

特点：高效，灵活。参考 C++、Python、JS、Go 等等。强类型，静态类型，

特性：有指针、运算符重载、模板、字符串模板、宏，异常。无 GC。有 async、coroutine，多线程。有 goto。

场景：机器学习、科学计算、后端、嵌入式、游戏、移动 app，桌面 app 等等。

计划：核心特性可编译到 C、C++、Python、JS、LLVM IR 等等。希望先支持一些 DL 和 CV 框架。



1. 语法规则

```
package main                # 定义包，与 Go 类似
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.*             # 稍后用神经网络库
import sys.console as console # 稍后用 sys.console.writeLine
```

Basis 与 Python 类似，初次使用需赋值。可省略类型，类似 C++ 1x 的 auto，编译期推断静态类型

```
bool isPrime(int n) :      # 注意，冒号不是装饰。A : B 和 A = B 相似，前者导出到包外，后者不导出到包外
    if n <= 1              # 类似 Python，用缩进控制，并省略冒号
        return false      # 如果写在同一行就需要冒号 if n<=1 : return false
                            # 虽然 if A {B} 也合法，但此时语义与 if A : B 不同
    for i = 2, i * i <= n, i++
        if n % i == 0      # 等价于 for i in 2 .. sys.math.sqrt(n).int()+1
            return false
    return true
```

```
bool isOdd(int x) : {x % 2 == 1}    # 写在一行时，花括号为闭包，可省略 return
string myDict[bool] : {true: 'good', false: 'bad'} # 目前字典需写明类型，字符串支持 ' ' 和 " "
```

Basis 同时支持 lambda 风格，下面演示

```
isEven : (x) => {x % 2 == 0}        # 省略类型，自动用 auto 机制推断
isEvenFoo : (int x) => bool {x % 2 == 0} # 写明类型的 lambda 风格
isEvenBar : (int x) => bool         # 多行的 lambda 风格
    return x % 2 == 0
```

```
myDictFoo : [bool] => string {true: 'good', false: 'bad'} # 字典也有 lambda 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 switch
    false : 'bad'
```

```
sayNumberSilly : (int x) => string # 所以，函数也可使用冒号语法，代替 switch
    6 : 'six'                      # 在只有 1 个参数时，可省略参数名，在此会执行 x == 6 比较
    x % 2 == 0 : 'some even number' # 这里写出参数，运行时，从上往下，逐行做判断，成功后跳出
    return 'no idea'              # 普通的语句
```

下面定义类，也都用 lambda 风格

```
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                                # 用 A = B 定义私有元素；浮点常数默认是 double 类型

# 将构造函数视为一种运算符重载，一定返回自己，无需写明输出
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

op + : (MY_COMPLEX v) => MY_COMPLEX                    # 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，定义栈对象的方法和 C++ 不同
    return res

op string :                                              # 类型转换重载，也无需写明输出
    return string(re) + '+' + string(im) + 'i'          # 用 + 连接字符串

op copy : { 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 string(this)                                  # 此外，string(x) 也可写成 x.string()，语义相同

# 函数定义省略一切类型，则相当于 (int argc, char** argv) => var 全自动推断
MY_COMPLEX.print : { console.WriteLine(this.string()) } # 目前这实际是静态地给类添加方法

# 也可类似 JS 定义类，此时需写明 return this
MY_COMPLEX_FOO : (double re, double im)                # 函数定义省略返回类型，自动推断返回类型
    double re : re, im : im                             # 用 A : B 定义公有元素
    return 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' : 'bad')                  # 三元操作符
    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()                                  # 也可以这样写
D = cpu(tensor(5,1))                                     # 也可以这样写
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

```

函数定义总结，可省略类型的部分或全部：

```
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          # 甚至可这样写
```

再以字典为例，4 种写法：

```
string myDict[bool] : {true: 'good', false: 'bad'}
myDictFoo : [bool] => string {true: 'good', false: 'bad'}
myDictFoo[bool] : string {true: 'good', false: 'bad'}
([bool] => string) myDict : {true: 'good', false: 'bad'}
```

但是，变量定义只支持 `int x = 6`，不支持 `x : int {6}`，因为会与省略输入的函数混淆。

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。

2. 语法的简单说明

每个代码文件的构造，首先是文件头：

```
package xxx
import xxx
```

后续的外层语句，有且仅有 4 种可能。其中 `:` 都可替换为 `=`，区别如前所述。

- 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>`，且可部分或全部省略

```
TYPE x : v
```

```
TYPE x :
    v
```

```
x : TYPE v
```

```
x : TYPE
    v
```

例如，之前的例程，外层语句如下：

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

在 `TYPE` 不完整时，如何推断？有 4 种可能：变量/函数/字典/类。目前要求字典的 `TYPE` 必须完整，解决字典的歧义。目前要求类必须写明基类，解决类的歧义。那么只剩下函数和变量的歧义。举例说明推断。

例一，由于 `v` 为 `{v}` 形式，因此 `MY_COMPLEX.print` 是函数。

```
MY_COMPLEX.print : {v}
```

例二，由于 `v` 在换行后，因此 `main` 是函数。

```
main :
    v
```

例三，在此 `x` 是变量。

```
x : v, 其中 v 是 v 或 [v]。
```