渐进函数式语言文档

语言概述

渐进函数式语言是一种支持渐进类型系统的函数式编程语言，基于 Lambda 表达式。该语言允许在同一个程序中混合使用动态类型和静态类型，并支持简单的函数返回类型推导。

核心特性

1. 渐进式类型注解

```
;; 完全动态类型
(define add (lambda (x y) (+ x y)))

;; 部分类型注解
(define add (lambda ((x : Int) y) (+ x y)))

;; 完全静态类型
(define add : (Int Int -> Int)
  (lambda ((x : Int) (y : Int)) (+ x y)))
```

2. 简单类型推导

```
(define x 42)          ; 推导为 Int
(define s "hello")     ; 推导为 String
(define f : (Int Int -> ?) (lambda (x y) (+ x y)))  ; 自动推导为 (Int Int -> Int)
```

3. 类型兼容性

```
(define process : (? -> ?)
  (lambda ((x : ?))
    (if (number? x)
        (+ x 1)
        x)))

(process 42)      ; OK: Int 兼容 ?
(process "hello") ; OK: String 兼容 ?
```

4. 联合类型支持

```
(define handle : ((Union Int String) -> String)
  (lambda ((x : (Union Int String)))
    (cond
      [(number? x) "number"]
      [(string? x) "string"])))
```

类型系统

基本类型

类型	描述	示例
Int	整数	`42`, `-10`, `0`
Number	数值类型	`3.14`, `42`
Bool	布尔值	`#t`, `#f`
String	字符串	`"hello"`, `""`
Char	字符	`#\a`, `#\空格`
?	未知类型	任意值

复合类型

函数类型

```
(Int -> Int)                    ; 单参数函数
(Int String -> Bool)            ; 多参数函数
(? -> ?)                        ; 动态函数类型
((Int -> Int) Int -> Int)       ; 高阶函数
```

列表类型

```
(List Int)          ; 整数列表
(List String)       ; 字符串列表
(List ?)            ; 动态元素类型的列表
(List (List Int))   ; 嵌套整数列表
```

配对类型

```
(Pair Int String)   ; 整数与字符串的配对
(Pair ? ?)          ; 动态配对
```

联合类型

```
(Union Int String)          ; 整数或字符串
(Union Int String Bool)     ; 三种类型之一
(Union (List Int) String)   ; 整数列表或字符串
```

类型兼容性规则

1. `?` 与任何类型兼容；
2. `Int` 与 `Number` 可互相兼容；
3. 函数类型支持逆变参数、协变返回值；
4. 复合类型兼容性通过递归结构检查；
5. 联合类型：只要值属于任一成员类型，即视为兼容。

类型检查相关函数

```
type-of     : (? -> String)              ; 获取值的类型字符串
is-type?    : (? Type -> Bool)           ; 判断值是否为某类型
cast        : (? FromType ToType -> ?)   ; 类型强制转换
```

类型谓词

```
number?     : (? -> Bool)
string?     : (? -> Bool)
boolean?    : (? -> Bool)
list?       : (? -> Bool)
pair?       : (? -> Bool)
procedure?  : (? -> Bool)
```

语法规范

程序结构

```
<program>     ::= <definition>* <expression>*
<definition>  ::= <variable-def> | <function-def>
<expression>  ::= <literal> | <variable> | <lambda> | <application>
                | <conditional> | <let-binding> | <type-annotation>
```

类型注解语法

```
(define x : Int 42)                  ; 变量注解
(: expression type)                  ; 表达式类型注解
expression :: type                   ; 内联类型注解
(lambda ((x : Int)) ...)            ; 函数参数注解
```

控制结构语法

```
(if condition then else)

(cond
  [cond1 expr1]
  [cond2 expr2]
  [else default])

(let ([x val1] [y val2]) body)
(let* ([x val1] [y (f x)]) body)
```

简单类型推断

```
;; 若函数返回类型为 (->?) ，解释器将尝试推导返回类型，不能推导保持 (->?)
(define add : (Int -> ?)
  (lambda ((x : Int)) (+ x 10)))

(type-of add) ; 推导结果: (Int -> Int)
```

错误处理

常见错误类型

```
(define add : (Int Int -> Int)
  (lambda ((x : Int) (y : Int)) (+ x y)))

(add "hello" 1)
;; Error: Type mismatch for parameter x: expected Int, got String
```

```
(define f : (Int -> String)
  (lambda ((x : Int)) (+ x 1)))
;; Error: Return type mismatch, expected String, got Int
```

```
(define x : String 123)
;; Error: Type mismatch in definition of x: expected String, got Int
```

```
(cast "hello" Int String)
;; Error: Cast failed: value type doesn't match from-type
```

渐进式函数辅助开发流程示例

第一阶段：快速原型

```
(define process-data
  (lambda (data)
    (map (lambda (item)
           (if (number? item)
               (* item 2)
               item))
         data)))
```

第二阶段：部分类型标注

```
(define process-data
  (lambda ((data : (List ?)))
    (map (lambda (item)
           (if (number? item)
               (* item 2)
               item))
         data)))
```

第三阶段：完全静态化

```
(define process-data : ((List (Union Number String)) -> (List (Union Number
String)))
  (lambda ((data : (List (Union Number String))))
    (map (lambda ((item : (Union Number String)))
           (cond
             [(number? item) (* item 2)]
             [else item]))
         data)))
```

内置函数

算术运算

```
+ : (Number Number -> Number)        ; 加法
- : (Number Number -> Number)        ; 减法
* : (Number Number -> Number)        ; 乘法
/ : (Number Number -> Number)        ; 除法
quotient : (Number Number -> Number) ; 整数除法
modulo : (Number Number -> Number)   ; 求余
```

比较运算

```
= : (Number Number -> Bool)          ; 数值相等
< : (Number Number -> Bool)          ; 小于
> : (Number Number -> Bool)          ; 大于
<= : (Number Number -> Bool)         ; 小于等于
>= : (Number Number -> Bool)         ; 大于等于
```

逻辑运算

```
and : (Bool Bool -> Bool)            ; 逻辑与
or : (Bool Bool -> Bool)             ; 逻辑或
not : (Bool -> Bool)                 ; 逻辑非
```

字符串操作

```
string-append : (String String -> String)    ; 字符串连接
string-length : (String -> Int)              ; 字符串长度
substring : (String Int Int -> String)       ; 子字符串
string=? : (String String -> Bool)           ; 字符串相等
```

列表操作

```
car : ((List ?) -> ?)                ; 取列表第一个元素
cdr : ((List ?) -> (List ?))         ; 取列表剩余部分
cons : (? (List ?) -> (List ?))      ; 构造列表
list : (? * -> (List ?))             ; 创建列表
null? : (? -> Bool)                  ; 判断是否为空列表
length : ((List ?) -> Int)           ; 列表长度
append : ((List ?) (List ?) -> (List ?))  ; 列表连接
```

类型检查函数

```
type-of : (? -> String)              ; 获取值的类型
is-type? : (? Type -> Bool)          ; 检查值是否为指定类型
cast : (? Type Type -> ?)            ; 类型转换
```

类型谓词

```
number? : (? -> Bool)                ; 是否为数字
string? : (? -> Bool)                ; 是否为字符串
boolean? : (? -> Bool)               ; 是否为布尔值
list? : (? -> Bool)                  ; 是否为列表
pair? : (? -> Bool)                  ; 是否为配对
procedure? : (? -> Bool)             ; 是否为函数
```