# 概述

Motoko编程语言是一种新的现代类型语言，专为希望构建下一代应用程序和服务以直接在 Internet Computer上运行的开发人员而设计。其具有以下特性：

* 原生的canister支持。canister被表示为actor，是封装了状态并通过异步消息进行通信的孤立对象。
* 以直接方式顺序编码。canister调用canister虽然是异步的，但是通过await可以表现的像同步调用一样。
* 现代类型系统。motoko具有健全的结构类型、泛型、变量类型和pattern匹配。
* 正交持久性。通过自动化整个存储过程简化了内存持久性。
* 支持软件升级。motoko支持了升级容器软件时允许您的堆自迁移语言的功能。
* motoko包括子类型、任意精度算术和垃圾收集的其他功能。
* 自动生成的 IDL 文件。motoko支持自动生成candid文件，该文件中暴露的接口允许其他人调用您的函数。

# 程序

程序<imp>;\* <dec>;\*具有类型T的返回值。

程序应满足如下前提：

* <dec>;\* 在由 <imp>;\* 导入引起的静态环境下具有类型 T的返回值。

程序先执行imports，并将值绑定到<imp>;\*中的标识符，然后执行一系列声明语句<dec>;\*并返回值。

# 注释

在 Motoko 中，注释必须以两道斜杠开始，并持续到本行的结尾。

|  |
| --- |
| // single line comment  x = 1 |

单行或多行注释可以是由 /\* 和 \*/ 分隔的任何字符序列：

|  |
| --- |
| /\* multi-line comments  look like this, as in C and friends \*/ |

文档注释以/// 开始，后跟一个空格，并附加到紧跟在它们后面的定义直到行尾。

|  |
| --- |
| /// I'm a documentation comment  /// for a function |

弃用注释以 /// @deprecated 开头，后跟一个空格，并附加到紧跟在它们后面的定义直到行尾。它们仅在公开声明前可被识别。

|  |
| --- |
| /// @deprecated |

# 关键字

如下关键字目前有对应其描述的功能。

* actor
* and
* assert
* assert
* await
* break
* case
* catch
* class
* continue
* debug
* debug\_show
* else
* flexible
* false
* for
* func
* if
* ignore
* in
* import
* not
* null
* object
* or
* label
* let
* loop
* private
* public
* query
* return
* shared
* stable
* system
* switch
* true
* try
* type
* var
* while

# 库

对module语法形式的限制可以使库没有副作用。库的导入是本地的，可以安全地对同一库的多个导入进行重复数据删除，而不产生副作用。

## module库

module库<imp>;\* module <id>? <obj-body>具有模块类型T的返回值。由导入序列 <import>;\* 后跟单个模块声明组成。

module库应满足如下前提：

* module <id>? <obj-body>具有模块种类的Object类型T的返回值。

module库先执行imports，并将值绑定到<imp>;\*中的标识符，然后执行module <id>? <obj-body>并返回值。

## actor class库

actor class 库<imp>;\* <shared-pat>? actor class <id> <typ-params>? <pat> (: <typ>)? <class-body>具有类型T的返回值。类型T具有如下形式：

|  |
| --- |
| module {  type <id> = U;  <id> : (V1,...,Vn) -> async U  } |

actor class库应满足如下前提：

* U的类型为<class-body>定义的对应的actor类型

actor class库先执行imports，并将值绑定到<imp>;\*中的标识符，然后执行如下派生module并返回值：

|  |
| --- |
| module {  <shared-pat>? actor class <id> <typ-params>? <pat> (: <typ>)? <class-body>  } |

注意：导入的actor class库中的<id>函数是异步函数，用于异步构造一个actor。

## import库

import语法import <id>? =? <url>声明了一个可选的<id>标识符，并将其绑定至文本字面量<url>。

如果<id>省略，则默认随机绑定一个新的标识符。

<url>指定了一些资源的文本字面量：

* "<filepath>"：使用相对路径指定的本地库，<id>将被绑定到<filepath>.mo指定的库模块。<filepath> 相对于封闭文件的绝对位置。例如import U "lib/Util"将本地的./lib/Util.mo绑定到了U标识符。
* "mo:<package-name>/<path>"：命名包中的命名模块，<id>将被绑定到<package-path>/<path>.mo指定的库模块。<package-path>由<package-name>进行映射，该映射在编译时通过flag --package <package-name> <package-path>进行指定。例如import L "mo:base/List"将base包别名中的List库绑定到了L标识符。
* "ic:<canisterid>"：由canister ID指定的外部canister，<id>将被绑定到actor实例，actor类型由canister的IDL接口文件定义。canister <canisterid>的IDL接口文件路径为<actorpath>/<canisterid>.did。<actorpath>在编译时通过flag --actor-idl <actorpath>进行指定。例如import C "ic:lg264-qjkae"使用标识符C引用了canister id为lg264-qjkae的actor，IDL接口文件为lg264-qjkae.did。
* "canister:<name>"：由命名别名指定的外部canister，是对容器别名 <name> 的符号引用。编译器假定 <name> 到 <canisterid> 的映射由命令行参数 --actor-alias <name> ic:<canisterid> 指定。因此"canister:<name>"等效于"ic:<cansterid>"。

注意：当使用 DFINITY Canister SDK 构建多容器项目时，Motoko 程序通常可以通过别名导入容器（例如 import C "canister:counter"），而无需指定低级容器 ID（例如 import C "ic:lg264- qjkae”）。SDK 工具负责向 Motoko 编译器提供适当的命令行参数。

# 字面量

## 整数

Tomoko中的整数写为十进制或十六进制。整数字面值有如下一些：

| **整数字面值** | **例子** |
| --- | --- |
| Decimal (十进制) | 98222, 98\_222, -98222 |
| Hex (十六进制) | 0xffff, 0xff\_ff |

注意：为了书写美观，允许在数字之间添加下划线进行分隔。

motoko中的整数默认为Nat类型。

## 浮点数

Tomoko中的浮点数写为十进制或十六进制。浮点数字面值有如下一些：

| **浮点数字面值** | **例子** |
| --- | --- |
| Decimal (十进制) | 9.82, 9.82e3, 9.82E3, 9.82e-3 |
| Hex (十六进制) | 0x9a.8b, 0x9a.8bP5, 0x9a.8bP-5, |

其中“e”（或“E”）以 10 为底的十进制指数为前缀； 'p'（或 'P'）作为以 2 为底的二进制指数的前缀。

注意：十进制或者十六进制的表示方式中，指数都是十进制表示法，即没有~~0x9a.8bP-a~~，而是0x9a.8bP-10。

## 布尔值

Motoko 中的布尔值字面量有两个可能的值：true 和 false。

## 字符

字符是由单引号(')分隔的：

* UTF-8 中的 Unicode 字符，
* 使用\进行转义的换行符、回车符、制表符、单引号或双引号，
* \x{hexnum}以十六进制形式包含有效的转义 ASCII字符，
* \u{hexnum}以十六进制形式包含有效的转义 Unicode 字符。

例如：

|  |
| --- |
| 'a'  'ℤ'  '\x00'  '\t'  '\u{00af}' |

## 字符串

字符串是由双引号(")进行分隔的字符序列：

|  |
| --- |
| "Löwe 老虎 Léopard"  "Hello, world!"  "Mary had\ta little \n\t lamb" |

## null

字面量null的类型为 Null。

由于对于任意的T，Null是? T的[子类型](#_子类型)，因此字面量null也具有? T的类型。

# 运算符

## 运算符种类

motoko对运算符和对应的可进行运算符操作的原始类型进行了如下的分类：

| Category | 可操作原始类型 | 支持的操作符 |
| --- | --- | --- |
| 算数运算符 | Float Int IntN Nat NatN | -(unop), +(unop), +, -, \*, /, %, \*\*, +=, -=, \*=, /=, %=, \*\*= |
| 逻辑运算符 | Bool | and, or, not |
| 位运算符 | IntN NatN | ^(unop), &, |, ^, <<, >>, <<>, <>>, &=, |=, ^=, <<=, >>=, <<>=, <>>=  +%, -%, \*%, \*\*%,+%=, +%=,\*%=, \*\*%= (这八个运算符在运算产生溢出时会进行位回绕) |
| 比较运算符 | Char Text Blob Float Int IntN Nat NatN Principal | <, >, <=, >=  (==和!=两个比较运算符没有种类，除了原始类型，他们也可以作用于复合类型上) |
| 文本拼接运算符 | Text | #, #= |

## 一元运算

### 算数运算

一元算数运算有以下几种：

* -：数值负号。
* +：数值正号。

### 位运算

一元位运算有以下一种：

* ^：位取反。不同于二元位异或运算，只能存在单个右操作数。

### 赋值运算

一元赋值运算有以下一种：

* ^=：位取反赋值运算。不同于二元位异或赋值运算，只能存在单个右操作数。

### Null Break

一元null break运算：

* !：null break，如果option类型值为null，则产生break。

## 二元运算

### 算数运算

二元算数运算包含以下几种：

* +：加法
* -：减法
* \*：乘法
* /：除法
* %：取模
* \*\*：幂运算
* +=：加法赋值运算
* -=：减法赋值运算
* \*=：乘法赋值运算
* /=：除法赋值运算
* %=：取模赋值运算
* \*\*=：幂赋值运算

### 位运算

二元位运算包含以下几种：

* &：位与
* |：位或
* ^：位异或
* <<：左移
* >>：右移，注意在>>的左边必须存在一个空格
* <<>：循环左移
* <>>：循环右移
* +%：溢出回绕加法
* -%：溢出回绕减法
* \*%：溢出回绕乘法
* \*\*%：溢出回绕幂运算
* &=：位与赋值运算
* |=：位或赋值运算
* ^=：位异或赋值运算
* <<=：左移赋值运算
* >>=：右移赋值运算
* <<>=：循环左移赋值运算
* <>>=：循环右移赋值运算
* +%=：溢出回绕加法赋值运算
* -%=：溢出回绕减法赋值运算
* \*%=：溢出回绕乘法赋值运算
* \*\*%=：溢出回绕幂运算赋值运算

### 拼接运算

二元文本拼接运算包含以下几种：

* #：文本拼接运算
* #=：文本拼接赋值运算

### 赋值运算

二元赋值运算包含以下几种：

* :=：值更新运算
* +=：加法赋值运算
* -=：减法赋值运算
* \*=：乘法赋值运算
* /=：除法赋值运算
* %=：取模赋值运算
* \*\*=：幂赋值运算
* &=：位与赋值运算
* |=：位或赋值运算
* ^=：位异或赋值运算
* <<=：左移赋值运算
* >>=：右移赋值运算
* <<>=：循环左移赋值运算
* <>>=：循环右移赋值运算
* +%=：溢出回绕加法赋值运算
* -%=：溢出回绕减法赋值运算
* \*%=：溢出回绕乘法赋值运算
* \*\*%=：溢出回绕幂赋值运算
* #=：文本拼接赋值运算

## 关系运算

关系运算包含以下几种：

* ==：值相等。
* !=：值不等。
* < ：值小于，注意<的左右必须存在空格。
* > ：值大于，注意>的左右必须存在空格。
* <=：值小于等于。
* >=：值大于等于。

注意：==和!=适用于所有shared类型的参数，包括非原始、复合类型，如不可变数组、variant等。

## 逻辑运算

逻辑运算包含以下几种：

* and：逻辑与。
* or：逻辑或。
* not：逻辑非。

# 数据类型

## 原始类型

### Bool

类型 Bool具有值 true 和 false，支持if表达式、while表达式、not表达式、and表达式、or表达式等。

### Char

Char 将字符表示为 Unicode 字符集中的代码点。支持比较运算符操作。使用Char.toText()可以将字符转换为长度为 1 的文本。

字符可以转换为Nat32，而0 .. 0x1FFFFF范围内的 Nat32可以转换为Char，超过此范围的Nat32转换时会产生陷阱。

### Text

Text类型表示 Unicode 字符序列（即字符串）。方法t.size()返回文本t中的字符数。支持拼接运算符操作和比较运算符操作。

可以通过 t.chars() 对字符串进行顺序迭代，例如for (c : Char in t.chars()) { ... c ... }。

### Blob

Blob 类型表示二进制 Blob 或字节序列。方法t.size()返回文本t中的字节数。支持比较运算符操作。

可以通过 t.vals() 对Vlob进行顺序迭代，例如for (v : Nat8 in b.vals()) { ... v ... }。

### Float

Float 类型表示的 64 位浮点值。支持算数运算符操作和比较运算符操作。

### Int

类型 Int是带符号整数。支持算数运算符操作和比较运算符操作。Int支持任意精度。

Nat是Int的子类型，每个 Nat 类型的表达式也是 Int 类型的表达式，每个 Nat 类型的值也是一个 Int 类型的值。

### Int8/Int16/Int32/Int64

Int8、Int16、Int32 和 Int64 类型分别表示具有 8、16、32 和 64 位精度的带符号整数。支持算数运算符操作和比较运算符操作以及位运算符操作。

在进行算数操作或位操作，可能会因为精度问题产生陷阱。

这些类型之间或者和其他数值类型之间都不存在子类型的关系。

### Nat

类型 Nat是自然数。支持算数运算符操作和比较运算符操作。Nat支持任意精度，当Nat使用减法时可能会产生下溢陷阱。

Nat是Int的子类型，每个 Nat 类型的表达式也是 Int 类型的表达式，每个 Nat 类型的值也是一个 Int 类型的值。

### Nat8/Nat16/Nat32/Nat64

Nat8、Nat16、Nat32 和 Nat64 类型分别表示具有 8、16、32 和 64 位精度的自然数。支持算数运算符操作和比较运算符操作以及位运算符操作。

在进行算数操作或位操作，可能会因为精度问题产生陷阱。

这些类型之间或者和其他数值类型之间都不存在子类型的关系。

### Principal

类型 Principal 表示不透明的身份主体，例如容器和用户。它们可用于识别共享函数的调用者并用于简单的身份验证。支持比较运算符操作。

尽管不透明，但身份主体可以转换为二进制 Blob 值，以实现更高效的散列和其他应用程序。

### Error

Error类型用于表示一个错误，错误只能在异步上下文中抛出和捕获，通常是share函数或async 表达式的主体。

Error暴漏了以下函数：

* E.reject : Text → Error：创建一个新的Error，该Error的code为#canister\_reject。
* E.code : Error → E.ErrorCode：获取指定Error的code。
* E.message : Error → Text。获取指定Error的文本信息。

注意：#canister\_reject 以外的代码错误（即系统错误）可能会被捕获并抛出，但这不是用户构建的。

## 复合类型

### Tuple

元组类型的语法为：( ((<id> :)? <typ>),\* )。

元组是一个将多个其他类型的值组合进一个复合类型的主要方式。元组长度固定：一旦声明，其长度不会增大或缩小。元组中的每一个位置都有一个类型，这些不同值的类型不必相同。元组类型声明如下：

|  |
| --- |
| let tup: (Nat, Nat, Nat)= (1,2,3); |

使用包含在圆括号中的逗号分隔的表达式列表来创建一个元组，声明时类型注解可以省略。例如：

|  |
| --- |
| let tup = (500, 6.4, 1); |

注意：表达式的值按照从左到右的顺序依次进行计算。

不支持单一元素元组，对于单一元素的元组，会自动变为该元素类型。例如：

|  |
| --- |
| let tup1: (Nat)= (1); // tup1类型为Nat，值为1  let tup2: (Nat,)= (1,); // tup2类型为Nat，值为1 |

元组类型支持添加组员标识符，该标识符无法作为索引，仅用于生成文档。例如：

|  |
| --- |
| let tup: (add1:Nat, add2:Nat, add3:Nat) = (1,2,3);  tup.0;  ~~tup.add1;~~ |

空元组()被称为单元类型。

#### 元组解构

可以使用pattern匹配（pattern matching）来解构（destructure）元组值。例如：

|  |
| --- |
| let tup = (500, 6.4, 1);  let (x, y, z) = tup; |

注意：元组解构只能用于初始化不可变变量，无法用于可变变量，例如：

|  |
| --- |
| let tup = (500, 6.4, 1);  ~~var (x, y, z) = tup;~~ |

具体见[元组pattern](#_元组)。

#### 成员访问

可以使用点号（.）后跟值的索引来直接访问它们。元组的第一个索引值是 0。例如：

|  |
| --- |
| let tup: (Nat, Nat, Nat)= (1,2,3);  tup.0; |

具体见[tuple投影表达式](#_tuple投影表达式)。

### Array

array类型的语法为[ var? <typ> ]。

数组是可以包含多个值的类型，数组中的每个元素的类型必须相同。数组类型声明如下：

|  |
| --- |
| let a1 = [1, 2, 3, 4, 5];  let a2: [Nat] = [1, 2, 3, 4, 5]; |

其中Nat 是每个元素的类型。

Motoko 中的数组是固定长度的，一旦声明，它们的长度不能增长或缩小。如果需要一个允许增长和缩小长度的类似数组的集合类型，那么可以使用标准库提供的Buffer 类型。

由于数组类型仅指定了数组的元素类型，而未指定长度，因此对于可变数组，可以使用一个不同长度相同元素类型的数组进行赋值。例如：

|  |
| --- |
| var a = [1, 2, 3, 4, 5];  a := [1, 2]; |

可以使用Array.tabulate<T>函数以及一个生成函数来创建一个数组。例如：

|  |
| --- |
| let array1 : [Nat] = [1, 2, 3, 4, 6, 7, 8] ;  let array2 : [Nat] = Array.tabulate<Nat>(7, func(i:Nat) : Nat {  if ( i == 2 or i == 5 ) { array1[i] \* i }  else { array1[i] }  }) ; |

#### 元素访问

可以使用索引来访问数组的元素。例如：

|  |
| --- |
| let a = [1, 2, 3, 4, 5];  let first = a[0];  let second = a[1]; |

当使用无效索引访问时，会产生一个陷阱。

具体见[array索引表达式](#_array索引表达式)。

#### 元素可变数组

[T]类型的数组的元素是不可变的，无法通过元素访问的方式修改元素的值。而[var T]类型的数组的元素是可变的，可以通过元素索引修改元素的值。例如：

|  |
| --- |
| let a : [var Nat] = [var 1, 2, 3] ;  a[0] := 3; |

注意：初始化列表中需要添加var。

可以使用Array.init<T>函数以及元素个数和初始化值来创建一个长度固定的元素可变数组。例如：

|  |
| --- |
| var size : Nat = 42 ;  let x : [var Nat] = Array.init<Nat>(size, 3); |

### Object

Object类型的语法为<sort>? { <typ-field>;\* }。通过列出其零个或多个命名类型字段来指定Object类型。在一个Object类型中，字段的名称必须是不同的，仅在字段顺序上不同的Object类型是等效的。

在Object声明中，只有public的字段才是该Object的类型字段，privae字段不是该Object的类型字段（可以认为内部变量），这会影响[子类型](#_子类型)的判断。例如：

|  |
| --- |
| object counter {  public var count = 0;  public func inc() { count += 1 };  };  // 类型为{var count : Nat; inc : () -> ()}  object counter {  var count = 0;  public func inc() { count += 1 };  };  // 类型为{inc : () -> ()} |

#### Object字段访问

具体见[object投影表达式](#_object投影表达式)。

#### Object解构

具体见[Object patten](#_Object)。

### Variant

variant类型的语法为{ <typ-tag>;\* }。通过将variant类型字段列为`<typ-tag>` 序列来指定variant类型。

variant类型类似rust中的可带有关联数据的enum类型。variant类型允许你通过列举可能的成员（variants）来定义一个类型。

variant类型进行如下定义：

|  |
| --- |
| type vari = {  #male;  #female;  };  let va : vari = #male; |

注意：

* variant类型中的所有标签名称都得不同。
* 仅在variant类型标签的顺序上不同的variant类型是相同的。
* "{ # }"表示一个空的variant类型。

#### 关联数据

可以将数据直接放进variant成员，使得数据和variant成员相关联。例如：

|  |
| --- |
| type vari = {  #male : Nat;  #female;  };  let va : vari = #male(1); |

如果variant成员没有关联数据，那么其关联成员类型默认为单元类型()。例如：

|  |
| --- |
| type vari = {  #male : Nat;  #female : ();  };  let va1 : vari = #female;  let va2 : vari = #female();  let va3 : vari = #female(()); |

注意：关联数据是不可变的。只能通过switch的pattern匹配取出关联数据。

### Option

option类型的语法为? <typ>。

option类型? type指示该类型值可以是null或者具有? v格式的值，其中v是type类型的值。例如：

|  |
| --- |
| let a : ? Nat = ?1;  let b : ? Nat = null; |

#### Null

Null 类型只有一个值，即字面量 null。Null类型是任意的?T类型的[子类型](#_子类型)。例如：

|  |
| --- |
| let b : ? Nat = null; |

#### options pattern

Option类型的值可以通过option pattern进行值提取。具体见[option pattern](#_Option)。

### Function

函数类型的语法为<shared>? <typ-params>? <typ1> → <typ2>。<typ1> 和 <typ2> 都可以引用 <typ-params> 中声明的类型参数。

具体见[function表达式](#_function表达式)、[function调用表达式](#_function调用表达式)、[function声明](#_function声明)。

### Async

async类型的语法为async <typ>。指定产生 <typ> 类型值的future。

具体见[async表达式](#_async表达式)、[await表达式](#_await表达式)。

### Any

类型 Any 是顶级类型，即所有类型的父类型。 所有值的类型为 Any。

### None

Type None 是底部类型，是所有其他类型的子类型。不存在值的类型为 None。

作为空类型，None 可用于指定无限循环或无条件陷阱的不可能返回值。

### Constructed type

Constructed type语法为<path> <typ-args>?，是由类型构造器标识符以及类型参数构造生成的类型。

其中<path> 可以是一个以.分隔的actor、object或module标识符序列。<typ-args>类型参数必须满足类型构造函数的类型参数的期望（如果有）。

例如：

|  |
| --- |
| type Natural = Nat; // Natural为没有类型参数的类型构造器  type List<T> = ?(T, List<T>); // List<T>为具有一个类型参数的类型构造器  type IntList<T <: Int> = ?(T, IntList<T>); // IntList<T <: Int>为具有类型参数的类型构造器，且该类型参数需要是Int子类型  let a : Natural = 1; // Natural类型为使用Natural类型构造器生成的类型  let b : List<Nat> = ?(1, null); // List<Nat>类型为使用List<T>类型构造器生成的类型  let c : IntList<Nat> = ?(1, null); // IntList<Nat>类型为使用IntList<T <: Int>类型构造器生成的类型 |

### Parenthesised type

函数的入参或者返回值，如果是类型( <tuple-typ> )，那么会变为<tuple-typ>类型。

在其他的位置，类型( <typ> )与类型<typ>具有相同的意义。

## 类型构造器

使用type C (< X0 <: T0>, …​, Xn <: Tn >)? = U语法可以声明一个类型构造函数,其中<:表示为类型约束，X0需要为T0的子类型。具体见[type声明](#_type声明)。

例如：

|  |
| --- |
| type Natural = Nat; // Natural为没有类型参数的类型构造器  type List<T> = ?(T, List<T>); // List<T>为具有一个类型参数的类型构造器  type IntList<T <: Int> = ?(T, IntList<T>); // IntList<T <: Int>为具有类型参数的类型构造器，且该类型参数需要是Int子类型  let a : Natural = 1; // Natural类型为使用Natural类型构造器生成的类型  let b : List<Nat> = ?(1, null); // List<Nat>类型为使用List<T>类型构造器生成的类型  let c : IntList<Nat> = ?(1, null); // IntList<Nat>类型为使用IntList<T <: Int>类型构造器生成的类型 |

在进行类型构造器声明的时候，要避免生成类型时产生循环。例如：

|  |
| --- |
| ~~type C = C;~~  ~~type D<T, U> = D<U, T>;~~  ~~type E<T> = F<T>;~~  ~~type F<T> = E<T>;~~  ~~type G<T> = Fst<G<T>, Any>;~~ |

如果在生成类型时如果会产生递归，那么就必须存在边界，不能无限膨胀。例如：

|  |
| --- |
| type List<T> = ?(T, List<T>);  ~~type Seq<T> = ?(T, Seq<[T]>);~~ |

## 子类型

如果T类型是U类型的子类型，那么记为T <: U。如果类型T是类型U的子类型，那么类型T的值可以成为类型U的值，例如：

|  |
| --- |
| let num1:Nat = 1;  let num2:Int = num1; // num1虽然类型为Nat，但是由于Nat <: Int，因此num1的值也是Int类型 |

以下条件之一为真时，T类型是U类型的子类型：

* 在声明中，使用了约束条件T <: U。
* 存在类型V，并且T <: V，并且V <: U。
* T是type C<X0 <: Y0, Xn <: Yn> = W类型构造器生成的C<V0, …​, Vn>类型，并且W是U的子类型。
* T和U是同一类型。
* U为Any类型。表示任一类型都是Any类型的子类型。
* T为None类型。None类型是任一类型的子类型。
* T是Nat类型，U是Int类型。
* T是Null类型，U是Option类型?W。
* T是不可变数组类型[V]，U是不可变数组类型[U]，并且V是U的子类型。
* T是可变数组类型[var V]，U是可变数组类型[var U]，并且V是U的子类型。
* T是元组类型(T0,T1,..Ti)，U是元组类型(U0,U1,...Ui)，并且对其中每一个类型，Ti是Ui的子类型。
* T是Option类型?V，U是Option类型?W，并且V是W的子类型。
* T是Async V类型，U是Async W类型，并且V是W的子类型。
* T是Object类型，U是Object类型，并且T和U拥有相同的分类，并且T中不可变字段类型V是U中对应不可变字段类型W的子类型，并且T中可变字段类型V和U中对应可变字段类型W是同一类型。注意：T 可以指定比 U 更多的字段。
* T是Variant类型，U是Variant类型，并且U中variant成员的类型V是T中variant成员的类型W的子类型。注意：U可以指定比T更多的字段。
* T是Function类型，U是Function类型，并且T和U拥有相同的shared声明(都有或都没有)，并且如果存在类型参数则拥有相同的类型约束，并且U的入参类型V是T的入参类型W的子类型，并且T的返回值类型X是U的返回值类型的子类型。

## Shared类型

因为shared function是可以被远程调用者访问，因此限制shared function的入参必须为shared类型，返回值必须具有async T类型，类型T必须是shared类型。

以下类型T是shared类型：

* Bool、Char、Text、Bolb、Float、Int、IntN、Nat、NatN、Principal类型。
* Any、None类型。
* actor类型。
* ?T类型，并且T也是shared类型。
* (T0,T1...,Tn)并且所有Ti都是shared类型。
* array类型，并且是不可变数组[T]，并且T也是shared类型。
* object类型，并且所有的字段是不可变的，并且这些字段是shared类型。
* variant类型，并且其所有的tag都是shared类型。
* function类型，并且该function是shared声明的。

注意：Error类型不是shared类型。

## Stable类型

stability将sharability扩展到了可变类型。在 actor 中使用 stable 限定符声明的字段类型必须具有stable类型。

以下类型T是stable类型：

* Bool、Char、Text、Bolb、Float、Int、IntN、Nat、NatN、Principal类型。
* Any、None类型。
* actor类型。
* ?T类型，并且T也是stable类型。
* (T0,T1...,Tn)并且所有Ti都是stable类型。
* array类型，是不可变数组或可变数组[var? T]，并且T也是stable类型。
* object类型，所有的字段是可变的或不可变的，并且所有的字段是stable类型。
* variant类型，并且其所有的tag都是stable类型。
* function类型，并且该function是shared声明的。

注意：

* Error类型不是stable类型。
* shared类型必然是stable类型，但是反之不成立。

## Well Formed类型

moto中所有的类型必须是well formed。类型T以及组成类型T的类型满足以下条件，那么类型T是well formed：

* 如果T是async U类型，那么U必须是shared类型。
* 如果T是shared query? U → V类型，那么U必须是shared类型，并且V是单元类型()或者async W类型。
* 如果T是T是type C<X0 <: U0, Xn <: Un> = W声明类型构造的C<V0, …​, Vn>类型，那么Vi必须是Ui的子类型。
* 如果T是actor类型，那么其中的所有字段必定是不可变的，并且function必然是shared函数。

## 类型语法

以下是motoko中完整的类型语法，列出了motoko的所有类型：

|  |
| --- |
| <typ> ::= type expressions  <path> <typ-args>? constructor  <sort>? { <typ-field>;\* } object  { <typ-tag>;\* } variant  { # } empty variant  [ var? <typ> ] array  Null null type  ? <typ> option  <shared>? <typ-params>? <typ> -> <typ> function  async <typ> future  ( ((<id> :)? <typ>),\* ) tuple  Any top  None bottom  Error errors/exceptions  ( type ) parenthesized type  <sort> ::= (actor | module | object)  <shared> ::= shared function type qualifier  shared query?  <typ-field> ::= object type fields  <id> : <typ> immutable  var <id> : <typ> mutable  <id> <typ-params>? <typ1> : <typ2> function (short-hand)  <typ-params> ::= type parameters  < typ-param,\* >  <typ-param>  <id> <: <typ> constrained type parameter  <id> unconstrained type parameter  <typ-args> ::= type arguments  < <typ>,\* >  <typ-tag> ::= variant type fields  # <id> : <typ> tag  # <id> unit tag (short-hand)  <path> ::= paths  <id> type identifier  <path> . <id> projection |

# pattern匹配

patter有以下几种用途：

* 绑定函数参数。
* 标识符绑定声明。
* 在switch的case分支中将值进行分解。

pattern匹配可能成功，也可能失败。

* 在函数参数绑定或者let标识符声明中，如果值无法匹配函数参数pattern或者let pattern，那么将会导致一个陷阱产生。
* 在 switch的case分支中，匹配该case的pattern失败会继续尝试匹配下一个 case，只有当没有这样的 case 时才会产生陷阱。

## 通配符

通配符pattern "\_"用于匹配单个值而不将其内容绑定到任何标识符。

## 标识符

标识符pattern "id"匹配单个值并将其绑定到标识符id。

## 字面量

字面量pattern将单个值与字面量常量值匹配，如果它们不是相等的值，则失败。

## 类型注解

注解pattern "<pat> : <typ>" 将<typ>类型的值与<pat>相匹配。

类型注解pattern不是动态类型检测，主要用于约束 <pat> 中绑定的标识符的类型，例如在函数的参数模式以及let标识符绑定中。

## 元组

元组pattern "( <pat>,\* )" 将n元组值与n元组pattern进行匹配（元组值和元组pattern必须具有相同数量的item）。元组pattern的每个item绑定的标识符必须是不同的。只有所有的item都成功匹配，整个元组pattern才算成功匹配。

空元组pattern ()称为单元pattern。

## Object

对象pattern "{ <pat-field>;\* }" 将对象值(public命名字段值的集合)与命名pattern字段序列进行匹配。对象pattern的每个字段所绑定的标识符必须是不同的。对象pattern中的pattern字段的名称必须不同。

pat-filed有以下两种新式：

* <id> (: <typ>) = <pat>：等号左边指定对象的命名字段。等号右边仍是一个pattern匹配，其与等号左边的字段值进行匹配。
* <id> (: <typ>)：简洁的形式，等价于<id>(: <typ>) = <id>(: <typ>)。

其中：如果不需要使用绑定的标识符，如果无需匹配type，则可以省略typ。因为要使用标识符，编译器需要知道标识符的类型。

例如：

|  |
| --- |
| func fullName({ first : Text; mid : Text; last : Text }) : Text {  first # " " # mid # " " # last  };  fullName(object {  public let first = "1";  public let mid = "2";  public let last = "3";  }) |

## Variant

variant pattern "# <id> <pat>?" 将形式为# <id'> v的variant值与variant pattern进行匹配。如果<pat>省略了，那么是单元pattern (()) 的简写。

如果variant值的标签 <id'> 与pattern的 <id> 不同（即 <id> <> <id'>），则pattern匹配失败。如果标签相等但variant值 v 与pattern <pat>? 不匹配，同样pattern匹配失败。

## Option

option pattern "? <pat>"用于匹配? <typ> option类型的值。

如果option值为null，则匹配失败。如果值为 ? v，那么匹配? <pat>的结果是 v 与 <pat> 匹配的结果。

字面量pattern null可以用于匹配option类型的值是否为null。

## Or

or pattern "<pat1> or <pat2>"是分离匹配pattern。

当一个值与<pat1> or <pat2>匹配时，如果<pat1>匹配成功，则返回<pat1>匹配结果，如果<pat1>匹配失败，则返回<pat2>匹配结果。

注意：<pat1> 和 <pat2> 都不能包含标识符(<id>) pattern，因此成功的匹配总是绑定零标识符。

## pattern匹配语法

以下是motoko中完整的pattern匹配语法，列出了motoko的所有pattern匹配：

|  |
| --- |
| <pat> ::= patterns  \_ wildcard  <id> variable  <unop>? <lit> literal  ( <pat>,\* ) tuple or brackets  { <pat-field>;\* } object pattern  # <id> <pat>? variant pattern  ? <pat> option  <pat> : <typ> type annotation  <pat> or <pat> disjunctive pattern  <pat-field> ::= object pattern fields  <id> (: <typ>) = <pat> field  <id> (: <typ>) punned field |

# 表达式

Motoko程序由声明和表达式构成，有效的程序包含一个actor表达式。

* 声明引入了不可变变量、可变状态、actors, objects, classes和其他类型。
* 表达式描述涉及上述概念的计算。

motoko语言对表达式做了如下限制：

* 单独的表达式是一个声明。
* 如果表达式不是一系列声明的最后一个声明，那么其返回值类型必须为单元类型()。
* 当表达式作为最后一个声明，该表达式的值将作为声明提供的值作为返回值。
* 可以显式的使用ignore关键字，将表达式的返回值类型变为单元类型。

例如：

|  |
| --- |
| let x = 1;  ignore x + 42;  let y = x + 1;  ignore y \* 42;  x \* y + x; |

## 标识符表达式

标识符表达式<id>具有T类型的返回值。

标识符表达式应满足如下前提：

* 类型T为<id>在当前范围内显示的或隐式的定义或推断的。

标识符表达式 <id> 计算为当前环境中绑定到 <id> 的值。

## 字面量表达式

字面量表达式 <lit>具有T类型的返回值。

字面量表达式应满足如下前提：

* 字面量需要在类型 T 的规定值范围内。

字面量表达式 <lit>计算为字面量本身。

## array字面量表达式

array字面量表达式 [ var? <exp>,\* ]具有[var? T]类型的返回值。

array字面量表达式应满足如下前提：

* 对于所有的<exp>表达式应具有T类型的返回值。

array字面量表达式依次计算<expi>得到结果vi，然后返回值[var? v0, …​ , vn]。

## array索引表达式

array索引表达式 <exp1> [ <exp2> ]具有var? T类型的返回值。

array索引表达式应满足如下前提：

* exp1的计算值具有[var? T]的类型。
* exp2的计算值具有Nat类型。
* exp2的计算值v2必须小于exp1计算得到的数组的size-1。

array索引表达式先计算<exp1>得到结果v1，然后计算exp2得到结果v2，然后根据以下规则计算返回值：

* 如果v1是[T]类型，则array索引表达式返回T类型的常量值。
* 如果v1是[var T]类型，则返回值v规则如下：
  + 如果array索引表达式作为赋值表达式的目标出现，则 v 是数组中的第 i 个可变位置；
  + 否则v是数组中的第 i 个位置的值。

## tuple字面量表达式

tuple字面量表达式 (<exp1>, ... , <expn>) 具有元组类型 (T1, ... , Tn)的返回值。

tuple字面量表达式应满足如下前提：

* <exp1>, …​, <expn>具有T1, …​, Tn类型的返回值。

tuple字面量表达式依次计算<expi>得到结果vi，然后返回值(v1, …​ , vn)。

## tuple投影表达式

tuple投影表达式<exp> . <nat>具有Ti类型的返回值。

tuple投影表达式应满足如下前提：

* <exp>返回(T0, ... , Tn-1)类型的值。
* <nat>是Nat类型数值字面量。
* 0<=i<n。

tuple投影表达式计算<exp>得到结果(v0, v1, ...vn)，然后返回值vi。

## object字面量表达式

object字面量表达式 { <exp-field>;\* } 具有object { <typ-field>;\* }类型的返回值。

object字面量也称为记录，其等效于object种类的object声明object { <dec-field>;\* }，类似于在<exp-field>;\*指定的每个字段添加public let前缀，每个var字段添加public前缀。但是与object声明的不同点在于，object字面量表达式只返回object种类的object类型值，不会存在<id>绑定。

## object投影表达式

object投影表达式<exp> . <id>具有var? T类型的返回值。

object投影表达式应满足如下前提：

* <exp>返回sort { var1? <id1> : T1, …​, var? <id> : T, …​, var? <idn> : Tn }类型的值。
* <id>为<idi>中的某一个。

object投影表达式先计算<exp>得到结果v，然后根据以下规则计算返回值：

* 如果<id>字段是T类型，则object投影表达式返回<id>字段对应的T类型的常量值。
* 如果<id>字段是var T类型，则返回值v规则如下：
  + 如果object投影表达式作为赋值表达式的目标出现，则 v 是<id>字段对应可变位置；
  + 否则v是<id>字段对应的值。

## variant字面量表达式

variant字面量表达式# <id> <exp>具有{# id T}类型的返回值。

variant字面量表达式应满足如下前提：

* <exp>具有类型T的返回值。

<exp>可以被省略，表达式# <id>是# <id> ()的语法糖。

variant字面量表达式计算<exp>得到结果v，然后返回{# id T}类型的值# <id> v。

## option字面量表达式

option字面量表达式 ? <exp> 具有? T类型的返回值。

option字面量表达式应满足如下前提：

* <exp>具有类型T的返回值。

option字面量表达式计算<exp>得到结果v，然后返回? T类型的值。

## 迭代器访问表达式

迭代器访问表达式<exp> . <id>具有T类型的返回值。

迭代器访问表达式应满足如下前提：

* <exp>具有类型U。
* U、<id>、T满足如下条件：

|  |  |  |  |
| --- | --- | --- | --- |
| U | <id> | T | Description |
| Text | size | Nat | size (or length) in characters |
| Text | chars | { next: () → Char? } | character iterator, first to last |
|  |  |  |  |
| Blob | size | Nat | size in bytes |
| Blob | vals | { next: () → Nat8? } | byte iterator, first to last |
|  |  |  |  |
| [var? T] | size | Nat | number of elements |
| [var? T] | get | Nat → T | indexed read function |
| [var? T] | keys | { next: () → Nat? } | index iterator, by ascending index |
| [var? T] | vals | { next: () → T? } | value iterator, by ascending index |
| [var T] | put | (Nat, T) → () | indexed write function (mutable arrays only) |

## 一元运算符表达式

[一元运算符](#_一元运算)表达式<unop> <exp>具有T类型的返回值。注意：不包括赋值运算和null break。

一元运算符表达式应满足如下前提：

* <exp>具有T类型的返回值。
* <unop>一元操作符可应用于类型T。

一元运算符表达式先计算<exp>得到结果v，然后再计算<unop> v的值。

## 一元复合赋值运算符表达式

[一元复合赋值运算符](#_赋值运算)表达式<unop> <exp>具有单元类型()的返回值。

一元复合赋值运算符表达式应满足如下前提：

* <exp>具有var T类型的返回值。
* <unop>一元操作符可应用于类型T。

一元复合赋值运算符表达式先计算<exp>得到结果r，然后再计算<unop> r得到结果v，最后将结果v赋值给r。

## 二元运算符表达式

[二元运算符](#_二元运算)表达式<exp1> <binop> <exp2>具有T类型的返回值。注意：不包括赋值运算。

二元运算符表达式应满足如下前提：

* <exp1>具有T类型的返回值。
* <exp2>具有T类型的返回值。
* <binop>二元操作符可应用于类型T。

二元运算符表达式先计算<exp1>得到结果v1，再计算<exp2>得到结果v2，最后再计算v1 <binop> v2的值。

## 二元复合赋值运算符表达式

[二元复合赋值运算符](#_赋值运算_1)表达式<exp1> <binop>= <exp2>具有单元类型()的返回值。

二元复合赋值运算符表达式应满足如下前提：

* <exp1>具有var T类型的返回值。
* <exp2>具有T类型的返回值。
* <binop>二元操作符可应用于类型T。

二元复合赋值运算符表达式先计算<exp1>得到结果r1，先计算<exp2>得到结果r2，然后再计算r1 <binop> r2得到结果v，最后将结果v赋值给r1。

## 关系运算符表达式

[关系运算符](#_关系运算)表达式<exp1> <relop> <exp2>具有bool类型的返回值。

关系运算符表达式应满足如下前提：

* <exp1>具有T类型的返回值。
* <exp2>具有T类型的返回值。
* <relop>比较操作符可应用于类型。

注意：如果relop是"=="或"!="，那么类型T是<exp1>的值类型T1和<exp2>的值类型T2共同具有的[子类型](#_子类型)T，并且T必须是[shared类型](#_Shared类型)。

比较运算符表达式先计算<exp1>得到结果v1，再计算<exp2>得到结果v2，最后再计算v1 <relop> v2的值。

## and表达式

and 表达式 <exp1> and <exp2> 具有 Bool 类型返回值。

and表达式应满足如下前提：

* <exp1>具有Bool类型返回值。
* <exp2>具有Bool类型返回值。

and表达式先计算<exp1>得到结果v1，如果v1为false，则返回false，否则计算<exp2>得到结果v2，最后返回结果v2。

## or表达式

or 表达式 <exp1> or <exp2> 具有 Bool 类型返回值。

or表达式应满足如下前提：

* <exp1>具有Bool类型返回值。
* <exp2>具有Bool类型返回值。

or表达式先计算<exp1>得到结果v1，如果v1为true，则返回true，否则计算<exp2>得到结果v2，最后返回结果v2。

## not表达式

not 表达式 not <exp> 具有 Bool 类型返回值。

not表达式应满足如下前提：

* <exp>具有Bool类型返回值。

not表达式先计算<exp>得到结果v，最后返回结果not v。

## 赋值表达式

赋值表达式<exp1> := <exp2>具有单元类型()的返回值。

赋值表达式应满足如下前提：

* <exp1>具有var T类型的返回值。
* <exp2>具有T类型的返回值。

赋值表达式先计算<exp1>得到结果r1，先计算<exp2>得到结果r2，然后将结果r2赋值给r1位置。

## function表达式

function表达式<shared-pat>? func < X0 <: T0, …​, Xn <: Tn > <pat1> (: U2)? =? <block-or-exp>具有<shared>? < X0 <: T0, …​, Xn <: Tn > U1→ U2类型的返回值。

function表达式应满足如下前提：

* <shared-pat>具有shared query? <pat>的形式，并且当且仅当存在shared query?时才有<shared>。
* <shared-pat>中的<pat>具有上下文类型 { caller : Principal }，用于取出调用者。
* <pat1>的类型为 U1。
* <block-or-exp>具有U2类型的返回值。
* 如果function是shared，那么<pat>和<pat1>必须是穷尽匹配的。
* 如果function是shared，那么<pat1>必须是shared类型，并且U2是单元类型()或者async W类型。
* T0、… 、Tn 和 U2 所有类型都是well-formed且约束良好的。

function表达式计算为一个函数值（又称为闭包），它将函数的代码以及函数调用时需要进行计算的环境绑定一起进行存储。

<shared-pat> 函数本身可能是shared <pat>或shared query <pat>：

* shared <pat>函数可以被远程调用者调用，除非触发陷阱，否则对被调用者的影响将在调用完成后持续存在。
* shared query <pat>函数可以被远程调用者调用，但对被调用者的影响是暂时的，一旦调用完成（无论是值还是错误），就会被丢弃。

函数的环境绑定如果是可变值，那么每次计算时都会获取最新值。例如：

|  |
| --- |
| var i = 1;  func show() : Nat {  i;  };  i := 2;  show(); // 返回值为2 |

## function调用表达式

function调用表达式<exp1> <T0,…​,Tn>? <exp2>具有类型T的返回值。

function调用表达式应满足如下前提：

* 函数<exp1>应该具有<shared>? < X0 <: V0, …​, Xn <: Vn > U1→ U2类型。
* 如果<T0,…​,Tn>?被省略，并且n > 0，那么需要存在最小的 T0, ... , Tn(由编译器推断)，使得Ti <: Vi。
* <exp2>具有[Ti/Xi]U1类型的返回值。
* 返回值类型T == [Ti/Xi]U2。

其中[Ti/Xi]表示类型生成参数Xi的类型为Ti。

function调用表达式先计算<exp1>得到结果r1，再计算<exp2>得到结果r2，r1是"<shared-pat>? func <X0 <: V0, …​, n <: Vn> <pat1> { <exp> }"定义的function，r2是值v2。如果r1的<shared-pat>具有shared query? <pat>格式，那么将{caller = p}与<pat>进行pattern匹配，其中p是函数调用者的Principal(通常是一个user或canister)。然后将v2与<pat1>进行pattern匹配，再执行并返回<exp>的结果。

## return 表达式

return表达式return <exp>?具有类型T的返回值。

return表达式应满足如下前提：

* <exp>如果省略，则等价于return ()。
* <exp>具有类型T的返回值。
* T是return表达式所在的最近的封闭函数的返回类型，（嵌套封闭函数中不能有async表达式），或者async T是return表达式所在的最近的封闭的async表达式的返回类型，（嵌套封闭async表达式中不能有函数）。

return表达式先计算<exp>得到结果r，然后以结果r退出相应的函数调用或完成async表达式。

## async表达式

async表达式async <block-or-exp>具有async T的返回值类型。

async表达式应满足如下前提：

* <block-or-exp>具有类型T的返回值。
* 类型T是[shared类型](#_Shared类型)。

在作用域内针对async <block-or-exp>的控制流标签针对的是async <block-or-exp>而不是<block-or-exp>。也可以定义<block-or-exp>自己的控制流标签。

async表达式执行时，在最近的封闭的actor或顶层actor中将消息排队来执行<block-or-exp>。它立即返回 async T 类型的 future，可以通过await该future来等待<block-or-exp>执行完成，并获取其执行结果。

## await表达式

await表达式await <block-or-exp>具有类型T的返回值。

await表达式应满足如下前提：

* <block-or-exp>具有async T类型的返回值。
* 类型T是[shared类型](#_Shared类型)。
* await表达式只能出现在async表达式中或者在shared函数的主体中。

await表达式执行<block-or-exp>得到结果r，结果r为一个future。如果future尚未完成，await表达式暂停对最近的async表达式或shared函数的执行，将其添加到future的等待队列，一旦future完成，暂停的执行将会继续。如果future 以值 v 完成，挂起的await会以值 v 恢复执行。如果 future 以错误值e(throw)完成，则挂起的await以重新抛出错误e恢复执行。

注意：在await之前的状态变更都会被提交，且无法更改。在暂停和恢复执行之间，封闭actor的状态可能会由于其他传入actor消息的并发处理而改变。防止非同步状态更改是程序员的责任。

## throw表达式

throw表达式throw <exp>具有None类型的返回值。即不返回。

throw表达式应满足如下前提：

* <exp> 具有Error类型的返回值。
* throw表达式只能出现在async表达式中或者在shared函数的主体中。

throw表达式执行<exp>得到结果r，结果r是错误值e，然后从最近的封闭 try <block-or-exp1> catch <pat> <block-or-exp2> 的 catch 子句开始，使用e其模式 <pat> 匹配值。如果没有这样的 try 表达式，则将 e 存储为最近的封闭async表达式或shared函数调用的异步值的错误结果。

## try表达式

try表达式try <block-or-exp1> catch <pat> <block-or-exp2>具有类型T的返回值。

try表达式应满足如下前提：

* <block-or-exp1>具有类型T的返回值。
* <pat>具有类型Error。
* <block-or-exp2>具有类型T的返回值。
* try表达式只能出现在async表达式中或者在shared函数的主体中。

try表达式先执行<block-or-exp1>获得结果r1，如果<block-or-exp1>的求值抛出一个未捕获的错误值 e，则 try 的结果是在 e 与 pat 匹配确定的绑定下对 <block-or-exp2> 求值的结果。

注意：由于Error类型是不透明的，所以模式匹配不能失败(确保<pat>是通配符pattern或标识符pattern）。

## if表达式

if表达式if <exp1> <block-or-exp2> (else <block-or-exp3>)?具有类型T的返回值。

if表达式应满足如下前提：

* <exp1>具有类型Bool的返回值。
* <block-or-exp2>具有类型T的返回值。
* <block-or-exp3>不存在且单元类型()是T的子类型或者<block-or-exp3>存在且具有类型T的返回值。

if表达式先计算<exp1>得到结果r1，如果r1的值为true，则返回<block-or-exp2>的计算结果。如果r1的值为false，如果不存在<block-or-exp3>，则返回单元类型()，如果存在<block-or-exp3>，则返回<block-or-exp3>的计算结果。

## switch表达式

switch表达式switch <exp> { (case <pat> <block-or-exp>;)+ }具有类型T的返回值。

switch表达式应满足如下前提：

* <exp>具有类型U的返回值。
* 对于每一个case分支case <pat> <block-or-exp>，<pat>的类型为U，<block-or-exp>具有类型T的返回值。

switch表达式先计算<exp>得到结果r，在case分支中匹配到第一个<pat>匹配的分支，并返回<block-or-exp>的计算结果。如果不存在case分支匹配r，则会产生陷阱。

## while表达式

while表达式while <exp1> <block-or-exp2>具有单元类型()的返回值。

while表达式应满足如下前提：

* <exp1>具有类型Bool的返回值。
* <block-or-exp2>具有单元类型()的返回值。

while表达式先计算<exp1>得到结果r1，如果r1的值为true，那么计算<block-or-exp2>，再重复开始的步骤，如果r1为false，则返回。

## loop表达式

loop表达式loop <block-or-exp>具有None类型的返回值。即不返回。

loop表达式计算<block-or-exp>，如果发生trap则返回，否则不断重复执行。

## loop while表达式

loop while表达式loop <block-or-exp1> while <exp2>具有单元类型()的返回值。

loop while表达式应满足如下前提：

* <exp2>具有类型Bool的返回值。
* <block-or-exp1>具有单元类型()的返回值。

loop while表达式先计算<block-or-exp1>得到结果r1，再计算<exp2>得到结果r2，如果r2的值为true，那么再计算<block-or-exp1>，重复开始的步骤，如果r2为false，则返回。

## for表达式

for表达式for ( <pat> in <exp1> ) <block-or-exp2>具有单元类型()的返回值。

for表达式应满足如下前提：

* <exp1>具有{ next : () → ?T }类型的返回值。
* <pat>的类型为T。
* <block-or-exp2>具有单元类型的返回值。

for表达式是以下声明的语法糖：

|  |
| --- |
| for ( <pat> in <exp1> ) <block-or-exp2> :=  {  let x = <exp1>;  label l loop {  switch (x.next()) {  case (? <pat>) <block-or-exp2>;  case (null) break l;  }  }  } |

如果 <exp1> 的求值产生陷阱，或x.next() 产生陷阱，或者 x.next() 的值与模式 <pat> 不匹配，或者 <block-or-exp2> 产生陷阱，for 循环都将产生陷阱。

## label表达式

label表达式label <id> (: <typ>)? <block-or-exp>具有T类型的返回值。

label表达式应满足如下前提：

* 如果(: <typ>)省略，则T应该为单元类型()，否则T == <typ>。
* <block-or-exp>具有类型T的返回值。

label表达式的执行结果为<block-or-exp>的执行结果。

## break表达式

break表达式break <id> <exp>?具有None类型的返回值。即不返回。

break表达式应满足如下前提：

* <exp>如果省略，其等价于break <id> ()。
* <exp>具有T类型的返回值。
* <id> 应该是类型为label <id> : T的label标识符。

break表达式先计算<exp>得到结果r，然后放弃计算剩余表达式直至label <id>的封闭处，然后将结果r作为该label表达式的结果。

## continue表达式

continue表达式continue <id>具有None类型的返回值。即不返回。

如果label表达式label <id> (: <typ>)? <exp>中的<exp>是一个循环结构：

* while (exp2) <block-or-exp1>
* loop <block-or-exp1> (while (<exp2>))?
* for (<pat> in <exp2>) <block-or-exp1>

循环的主体 <exp1>被隐式的包含在循环标签label <id\_continue> <exp1>中，因此允许通过continue <id> 表达式(类似<exp1>中执行break <id\_continue>)来提前结束<exp1>执行并继续下一个循环。<id\_continue> 是新的label标识符，只能被 continue <id> 引用。

## null break表达式

null break表达式<exp> !返回类型T或在<exp>值为null时更改控制流。

null break表达式应满足如下前提：

* <exp>具有类型?T的返回值。
* null break表达式只能出现在option块表达式的块中。

null break表达式先计算<exp>得到结果r，如果r的值为null，放弃计算剩余表达式直至option块的封闭处，然后将结果null作为该option块表达式的结果，否则，r的值为?v，返回值v并继续执行。

在嵌套的option块表达式中，null break中断与其最近的封闭option块表达式。

## 块表达式

块表达式{ <dec>;\* }具有类型T的返回值。

块表达式应满足如下前提：

* 最后一个声明的类型为 T。
* 非最后一个声明的类型为()。
* 块中所有的值标识符必须不同。
* 块中所有的类型标识符必须不同。
* 块中所有的标识符的作用域在块内。
* 无法从声明中推断类型的标识符，需要进行类型注解。

块表达式安装程序顺序依次执行声明，如果存在声明产生陷阱，则执行中断产生陷阱，否则返回最后一个声明的值。

## do块表达式

do块表达式do <block>具有类型T的返回值。do块表达式允许在语法不允许直接使用块表达式的位置使用块作为表达式。

do块表达式应满足如下前提：

* <block>具有类型T的返回值。

do块表达式执行<block>，然后返回其值。

例如

|  |
| --- |
| let z = do {  let x = 1;  let y = x + 1;  x \* y + x  }; |

z的值为3。

## option块表达式

option块表达式do ? <block>具有类型?T的返回值。option块表达式可以在其作用域中有效的处理null值(配合null break表达式)。

option块表达式应满足如下前提：

* <block>具有类型T的返回值。

option块表达式执行<block>得到结果r，然后返回值?r。

在嵌套的option块表达式中，null break中断与其最近的封闭option块表达式。

例如：

|  |
| --- |
| let z = do ? {  let x : ?Nat = ?1;  let y = x !;  y  }; |

z的值为?1，类型为?Nat。

## assert表达式

assert表达式assert <block-or-exp>具有单元类型()的返回值。

assert表达式应满足如下前提：

* <block-or-exp>具有Bool类型的返回值。

assert表达式先计算<block-or-exp>得到结果r，r是Boolean值v，如果v是true，则返回值()，否则产生一个陷阱。

## ignore表达式

ignore表达式<block-or-exp>具有单元类型()的返回值。

ignore表达式应满足如下前提：

* <block-or-exp>具有Any类型的返回值。

ignore表达式执行<block-or-exp>，但是会忽略其返回值。

## debug表达式

debug表达式debug <block-or-exp>具有单元类型()的返回值。

debug表达式应满足如下前提：

* <block-or-exp>具有单元类型()的返回值。

当使用（默认）标志 --debug 编译或解释程序, <block-or-exp>中的内容会被编译执行。当使用标志 --release 编译或解释程序时，debug表达式立即返回单元类型值()，<block-or-exp>中的代码永远不会执行。

## parentheses表达式

parentheses表达式 ( <exp> ) 具有类型T的返回值。

parentheses表达式应满足如下前提：

* <exp>具有类型T的返回值。

parentheses表达式的执行结果等价于<exp>的执行结果。

## actor reference表达式

actor reference表达式actor <exp>具有类型T的返回值。

actor reference表达式应满足如下前提：

* actor reference表达式通常在需要 T 类型表达式的上下文中使用，通常是类型注解，类型声明，函数参数。
* T 的类型为actor种类的object类型actor { … }。
* <exp>具有Text类型的返回值。

注意：

* <exp>的执行结果必须是canister标识符的文本格式，否则会产生陷阱。actor reference表达式的结果是表示该canister的actor。
* 容器标识符的有效性以及其类型T的断言保证了承诺和信任。
* 无效的容器标识符或类型可能会在调用actor声明的接口上的函数时显示为稍后的动态失败。
* actor 的参数不应包含用于指定导入的"ic:"。例如使用"lg264-qjkae"而不是"ic:lg264-qjkae"。

## type annotation表达式

type annotation表达式<exp> : <typ>具有类型T的返回值。

type annotation表达式应满足如下前提：

* <typ>是类型T。
* <exp>具有类型U，并且U是T的[子类型](#_子类型)。

当类型检查器不能以其他方式确定 <exp> 的类型时，或者当想要将推断的类型<exp>的U 限制为信息较少的父类型T时，类型注解可用于帮助类型检查器提供 U <：T。

注意：类型注解不存在任何运行时的开销。

## declaration表达式

declaration表达式<dec>具有类型T的返回值。

declaration表达式应满足如下前提：

* <dec>具有类型T的返回值。

declaration表达式计算并返回 <dec> 的结果，但丢弃 <dec> 引入的标识符绑定。

注意：declaration表达式 <dec> 实际上是块表达式 { <dec> } 的简写。

## 表达式语法

以下是motoko中完整的表达式语法，列出了motoko的所有表达式：

|  |
| --- |
| <exp> ::= expressions  <id> variable  <lit> literal  [ var? <exp>,\* ] array  <exp> [ <exp> ] array indexing  ( <exp>,\* ) tuple  <exp> . <nat> tuple projection  { <exp-field>;\* } object  <exp> . <id> object projection/member access  ? <exp> option injection  # id <exp>? variant injection  <unop> <exp> unary operator  <unop>= <exp> unary update operator  <exp> <binop> <exp> binary operator  <exp> <binop>= <exp> binary update operator  <exp> <relop> <exp> binary relational operator  <exp> and <exp> conjunction  <exp> or <exp> disjunction  not <exp> negation  <exp> := <exp> assignment  <shared-pat>? func <func\_exp> function expression  <exp> <typ-args>? <exp> function call  if <exp> <block-or-exp> (else <block-or-exp>)? conditional  switch <exp> { (case <pat> <block-or-exp>;)+ } switch  while <exp> <block-or-exp> while loop  loop <block-or-exp> (while <exp>)? loop  for ( <pat> in <exp> ) <block-or-exp> iteration  label <id> (: <typ>)? <block-or-exp> label  break <id> <exp>? break  continue <id> continue  <exp> ! null break  return <exp>? return  do <block> block as expression  do ? <block> option block  async <block-or-exp> async expression  await <block-or-exp> await future (only in async)  throw <exp> raise an error (only in async)  try <block-or-exp> catch <pat> <block-or-exp> catch an error (only in async)  assert <block-or-exp> assertion  ignore <block-or-exp> ignore value  debug <block-or-exp> debug expression  ( <exp> ) parentheses  actor <exp> actor reference  <exp> : <typ> type annotation  <dec> declaration  <block-or-exp> ::=  <block>  <exp>  <block> ::=  { <dec>;\* }  <pat> ::= patterns  \_ wildcard  <id> variable  <unop>? <lit> literal  ( <pat>,\* ) tuple or brackets  { <pat-field>;\* } object pattern  # <id> <pat>? variant pattern  ? <pat> option  <pat> : <typ> type annotation  <pat> or <pat> disjunctive pattern  <pat-field> ::= object pattern fields  <id> (: <typ>) = <pat> field  <id> (: <typ>) punned field  <exp-field> ::= object expression fields  var? <id> (: <typ>) = <exp> field  ~~var? <id> (: <typ>) punned field //实验后语法貌似不支持，不知道啥情况~~ |

# 声明

Motoko程序由声明和表达式构成，单独的表达式也是声明。Motoko程序由如下声明构成：

* let声明：用于声明一个或者多个不可变变量。
* var声明：用于声明一个可变变量。
* type声明：用于声明一个类型构造器。
* object声明：用于声明一个具有定义的Object类型的object实例/actor实例/module实例。
* function声明：是一个值为函数表达式的let声明或匿名声明的语法糖。
* class声明：是type声明和function声明的语法糖，该type声明了object类型构造器，function是该object类型实例的生成器。
* 表达式声明：会被隐式的转换为let声明。

程序都由一个声明列表组成，每个声明都有一个返回值，声明列表中的最后一个声明的返回值作为程序的返回值，声明列表由显示的声明和隐式的声明(表达式)构成。

## 表达式声明

单独的表达式是一个声明。表达式的值将作为声明返回值。

如果表达式不是一系列声明的最后一个声明，那么其返回值类型必须为单元类型()。

可以显式的使用ignore关键字，将声明返回值类型变为单元类型。

例如：

|  |
| --- |
| 1+1 |

## let声明

let 声明let <pat> = <exp>具有T类型的返回值，并且声明了pattern匹配的标识符绑定。

let声明应满足如下前提：

* <exp> 的类型为 T。
* <pat> 的类型为 T。

motoko将会先计算<exp>的值，并将该值作为声明返回值，然后将值与<pat>进行pattern匹配。<pat>通常为以下几种：

* [标识符pattern](#_标识符)。此时该值会绑定到该标识符上。
* [元组pattern](#_元组)。用于解构元组。
* [对象pattern](#_对象)。用于解构对象。

使用let声明绑定的所有标识符的值都是不可变的。

例如：

|  |
| --- |
| let foo1 = 1;  let foo2: Nat = 1; |

## var声明

var声明 var <id> (: <typ>)? = <exp>具有单元类型()的返回值，并且使用初始值声明了一个可变标识符。

var声明应满足如下前提：

* <exp> 的类型为 T。
* 如果存在类型注解":<typ>"，那么必须T==<typ>。

motoko将会先计算<exp>的值，并将该值作为可变标识符的初始值。var声明的返回值永远是单元类型()。

例如：

|  |
| --- |
| var x1 = 0  var x2 : Nat = 0 |

## type声明

type声明 type <id> <typ-params>? = <typ>具有单元类型()的返回值，并且声明了一个新的类型构造函数 <id>，该类型构造函数带有可选的类型参数 <typ-params> 和类型定义 <typ>。

例如：

|  |
| --- |
| type Person = { first : Text; last : Text };  type List<T> = ?(T, List<T>);  type Fst<T, U> = T;  type Ok<T> = Fst<Any, Ok<T>>; |

## object声明

object声明 <sort> <id>? =? { <dec-field>;\* }具有T类型的返回值，并且声明一个带有可选标识符<id>和零个或多个声明<dec-field>的对象。限定符 <sort>指定对象类型的种类（actor、module 或 object 之一），可以使用public或private声明字段的可见性，如果省略，则默认为private。<dec-field>具有<vis>? <stab>? <dec>的格式，<vis>指定字段的可见类型(private、public或system之一)，<stab>指定actor字段的升级方式(flexible或stable之一)。

object声明应满足如下前提：

* T类型具有如下形式 <sort> { [var0] id0 : T0, …​ , [varn] idn : T0 }
* T类型仅包含"{}"中的所有的public并且具有标识符绑定的声明产生的字段，标识符为字段名称，标识符类型为字段类型。
* 如果T是actor类型，那么T中的所有字段必定是不可变的，并且function必然是shared函数。
* 如果T是module类型，那么"{}"中的所有的声明必须都是[static声明](#_Staic声明)。

motoko将会运行"{}"块中的所有声明， 并将类型T的值作为声明返回值，T类型仅包含"{}"中的所有的public并且具有标识符绑定的声明产生的字段，标识符为字段名称，标识符类型为字段类型。如果存在<id>，则将返回值绑定到该标识符，否则进行空绑定。

默认情况下，actor种类的object声明中字段隐式的声明为flexible，在actor升级后，actor对象中该字段的内容不会被保留，可以通过声明为stable来解决该问题，对于stable修饰的类型字段，该类型必须为[stable类型](#_Stable类型)。注意：stable字段不一定必须public字段。canister升级时，其stable修饰的字段升级过程如下：

* 如果stable修饰字段先前就是stable的，那么新的值从旧的值继承。
* 如果stable修饰字段先前不是stable的，那么执行<dec>初始化其值。

注意：同一个标识符在升级前后如果类型不同，那么旧的类型必须是新的类型的子类型。

在actor种类的object声明中字段还可以使用system进行修饰，system修饰的字段必须拥有以下标识符和类型：

| name | type | description |
| --- | --- | --- |
| preupgrade | () → () | 在旧actor的stable变量转移到新的stable变量前调用 |
| postupgrade | () → () | 在旧actor的stable变量转移到新的stable变量后立即调用 |

这些系统方法提供了保存和恢复非stable类型的数据结构的机会。在升级期间，在对 preupgrade() 或 postupgrade() 的隐式调用中发生的陷阱会导致整个升级产生陷阱，从而保留升级前的actor。

注意：由于 actor 构造是异步的，actor 种类的object声明只能出现在异步上下文 (非query的shared函数或者async表达式) 中。一定要注意actor构造时的异步情况。

例如：

|  |
| --- |
| object {  public object a {    };  object b {    };  }  /\*  类型为  {a : {}}  \*/ |

## function声明

function声明 <shared-pat>? func <id>? <typ-params>? <pat> (: <typ>)? =? <exp> 是函数表达式的let命名声明或匿名表达式声明的语法糖，等价于：

|  |
| --- |
| let <id> = <shared-pat>? func <typ-params>? <pat> (: <typ>)? =? <block-or-exp> // 具有id标识符  <shared-pat>? func <typ-params>? <pat> (: <typ>)? =? <block-or-exp> // 不具有id标识符 |

function声明具有function类型的返回值。

注意：shared function只能作为actor的public字段。

## class声明

class声明<shared-pat>? <sort>? class <id>? <typ-params>? <pat> (: <typ>)? <class-body>是一对type声明和function声明的语法糖，该声明创建一个object类型构造器(type声明)，以及object类型实例构造函数(function声明)：

|  |
| --- |
| type <id> <typ-params> = <sort> { <typ-field>;\* }; // 定义了object类型构造器  <shared-pat>? func <id> <typ-params>? <pat> : async? <id> <typ-args> = async? <sort> <id\_this>? <obj-body> // 定义了object实例生成函数 |

* 当存在<shared-pat>时，需要 <sort> == actor，并且提供 actor 构造函数的调用者的caller的访问。
* <pat> : async? <id> <typ-args>是function的函数签名。<pat>为入参，async? <id> <typ-args>表示返回的object类型。
* async? <sort> <id\_this>? <obj-body>是function的函数定义，对应object实例生成的定义。

class声明具有function类型的返回值。

注意：

* <shared-pat>不得采用shared query <pat>?的形式，构造函数与函数不同，不能是查询函数。
* (: <typ>)如果存在，那么<class-body>定义生成的object类型，必须是<type>的[子类型](#_子类型)，因此类型注解<type>仅用于检查。
* 当<sort>为actor时，具有以下规则：
  + 即使<shared-pat>省略，也默认为shared。
  + <typ-params>必须省略，actor class不允许使用类型参数。
  + <pat>中的类型必须为[shared类型](#_Shared类型)。
  + (: <typ>)如果存在，必须是async T类型，T为actor种类的object类型。因为actor的实例化是异步的。

## Staic声明

static声明旨在无副作用的允许合并重复的库导入（也称为重复数据删除）。

以下声明是static的：

* type声明。
* class声明。
* function声明。
* module和object声明(actor不是)。
* 使用static pattern和static表达式的let声明。
* static表达式。

以下表达式是static表达式：

* 字面量表达式。
* 元素不可变数组。
* 模块表达式。
* 函数表达式。
* static声明。
* 由static表达式构成的tuple。
* 由static表达式构成的object。
* 由static表达式构成的variant。
* 由static表达式构成的option。
* static表达式的字段访问。
* ignore修饰的static表达式。
* 所有表达式都是static的块表达式。
* 带有static表达式的类型注解。

以下pattern是static pattern：

* id标识符。
* 通配符。
* 由static pattern构成的tuple。
* 带有static pattern的类型注解。

## 声明语法

以下是motoko中完整的声明语法，列出了motoko的所有声明：

|  |
| --- |
| <dec> ::= declaration  <exp> expression  let <pat> = <exp> immutable  var <id> (: <typ>)? = <exp> mutable  <sort> <id>? =? <obj-body> object  <shared-pat>? func <id>? <typ-params>? <pat> (: <typ>)? =? <exp> function  type <id> <typ-params>? = <typ> type  <shared-pat>? <sort>? class <id>? <typ-params>? <pat> (: <typ>)? <class-body> class  <sort> ::= (actor | module | object)  <obj-body> ::= object body  { <dec-field>;\* } field declarations  <class-body> ::= class body  = <id>? <obj-body> object body, optionally binding <id> to \_this\_ instance  <obj-body> object body  <shared-pat> ::=  shared query? <pat>?  <dec-field> ::= object declaration fields  <vis>? <stab>? <dec> field  <vis> ::= field visibility  public  private  system  <stab> ::= field stability (actor only)  stable  flexible |

# 语法规范

下面是motoko的完整语法规范：

|  |
| --- |
| <list(X, SEP)> ::=  <empty>  X  X SEP <list(X, SEP)>  <list1(X, SEP)> ::=  X  X SEP <list(X, SEP)>  <obj\_sort> ::=  'object'  'actor'  'module'  <func\_sort\_opt> ::=  <empty>  'shared' 'query'?  'query'  <shared\_pat\_opt> ::=  <empty>  'shared' 'query'? <pat\_plain>?  'query' <pat\_plain>?  <typ\_obj> ::=  '{' <list(<typ\_field>, ';')> '}'  <typ\_variant> ::=  '{' '#' '}'  '{' <list1(<typ\_tag>, ';')> '}'  <typ\_nullary> ::=  '(' <list(<typ\_item>, ',')> ')'  <id> ('.' <id>)\* <typ\_args>?  '[' 'var'? <typ> ']'  <typ\_obj>  <typ\_variant>  <typ\_un> ::=  <typ\_nullary>  '?' <typ\_un>  <typ\_pre> ::=  <typ\_un>  'async' <typ\_pre>  <obj\_sort> <typ\_obj>  <typ> ::=  <typ\_pre>  <func\_sort\_opt> ('<' <list(<typ\_bind>, ',')> '>')? <typ\_un> '->' <typ>  <typ\_item> ::=  <id> ':' <typ>  <typ>  <typ\_args> ::=  '<' <list(<typ>, ',')> '>'  <typ\_field> ::=  'var'? <id> ':' <typ>  <id> ('<' <list(<typ\_bind>, ',')> '>')? <typ\_nullary> ':' <typ>  <typ\_tag> ::=  '#' <id> (':' <typ>)?  <typ\_bind> ::=  <id> '<:' <typ>  <id>  <lit> ::=  'null'  <bool>  <nat>  <float>  <char>  <text>  <unop> ::=  '+'  '-'  '^'  <binop> ::=  '+'  '-'  '\*'  '/'  '%'  '\*\*'  '+%'  '-%'  '\*%'  '\*\*%'  '&'  '|'  '^'  '<<'  ' >>'  '<<>'  '<>>'  '#'  <relop> ::=  '=='  '!='  ' < '  '<='  ' > '  '>='  <unassign> ::=  '+='  '-='  '^='  <binassign> ::=  '+='  '-='  '\*='  '/='  '%='  '\*\*-'  '+%='  '-%='  '\*%='  '\*\*%='  '&='  '|='  '^='  '<<='  '>>='  '<<>='  '<>>='  '@='  <exp\_obj> ::=  '{' <list(<exp\_field>, ';')> '}'  <exp\_plain> ::=  <lit>  '(' <list(<exp>, ',')> ')'  <exp\_nullary> ::=  <exp\_obj>  <exp\_plain>  <id>  <exp\_post> ::=  <exp\_nullary>  '[' 'var'? <list(<exp\_nonvar>, ',')> ']'  <exp\_post> '[' <exp> ']'  <exp\_post> '.'<nat>  <exp\_post> '.' <id>  <exp\_post> ('<' <list(<typ>, ',')> '>')? <exp\_nullary>  <exp\_post> BANG  <exp\_un> ::=  <exp\_post>  '#' <id>  '#' <id> <exp\_nullary>  '?' <exp\_un>  <unop> <exp\_un>  <unassign> <exp\_un>  'actor' <exp\_plain>  'not' <exp\_un>  'debug\_show' <exp\_un>  <exp\_bin> ::=  <exp\_un>  <exp\_bin> <binop> <exp\_bin>  <exp\_bin> <relop> <exp\_bin>  <exp\_bin> 'and' <exp\_bin>  <exp\_bin> 'or' <exp\_bin>  <exp\_bin> ':' <typ>  <exp\_nondec> ::=  <exp\_bin>  <exp\_bin> ':=' <exp>  <exp\_bin> <binassign> <exp>  'return' <exp>?  'async' <exp\_nest>  'await' <exp\_nest>  'assert' <exp\_nest>  'label' <id> (':' <typ>)? <exp\_nest>  'break' <id> <exp\_nullary>?  'continue' <id>  'debug' <exp\_nest>  'if' <exp\_nullary> <exp\_nest>  'if' <exp\_nullary> <exp\_nest> 'else' <exp\_nest>  'try' <exp\_nest> <catch>  'throw' <exp\_nest>  'switch' <exp\_nullary> '{' <list(<case>, ';')> '}'  'while' <exp\_nullary> <exp\_nest>  'loop' <exp\_nest>  'loop' <exp\_nest> 'while' <exp\_nest>  'for' '(' <pat> 'in' <exp> ')' <exp\_nest>  'ignore' <exp\_nest>  'do' <block>  'do' '?' <block>  <exp\_nonvar> ::=  <exp\_nondec>  <dec\_nonvar>  <exp> ::=  <exp\_nonvar>  <dec\_var>  <exp\_nest> ::=  <block>  <exp>  <block> ::=  '{' <list(<dec>, ';')> '}'  <case> ::=  'case' <pat\_nullary> <exp\_nest>  <catch> ::=  'catch' <pat\_nullary> <exp\_nest>  <exp\_field> ::=  'var'? <id> (':' <typ>)?  'var'? <id> (':' <typ>)? '=' <exp>  <dec\_field> ::=  <vis> <stab> <dec>  <vis> ::=  <empty>  'private'  'public'  'system'  <stab> ::=  <empty>  'flexible'  'stable'  <pat\_plain> ::=  '\_'  <id>  <lit>  '(' <list(<pat\_bin>, ',')> ')'  <pat\_nullary> ::=  <pat\_plain>  '{' <list(<pat\_field>, ';')> '}'  <pat\_un> ::=  <pat\_nullary>  '#' <id>  '#' <id> <pat\_nullary>  '?' <pat\_un>  <unop> <lit>  <pat\_bin> ::=  <pat\_un>  <pat\_bin> 'or' <pat\_bin>  <pat\_bin> ':' <typ>  <pat> ::=  <pat\_bin>  <pat\_field> ::=  <id> (':' <typ>)?  <id> (':' <typ>)? '=' <pat>  <dec\_var> ::=  'var' <id> (':' <typ>)? '=' <exp>  <dec\_nonvar> ::=  'let' <pat> '=' <exp>  'type' <id> ('<' <list(<typ\_bind>, ',')> '>')? '=' <typ>  <obj\_sort> <id>? '='? <obj\_body>  <shared\_pat\_opt> 'func' <id>? ('<' <list(<typ\_bind>, ',')> '>')? <pat\_plain> (':' <typ>)? <func\_body>  <shared\_pat\_opt> <obj\_sort>? 'class' <id>? ('<' <list(<typ\_bind>, ',')> '>')? <pat\_plain> (':' <typ>)? <class\_body>  <dec> ::=  <dec\_var>  <dec\_nonvar>  <exp\_nondec>  <func\_body> ::=  '=' <exp>  <block>  <obj\_body> ::=  '{' <list(<dec\_field>, ';')> '}'  <class\_body> ::=  '=' <id>? <obj\_body>  <obj\_body>  <imp> ::=  'import' <id>? '='? <text>  <prog> ::=  <list(<imp>, ';')> <list(<dec>, ';')> |