The Basis Programming Language 草稿

愿望：灵活，高效。参考 C++、Python、JS、C#、Go、Swift、Rust，Mathematica 等等。强类型，静态类型。有继承、多态，泛型。

特性：无 GC，无生命期管理。有依赖管理、指针、运算符重载、字符串模板、unicode、宏、模板、元编程、反射、trait、async、coroutine、多线程、异构计算（包括 GPU TPU 等等）、goto，异常。可选边界和溢出检查。

场景：机器学习、科学计算、后端、嵌入式、脚本、游戏、移动和桌面开发，等等。

计划：编译到 IR 再编译到多种目标。在生产中测试和完善。计划先接入 openCV 和 MXNet 的 C++ 接口，再加上异步和多线程，代替目前所用的 Python 代码。

1. Hello World

Basis 与 Python 类似，初次使用需定义，但可省略类型（此时类似 C++ 1x 的 auto，编译期推断静态类型）。注意，Basis 是“披着 Python 皮的 C++”，不是“静态类型的 Python”。Basis 的大部分特性可编译到 C++ 1x。

```
module main
import sys.console as console


tellPrime : (n)
    n ≤ 1 : 'bad'
    for i = 2, i * i ≤ n, i++
        n % i == 0 : 'bad'
    return 'good'



main :
    print = console.writeLine
    print(tellPrime(17))
```

```
module main
import sys.object.*
import sys.console as console

string tellPrime(int n) {
    if (n ≤ 1) return 'bad'
    for (i = 2; i * i ≤ n; i++) {
        if (n % i == 0) return 'bad'
    }
    return 'good'
}

void main() {
    print = console.writeLine
    print(tellPrime(17))
}
```

```
module main
import sys.object.*
import sys.console as console

tellPrime : (int n) ⇒ string {
    if n ≤ 1 { return 'bad' }
    for i = 2, i * i ≤ n, i++ {
        if n % i == 0 { return 'bad' }
    }
    return 'good'
}

void main() {
    print = console.writeLine
    print(tellPrime(17))
}
```

上文左中右，都是合法的 Basis 程序，语义相同。在 Basis 中，您可自由混搭出各种风格，自由选择喜爱的编程风格。

显然，Basis 的编译器会很复杂，也可能会有未知的歧义，欢迎您关注和参与 Basis 项目，多提宝贵意见。

值得一提的是，Basis 中的冒号代表“定义映射”，如左边所示，它甚至可代替 if 语句。

2. 基本语法例程

Basis 的设计不追求“凡事只有一种方法”，而是希望提供多种选择。例如，定义函数至少有 4 种方法：

```
TYPE f(TYPE x) :
```

```
f(TYPE x) : TYPE
```

```
f : (TYPE x) TYPE
```

```
(TYPE x) TYPE f :
```

本文使用 Python 风格，用缩进控制域。但 Basis 也支持用 { } 控制域的 Cxx 风格。

下文的程序，旨在演示多种语义风格。在实际编程中，建议统一风格。

```
module main                          # 定义模块
import sys.object.*                  # 注意，int 等等不是关键字，而是来自 sys.object.int 等等
import sys.nn.*                      # 稍后用到神经网络库（也会有通往流行 DL 框架的接口）
import sys.console as console        # 稍后用到 sys.console.writeLine
                                     # 此外，远程模块可类似 import "xxx://xxx" as xxx 引入

bool isPrime(int n) :                # 函数用 A : B 定义
    if n ≤ 1                         # 类似 Python，并省略冒号
        return false
    for i = 2, i * i ≤ n, i++        # 等价于 for i in 2 .. sys.math.sqrt(n).int()+1
        if n % i == 0 : return false # 这样可将 if 写成一行，稍后用冒号有更简练的写法
    return true

bool isOdd(int x) : {x % 2 == 1}     # 花括号永远代表闭包，写在单行时，会自动 return 结果
bool isOddAAA(int x) : x % 2 == 1    # 若写成单行，且不用花括号，则必须省略 return 关键字
bool isOddBBB(int x) : (x % 2 == 1)  # 可选加小括号，更清晰

string myDict[bool] = [true: 'good', false: 'bad']   # 字典必须写明类型；字符串支持 ' ' 和 " "
                                                     # 字典用中括号列表
int[] arr = [1, 2, 3]                                # 数组也用中括号列表

# 同时 Basis 支持多种风格，可自由使用，下文的箭头用 => 输入

isEven : x ⇒ {x % 2 == 0}            # 箭头风格，这里省略类型，自动用 auto 机制推断
isEvenFoo : int x ⇒ bool {x % 2 == 0}# 写明类型的箭头风格，
isEvenBar : int x ⇒ bool             # 多行的箭头风格
    return x % 2 == 0

isEvenXXX : x ⇒ x % 2 == 0           # 如前所述，这里的花括号也可省略，最终效果很干净

isEvenAAA : (int x) bool {x % 2 == 0}# 也可省略箭头，那么给参数加括号
isEvenBBB : bool (int x) {x % 2 == 0}# 也可将 (x) y 写成 y (x)

# 下文介绍 Basis 的特点：冒号语法

myDictFoo : [bool] string [true: 'good', false: 'bad']      # 字典也有箭头风格
myDictBar : [bool] string            # 多行的字典，有点像函数。字典和函数都是映射，都是运算符重载
    true    : 'good'                # 因此，这种冒号语法，可视为某种自动 return 的 if 语句
    false   : 'bad'                 # 显然，不能写成 false = 'bad'，这是 : 和 = 的区别之一例

sayNumberSilly : (int x) string      # 函数也可用冒号语法，在冒号的左边需写明完整的条件
    x == 6      : 'six'              # 实际语义，完全等于自动 return 的 if 语句
    x % 2 == 0  : 'some even number' # 但，冒号语法，更符合"定义映射"的思想
                : 'no idea'          # 若左边是 true，可省略，并推荐省略

isEvenBaz(int x) : bool              # 这是一种常见风格，Basis 同样支持
```

```
    return x % 2 == 0

# 下文介绍简单的 OOP 特性，后续章节介绍多态等高级特性

MY_COMPLEX : <object>                   # 定义类，必须写明基类，这里继承 sys.object.object
    double re = 0.0, im = 0.0           # 浮点常数默认是 double 类型

    # 将构造函数视为一种运算符重载，这里的函数定义省略返回类型，自动推断返回类型
    op (double re, double im) :         # 也可写成 op : (double re, double im)
        this.re = re                    # 有歧义时，加上 this
        this.im = im
        return new(this)                # 记得最后返回一个新对象

    op + (MY_COMPLEX v) :               # 注意，对象都是传引用，在此无需说明是传引用
        res = MY_COMPLEX(re + v.re, im + v.im)    # 注意，生成对象的语法和 C++ 不同
        return res

    op string :                                   # 类型转换重载
        return string(re) + '+' + string(im) + 'i'    # 用 + 连接字符串

    op copy : { return copy(this) }         # 默认赋值时只做浅拷贝。这里定义深拷贝（用默认的深拷贝）

    toStringFoo : string                    # 这是一个普通的类方法，实际是 () ⇒ string，做了省略
        return this.string()                # 注意 x.string() 等于 string(x)，语义相同

# 下面的函数定义省略一切类型，则等于 int argc, char** argv ⇒ var 全自动推断
MY_COMPLEX.print : { console.writeLine(toStringFoo()) }      # 目前这实际是静态地给类添加方法

MY_COMPLEX_FOO : (double re, double im)   # 也可模仿 JS，由函数定义类
    double re = re, im = im               # 这样的类会自动继承 sys.object.function
    return this.new()                     # 记得返回新对象，this.new() 也等于 new(this)

main :
    console.writeLine('hello world')

    print = console.writeLine                     # 这实际是传递一个函数指针
    print(isPrime(17) : 'good' else 'bad')        # 三元操作符，注意和 C 的不同
    print('this is ' + myDict[isOdd(x: 17)])      # 类似 Python，可用明确的参数名调用函数

    COMPLEX = MY_COMPLEX                  # 不需要 typedef 关键字
    COMPLEX aa = COMPLEX(1, 2)            # 完整的写法
    bb = COMPLEX(3, 4)                    # 简单的写法
    (aa+bb).print()                       # 调用类的方法

    A = tensor(shape: (10, 5), ctx: gpu)  # 完整的写法，sys.nn.tensor 和 sys.nn.gpu 都是类

    B = tensor(5, 1, cpu)                 # 简单的写法
    C = tensor(5, 1).cpu()                # 也可这样写
    D = cpu(tensor(5,1))                  # 也可这样写
```

```
E = cpu(5, 1)                          # 偷懒的写法

X = cpu(A * gpu(B))                    # 在 gpu 运算，再传回 cpu
print(string(X))                       # 强类型，所以需明确转为 string
```

例程到此结束。下面总结函数定义，可省略部分或全部类型：

```
f : (TYPE x) TYPE          # 完整的函数定义
f : TYPE                   # 自动输入，等于 f : int argc, char** argv ⇒ TYPE
f : (TYPE x)               # 自动输出，等于 f : TYPE x ⇒ var
f : (x)                    # 等于 f : var x ⇒ var
f :                        # 自动输入输出，等价于 f : int argc, char** argv ⇒ var
f : () ⇒ void              # 这是一个必须无输出，无输出的函数
```

若函数中未使用 argc，argv，会自动优化。

还有更多写法，因为，可说明 f 的类型，也可说明 f(TYPE x) 的类型：

```
TYPE f(TYPE x) :           # 传统写法
(TYPE x) TYPE f :          # 因为 TYPE x ⇒ TYPE 也是一种 TYPE
f(TYPE x) : TYPE           # 还可这样写
```

再以字典为例，下面只是部分写法：

```
string myDict[bool] = [true: 'good', false: 'bad']
myDict = [bool] string [true: 'good', false: 'bad']
myDict = string[bool] [true: 'good', false: 'bad']
myDict[bool] = string [true: 'good', false: 'bad']
string[bool] myDict = [true: 'good', false: 'bad']
[bool] string myDict = [true: 'good', false: 'bad']
```

从运算符角度而言，重载 () 的是函数。重载 [] 的是字典。重载 . 的是类。其余是普通变量。

3. 基本语法的简单说明

每个代码文件的构造，首先是文件头：

```
module xxx
import xxx
```

后续的外层语句，有且仅有 4 种可能。其中 : 也可能是 =。

TYPE x : v	TYPE x : v	x : TYPE v	x : TYPE v

- 目前 x 包括 x x(a) x[a] x.y
- 目前 v 包括 v {v} [v]
- 目前 TYPE 包括 x，x ⇒ y，[x] ⇒ y，x[n]，<x>，且可部分或全部省略
- 同时 TYPE 可省略箭头，例如下列三者等价：x ⇒ y，(x) y，y (x)

举例，之前的例程，外层语句如下：

```
package xxx
import xxx
TYPE x :        # x 为 isPrime(int n)，TYPE 为 bool
    v
TYPE x : {v}    # x 为 isOdd(int x)，TYPE 为 bool
TYPE x = [v]    # x 为 myDict[bool]，TYPE 为 string
TYPE x = [v]    # x 为 arr，TYPE 为 int[]
x : TYPE {v}    # x 为 isEven，TYPE 为 x ⇒ 这实际是半省略的形式，等价于 var x ⇒ var
x : TYPE {v}    # x 为 isEvenFoo，TYPE 为 int x ⇒ bool
……
x : TYPE        # x 为 MY_COMPLEX，TYPE 为 <object>
    v
……
```

在 TYPE 不完整时，如何推断？目前有 4 种可能：变量，函数，字典、类。

- 字典必须写明类型。于是解决字典的歧义。
- 类必须写明基类。于是解决类的歧义。
- 变量必须用 A = B 定义，函数必须用 A : B 定义。于是解决余下的歧义。