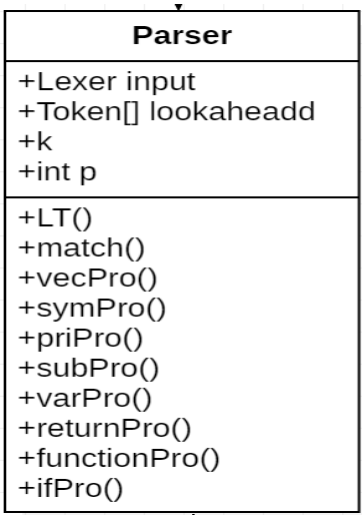
读入的字符如果是一个字母，进入 getName()函数进行处理，判断下一个字符是否是字母或者数字，如果是的话 append，否则判断这个字符串是关键字、变量名还是字符串的内容（根据是否已经读取了双引号和等号）。

举例来说，如果这个字符串等于“var”或”vec”或”return”这些关键字时，其类型即为这些关键字。否则如果词法解析在前面的 Token 中还未读取到等号和双引号，则认为该字符串代表的是变量名，其类型为 NAME，否则认为其代表 string 类型，类型为 STR。

读入的字符如果是一个整数，用一个标志位标志是否已经读取了双引号，如果已经读取，表面此时在字符串类型，进入 getName()函数进行处理。否则，认为是整数，返回 Token，并填充 value 字段。

最后将读到的 Token 序列传递给语法解析器。

3.2 语法分析器



LL(2)语法解析：

由于文法中有一些只往前看一个符号无法对其类型进行判断，所以引入了 LL(2)语法解析器。首先构建环形向前看缓冲区，用数组缓冲输入的符号，下标 p 会沿着已知大小的缓冲区移动。处理完一个词法单元，解析器只需要自增 p，然后从缓冲区的末尾补充新的词法单元。举例来说，k=2 时下标最多只能到 1，当 p 到 2 时采用对 2 取模运算，变为 0，解决下标的折回问题。

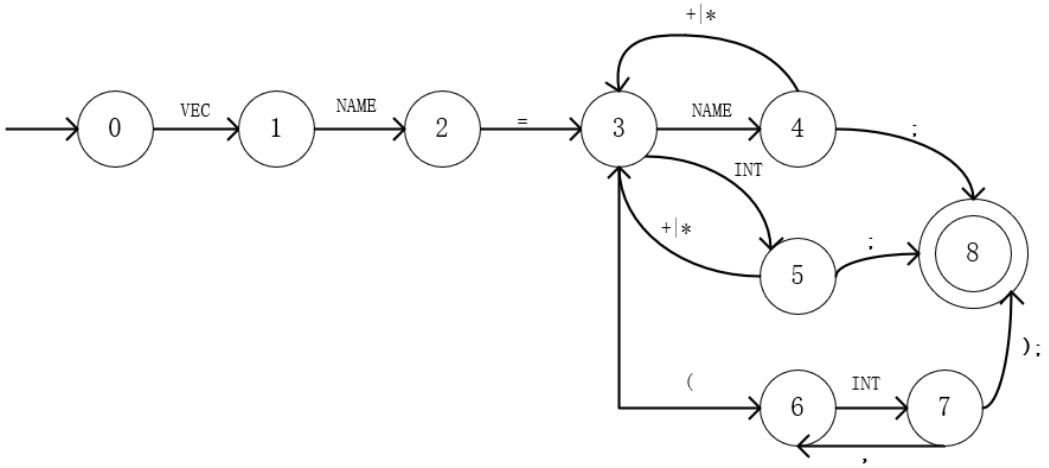
consume 函数，将下一个词法单元读入缓冲区，然后更改下标 p，改为 p=(p+1)%k;

LT 函数： 需要获取缓冲区中下一个或者下面第二个 Token，使用函数 LT，输入参数 i 即向后读第一个 Token，获取 lookahead[(p+i-1) % k]。

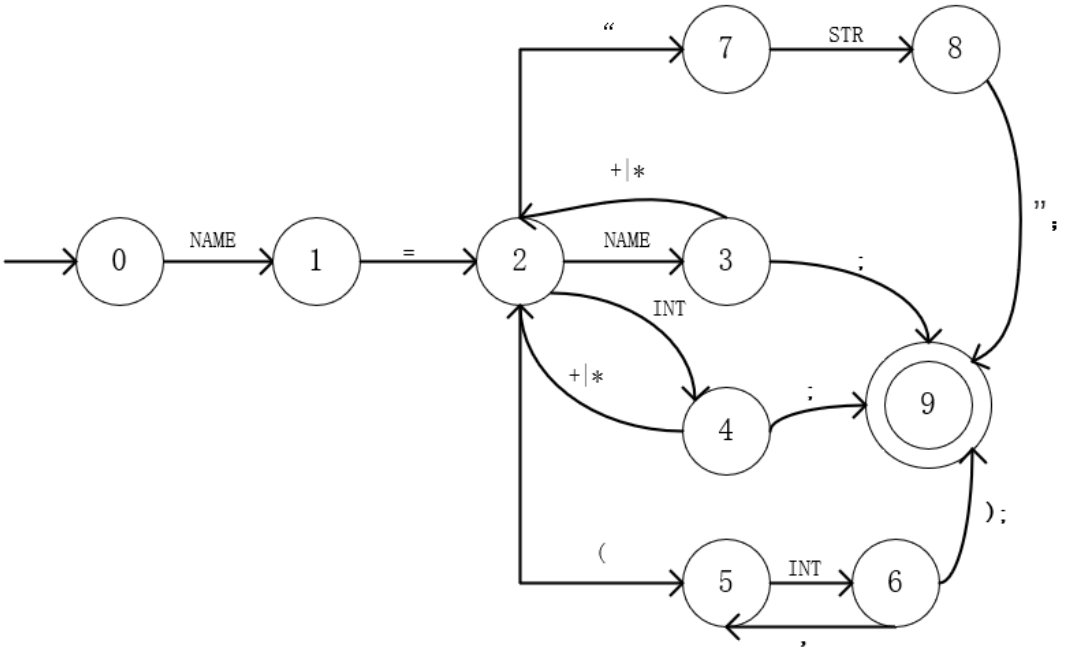
match 函数：

match 函数作用是对当前的 Token 进行匹配，如果是相同的 type，则跳过这个 Token，进入 consume 函数，即更改下标 p 以及 lookahead 数组中的内容。

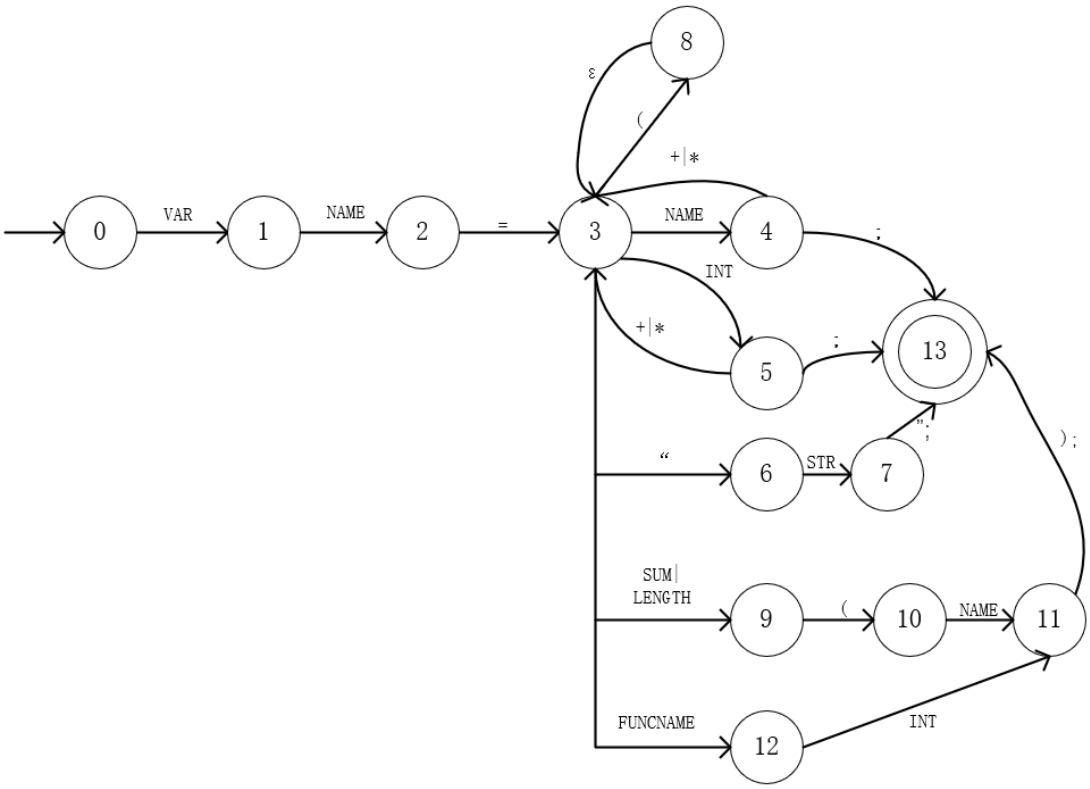
在语法解析时，根据句首不同的符号进入不同的函数进行处理。下面详细介绍以下 vecPro()，varPro()，symPro()三个最重要的函数。其状态机如下图所示：



当句首的 Token 类型为 vec 时，说明这条语句是一个定义向量类型变量的语句，其状态机形式如上图，只需要进入该函数，对不同的 Token 进行匹配，然后在符号表中保存该变量符号即可。



这是发现句首 Token 为 NAME 时使用，该条语句是更改变量的赋值，处理方式如上。



当发现句首 Token 为 var 时，说明该语句是对变量的定义，状态机如上图，只需要在函数中对 Token 进行匹配解析即可。

求带嵌套 list 的个数的计算方法

举例而言定义方法为：var x = (1,2,(2,2),2,3,(2,3),"qwe");

我们需要识别这种列表的元素个数。

其匹配模式如下：

list : ‘(’ elements’)’ ;

elements : element (‘,’ element)*;

element: INT | “STR” | list ;

首先将 innerList 标志位设置为 list，意为当前不在子列表中，然后进入 list()函数，传入参数 innerList 为布尔类型。

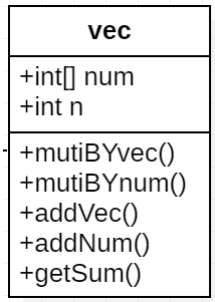
在 list 函数中，由于进入这个函数的可以是最外层的列表，也可能是列表中的子列表，所以需要标志位 innerList 作为传入的参数。

```
private void list(boolean innerList) {
            match(LookaheadLexer.LROUND);
             elements(innerList);
             match(LookaheadLexer.RROUND);
             innerList = false;
}
```

这样在进入 elements 函数中，就可以进行解析，如果是最外部的列表，可以直接++，否则不做处理。

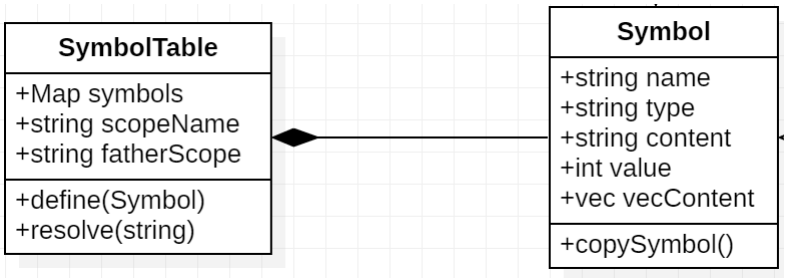
在进入 element 函数中，如果解析发现后面跟着的 Token 是”(”，说明接下来是内嵌的列表，只应该计算成一个元素，列表内的元素不应全加，所以设置 innerList 为 true 即可。

3.3 向量的类



由于现在的很多数据都以向量的形式进行存储和计算，所以内嵌了一个向量的数据类型、封装一些向量类型的方法很有必要。
新建的向量类型属性：以数组的形式存数值，以整形存向量的大小。
新建的向量类型的方法：获取向量所有元素相加之和，向量与向量相加、向量与向量相乘。向量与整数相加、向量与整数相乘。
向量与向量相乘：得到其内积，为一个整数
向量与整数相乘：向量中的每个元素都乘该整数
向量与向量相加：向量中对应元素相加
向量与整数相加：向量中的每个元素加上该整数
向量元素求和：对向量中的所有元素求和。

3.4 符号类和符号表



新建一个符号类属性：符号的名字，符号的类型（包括 int/string/list/vec/return/function)。符号的字符类型的 content，符号的整数值、符号的向量内容，以向量形式存储。

新建的符号类的方法：值得一提的是，新增了一个复制变量的方法，将参数中的符号中的所有属性赋值给本符号，实现变量之间的赋值。

符号表的类：其中有一个字段为一个字典，键为符号的名字，值为 Symbol 类型，即该符号。还保存有该符号表的 scopeName，以及其 fatherScope，这是找寻父作用域时起作用。

主要方法：define 和 resolve。define 函数即在该符号表中的字典里，将该符号进行定义，将其键值对保存在字典中，传入参数为 Symbol，存入其中的字典。

resolve 函数是在字典中对变量名解析，传入参数为变量名，返回该变量的符号类型。

3.5 函数的处理以及函数的子符号表

代码示例：

```
function getCredit(int);
var credit = getCredit(89);
```

在写函数体前，需要先获取传入函数的参数，否则需要将函数体的代码进行存储等获取参数后再做解析，比较麻烦。所以我们设计成先获取传入函数中的参数，再编写函数体。

但这样带来的问题是，无法区分某个 NAME 的 Token 是变量名还是函数，所以在使用函数之前，必须先对函数进行定义，让我们将function后的字符串识别为函数名称，然后读取第二行时，我们可在全局符号表中解析 getCredit 为函数名，所以后面跟着的括号后的可以解析为参数，等号左边的符号解析为 return 类型的符号，方便后续进行处理。

然后在全局符号表中定义 PARAM 的值为参数的值，方便函数体内部使用该变量。

```
sub getCredit{
    var temp = PARAM;
    var b = a;
    print(b);
    var credit = 0;
    if temp > 89:   credit = 50;
    if temp < 90:   credit = 50-(90-temp);
    return temp;
}
```

在函数体内，我们会在未识别到"}"符号时，设置一个标志位，告诉程序现在是在函数作用域内，首先使用函数作用域进行解析，如果在函数符号表中没有对该变量进行定义，则读取函数符号表的 fatherScope 字段，然后在 SymbolTable[ ]数组中遍历，找到其中 scopeName 与函数符号表的 fatherScope 字段匹配的那张符号表，再在那张符号表中解析，直至当前符号表的 scopeName 为 global，认为该符号未定义。

举例来说，在函数体内并未对 PARAM 符号进行定义，在函数符号表中解析发现未定义，会找寻符号表的 father Scope 字段，发现是 global，然后在符号表数组中遍历找寻 scopeName 字段为 global 的符号表，对 PARAM 进行解析，得到其符号。

对于 return 的解析：当得到 return 字段后面的符号 name 时，使用函数内符号表对 name 解析得到 symA。然后遍历 global 符号表，找寻其中变量类型为 return 的符号 symB，将 return 字段后的符号 symA copy 给 symB。就完成了函数的参数返回。