**符号元语言Alpha草案报告**

罗俊逸，胡常齐

*致力于设计符号元语言的优秀方案*

**2017年xx月xx日星期六**

**摘要**

此报告给出了符号元语言Alpha的定义描述。Alpha是一个基于静态作用域的Lisp方言。它的设计遵循简约之法则，提供最简洁的语法和具有最强派生能力的原生功能。

背景部分阐述了Alpha语言和本报告产生的背景和动机。

第一章简略地描述了Alpha语言地语义和语法。

第二章详细介绍了Alpha语法层面的符号约定。

第三章和第四章详细介绍了Alpha的对象和原生过程。

附录部分提供了典型的Alpha语言示例。

**内容**

[背景 3](#_Toc475268284)

[1. Alpha语言概述 3](#_Toc475268285)

[1.1. 语义 3](#_Toc475268286)

[1.2. 语法 4](#_Toc475268287)

[2. 符号约定 4](#_Toc475268288)

[2.1. 标识符 4](#_Toc475268289)

[2.2. 保留常量 5](#_Toc475268290)

[2.3. 字符和字符串 5](#_Toc475268291)

[2.4. 普通数值 5](#_Toc475268292)

[2.5. 广义点对和空值 6](#_Toc475268293)

[2.6. 空白和注释 6](#_Toc475268294)

[3. 原生对象 6](#_Toc475268295)

[3.1. 类型 6](#_Toc475268296)

[3.2. 空值 7](#_Toc475268297)

[3.3. 布尔值 7](#_Toc475268298)

[3.4. 字节 7](#_Toc475268299)

[3.5. 数值 7](#_Toc475268300)

[3.6. 字符 7](#_Toc475268301)

[3.7. 点对 8](#_Toc475268302)

[3.8. 字节向量 8](#_Toc475268303)

[3.9. 向量 8](#_Toc475268304)

[3.10. 字符串 8](#_Toc475268305)

[3.11. 标识符 9](#_Toc475268306)

[3.12. 过程 9](#_Toc475268307)

[3.13. 作用域 10](#_Toc475268308)

[4. 原生过程 10](#_Toc475268309)

[4.1. 变量 10](#_Toc475268310)

[4.2. 数据和运算 11](#_Toc475268311)

[4.3. 复合结构 13](#_Toc475268312)

[4.4. 控制台和文件流 14](#_Toc475268313)

[4.5. 流程控制 14](#_Toc475268314)

[4.6. 动态调用和执行 15](#_Toc475268315)

[4.7. 系统扩展 15](#_Toc475268316)

[Alpha语言示例 15](#_Toc475268317)

[参考文献 15](#_Toc475268318)

# 背景

截止20世纪50年代，计算机程序语言经历了机器指令码序列、汇编符号序列、Fortran语言的这条发展路线，其核心思想是“程序服务于机器”。此领域中有学者认为“符号演算系统可以派生出智能”。于是，人们对一种脱离机器的原生冗余，以数学、逻辑、符号、形式为本的“机器服务于程序”的语言系统（本报告称之为“元语言符号系统”或“符号元语言”）的需求逐渐变得迫切。

1958年，人工智能之父John McCarthy基于S-表达式和Lambda演算创造了Lisp语言。其名称源自列表处理器（List Processor）的缩写。LISP有很多种方言，各个实现中的语言不完全一样。LISP语言的主要现代版本包括[Common Lisp](https://zh.wikipedia.org/wiki/Common_Lisp), [Scheme](https://zh.wikipedia.org/wiki/Scheme)，[Racket](https://zh.wikipedia.org/wiki/Racket)以及[Clojure](https://zh.wikipedia.org/wiki/Clojure)。1980年代[盖伊·史提尔二世](https://zh.wikipedia.org/wiki/%E8%93%8B%E4%BC%8A%C2%B7%E5%8F%B2%E6%8F%90%E7%88%BE%E4%BA%8C%E4%B8%96)编写了Common Lisp试图进行标准化，这个标准被大多数解释器和编译器所接受。还有一种是编辑器[Emacs](https://zh.wikipedia.org/wiki/Emacs)所派生出来的[Emacs Lisp](https://zh.wikipedia.org/wiki/Emacs_Lisp)非常流行，并创建了自己的标准。

Lisp有两大特点：

* S-表达式是程序和数据内在结构的直接体现。本报告称Lisp为“符号元语言”，其“元原理”性的主体就是S-表达式。
* 基于S-表达式的简洁和灵活，Lisp异常强大，强大到可以“自行定义符号语法”。

然而S-表达式层层括号的友好性可谓仁者见仁，也遭到许多“自然语言流派”的批评。Lisp的强大和灵活也让它的标准化难以控制。Lisp与人类的矛盾等价于“灵活性”和“标准化”的矛盾，等价于“解耦”和“封装”的矛盾。

笔者认为，对于满足某种完备性的符号计算系统，解耦和封装是等价的，但它们之间相互推演的复杂性有天壤之别。从解耦的“元子”封装成“模块”是一个简单而自然的过程，可谓是“构造”，但用模块去实现元子的功能却相当复杂，可谓是“重构”。例如用1比1的单位方形拼成m比n的矩形很简单，而用m比n的矩形拼成成1比1的方形就相当复杂甚至无解。

S-表达式是Lisp的形式基础和基本“语法糖”，本报告不尝试对S-表达式进行改进。

本报告的目的是推进Lisp的标准化进程，但不是像Common Lisp那样直接限定符号和封装功能来构成一部Lisp方言。笔者认为，符号元语言的原生功能应当尽可能简单而必要，而“标准”实质上是一系列的词汇与功能的映射，应该由原生功能或更基础的标准迭代地“构造”出来。这种标准化的过程很自然，不以牺牲Lisp的灵活和强大为代价去“限制”出一套只能通过“重构”来实现某些边缘需求的标准。

基于此目的，笔者以“奥卡姆剃刀”为指导思想，以尽可能地解耦和降冗为目标，设计了Lisp的方言——符号元语言Alpha。

**描述**

# Alpha语言概述

## 语义

Alpha不是一门具有具体标识符约定的语言，更不是一个具体实现，而是一个Lisp的特化标准。本节给出了Alpha语义的概述。详细的交叉内容请见后面章节。

Alpha是一门基于静态作用域的程序语言。作用域呈树状嵌套结构，变量的定义、修改、访问遵循闭包原则。

Alpha是一门动态类型语言。类型被绑定于值（对象）而非变量或表达式。

Alpha中所有对象都具有“第一类”性质，包括标识符、过程、作用域、类型。所有对象都能够被动态创建，都能够被赋予变量和被变量擦除，都能够被复合结构存储和丢弃，都能够作为过程的参数和返回值，等等。

基于这一特性，Alpha中所有对象的引用原理是一致的，整个Alpha系统是一个由“对象”和“引用”构成的有向图，一个对象的生命周期一直持续到其引用计数减少为0。因此，为了避免内存泄漏，用户需手动拆开显式或隐式的循环引用。这种机制在实现上也称作“半自动垃圾回收”。

Alpha的一个重要特性：过程不同于传统意义上的函数或宏，而是他们的综合或解耦。过程具有“第一类”性，一个变量是不是过程在执行时才知道。形式上调用一个构造的过程时，参数部分的语法表达式（实参的形式结构）对象会原封不动地传递，此时该过程类似于语法宏；系统内置逻辑或者动态机制（初存于apply的原生过程）调用一个构造的回调过程时，参数部分由调用时的逻辑指定或任意指定，此时该过程类似于函数。

另一个重要特性：作用域具有“第一类”性。自定义过程的每次调用都会产生一个新的作用域，这是闭包的基础，在此不详述。动态执行时可以自由地设定作用域作为“执行场景”。除此之外，作用域可以被显式地获取和构造，而且是“闭包安全”的：仅允许获过程的调用场景和基于某一指定作用域显式构造其子域。这一特性让Alpha可以较支持“续延”的语言（如Scheme）实现更灵活的标识符权限控制，同时保证安全，避免混乱。

还有一个重要特性：类型具有“第一类”性。所有对象的类型同样是对象，每个对象都引用着自己的类型，每个类型都引用着自己的构造过程，用于显式构造该类型的对象。

Alpha的数值系统是基于浮点数的复数系统。在Alpha中，自然数即整数，整数即有理数，有理数即实数，实数即复数。所以，用户不必关心数值的存储，可以全身心研究数值计算。

Alpha支持字节处理和位运算。

## 语法

和大多数Lisp方言一样，Alpha的语法对象集是其数据对象集的真子集。语法形式遵循S-表达式——语法对象包括元子自运算对象和括号化的广义表，而广义表实为“右嵌套点对链”。一段程序代码解析后必须是一个语法对象，执行一段代码就是执行该代码解析后的语法对象。

Alpha的语法原理大致可归纳为三种执行时映射：自映射、标识符映射和过程映射。

### 自映射

在符号语言中，如果一类对象被执行时意义不变，则该类对象被称为是“自运算”的或“自映射”的。自映射对象是符号语言的语法基础，是语法结构的元子单元。

在Alpha中，具有自映射性质的对象及其语法形式有：

() [] {} 空值的括号形式  
#t #0 #nil 保留常量  
‘A’ "Hello, world!" 字符和字符串  
3.1415 普通数值

### 标识符映射

一个标识符字面量的执行实质上是提取该标识符变量所存储的数据值。这也称作“变量映射”、“名词指代”，是符号元语言和众多高级语言的基本功能。标识符举例：

Lambda

### 过程映射

一个广义表的执行实质上是以表头作为调用体，剩余部分作为参数的映射行为。在Alpha中，调用体只能是过程，所以映射的值即是过程的返回值。例如：

执行(+ 1 1) ⬄ 调用+传参(1 1)

Alpha的语法也支持点对的字面化，一个点对的执行是以左值为调用体，右值为参数的过程映射。例如：

执行(a , b) ⬄ 调用a传参b

上面两种语法被大部分Lisp方言所支持，除此之外，Alpha还支持更灵活的过程映射。众所周知，S-表达式的广义表在Lisp中是以空值结尾的右嵌套点对链，为了提高语法解析效率和统一规则，Alpha支持一种新的字面量形式——广义点对。

广义点对的定义：一个以任意语法对象结尾的右嵌套点对链。广义点对可以被字面化，广义表和点对的字面量都是其特殊形式。广义点对的字面化规则为：点对分隔符出现在第一个既不是点对也不是空值的右值的前面。广义点对的字面形式举例：

(a b , c) ⬄ (a , (b , c))

现在给出广义点对的执行原理：一个广义点对的执行是以第一个左值为调用体，第一个右值为参数的过程映射。例如：

执行(a b c , d) ⬄ 调用a传参(b c , d)

# 符号约定

这里给出用Alpha写程序时的符号约定。

## 标识符

标识符是字母、数字和“扩展标识符字符”组成的满足下列条件的特化的字符序列：

* 不能以有效数字作为前缀
* 不能以若干扩展字符后跟有效数字作为前缀

Alpha的实现必须支持以下扩展标识符：

! $ % & \* + - . / : < = > ? @ ^ \_ ~ |

标识符举例：

+ +soup+ <=? ->string a34kTMNs lambda list->vector qV17a the-word-recursion-has-many-meanings ||

## 保留常量

由#后面跟随一个标识符或数字组成，保留常量中的所有英文字母对大小写不敏感。Alpha约定如下保留常量：

#n #null #nil 空值

#0 #f #fals 布尔假

#1 #t #tru 布尔真

#+inf #-inf 实无穷

#nan 非确定数值

## 字符和字符串

字符字面量由单引号中包含一个字符、转义符组成。其中约定如下转义符：

‘\'’ 单引号

‘\"’ 双引号

‘\\’ 反斜杠

‘\b’ 退格符

‘\f’ 分页符

‘\n’ 换行符

‘\r’ 回车符

‘\t’ 水平制表符

‘\v’ 垂直制表符

‘\x…’ 后跟2位16进制数表示该ASCII编码的字符

‘\u…’ 后跟4位16进制数表示该Unicode或UTF-8编码的字符；后跟括号（小、中、大均可）括住任意长于4位的16进制数表示该UTF-16编码的字符

字符字面量举例：

‘a’ ‘A’ ‘(’ ‘ ’ ‘\u03BB’

字符串字面量由双引号和中间任意个字符、转义符组成。

对于字符而言\"是非必要转义符，'"'合法；对于字符串而言\'是非必要转义符，"'"合法。

字符串字面量举例：

"The word \"recursion\" has many meanings."

"Another example:\ntwo lines of text"

"Here's text \

containing just one line"

"\u03B1 is named GREEK SMALL LETTER ALPHA."

## 普通数值

普通数值字面量是指除开 #+inf #-inf #nan 的数值字面量。

Alpha约定：实数以64位浮点数形式存储，且遵循IEEE754标准的双精度模式，复数包括实部和虚部两个实数。

### 精度和范围

根据IEEE754标准，Alpha浮点数的64个二进制位，从最左边开始，是这样组成的：

* 第1位：符号位，0表示正数，1表示负数
* 第2位到第12位：存储指数部分
* 第13位到第64位：存储小数部分（即有效数字）

IEEE754规定，有效数字第一位默认总是1，不保存在64位浮点数之中。也就是说，有效数字总是1.xx...xx的形式，其中xx...xx的部分保存在64位浮点数之中，最长可能为52位。因此，Alpha提供的有效数字最长为53个二进制位。

最多53个有效二进制位，意味着绝对值小于2的53次方的整数，都可以精确表示。而指数部分是11位，意味着最大指数是2047（2^11-1），分出一半表示负数，则Alpha能够表示的浮点数范围是(2^1024,2^-1023)。

### 整数的进制记法

使用字面量时，Alpha对整数提供四种进制的表示方法：

123 十进制，没有前导

0b11 二进制，有前缀0b或0B

0o377 八进制，有前缀0o或0O

0xff 十六进制，有前缀0x或0X，字母对大小写不敏感

### 实数的科学记法

实数的科学记法为：*N*[e(E)*I*]。允许字母e或E的后面跟一个整数，表示指数部分，例如：

123e3 123e-3 -3.1E+12 .1e-23

其中指数部分（*I*）是整数的十进制记法。

### 复数的一般记法

复数的一般记法为：*R*[+(-)*V*i(I)]。例如：

12+5i 4.33e7-50I

其中实部（*R*）和虚部（*V*）是实数的科学记法。

## 广义点对和空值

三种括号（小、中、大）都可以作为广义点对的边界符，没有特别的意义，但要求首位边界相对应。广义点对的分隔符是逗号（,）：

[hello {()} world] 合法

(hello [] world} 非法

广义点对的边界符和与之相邻的元素之间可以紧挨或有任意空白符，但内部元素之间必须要有至少一个空白符：

(1234 asdf) 合法

[ 1234 asdf] 合法

[1234 asdf ] 合法

{ 1234 asdf } 合法

(1234asdf) 非法

(a,b) 合法

特别的，一个不含任何元素的广义表字面量满足自映射，表示空值：

() [] {}

## 空白和注释

空白字符包括空格、制表符和换行符，实现可以提供额外的空白字符，例如分页符。空白用于改善可读性，并且必要时用于将元语法对象彼此分离，元语法对象是不可分割的词汇单元，例如标识符或数字。空白可以发生在任意两个元语法对象之间，但不在其内，除了字符串。

词法语法包括多个注释形式。注释的语法解析与空白完全相同。

分号（;）表示行注释的开始。注释将继续到分号出现的行的结尾。

指示注释的另一种方式是用井号加分号（#;）标记一个语法对象前缀和可选的空白。注释由注释前缀（#;）、若干空白和一个语法对象组成。此注释对于注释代码段很有用。

块注释由前缀（#`）和后继（`#）围成。

#`

The FACT procedure computes the factorial of a non-negative integer.

`#

(define fact (make Procedure scp . (seq

(define n (eval (car .) scp))

(if #;(= n 1) (= n 0)

1 ;Base case: return 1

(\* n (fact (- n 1)))

)

)))

# 原生对象

在Alpha中，一切皆对象，对象间的引用关系是一致的，所有对象构成一个有向图。Alpha的原生对象集尽量解耦为逻辑上必须的最简单的几种对象，同时包含机器的最小存储单元和内部存储器的线性冗余。

## 类型

Alpha的类型本身就是对象，跟其他对象没有本质区别。

在其他经典语言中，类型之间可以有很复杂的拓扑关系，例如基于类的继承、基于原型的继承等等。经过解耦，Alpha采用最简单的类型结构，而那些复杂的拓扑关系，可作为软规范由用户自行构造。



图 ‑1 类型的引用关系

初存于make变量的过程用来构造指定类型的对象。

(make

*type* :Type

[*argument* …] :Object

) :Object

每个类型都绑定着一个构造过程，make过程会调用*type*的构造过程，并将[*argument* …]传给它，构造过程的返回值会作为内核封装在新构造的对象中，最后make过程返回这个新对象，该对象的类型即为*type*。访问新对象的内核需要使用*type*的访问器，详见§4.3.1。

类型的类型初存于Type变量，构造时需指定类型的构造过程：

(make Type

*constructor* :Procedure 构造过程

)

这个过程会构造一个自定义类型，该类型的构造过程就是constructor，该类型的类型是Type，构造该类型的对象仍旧调用make过程，例如：

(define MyType (make Type (make Procedure . .

(cons (car .) (car (cdr .)))

)))

(make MyType “Hello, ” “world!”)

特别的，Type的类型就是Type自身。

## 空值

空值的意义是“无意义”、“终结”、“空集”等，可记作：

() [] {} #n #null #nil

其类型初存于Null变量，构造形式属：

(make Null)

空值是Alpha语法结构的构成元素，广义表字面量解析后实质是“右嵌套点对链”，其末端是空值。

在Alpha中，空值仅有一份拷贝。



图 ‑2 空值的引用关系

## 布尔值

由于布尔值的特殊意义，Alpha原生对象集将其显式保留了下来，尽管其派生物已经完全可以取代它的作用。表示“假”和“真”的标准布尔对象可记作：

#0 #f #false

#1 #t #true

布尔真值和假值在系统中只有一份拷贝，其类型初存于Boolean变量，构造形式属自构造：

(make Boolean

*self* :Boolean 布尔值自身

)

布尔值主要用于条件判断。



图 ‑3 布尔值的引用关系

## 字节

顾名思义，现代计算机内部存储的次基本单元，含有8个二进制位。在Alpha中字节的类型初存于Byte变量，构造形式属默认构造：

(make Byte)

字节可用于二进制文件的读写、高效率的位运算等。



图 ‑4 字节的引用关系

## 数值

Alpha的数值系统是基于浮点数的复数系统。数值的类型初存于Number变量，构造形式属自构造：

(make Number

*self* :Number 数值自身

)

Alpha的普通数值记作123, 3.14159, 79e-20, 8+5i等等，特殊数值有：

#+inf 正无穷

#-inf 负无穷

#nan 非确切数值

数值可参与各种数学运算和顺序运算。



图 ‑5 字节的引用关系

## 字符

基于ASCII和Unicode标准的编码的对应集，可记作：

‘a’ ‘A’ ‘(’ ‘ ’ ‘\u03BB’

字符的类型初存于Character变量，构造形式属自构造：

(make Character

*self* :Character 字符自身

)

字符及相关运算是Alpha符号处理的基础。



图 ‑6 字符的引用关系

## 点对

点对是最简单、最基础的复合数据结构，由一个左值和一个右值构成。点对类型初存于Pair变量，构造形式属默认构造：

(make Pair)

Alpha的原生功能包含了对点对的访问操作：

car 获取左值

cdr 获取右值

set-car! 修改左值

set-cdr! 修改右值

点对是Alpha语法结构的构成元素，广义表字面量解析后实质是“右嵌套点对链”。



图 ‑7 点对的引用关系

## 字节向量

顾名思义，字节的数组，一种仅以字节为元素的特殊的线性冗余结构。字节向量的类型初存于ByteVector变量，构造时指定需要分配的连续存储空间的长度：

(make ByteVector

*size* :Number向量的长度

)

Alpha的原生功能包含了对字节向量的长度和各个字节元素位的访问操作：

bytes-size 获取字节向量长度

bytes-ref 获取字节向量元素

bytes-set! 修改字节向量元素



图 ‑8 字节向量的引用关系

## 向量

即数组，支持顺序随机访问的线性冗余结构，向量的每个元素可以是任意对象。向量类型初存于Vector变量，构造时指定需要分配的连续存储空间的长度：

(make Vector

*size* :Number 向量的长度

)

Alpha的原生功能包含了对向量的长度和各个元素的访问操作：

vector-size 获取向量长度

vector-ref 获取向量元素

vector-set! 修改向量元素



图 ‑9 向量的引用关系

## 字符串

字符的数组，一种仅以字符为元素的特殊的线性冗余结构。字符串对象有字面量，如：

“Hello, world!”

字符串类型初存于String变量，构造时需指定要分配的连续存储空间的长度：

(make String

*size* :Number 字符串的长度

)

Alpha的原生功能包含了对字符串的长度和各个字符元素位的访问操作：

string-size 获取字符串长度

string-ref 获取字符串元素

string-set! 修改字符串元素



图 ‑10 字符串的引用关系

## 标识符

高级语言的最大特点是将变量“词汇化”，而符号元语言的特点则是将词汇本身“对象化”，这样词汇对象又可以存于变量，从原理上形成一个完备的有向闭环。

Alpha语言的词汇对象化后的产物即是标识符对象。前面语法部分介绍了，标识符实质上是字符串特化后的映射集，拥有满足自映射的字面量，如：

love-alpha!

标识符类型初存于Identifier，构造形式属自构造：

(make Identifier

*self* :Identifier 标识符自身

)



图 ‑11 标识符的引用关系

## 过程

过程是Alpha广义映射的最基本单元，分为原生过程和自定义过程。

原生过程集构成了Alpha的所有原生功能。原生过程无需构造，初存于各自的变量。值得一提的是，原生过程define和set!可以修改变量的值，意味着原生过程的宿主变量是可修改的从而导致原生过程有丢失的可能。

自定义过程是使用Alpha描述算法的最基本工具。由于自定义过程是执行时产生的，因此它依附于所在的作用域，称为“诞生场景”。自定义过程还绑定着两个标识符，在过程被调用时，一个是默认存储着“调用场景”的变量，另一个是默认存储实际参数的变量，原理见§1.2.3。特别的，系统原生功能回调用户构造的过程时，“调用场景”被空值取代。自定义过程还绑定着一个能字面化的对象，称为“过程主体”，即是过程被调用时的执行主体，过程的调用即是对过程主体的执行。

过程类型初存于Procedure变量，构造时需指构造场景、定两个形参标识符和一个过程主体：

(make Procedure

*environment* :Scope 诞生场景

scene\_var =Identifier 存储调用场景

*args\_var* =Identifier 存储实际参数

*body* =Object 过程主体

)

由于过程是第一类对象，Alpha的编程模式也可称为“过程式编程”。



图 ‑12 过程的引用关系

## 作用域

作为§1.1介绍的三大特性之一，作用域的对象化，使得Alpha的流程控制变得方便和直观。每个作用域依附于它的外围作用域存在，类型初存于Scope变量，构造时需指定外围作用域：

(make Scope

*environment* :Scope 外围作用域

)

作用域对象可用于动态的过程调用和程序执行。



图 ‑13 作用域的引用关系

这里给出各类原生对象间的关系总览：



图 ‑14 各类型引用关系总览

# 原生过程

这里介绍初存于各个变量的原生过程，它们构成了Alpha的原生功能。原生过程要求传参的格式必须符合经典广义表，不能是特殊的广义点对。

## 变量

变量系统是符号化语言的基础，是高级语言词汇化的直接体现。变量系统的实质是实现标识符与任意对象的键值映射。

(defined?

*variable* =Identifier

) :Boolean

defined?过程用于检测某变量目前是否已经被定义，已被定义则返回真，否则返回假。

(locally-defined?

*variable* =Idenfifier

) :Boolean

locally-defined?过程用于检测某变量是否再当前作用域被定义，已被定义则返回真，否则返回假。

(define

*variable* :Identifier

[*value*] :Object

) :Boolean

define过程用于在当前作用域定义一个变量，并对其赋值。若变量已在当前作用域被定义则返回假，否则返回真。

(set!

*variable* :Identifier

*value* :Object

) :Null

set!过程用于修改变量的值。若变量不存在则抛出异常

(erase!

*variable* :Identifier

) :Boolean

erase!过程用于在当前作用域擦除变量。若变量未在当前作用域被定义则返回假，否则返回真。

## 数据和运算

这里介绍基本数据类型的内在联系及其运算。

### 解析和串化

程序即数据，这是以S-表达式为语法的符号元语言能够支持的优秀特性。此部分介绍的两个原生过程即是Alpha语言对此特性的支持。

(parse

*code* :String

) :Object

parse过程可以将一段合法的Alpha代码串解析成对应语法对象结构。如果*code*不符合语法规则，则抛出异常。

(stringify

*datum* :Object

) :String

stringify和parse互为逆过程，用于把纯语法对象结构变回代码字符串。如果*datum*不是纯语法对象结构，则抛出异常。

### 数据枢纽

这里的“枢纽”指数据的本质和不同冗余数据类型之间的内在联系。元语言符号系统是一种特殊的形式系统，任何形式系统的数学原理都是基于“两极化差异”和“二元组”。在Alpha中保留的布尔类型正是 “两极化差异”的直接承载者。

(booleans<>byte

*value* :8-length Boolean Vector or Byte

) :the opposite of *value*

booleans<>byte过程诠释了字节的形式本质，用于长度为8的布尔向量和字节的相互转换。

(tear-complex

*complex\_number* :Number

) :Vector

tear-complex过程把一个复数拆成实部和虚部两个实数，并存放在向量中。

(tear-floating

*floating\_point* :Number

) :Vector

tear-floating过程把一个浮点数拆成遵循IEEE 754存储标准的有效数字长度、纯小数底数和指数，并存放在向量中。其中有效数字长度是自然数，指数是整数。

(tear-integer

*integer* :Number

) :Vector

tear-integer过程把一个整数拆成符号和绝对值，并存放在向量中。其中符号是布尔值，绝对值是自然数。

(booleans<>natural

*value* :Boolean Vector or Number

) :the opposite of *value*

booleans<>natural过程诠释了自然数的形式本质，用于布尔向量和自然数的相互转换。布尔向量对应于自然数的二进制序列。

(natural<>character

*value* :natural Number or Character

) :the opposite of *value*

natural<>character过程用于字符和对应编码的相互转换。

(string<>identifier

*value* :String or Identifier

) : the opposite of *value*

string<>identifier过程用于标识符和对应字符串的相互转换。



图 ‑1 类型转换关系

### 位操作

* 位运算

(~ *B*) 取反

(| *B1* *B2*) 按位或

(^ *B1* *B2*) 按位异或

(& *B1* *B2*) 按位与

* 位移，移后空缺的部分全部补0。

(<< *B* *N*) 左移

(>> *B* *N*) 右移

### 逻辑运算

(! *L*) 否定

(|| *L1* *L2*) 逻辑或

(^^ *L1* *L2*) 逻辑异或

(&& *L1* *L2*) 逻辑与

### 顺序运算

(< *N1* *N2*) 小于

(<= *N1* *N2*) 小于或等于

(= *N1* *N2*) 等于

(>= *N1* *N2*) 大于或等于

(> *N1* *N2*) 大于

如果参与顺序比较的数值带有非0虚部，则始终返回假。

### 数学运算

* 基本四则运算

(+ *N1* *N2*) 加

(- *N1* *N2*) 减

(\* *N1* *N2*) 乘

(/ *N1* *N2*) 除

* 乘方、开方和对数

(exp *N*) 自然对数的底数的*N*次方

(pow *N1* *N2*) *N1*的*N2*次方

(root *N1* *N2*) *N1*的*N2*次方根

(lg *N*) *N*的自然对数

(log *N1* *N2*) 以*N1*为底数的*N2*的对数

* 三角、双曲及反函数

(cos *N*) 余弦

(acos *N*) 反余弦

(cosh *N*) 双曲余弦

(acosh *N*) 反双曲余弦

(sin *N*) 正弦

(asin *N*) 反正弦

(sinh *N*) 双曲正弦

(asinh *N*) 反双曲正弦

(tan *N*) 正切

(atan *N*) 反正切

(tanh *N*) 双曲正切

(atanh *N*) 反双曲正切

* 近似值

(floor *N*) 小于*N*的最大整数

(round *N*) 四舍五入后的整数

(ceil *N*) *N*向上取整后的值

(trunc *N*) *N*的整数部分，去除小数

如果参与近似运算的数值带有非0虚部，则返回对实虚分别取近似值的复数。

* 复数的值

(real *N*) 实部

(imag *N*) 虚部

(abs *N*) 绝对值

(arg *N*) 相角

(norm *N*) 范数值

(conj *N*) 共轭值

(polar *rho* [*theta*]) 以距离和相角定义的复数

(proj *N*) 复数到Riemann球体上的投影。

## 复合结构

这里介绍Alpha各种必要的复合结构的相关过程。

### 类型系统

Alpha的类型具有“第一类”性，类型也是对象。类型的相关功能被赋予几个简单而必要的过程。

(make

*type* :Type

[*arguments* …] :Object

) :Object

make过程用于创建指定类型的对象。[*arguments* …]会作为参数传递给*type*的构造过程，构造过程生成的对象称为被创建的新对象的内核。

(make-accessor

*type* :Type

*callback* :Procedure

) :Procedure

make-accessor过程用于生成指定类型的访问器过程。生成的访问器有如下规则：

* 调用时会回调*callback*
* 第一个参数必须是*type*类型的对象，即需要访问的对象，该对象的内核会作为第一个参数传给*callback*
* 后续参数会作为后续参数传给*callback*
* 返回*callback*的返回值

(modify-linearize

*type* :Type

*callback* :Procedure

) :Null

modify-linearize过程用于修饰指定类型的线性化规则。类型线性化的作用是任何类型均可参与原生的实数顺序比较运算和逻辑运算。*callback*必须仅有一个参数，该参数必须是*type*类型的对象，返回值必须是一个实数，即线性化的结果值。

(modify-stringify

*type* :Type

*callback* :Procedure

) :Null

modify-stringify过程用于修饰指定类型的字符串化规则。类型字符串化为任意类型的对象参与原生输出流的输出行为提供接口。*callback*必须仅有一个参数，该参数必须是*type*类型的对象，返回值必须是一个字符串，即字符串化的结果值。

### 点对

点对相关功能有左右值的获取和修改：

(car *pair*)

(set-car! *pair* *car\_value*)

(cdr *pair*)

(set-cdr! *pair* *cdr\_value*)

### 线性结构

线性结构的相关功能有各种线性冗余结构的长度的获取和元素的获取和修改：

(bytes-size *byte\_vector*)

(bytes-ref *byte\_vector* *index*)

(bytes-set! *byte\_vector* *index* *value*)

(vector-size *vector*)

(vector-ref *vector* *index*)

(vector-set! *vector* *index* *value*)

(string-size *string*)

(string-ref *string* *index*)

(string-set! *string* *index* *value*)

## 控制台和文件流

这里介绍简单而必要的原生的流功能。

### 控制台输入输出

(input) :Object

input过程用于从控制台（标准输入流）读取一个Alpha语法字面量的解析后的对象。若出现了非法字符或不满足语境的字符，则抛出异常。

(output

*value* :Object

) :Null

output过程用于向控制台（标准输出流）写入一个对象字符串化后的值。如果*value*的类型未经过字符串化修饰，则抛出异常。

### 文件流

原生的文件流功能包含文本文件和二进制文件的读写。由于文件读写规则相对复杂，放在本报告的下一个版本编写，Alpha的草案版解释器的原生库将不包含此部分的功能。

## 流程控制

### 局部基本流程

seq、if、loop和break构成了单一作用域内的基本控制流程。

(seq

*A* :Object

[*B* …] :Object

) :Object

顺序、序列。在当前作用域依次执行A, B …，并返回最后一个元素的执行结果。所有元素的执行均在当前作用域。

(if

*condition* :Boolean

*then* :Object

[*else*] :Object

) :Object

选择、条件、分支。若*condition*为真，则执行*then*，否则执行*else*。若condition不是布尔值，则对其进行其类型绑定的线性化求值，若其类型没有绑定线性化，则执行*else*。若判断执行的部分是*else*而*else*未传，则返回null。两个分支的执行均在当前作用域。

(loop

*body* :Object

) :Object

循环。不停地在当前作用域执行*body*，直到遇到中断。若遇到break中断则返回break携带的值。

(break

[*result*] :Object

)

一种中断。携带*result*跳出当前作用域最近的一层loop。

### 异常处理

try和throw构成了Alpha的异常处理机制。

(try

*body* :Object

[*catcher*] :Procedure

) :Object

容错机制。在当前作用域执行*body*，若没有任何异常，则返回*body*的执行结果，否则调用*catcher*并传入捕获到的异常，并返回*catcher*的返回值。

(throw

[*exception*] :Object

)

一种中断。手动抛出异常exception，携带exception进入最近一层try的catcher部分。

### 经典续延方案

(call-with-current-continuation

*callback* :Procedure

) :Object

回归机制。调用*callback*并传入一个需要一个参数的“回归过程”，若在*callback*中调用了这个回归过程，则终止*callback*的调用并返回回归过程传入的值，否则一直调用完*callback*并返回其返回值。

### 全局中断

(exit

[*result*] :Object

)

最高级中断。直接结束整个程序的执行，并以*result*作为执行结果。

## 动态调用和执行

(apply

*procedure* :Procedure

[*arguments* :generalized pair

[*environment*]] :Scope

) :Object

apply过程用于对指定过程的动态调用。传参可包含实参自变量和调用场景，调用场景默认为当前作用域。

(eval

*program* :Object

[*environment*] :Scope

) :Object

eval过程用于动态执行一段语法对象。传参可包含调用场景，默认为当前作用域。

## 系统扩展

此部分功能在报告的下一个版本正式编写，这里给出一个指导方案：

(use

*module :String*

[*item* …] =Identifier

) :Null

设想的use过程用于在当前作用域使用Alpha扩展库。扩展库以动态库的形式存在，*module*是其路径。使用时只传*module*会在当前作用域定义库中所有过程的变量，除此之外也可以指定仅使用某些过程。

# Alpha语言示例

* 你好，世界！

(output “Hello, world!”)

# 参考文献

1. Structure and Interpretation of Computer Programs, SICP
2. the Revised⁷ Report on the Algorithmic Language Scheme, R7RS