NumbLang: 一款轻量级编程语言

摘要—本文实现了 NumbLang 这一轻量级的编程语言，该语言具备过程式语言与函数式语言的一些特点，可以作为一个参考给想要学习编程语言原理但又不知从而入手的用户使用。NumbLang 具有支持函数式编程的特点，支持列表和字典等高级数据结构；同时，NumbLang 具有良好的可扩展性，可以很方便地对数据类型进行扩展。

关键字—PL，词法分析，语法分析，语义分析

I. 语言设计的背景

背景与意义

计算机程序设计语言是人的思维与计算机执行之间的桥梁，它可以把人们想要完成的工作交由计算机完成。计算机程序设计语言分编译型语言和解释性语言：编译型语言在程序执行之前，先会通过编译器对程序执行一个编译的过程，把程序转变成机器语言，运行时不需要翻译，而是直接执行，最典型的例子就是 C 语言；解释型语言就没有这个编译的过程，而是在程序运行的时候，通过解释器对程序逐行作出解释，然后直接运行，最典型的例子是 Python。编译型语言执行效率高，而解释型语言的优点是较为灵活。

根据 2021 年 12 月的 TOIBE 指数[1]，最流行的编程语言的前 5 名分别为：Python,C,Java,C++ 和 C#。近年来，由于人工智能等领域的快速发展，以及语言自身的特点，Python 变得非常火热。Python 最大的优点之一是具有伪代码的本质，它使程序设计者在开发 Python 程序时，专注的是解决问题，而不是搞明白语言本身。Python 采用 C 语言的样式进行开发，但是 Python 不再有 C 语言中的指针等复杂的数据类型存在。例如，同样一个程序，使用 C 语言可能需要 1000 行代码，使用 Java 语言需要 100 行代码，而使用 Python 则只需要 20 行代码。这也就是很多程序设计新手选择学习 Python 作为入门语言的原因。除此之外，Python 语言也没有那么复杂的逻辑，代码简洁规范，关键字也相对较少，说明文档还非常简单，极易上手。综上所述，Python 语言的成功有以下几点因素：

- 语法简单、极易入门
- 代码简洁
- 支持面向过程的函数式编程
- 支持面向对象编程
- 多平台配适、移植性强
- 扩展性强
- Python 库资源丰富

随着时间的推移和 Python 语言的发展，Python 的体量也在逐渐增大，对于一个初学者，如果想要学习其背后的语言机理，直接解读 Python 语言源码，可能难度较高。且 Python 源代码的耦合性较高，多种功能细分很详细，对于初学者很不友好。因此，本文意在设计一种轻量级编程语言 NumbLang，它具有部分 Python 的特点，它能够支持函数作为“一等公民”存在；又能够保证语言文法本身的简洁性，不至于十分庞大。具体而言，NumbLang 具有以下特性：

- 函数作为“一等公民”、高阶函数
- 语法简洁、入门简单
- 变量绑定
- 支持注释
- 闭包
- 柯里化
- 支持算数表达式
- 支持字符串、列表、字典
- 内置函数

NumbLang 中函数作为“一等公民”存在，它就像数据一样可以作为变量值、函数参数和返回值。由于函数可以作为返回值，因此 NumbLang 支持高阶函数，

[1] https://www.tiobe.com/tiobe-index/

也就是说函数可以通过多次调用并得到执行。使用方法见本章 NumbLang 使用示例一节。

NumbLang 支持常见的数据结构如：算数表达式、字符串、列表、哈希等。并且 NumbLang 具体良好的可扩展性，更多的数据类型可以在不用改变核心代码的基础上进行集成。

除此之外，NumbLang 中内置了两个函数 len 和 type，可以用来对 NumbLang 中的数据进行长度和类型的求解。NumbLang 具有很好地扩展性，更多的内置函数可以在将来进行添加。

这些特点将在 NumbLang 使用示例中进行介绍。另外，本语言被命名为 NumbLang，意为 “愚笨的语言”，是因为该语言尚处开发阶段，该语言还存在或多或少的不足，这些不足或将在将来进行完善。

NumbLang 使用示例

1) 变量的绑定: 使用 let 关键字进行绑定，以下是使用 let 进行变量绑定的例子。

let 使用示例

```
let age = 1;
let name = "Jack";
let result = 10 * (20/2 + 5);
```

除了通常的整数，布尔值，字符串外，像 Python 一样，NumbLang 还提供了列表（list）和字典（dict）的数据结构支持。

list 和 dict 使用示例

```
let myArray = [1,2,3,4,5];
let myDict = {"name":"Jack", "age":28};
```

具体值的获取通过下列方式：

list 和 dict 值的获取

```
myArray[0]; // -> 1
myDict["name"]; // -> Jack
```

2) 函数的使用: 在 NumbLang 中，使用 fn 关键字进行函数的定义，且函数可以如同数据一样被绑定：

函数的定义与使用

```
let add = fn(a,b){ return a + b; };
add(2,3); // -> 5
```

上面的例子定义了一个函数 add, 这个函数接收两个参数：a 和 b。在收到实参之后，函数将两个数进行相加之后返回。add(2,3) 是函数的使用，将参数 (2,3) 与形参进行结合，函数算出最终的结果为 5。

3) 高阶函数: 就像前面提到的一样，NumbLang 支持高阶函数。

高阶函数使用示例

```
//高阶函数
let exeTwice = fn(f, x){
    return f(f(x));
};
let addTwo = fn(x) { return x + 2;};
let mulThree = fn(x) { return x * 3;};

exeTwice(addTwo,3); // -> 7
exeTwice(mulThree,12); // -> 108
```

上述例子演示了高阶函数的使用。在上面这个例子中，首先定义了一个函数 exeTwice, 这个函数接收两个参数，其中第一个参数为一个函数。在函数体中，执行了接收的函数两次。在 exeTwice(addTwo,3) 这个示例中，addTwo 被执行两次，参数为 3，所以结果为 7；在 exeTwice(mulTree,12) 中，mulThree 被执行两次，参数 x 为 12，所以结果为 $12 \times 3 \times 3 = 108$.

4) 闭包: 闭包（Closure）是在支持头等函数的编程语言中实现词法绑定的一种技术。下面的例子用于展示闭包。

闭包示例

```
//闭包
let newAdder = fn(x) {
    return fn(y){x + y; };
};
let addTwo = newAdder(2);
x; // Error: x doesn't exit!
addTwo(3); // -> 5
```

在上述例子中，x 的值只在 newAdder 函数内可见，对于外层不可见，因此可以正常执行 addTwo(3)，而直接获取 x 的值则会发生错误。

5) 柯里化: 柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

高阶函数使用示例

```
//柯里化
let a = fn(f,x){return f(x);};
let f = fn(x){
    return fn(y){
        return fn(z){
            return x + y * z;
        };
    };
};
//第一种方式
a(f,2)(3)(8); // -> 26
//第二种方式
a(f(3)(8),2); // -> 19
```

上述示例演示了 NumbLang 中的柯里化结果。在第一种方式中，a(f,2) 将函数 f 中的 x 固定为 2，然后分别固定 y,z 的值分别为 3，8 并最终求值得到 $2+3\times 8=26$; 第二种方式中，首先固定 x 为 3，y 为 8，传入 a 中后固定 z 为 2，所以最终结果为 $3+8\times 2=19$。

有些语言(如 Scheme)使用 S-表达式(S-expression)这一半结构化地数据来进行解析，这样做的好处很明显：可以很方便地构造解析器。但是，对于很少接触 S-表达式地用户而言，使用 S-表达式可能会出现一些水土不服地状况。为了使用户更好更快地熟悉 NumbLang，本语言采用中缀表达式来进行数据的解析。

6) 内置函数的使用： NumbLang 目前具有两个内置函数：len() 和 type()，分别用来求解长度和类型。以下是使用示例：

内置函数使用示例

```
//内置函数测试
let a = 10+2;
let b = false;
let c = fn(x,y){ return x + y;};
let d = c(2,3);
let e = 1.23;
let f = [1,2,3,4,5];
let g = "hello,world";
let h = {"name":"Jack", "age":100};

len(a); // -> 12
len(b); // -> false
len(c); // -> 2
len(d); // -> 5
len(e); // -> 1.23
len(f); // -> 5
len(g); // -> 11
len(h); // -> 2
len(h["name"]); // -> 4

type(a); // -> integer
type(b); // -> boolean
type(c); // -> function
type(d); // -> integer
type(e); // -> float
type(f); // -> list
type(g); // -> string
type(h); // -> dict
type(h["name"]); // -> string
```

len 对于整形、浮点以及布尔值的求解为其原值，作用于函数则是求解函数的参数个数，作用于列表和字典则是求解内含元素个数。type 函数返回作用对象的类型。

以上是 NumbLang 语言的使用示例。本文接下来的安排如下：第二章对 NumbLang 的文法进行介绍，第三章对 NumbLang 的指称语义进行介绍，在第四章介绍 NumbLand 的具体实现，最后在第五章给出 NumbLang 的运行示例。

II. NumbLang 文法

本章对 NumbLang 的文法进行介绍。首先介绍 NumbLang 中的关键字以及操作符，之后介绍 NumbLang 的 BNF 文法。

1) 关键字与操作符： 在词法分析阶段，关键字与操作符被解析成一系列的 Token 流，这些 Token 流被送入语法解析器进行解析，得到抽象语法树（Abstract Syntax Tree，AST）。因此，对关键字与操作符的辨识是基础且重要的工作。NumbLang 中支持的关键字以及对应的含义和 Token 类型在表 I中进行了总结。NumbLang 中的操作符是一个广泛的概念，包括一元运算符，二元运算符以及界定符等，具体见表 II。NumbLang 具有很好的扩展性，可以在 Token 集合中简单地增加类型而不需要对现有实现进行很多的改动即可实现关键字和操作符的扩展。目前关键字和操作符的数量没有设计得十分庞大，是因为时间较为紧迫以及保持实现的简洁。

2) NumbLang 文法： NumbLang 语法简洁，并没有涉及太过复杂的语法。语法的简洁保证了实现的简

表 I: NumbLang 关键字

关键字	意义	Token 类型
fn	函数定义	FUNCTION
let	值束定	LET
true	真值	TRUE
false	假值	FALSE
if	if 语句	IF
else	else 语句	ELSE
return	返回语句	RETURN

表 II: NumbLang 操作符

+	−	*	/	(	)
>	<	{	}	==	! =
~	=	,	;	&&	‖

便，同时保证了 I章中提到的语法特点。NumbLang 的语法使用 BNF 进行语法描述。

首先是程序块，对总的程序块进行细分，可以细分成各种 Statement。

⟨*program*⟩ ::= ⟨*program-block*⟩

⟨*program-block*⟩ ::= ⟨*block*⟩

⟨*block*⟩ ::= ⟨*single-statment*⟩*

⟨*single-statement*⟩ ::= ⟨*let-statement*⟩
| ⟨*callFunction-statement*⟩
| ⟨*function-definition*⟩
| ⟨*expression-statement*⟩
| ⟨*return-statement*⟩
| ⟨*identifier*⟩
| ⟨*if-statement*⟩ ';'

在 NumbLang 中，向外共有以上几种 Statement(内部还另有辅助 Statement)。

Let-Statement 专门用来处理赋值语句。

⟨*let-statement*⟩ ::= 'let' ⟨*identifier*⟩ '='
⟨*callFunction-statement*⟩
| ⟨*function-definition*⟩
| ⟨*expression-statement*⟩
| ⟨*identifier*⟩

需要注意的是，在 NumbLang 中，被束定对象必须为变量，因为对非变量进行束定是没有意义的。

函数调用语句 callFunction-statement 的文法如下。

⟨*callFunction-statement*⟩ ::= ⟨*ientifier*⟩
| ⟨*function-definition*⟩ '('
⟨*parameters*⟩ ')'

⟨*parameters*⟩ ::= {⟨*expressoin-statement*⟩ |
callFunction}*

NumLang 支持定义函数之后立即调用。

接着是 if 语句的文法。

⟨*if-statement*⟩ ::= 'if' '{' ⟨*program-block*⟩ '}'
'else' '{' ⟨*program-block*⟩ '}'

函数定义与返回语句的文法如下。

⟨*function-definition*⟩ ::= 'fn' '(' ⟨*arguments*⟩
')' '{' ⟨*function-body*⟩ '}'

⟨*arguments*⟩ ::= ⟨*identifier*⟩
| ⟨*expression-statement*⟩
| ⟨*callFunction-statement*⟩,
{ ',' ⟨*identifier*⟩
| ⟨*expression-statement*⟩
| ⟨*callFunction-statement*⟩
| ⟨*expression-statement*⟩
} *

⟨*functoin-body*⟩ ::= ⟨*program-block*⟩

⟨*return-statement*⟩ ::= 'return'
⟨*single-statement*⟩

表达式的处理是语法分析中很重要的一环。为了区分不同操作符的优先级，NumbLang 表达式的文法如下。

⟨*expression-statement*⟩ ::= ⟨*compare*⟩ '==' ⟨*expression-statement*⟩
| ⟨*compare*⟩ '!=' ⟨*expression-statement*⟩
| ⟨*compare*⟩

⟨*compare*⟩ ::= ⟨*arithmetic*⟩ '>' ⟨*compare*⟩
| ⟨*arithmetic*⟩ '⟨' <*compare*⟩
| ⟨*arithmetic*⟩

⟨*arithmetic*⟩ ::= ⟨*p-arithmetic*⟩ '+' ⟨*arithmetic*⟩
| ⟨*p-arithmetic*⟩ '-' ⟨*arithmetic*⟩
| ⟨*p-arithmetic*⟩

⟨*p-arithmetic*⟩ ::= ⟨*basic-type*⟩ '*' ⟨*p-arithmetic*⟩
| ⟨*basic*⟩ '/' ⟨*p-arithmetic*⟩
| ⟨*basic*⟩

在 NumbLang 中，整数、浮点数等作为原子存在而不进行更细的定义，对它们的解析在词法分析阶段即完成。

最后是基本数据和复杂数据结的定义。

⟨*basic*⟩ ::= ⟨*number*⟩
| ⟨*list*⟩
| ⟨*dict*⟩
| ⟨*identifier*⟩
| ⟨*callFunction-statement*⟩

⟨*number*⟩ ::= ⟨*interger*⟩
| ⟨*float*⟩
| ⟨*true*⟩
| ⟨*false*⟩

⟨*list*⟩ ::= '[' ⟨*expression-statement*⟩* ']'

⟨*dict*⟩ ::= { ⟨*identifier*⟩ ':' ⟨*expression-statement*⟩ } *

以上就是 NumbLang 的所有文法。有了文法之后，我们需要对文法的语义进行说明。接下来的一章对 NumbLang 的指称语义进行详细说明。

III. NumbLang 指称语义

本章对 NumbLang 的指称语义进行说明。NumbLang 的语法分析利用的是自顶向下语法分析的方法，因此，指称语义自顶向下给出。本章依次给出指称语义域、语义函数以及辅助函数。NumbLang 的指称语义实现借鉴了部分 [2]。

语义域

NumbLang 的语义域如下。

```
1:  Value = True
2:  + Integer
3:  + Float
4:  + String
5:  + List
6:  + Dict
7:  + Function
8:
9:  Statement =
10: Let_Statement
11: + If_Statement
12: + Block_Statement
13: + Call_Statement
14: + Dict_Statement
15: + Expression_Statement
16: + Function_Statement
17: + Get_Element_Statement
18: + Identifier_Statement
19: + List_Statement
20: + Return_Statement
21: + String_Statement
22:
23: Interval_Types =
24: Boolean_Internal
25: + Dict_Internal
26: + Function_Internal
27: + Integer_Internal
28: + List_Internal
29: + Return_Internal
30: + String_Internal
31: + Null_Internal
32:
33: Bindable = Internal_Types
34:
35: Environment = Identifier→Bindable
36:
37: Opr = ASSIGN + ADD + SUB
38: + MUL + DIV + GT + GE
39: + LT + LE + EQU + NEQU
40: + OPPOSITE + OR + AND
41: + COMMA + SEMICOLON
```

```
    + COLONS + LPAREN + RPAREN
    + LSBRACE + RSBRACE
    + LBRACE + RBRACE
    + COMMENT + EOF

    KeyWords = ILLEGAL + FUNCTION
    + LET + TRUE + FALSE
    + IF + ELSE + RETURN
    + NUMBER + FLOAT + IDEN
    + STRING + LIST + DICT
    + CALLFUNC

    Token = opr Opr + keyword KeyWords
```

Environment 的辅助函数:

```
empty_env:Environment

set :
Identifier × Bindable
→ Environment

map_to_outer:
Environment × Environment
→ Environment

get:
Identifier × Environment
→ Bindable
```

约定: lambda 代表的是数学符号 λ,perp 代表的是数学符号 $\perp$,epsilon 代表的是 ϵ。那么这些函数定义如下:

```
empty_env = lambda I.epsilon
set(I,Bind) =
  I.I=Bind in env(I)
map_to_outer(env,env' )
 = map(env,env')
get(env,I) =
    let get_value(env.contains(I))
        = env(I)
       get_value(env' .contains(I))
        = env' (I)
       get_value(epsilon) = perp
    in env(I)
```

其中 env(I) 表示在环境 env 范围内的标识符,get_value 为辅助求 I 值的函数。对 Statement 求值的函数:

```
evaluate: Statement × Environment
→ (Environment → Environment)
```

为了细分 Statement 的定义, 分别对不同类别的 Statement 增加不同的求值辅助函数。对 If-statement 就行求值的函数:

```
evalute_if_statement:
If_Statement × Environment
→(Environment→Environment)

evaluate_if_statement =
execute[
if (condition)
    {Bolock_Statement}
else
    {Block_Statement}
] env

execute[
if (condition)
    {If_Bock_Statement}
else
    {Else_Block_Statement}
] env
=
if evaluate[condition] env
then
evaluate[If_Block_Statement] env
else
evaluate[Else_Block_Statement] env
```

对 Let-Statement 求值的函数:

```
evaluate_let_statement =
execute[let identifier = statement] env

execute[let identifier = statement] env
=set(identifier,evaluate[statement]) env
```

对 Block-Statement 的求值:

```
evaluate_block_statement[block_statement]
= execute[block_statement]

execute[block_statement] env =
    if epsilon then perp
    else
    evaluate(head(block_statement))
            evaluate(
            block_statement -
```

```
10:                 head(block_statement)
11:                 )
```

其中 head(block_statement) 表示取语句块中的第一条语句。对 String 和 Return 语句的求值函数：

```
1:     evaluate[string_statement] env
2:     = execute[string_statement]
3:
4:     execute[string_statement] env
5:     = String_Internal(string_statement)
6:     env
7:
8:     evalute[return statement] env
9:     = execute[return statement] env
10:
11:    execute[return statement] env
12:    = evaluate[statement] env
```

之后的函数作用类似，不再一一说明：

```
1:     evaluate[list_statement] env
2:     = execute[list_statement] env
3:
4:     execute[list_statement] env =
5:     if epsilon then perp
6:     else
7:         evaluate(head(list_statement))
8:         evaluate(
9:         list_statement -
10:        head(list_statement)
11:        )
12:
13:    evaluate[function_statement] env
14:    = execute[function_statement] env
15:
16:    execute[fuction_statement] env =
17:    Function_Internal(
18:    get_para[function_statement],
19:    get_body[function_statement]
20:    )
21:
22:    get_para[function_statement] env
23:    = recurve_para[
24:    function_statement.parameters
25:    ] env
26:
27:    recurve_para[paras] =
28:    if epsilon then perp
29:    else
30:        head(paras) +
31:        recurve[paras - head(paras)]
32:
33:    get_body[function_statement] env
34:    = recurve_body[
35:    function_statement.body
36:    ] env
37:
38:    recurve_body[body] =
39:        if epsilon then perp
40:        else
41:            curStmt +
42:            body - head(body)
43:
44:    evaluate[dict_statement] env
45:    = execute[dict_statement] env
46:
47:    execute[dict_statement] env
48:    = bind_process(
49:    dict_statement.key,
50:    dict_statement.value
51:    ) env
52:
53:    bind_process(key,value) env
54:    = set(
55:    evaluate[key],
56:    evaluate[value]
57:    ) env
58:
59:    evaluate[call_statement] env
60:    = execute[call_statement] env
61:
62:    execute[call_statement] env
63:    = map_arguments(
64:    call_statement.func,
65:    call_statement.args) env
66:
67:    map_arguments(func,args) env
68:    = if epsilon then perp
69:    else
70:        map_to_outer(env,env' )
71:        set(
72:        head(func.paras),
73:        head(args)) env'
74:        map_arguments(
75:        func.paras-head(func.paras),
```

```
76:         args-head(args)) env
77:         evaluate(func.body) env'
```

注意，在这里将新的环境设置为 env'，将参数映射之后执行函数主体，这样可以保证闭包这一性质。

```
1:     evaluate[expression_statement] env
2:     = execute[expression_statement] env
3:
4:     execute[expression_statement]
5:     = oprLeftRight(
6:     expression_statement.left,
7:     expression_statement.right,
8:     expression_statement.opr)
9:
10:    oprLeftRight(left,right,opr) env =
11:    lambda T.
12:    if T = Boolean
13:    then revalBoolean(left,right,opr)
14:    env
15:    else if T = Integer
16:    then revalInteger(left,right,opr)
17:    env
18:    else if T = Float
19:    then revalInteger(left,right,opr)
20:    env
21:    else if T = String
22:    then revalString(left,right,opr)
23:    env
24:    else if T = List then
25:    revalList(left,right,opr)
26:    env
27:    else perp
28:
29:    revalBoolean(left,right,opr) env
30:    =
31:    if opr = '||'
32:    then Or(
33:    evaluate[left],
34:    evaluate[right]
35:    ) env
36:    else if opr = '&&'
37:    then And(
38:    evaluate[left],
39:    evaluate[right]
40:    ) env
41:    else if opr = '=='
42:    then Equ(
43:    evaluate[left],
44:    evaluate[right]
45:    ) env
46:    else if opr = '!='
47:    then Neq(
48:    evaluate[left],
49:    evaluate[right]
50:    ) env
51:    else perp
52:
53:    Or:
54:    Boolean_Internal ×
55:    Boolean_Internal
56:    → Boolean_Internal
57:
58:    And:
59:    Boolean_Internal ×
60:    Boolean_Internal
61:    → Boolean_Internal
62:
63:    Equ:Boolean_Internal ×
64:    Boolean_Internal
65:    → Boolean_Internal
66:
67:    Neq:
68:    Boolean_Internal ×
69:    Boolean_Internal
70:    → Boolean_Internal
```

上述辅助函数分别用来进行布尔变量之间的运算，比较简单，在此不再赘述。

```
1:     revalInteger(left,right,opr)
2:     env
3:     =
4:         if opr = '+'
5:         then Add(
6:         evaluate[left],
7:         evaluate[right]
8:         ) env
9:         else if opr = '- '
10:        then Sub(
11:        evaluate[left],
12:        evaluate[right]
13:        ) env
14:        else if opr = '*'
15:        then Mul(
16:        evaluate[left],
```

```
        evaluate[right]
        ) env
        else if opr = '/'
        then Div(
        evaluate[left],
        evaluate[right]
        ) env
        else if opr = '=='
        then IEqu(
        evaluate[left],
        evaluate[right]
        ) env
        else if opr = '!='
        then INeq(
        evaluate[left],
        evaluate[right]
        ) env
        else perp
```

其中：

```
Add:
Integer_Internal ×
Integer_Internal
→ Integer_Internal

Sub: Integer_Internal ×
Integer_Internal
→ Integer_Internal

Mul:Integer_Internal ×
Integer_Internal
→ Integer_Internal

Div: Integer_Internal ×
Integer_Internal
→ Integer_Internal

IEqu:Integer_Internal ×
Integer_Internal
→ Boolean_Internal

INeq:Integer_Internal ×
Integer_Internal
→ Boolean_Internal
```

Float 与 Integer 类似：

```
revalFloat(left,right,opr)
env
=
    let right_value
    = evaluate[right]
    let left_value
    = evaluate[left]
    operate(opr,
    left_value,
    right_value
    ) env
```

其中 operate() 函数根据操作符进行操作，与 revalInteger() 相似，不再赘述。

```
revalList(left,right,opr)
env
= if opr == '+'
then addList(
left,
right
)
    else perp
```

其中 addList 辅助函数将一个列表拼接到另一个列表之后。目前只支持列表的’+’运算，后续可对列表支持的运算进行扩展。

```
revalString(left,right,opr)
env
=if opr = '+'
then addString(left,
right)
    else perp
```

为了得到每个 Statement, 在语义域上定义函数：

```
parse_statement:
Token × Token*
→ Statement
```

同样为了对 Statement 进行细分，增加以下辅助函数：

```
parse_let_statement:
Token × Token *
→ Let_Statement

parse_return_statement:
Token × Token*
→ Return_Statement
```

```
    parse_if_statement:
    Token × Token*
    → If_Statement

    parse_identifier:
    Token × Token*
    → Identifier_Statement

    parse_block_statement:
    Token × Token*
    → Block_Statement

    parse_function_statement:
    Token × Token*
    → Function_Statement

    parse_expression_statement:
    Token × Token*
    → Expression_Statement

    parse_list_statement:
    Token × Token*
    → List_Statement

    parse_dict_statement:
    Token × Token*
    → Dict_Statement
```

现在分别进行定义。

```
    parse_let_statement
    [curToken]
    =
    nextToken()
        let iden
        =parse_identifier
        [curToken]
        let stmt =
        parse_expression_statement
        [curToken]
        Let_Statement(iden,stmt)
```

其中 nextToken() 表示取下一个 Token，此时，curToken 的值跟着改变，下同。

```
    parse_return_statement
    [curToken]
    =
    nextToken()
        let returnVal
        =parse_expression_statement
        [curToken]
        Return_Statement(returnVal)

parse_if_statement
[curToken]
=
    nextToken()
    let condiction
    =parse_expression_statement
    [curToken]
    let if_block_statement
    = parse_block_statement
    [cutToken]
    let else_block_statement
    = if curToken = 'else'
    then parse_block_statement
    [curToken]
    else perp
    If_Statement(condition,
    if_block_statement,
    else_block_statement
    )

parse_identifier
[curToekn]
=
    let iden
    = curToken
    nextToken()
    Identifier_Statement(
    iden
    )

parse_block_statement
[curToken]
=
    if epsilon then
    perp
    else
let stmt
= parse_statement
[cutToken]
nextToken()
    Block_Statement(stmt +
    parse_block_statement
```

```
        [curToken]
        )

    parse_function_statement
    [curToken]
    =
        let args
        = parse_argument
        [curToken]
        nextToken()
        let body
        = parse_block_statement
        [curToken]
        Function_Statement(args,
        body
        )
```

其中的辅助函数 parse_argument 用来求解函数的参数。

```
    parse_argument:
    Token × Token*
    → Statement *
```

求取 Expressiono-Statement:

```
    parse_expression_statement
    [curToken]
    =
        let left
        =
        parse_compare
        [curToken]
        let opr = lambda T.
        if T=' =='
        then EQU
        else if T=' !='
        then NEQ
        else perp
        nextToken()
        let right
        = parse_epression_statement
        [curToken]
        Expression_Statement(left,
        opr,
        right)

    parse_compare[curToken]
    =
        let left
        = parse_arithmetic
        [curToken]
        let opr = lambda T.
        if T =' >'  then
        GT
        else if T =' <'
        then
        LT
        else perp
        nextToken()
        let right
        = parse_comparse
        [curToken]
        Expression_Statement(left,
        opr,
        right
        )

    parse_arithmetic[curToken]
    =
        let left
        = parse_priority_arithmetic
        [curToken]
        left opr = lambda T.if
        T=' +'  then ADD else
        if T=' - ' then SUB
        else perp
        nextToken()
        let right
        = parse_arithmetic
        [curToken]
        Expression_Statement(left,
        opr,
        right
        )

    parse_pritority_arithmetic
    [curToken]
    =
        let left
        = parse_meata
        [curToken]
        let opr
        =  T.if T=' *'  then
        MUL else if T=' /'  then
        DIV else if  '~'  the
```

```
        n OPPSITE else then
        perp
        nextToken()
        let right
        = parse_pritority_arithmetic
        [curToken]
        Expression_Statement(left,
        opr,
        right)
```

其中 parse_meta 为识别基本数据类型的辅助函数。

```
    parse_list_statement
    [curToken]
    =
        if epsilon then perp
        else
        let stmt
        = parse_statement
        [curToken]
        nextToken()
        List_Statement(
        stmt +
        parse_list_statement
        [curToken])

    parse_dict_statement
    [curToken]
    =
    if epsilon then perp
    else
        let key
        = parse_string
        [curToken]
        nextToken()
        let value
        = parse_statement
        [curToken]
        nextToken()
        Dict_Statement(<key,value> +
        parse_dict_statement
        [curToken]
        )
```

其中 parse_string 表示的是解析当前的字符串。本文将 Identifier,Interger 等视为原子，不再进一步进行定义。

```
    nextToken: String → Token
```

为了对不同 Token 进行细分，定义以下辅助函数：

```
    readNumber: String → Token
    readOpr: String → Token
    readKeyWord : String → Token
    readIden: String → Token
```

以上几个函数分别解析出不同的 Token。

IV. NumbLang 的实现

在第 II章和第 III章中，我们对 NumbLang 的文法以及指称语义进行了说明。在本章中，我们将会对实现 NumbLang 的关键部分进行介绍。

词法分析

词法解析是实现 NumbLang 的第一步，在这一步中，词法分析器将输入的源文件解析成 Token 流，交给语法分析器进行语法分析。下面将对词法分析过程涉及到的几个自动机进行说明。

1）整数的有限状态自动机。NumbLang 支持整型变量，在词法分析阶段对整型变量进行识别并求值。图 1是整型变量识别并求值的自动机。其中，Action 1-1 的

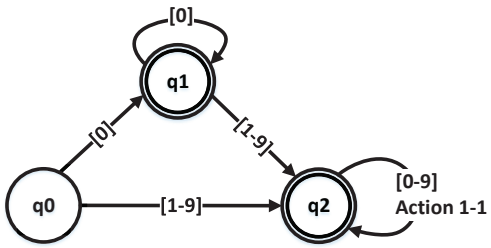

图 1: 识别整数

语义操作为对输入数据乘以 10，并加上当前字符。

2）浮点数的有限状态自动机。NumbLang 支持浮点类型，图 2是识别浮点数并求值的状态机。其中，Action 2-1 的操作为对输入数据乘以 10，并加上当前字符；Action 2-2 的操作为设置一个计数器 n（初值为 1，每次循环自增 1），每次输入值乘以 10^{-n}，并加上之前的值。

3）注释的有限状态自动机。NumbLang 支持行注释和块注释两种注释。行注释只允许单行注释，而块注释允许多行注释。图 3是注释识别的自动机。

4）字符串的有限状态自动机。字符串识别的有限状态自动机见图 4。

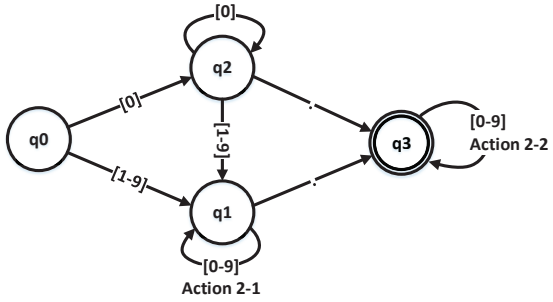

图 2: 识别浮点数

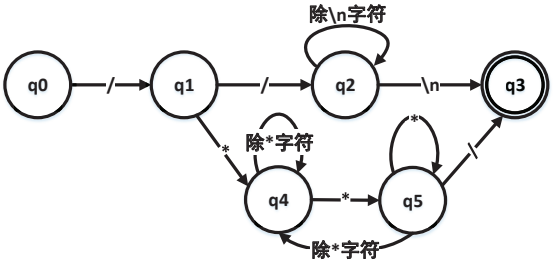

图 3: 识别注释

5）关键字的有限状态自动机。关键字的有限状态自动机见图 5。其中，Action 4-1 的操作为判断是否为关键字。

语法分析

语法分析过程中最重要也是最复杂的部分是抽象语法树（Abstract Syntax Tree，AST）的建立。而 AST 的建立与算数表达式的解析过程密切相关，因此，本文重点对算数表达式解析的过程给出说明。[1] 提出了自定向下的优先符算法，可以用来解析具有不同优先级的算符。但是本文为了实现上的方便，不打算使用 [1] 中提出的方法，但是可以借鉴其实现思路。本文采用递归的、自顶向下的方法实现表达式的解析。具体流程见算法 1。算法 1自定向下递归地生成 AST。算法首先将上

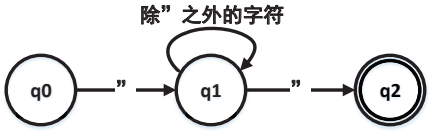

图 4: 识别字符串-1

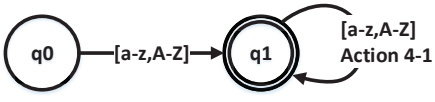

图 5: 识别字符串-2

Algorithm 1 语法树生成算法核心

Require: TreeRoot root, Token curToken

```
1:  IF satisfied(curToken)
2:      root.val ← curToken.val
3:      ▷ Make a tree called tmp
4:      tmp.left ← root
5:      ▷ Recursively build tmp's right
6:      IF tmp.right == NULL
7:          RETURN root
8:      ELSE
9:          RETURN tmp
10: ELSE
11:     RETURN NULL
```

Ensure: The root of the AST

层根节点值进行固定 (第 2 行)，接着申请一个临时的树节点，并将根节点设置作为其左子树 (3-4 行)。接着递归地建立临时树的右子树，最终如果发现右子树为空，那么返回原来的根节点，否则返回临时树的根节点 (5-9 行)。语法解析中还有其余解析模块，但是算法较为简单，具体见 NumbLang 的实现 [3]。

Statement 求值

正如第 III章指称语义所说的那样，对 Statement 求值的函数为 evaluate。由于 Statement 的多样性，所以我们对 evaluate 进行了细分。其中较为复杂的部分是对函数调用的处理，因为我们要满足柯里化。因此，本节对函数调用部分的实现进行说明，其余部分的实现细节见 [3]。

函数调用可能涉及连续的函数调用，因此在语法解析的时候，将函数的实参用一个二维的列表进行存储，供求值时进行使用。函数调用的处理算法见算法 2。在算法 2中，首先根据当前环境获取实参 (第 1 行)，之后将实参与形参映射到一个新的环境 local 中，local 环境的外层环境时 env(3-4 行)，之后执行函数体 (5 行)。如果参数执行完毕，那么返回执行的结果，否则递归地进

Algorithm 2 函数调用处理算法

Require: FunctionInternal func, Arguments[n][m] args, Environment env

```
1:  paras gets getParameters(args.get(0),env) //根据环境 env 获取实参
2:  ▷ Remove head of the args
3:  local ← getNewEnvironment(env) //获取新的环境
4:  ▷ Map <func's arguments, paras> to local
5:  res ← execute(func.body,local)
6:  IF isEmpty(args)
7:      ▷ Restore args
8:      RETURN res
9:  ELSE
10:     ans ← Do this again
11:     ▷ Restore args
12:     RETURN ans
```

行操作，并最终返回结果 (6-12 行)。需要注意的是，在函数返回前，需要将移除的参数重置 (7,11 行)，否则会发生函数不可重入的情况。

NumbLang 使用 Java 进行实现，本章只对其实现的核心关键步骤进行了说明，具体实现可以参见 [3]。

V. NumbLang 运行示例

本章将会对 NumbLang 的词法分析、语法分析和语义分析（求值）这三个过程进行展示。

词法分析

词法分析的输入为：

词法分析输入

```
1: //测试词法输入
2: let a = 5.12;
3: let b = fn(x){
4:   return x * x;
5: };
6: let c = [1,2,4*5,true,false,3==2];
7: let d = {"name":"Bob","age":25};
```

词法分析的输出如下：

词法分析输出

```
1: COMMENT COMMENT
2: let LET
3: a IDEN
4: = ASSIGN
5: 5.12 FLOAT
6: ; SEMICOLON
7: let LET
8: b IDEN
9: = ASSIGN
10: fn FUNCTOIN
11: ( LPAREN
12: x IDEN
13: ) RPAREN
14: { LBRACE
15: return RETURN
16: x IDEN
17: * MUL
18: x IDEN
19: ; SEMICOLON
20: } RBRACE
21: ; SEMICOLON
```

为了节省篇幅，这里仅展示部分结果。可以看到，词法分析将我们输入的源文件解析成了一个 Token 流，这个 Token 流包含了每个单词的内容以及类别。

语法分析

在语法分析阶段，语法分析器会接收词法分析器提供的 Token 单词，根据第 II章提供的语法进行语法解析。输入下列源文件来测试语法分析结果：

语法分析输入

```
1: //定义加法函数
2: let add = fn(x,y){
3:     return x + y;
4: };
5: add(2,3+4);
6: let b = if (3 > 4){
7:     15;
8: } else
9: {
10:    16;
11: };
```

为了节省篇幅，这里只展示 add(2,3+4) 的解析结果。解析结果见图 6。

语法解析的结果是 add(2,3+4) 是一个函数调用 (Function Invocation)，并且函数名是一个名为 add 的

```
Function Invocation
  function:
    Identifier
      name:
        add
    Layer1
      Argument1
        Experssion
          value:
            2
      Argument2
        Experssion
          Operatioin:
            ADD
          Left:
            Experssion
              value:
                3
          Right:
            Experssion
              value:
                4
```

图 6: add(2,3+4) 解析结果

标识符；该函数共有两个实参，参数 1 是 2，参数 2 是一个复合表达式 3+4。可以看到，语法解析器对 add(2,3+4) 的解析结果是符合期望的。

语义分析 (求值)

语义分析部分对语法解析得到的 Statement 进行求值，具体的过程在第 III章中已做详尽说明。输入下列源文件进行测试：

语义分析输入

```
1: //语义分析输入
2: let newAdder = fn(x){
3:     return fn(y){
4:         return x + y;
5:     };
6: };
7: let addTwo = newAdder(2);
8: let list = [1,4/2,addTwo];
9: let dict = {"name":"Tom", "age":25};
10: let a = list[2];
11: let b = dict["age"];
12: a;
13: a(b);
```

执行结果见图 7。可以看到，语义分析首先识别出 newAdder 和 addTwo 是两个函数。随后计算出列表

```
A function.
A function.
[ 1 , 2, A function.]
{
  key:
    name
  value:
    Tom
  key:
    age
  value:
    25
}

A function.
25
A function.
27
```

图 7: 语义分析输出

list 和字典 dict 的值。最后将 addTwo 对应的函数与 a 绑定，并且传入 b=25 进行计算，得到最终结果 27。

以上对 NumbLang 的词法分析、语法分析和语义分析进行了演示，更多测试可以通过 [3] 获得。

VI. 总结与展望

NumbLang 的文法简单，可以为想要深入学习程序设计语言原理的学者提供一个设计的参考。NumbLang 支持常见的数据类型，如整形，浮点，布尔，字典等，同时 NumbLang 支持函数式编程。但是由于时间的紧迫性，NumbLang 同样存在诸多不足，这些不足表现在：

- 内置函数较少
- 没有垃圾回收机制
- 错误提示不够人性化

尽管 NumbLang 有 len 和 type 两个内置函数，但想要在实际生产生活中应用 NumbLang，这是远远不够的。因此，NumbLang 内置函数的扩充是今后努力的一个方向。同时，NumbLang 是基于 Java 开发的，NumbLang 现阶段的垃圾回收依赖于 Java 的垃圾回收机制，因此缺少本身的垃圾回收机制。另外，在发生语法解析错误时，NumbLang 给出的提示尽管能在一定程度上找到问题发生的原因，但仍然不够人性化，因此可以逐步完善 NumbLang 的错误提示。

参考文献

[1] Vaughan R. Pratt. 1973. Top down operator precedence. In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on

Principles of programming languages (POPL '73). Association for Computing Machinery, New York, NY, USA, 41–51.

[2] 麦中凡, & 吕卫锋. (2011). 程序设计语言原理 (2nd ed.). 北京: 北京航空航天大学出版社.

[3] https://gitee.com/salvete/numb-lang.git