文档编号： 0.1
日期： 2024-04-03

编程语言 X

目录

目录 ii

表格列表 iv

图片列表 v

1 词法约定 1
- 1.1 注释 . . . 1
- 1.2 标识符 . . . 1
- 1.3 关键字 . . . 1
- 1.4 运算符 . . . 2
- 1.5 字面量 . . . 2

2 基本概念 6
- 2.1 作用域 . . . 6
- 2.2 名称查找 . . . 6

3 类型与修饰符 7
- 3.1 类型、值和对象 . . . 7
- 3.2 修饰符 . . . 10

4 表达式 11
- 4.1 基本表达式 . . . 12
- 4.2 后缀运算符 . . . 13
- 4.3 加法运算符 . . . 13
- 4.4 顶层运算符 . . . 13
- 4.5 分号运算符 . . . 14

5 语句 15
- 5.1 块 . . . 15
- 5.2 绑定语句 . . . 15
- 5.3 `if` 语句 . . . 16
- 5.4 `match` 语句 . . . 16
- 5.5 `while` 语句 . . . 17
- 5.6 `for` 语句 . . . 17
- 5.7 控制语句 . . . 17

6 模式匹配 **18**
6.1 空模式 . . . 18
6.2 表达式模式 . . . 18
6.3 数组模式 . . . 18
6.4 元组模式 . . . 19
6.5 对象模式 . . . 19
6.6 绑定模式 . . . 19
6.7 类型断言 . . . 20
6.8 包含断言 . . . 21
6.9 条件断言 . . . 21

7 声明 **22**
7.1 类型声明 . . . 22
7.2 类声明 . . . 22

8 函数 **23**

9 概念 **24**

10 类 **25**

11 枚举 **26**

12 属性与修饰符 **28**
12.1 `noreturn` . . . 28

13 访问控制 **29**

14 `core` 库介绍 **30**

15 序列库 **31**

索引 **32**

语法产生式索引 **33**

表格列表

1 关键字 2

图片列表

1 词法约定 [lex]

1 程序文本指将被翻译为 *X* 程序的文本的整体或者一部分。它存储在源文件中。

1.1 注释 [lex.comment]

1 有两种形式的注释：以/* 开始，*/结束的块注释和以// 开始，到行末结束的行注释。注释内部的 /* 或 // 等不具有特殊含义。注释在将程序文本分割为标记以后和空白一起被删除。

1.2 标识符 [lex.identifier]

Identifier:

NormalIdentifier

LambdaIdentifier

NormalIdentifier:

*IdentifierHead IdentifierTail**

IdentifierHead:

Unicode(Lu, Ll, Lt, Lm, Lo, Nl)

_

Alnum:

a 至 z

A 至 Z

Digit

IdentifierTail:

IdentifierHead

Unicode(Mn, Mc, Nd, Pc, Cf)

LambdaIdentifier:

$ *Digit+*

$ *Identifier*

1 任意长的由字母和数字组成的序列是标识符。大写和小写字母被认为是不同的。每个字符都是有效的。

2 以 $ 开始后接十进制数字或标识符的序列也是标识符。这些标识符只能在 lambda 作用域中使用。

1.3 关键字 [lex.keyword]

1 表 1 中的标识符被保留做关键字；此外，以 __ 开始的标识符被保留作为关键字。只有在某些特定语法结构中的关键字可以作为标识符使用。其他情况下的关键字不能作为标识符。

表 1 — 关键字

`_`	`bitand`	`bitand_eq`	`bitor`	`bitor_eq`
`bool`	`clone`	`concept`	`div`	`div_eq`
`else`	`false`	`float`	`if`	`int`
`let`	`lvalue`	`mod`	`mod_eq`	`mut`
`move`	`nil`	`return`	`scope`	`string`
`switch`	`then`	`this`	`throw`	`throws`
`true`	`type`	`typeof`	`never`	`var`
`void`	`xor`	`xor_eq`		

1.4 运算符 [lex.op]

Operator:

`` ` ? *RawOperator* ``

RawOperator: one of

CustomOperator `, ; : ( ) [ ] { }`

CustomOperator:

CustomOperatorPart+

CustomOperatorPart: one of

`~ ! # % & * - | + = / ? < > .`

[1] `.`, `...` 和 `=` 被保留不能重载为运算符。

1.5 字面量 [lex.literal]

Literal:

IntegerLiteral Suffix?
FloatingLiteral Suffix?
StringLiteral Suffix?
SymbolLiteral

1.5.1 整数字面量 [literal.integer]

IntegerLiteral:

DecimalLiteral
BinaryLiteral
HexadecimalLiteral

DecimalLiteral:

*Digit (' ? Digit)**

Digit: one of

`0 1 2 3 4 5 6 7 8 9`

BinaryLiteral:

`0b` *BinaryDigit* (`'` *?* *BinaryDigit*)*

BinaryDigit:

`0`

`1`

HexadecimalLiteral:

`0x` *HexadecimalDigit* (`'` *?* *HexadecimalDigit*)* `0X` *HexadecimalDigit* (`'` *?* *HexadecimalDigit*)*

HexadecimalDigit: one of

`0 1 2 3 4 5 6 7 8 9`

`A B C D E F`

`a b c d e f`

1 整数字面量由一系列数字构成。其中的单引号用作分隔并且不影响字面量的值。字面量的前缀用于指示它的进制。最左边的数字具有最大的权重值。十进制字面量由若干十进制数字构成；十六进制字面量由前缀 `0x` 或 `0X` 后跟若干十六进制数字构成；二进制字面量前缀 `0b` 后跟若干二进制数字构成。*X* 不支持八进制字面量。

1.5.2 浮点字面量 [literal.floating]

1 浮点字面量用于表示浮点数。与很多其他语言不同的是，浮点字面量的小数点前后不允许省略数字。

1.5.3 字符串字面量 [literal.string]

StringLiteral:

`"` *Schar** `"`

`@"` *Rchar** `"`

`$"` *SIchar** `"`

`$@"` *RIchar** `"`

*Mdelim Mchar** *Mdelim*

Schar:

除了`\` 和`"` 以外的非换行可打印字符

EscapeSeq

Rchar:

除了`"` 以外的非换行可打印字符 `""`

SIchar:

除了\ 、" 、{ 和} 以外的非换行可打印字符

EscapeSeq

{ *Expression* }

{{

}}

RIchar:

除了" 、{ 和} 以外的非换行可打印字符

""

{ *Expression* }

{{

}}

Mdelim:

""" "*

Mchar:

任意可打印字符

1 有三种类型的字符串字面量：普通字符串字面量、原始字符串字面量、多行字符串字面量。*普通字符串字面量*使用反斜杠开始的转义序列表示其他字符。*原始字符串字面量*中所有可打印字符将会表示这个字符本身，除了"" 表示单个双引号的转义序列之外。这两类字符串字面量不允许包含换行符。

2 上述两类字符串字面量可以前加 $ 表示插值字符串。插值字符串中的 {*e*} 序列会被解释为一个字符串插值，其中 *e* 为表达式。该表达式将被求值后转换为字符串插入当前位置。

3 *多行字符串字面量*表示横跨多行的字符串。它以任意数量但不少于三个的" 开始，并以等量的" 结束。每一行包含换行符都属于该字符串字面量，但是除了以下字符：

(3.1) — 开头引号序列之后紧邻的换行会被删除。

(3.2) — 如果结尾引号序列所在行之前全都是空白字符，则这些字符连同上一行的换行会被删除。

(3.3) — 如果每一行的空白字符都以结尾引号序列之前的空白字符为前缀，则这些字符都会被删除。

如果结尾行包含空白字符且之前的某一行的空白字符不以这些空白字符为前缀，则这是一个编译错误。

[例：

如下字面量为合法的多行字符串字面量：

```
"""
abc
"""
```

其等价于"abc"。

]

1.5.4 符号字面量 [literal.symbol]

SymbolLiteral:

' *NormalIdentifier*

符号字面量后跟的标识符可以与关键字相同。

1.5.5 字面量后缀 [literal.suffix]

Suffix:

Alnums

整数、浮点数与字符串能带有后缀，指示不同的类别。这些后缀可能是内建的或是用户自定义的。

2 基本概念 [basic]

1 实体包括对象、函数、类型、模块、运算符、扩展。

2.1 作用域 [scope]

1 作用域是一段程序文本。当一个名称的完整声明结束后，将这个名称插入到最小的、它具有的作用域之中。这意味着，若无特别说明，在这之后到那个作用域结束之前，这个名称可以引用被声明的实体。

2.1.1 语句作用域 [scope.stmt]

1 每一个语句都是作用域。变量、函数和类型具有语句作用域。

2.1.2 模块作用域 [scope.module]

2.1.3 类型作用域 [scope.type]

2.1.4 枚举作用域 [scope.enum]

2.1.5 Lambda 作用域 [scope.lambda]

1 只有 lambda 参数具有 lambda 作用域。Lambda 作用域是 lambda 表达式（4.1.2）的 *LambdaBody* 部分；或者函数调用运算符（**??**）的 *Block* 部分。

2 Lambda 参数在 lambda 作用域的任意位置都可以使用，但只有在 lambda 没有形参声明的时候才能使用。形如 `$`*n* 的 lambda 参数指代该 lambda 的第 *n* 个未命名形参；形如 `$`*id* 的 lambda 参数指代该 lambda 的名称为 *id* 的命名形参。

2.1.6 序列作用域 [scope.sequence]

1 只有 `$` 具有序列作用域。序列作用域是具有形式 `r[...]` 或 `r(...)` 或 `r.m(...)` 的 `...` 部分，其中 `r` 实现了 `core.Sequence`。

2 在序列作用域中，`$` 在任意位置都可以使用并且等价于 `r.Size`。

2.2 名称查找 [name.lookup]

1 名称查找用于解析一个 *IDExpr* 具体指代的实体。

3 类型与修饰符 [typequal]

3.1 类型、值和对象 [type]

Type:

FundaType

SpecialType

CompType

$$\mathscr{V} = \{\langle T, Q, v\rangle \mid T \in \mathscr{T}, Q \subset \mathbb{Q}, v \in T\}$$

1 类型是一个有限集合。修饰符是一个标识符。值是类型、修饰符集合和这个类型的成员的三元组。[注：实际上，值并不由这三个项完全表征。例如 `lvalue` 表示有与这个值相关联的对象，这个对象不能由这个值确定。] T 称为值 v 的类型。

3.1.1 基本类型 [type.funda]

FundaType:

`void`

`never`

`bool`

`int`

`uint`

`int <` *Expression* `,` *Expression* `>`

`float`

`float <` *Expression* `>`

1 `void` 标识只有唯一一个值的类型。

$$\texttt{void} \coloneqq \{\text{void}\}$$

2 `never` 标识没有值的类型。

$$\texttt{never} \coloneqq \{\text{never}\}$$

3 `bool` 标识布尔值。

$$\texttt{bool} \coloneqq \{\text{true}, \text{false}\}$$

4 `int` 称作整数类型。

$$\texttt{int} \coloneqq \{x \in \mathfrak{Z} \mid l \leq x \leq h\}$$

其中 l 和 h 为待推导常数。在本规范中，带有特定 l 和 h 的 `int` 类型将记作 $\texttt{int}_{l,h}$。如果 $l = h$，则记作 $\texttt{int}_{l}$。存在实现定义的常数 m 和 M。l 和 h 须满足

$$l \geq m$$

$$h \leq M$$

$$0 \leq h - l \leq M$$

`uint` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l >= 0$。

`byte` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l >= 0$ 且 $h <= 255$。

5 `float` 称作浮点类型。

$$\texttt{float}^{*} \subset \mathfrak{R}$$

$$\texttt{float}^{\dagger} \subset \{+\infty, -\infty, \mathrm{NaN}\}$$

$$\texttt{float} \coloneqq \texttt{float}^{*} \cup \texttt{float}^{\dagger}$$

`float32` 与 `float` 类似，但其可用的值范围更小。

3.1.2 特殊类型 [type.special]

SpecialType:

`self`

3.1.3 复合类型 [type.comp]

CompType:

Type `?`

Type `[` *Type* `]` `(` *UnnamedTypes* `)`

`(` *NamedTypes* `)`

`(` *UnnamedTypes* `,` *NamedTypes* `)`

`{` *UnnamedTypes* `}` *Type* `*`

UnnamedTypes:

Type

UnnamedTypes `,` *Type*

NamedTypes:
Identifier : *Type*
NamedTypes , *Identifier* : *Type*

1 `T?` 为可空类型。

$$T\texttt{?} \coloneqq \{\langle t\rangle \mid t \in T\} \cup \{\text{nil}\}$$

2 `T[]` 为数组类型。

$$T\texttt{[]} \coloneqq \bigcup_{n=0}^{\infty} T^n$$

3 `T[U]` 为字典类型。

$$T\texttt{[}U\texttt{]} \coloneqq T^U$$

4 `(T1, …, Tm)` 称作元组类型。

$$(T_1\texttt{,}\ldots\texttt{,}T_m\texttt{,}) \coloneqq \prod_{i=1}^{m} T_i$$

5 `T1|…|Tn` 称作联合类型。其中各个 $\texttt{T}_i$ 是无序的。重复的 $\texttt{T}_i$ 将被视为一个。

$$T_1\texttt{|}\ldots\texttt{|}T_n \coloneqq \bigcup_{i=1}^{n}\{\langle T_i, t_i\rangle \mid t_i \in T_i\}$$

3.1.4 公共类型 [type.common]

1 存在函数

$$\otimes : \mathcal{T} \times \mathcal{T} \to \mathcal{T} \cup \{*\}$$

满足交换律。如果对于类型 T_1 和 T_2，$T_1 \otimes T_2 \neq *$，称 $T_1 \otimes T_2$ 为 T_1 和 T_2 的公共类型。

2 函数 $\otimes$ 按照如下顺序确定：

(2.1) — $T \otimes T \coloneqq T$

(2.2) — $\texttt{never} \otimes T \coloneqq T$

(2.3) — $\texttt{void} \otimes T \coloneqq *$

(2.4) — $\texttt{int}_{l_1,h_1} \otimes \texttt{int}_{l_2,h_2} \coloneqq \texttt{int}_{\min\{l_1,l_2\},\max\{h_1,h_2\}}$

(2.5) — $\texttt{int}_{l,h} \otimes \texttt{float} \coloneqq \texttt{float}$

(2.6) — $\texttt{int}_{l,h} \otimes \texttt{float32} \coloneqq \texttt{float32}$

(2.7) — $\texttt{float} \otimes \texttt{float32} \coloneqq \texttt{float}$

(2.8) — $T \otimes T\texttt{?} \coloneqq T\texttt{?}$

(2.9) — $T \otimes T \,|\, T_1 \,|\, \ldots \,|\, T_{n-1} \coloneqq T \,|\, T_1 \,|\, \ldots \,|\, T_{n-1}$

(2.10) — $T \otimes T_1 \,|\, T_2 \,|\, \ldots \,|\, T_n \coloneqq T \,|\, T_1 \,|\, \ldots \,|\, T_n$

(2.11) — $T \otimes U \coloneqq T \,|\, U$

3.2 修饰符 [qualifier]

3.2.1 `mut` [qual.mut]

1 `mut` 表示该值是可变的。

4 表达式 [expr]

Expression:
PrimaryExpr
Operator Expression
Expression Operator
Expression Operator Expression

1 表达式由运算符与操作数按照顺序组合在一起，它表示一个计算过程。运算符是若干标点符号的组合、一个标识符或是语言规定的特殊结构。运算符可以是前缀、后缀或者是中缀的，这决定了运算符与其操作数的结合方式。运算符组是运算符的集合，每一个运算符都属于某个运算符组。运算符组之间有一个弱偏序关系，决定了它们的优先级。完全由中缀运算符构成的运算符组有结合性：左结合的组每个运算符从左到右选择操作数；右结合的组从右到左；无结合的组两个运算符不能选择同一个操作数。

2 4.2 到 4.5 各节按照优先级从高到低依次对运算符组进行描述。若无特别说明，每一节描述一个运算符组。

3 解析表达式按照如下顺序：

(3.1) — 构造基本表达式、运算符和 ID 表达式。

(3.2) — 将 . 和后面的 ID 表达式结合为一个后缀运算符。

(3.3) — 将每个重载为运算符的标识符[1]替换为运算符。

(3.4) — 对于每组连续的基本表达式，如果除了第一个以外都形如 `(...)`、`[...]` 或 `{...}`，将它们替换为后缀运算符；否则这是一个编译错误。[注：因为块也是基本表达式，它也会被替换为后缀运算符。然后如果有的话，它会与之前的函数调用运算符结合成一个函数调用运算符。]

(3.5) — 确定每个运算符是前缀、后缀或中缀的。对于每组连续的运算符，如果存在一个运算符满足：

(3.5.1) — 它要么是 ；要么两边都有空白；

(3.5.2) — 它可以作为中缀运算符；

(3.5.3) — 它前面的所有运算符都可以作为后缀运算符；[注：由上文已经限定为前缀运算符的运算符在这里不能作为后缀运算符；反之亦然]

(3.5.4) — 它后面的所有运算符都可以作为前缀运算符。

那么这个运算符是中缀运算符，这个运算符之前的所有运算符是后缀运算符，这个运算符之后的所有运算符是前缀运算符。如果存在多于一个或者不存在这样的运算符，这是一个编译错误。

1) 这里指本身作为 ID 表达式的而不是作为 ID 表达式的一部分的标识符。

(3.6) — 运算符按照优先级选择操作数。对于优先级最高（可能不止一个）的运算符组：

(3.6.1) — 每个运算符选择各自的操作数。如果不同的组之间选择了相同的操作数，这是一个编译错误。如果选择的表达式不是基本表达式也不是优先级高于这个运算符组的运算符构成的表达式，这是一个编译错误。

(3.6.2) — 如果这个组由前缀、中缀和后缀混合而成或是无结合的中缀运算符组，选择相同的操作数的运算符引发一个错误；

(3.6.3) — 如果这个组是前缀组或者是右结合的中缀组，每个运算符从右到左把自己和操作数替换为一个子表达式；

(3.6.4) — 如果这个组是后缀组或者是左结合的中缀组，每个运算符从左到右把自己和操作数替换为一个子表达式。

(3.7) — 如果最后仍然剩余大于一个子表达式，这是一个编译错误。

$$\triangleright : \mathscr{E} \times \Omega \to (\mathscr{V} \cup \mathscr{V}^\dagger \cup \{*\}) \times \Omega$$

4 $\mathrm{e} \triangleright \omega$ 称作*在环境 ω 下对 e 求值*。设 $\mathrm{e} \triangleright \omega = \langle v, \omega' \rangle$。如果 $v \in \mathscr{V}$，称*求值正常结束*，v 是 e 的*值*；否则，称*求值以抛出 v 异常结束*，e 的值为 `never()`。$v = *$ 意味着求值过程中程序终止了。若无特别说明，对表达式 e 的子表达式 e_0 求值以抛出 v 异常结束也会导致对 e 的求值以抛出 v 异常结束。

5 ω 是求值之前的环境，ω' 是求值之后的环境。如果 $\omega = \omega'$，称 e *是无副作用的*；如果 e 是无副作用的且 v 与 ω 无关，称 e *是纯的*。

6 下文的数学定义式中，$\coloneqq$ 右边的 e 表示对 e 求值之后的 v；$\otimes e$ 表示求值后的环境；$\triangleright e$ 表示求值的结果。如果表达式是无副作用的，将 ω 部分省略。

7 *丢弃表达式 e 的结果*指，在决定 e 的类型时，直接将它确定为 `never` 而跳过所有步骤；在对 e 求值时，进行所有步骤，但是如果求值正常结束，丢弃最后的值。

4.1 基本表达式 [expr.primary]

PrimaryExpr:
LiteralExpression
Identifier
(*Expression*)
LambdaExpr

$$(e) \triangleright \omega \coloneqq e \triangleright \omega$$

1 括号包起的表达式与其内部的表达式完全等价。

4.1.1 字面量表达式 [expr.lit]

LiteralExpression:
    IntegerLiteral
    FloatLiteral
    StringLiteral
    `true`
    `false`
    `nil`
    `()`

1 字面量本身是基本表达式。

2 整数字面量 i 的值为 i，类型为 $\texttt{int}_i$。如果 i 超出了能够选择的范围，这是一个编译错误。

3 浮点字面量 f 的值为最接近 f 的浮点数值，类型为 `float`。如果 f 太大，则其值为 `float.Infinity`；如果 f 太小，其值为 `0`。

4 字符串字面量 s 的值为对应的字符串，类型为 `string`。

5 布尔字面量的类型为 `bool`。`true` 和 `false` 分别对应其值。

6 `nil` 的类型为 T`?`，其中 T 为待推导的类型参数。

7 `()` 的类型为 `void`。

4.1.2 Lambda 表达式 [expr.lambda]

LambdaExpr:
    SimpleLambdaExpr
    CompoundLambdaExpr

SimpleLambdaExpr:
    LambdaParameter`?` `=>` *LambdaBody*

LambdaParameter:

4.2 后缀运算符 [expr.suffix]

4.3 加法运算符 [expr.add]

1 运算符 `+` 和 `-` 分别表示加法和减法。

4.4 顶层运算符 [expr.toplvl]

1 顶层运算符的结果类型都是 `void`。

4.5 分号运算符 [expr.semi]

SemicolonExpression:

Expression ; *Expression*

Binding ; *Expression*

$$x\ ;\ y \rhd \omega \coloneqq y \rhd \otimes\mathrm{discard}(x)$$

1 分号表达式中，分号左侧可以为一个表达式或绑定。如果分号左侧为绑定，则该绑定会被插入到当前作用域中。如果左侧为表达式，则该表达式将被求值且结果会被丢弃。在那之后，将对右侧表达式进行求值并将其值作为整个表达式的值。

5 语句 [stmt]

Statement:
Block
BindingStmt
IfStmt
SwitchStmt
AssertStmt

1 语句是一类特殊的基本表达式。在语句中的子块将优先作为语句的一部分而不是其中的表达式的一部分。[例：

```
// 错误: { true } 被认为是 if 语句的第一个子块，而不是它的条件表达式的一部分
if x.filter{ true }
```

]

5.1 块 [stmt.block]

Block:
`{` *BlockItem** `}`

BlockItem:
Expression `;` `?`
BlockDecl

1 块是由大括号包裹的一系列声明和表达式的序列。块定义了一个块作用域。块的求值按照顺序进行，整个语句的值是最后一个项目的值。所有不是最后一项的表达式项的值被丢弃；这些表达式必须以；结尾。如果最后一个项目是一个声明，这个块的类型为 `void`。

5.2 绑定语句 [stmt.bind]

BindingStmt:
Binding `;`

Binding:
Pattern `=` *Expression* `;`

1 绑定形如 p `=` e，其中 p 是包含至少一个绑定模式的模式。

2 绑定语句将一个绑定插入当前作用域中。该绑定必须不能失败。

5.3 `if` 语句 [stmt.if]

IfStmt:

`if` *Condition* `then` *Expression*
`if` *Condition* `then` *Expression* `else` *Expression*
`if` *Condition Block*
`if` *Condition Block* `else` *Block*

Condition:

Expression
Binding

1 条件可以为任意对象声明后跟一个表达式或模式匹配。如果条件为表达式 e[2]，那么这个表达式的类型必须实现 `core.Boolean`。条件成立当且仅当 e 求值为真（**??**）。如果条件是绑定，那条件成立当且仅当绑定成功。该绑定必须可以失败。

2 将 `if` 语句的第一个表达式记作 T，第二个表达式（如果有）记作 F。对 `if` 语句的求值按以下顺序进行：

(2.1) — 如果条件成立，对 T 求值，然后将它的值作为整个语句的值。

(2.2) — 否则，如果存在 F，那么对它求值，然后将它的值作为整个语句的值。

(2.3) — 否则，整个语句的值为 `()`。

只有一个表达式会被求值。整个语句的类型是 T 和 F 的公共类型（如果 F 不存在的话视为 `void`）。

5.4 `match` 语句 [stmt.match]

MatchStmt:

`match` *Expression MatchBlock*

MatchBlock:

`{` *BlockItem* MatchItem** `}`

MatchItem:

Matcher Expression+

Matcher:

Pattern `->`

1 *`match` 语句*对其后跟的表达式进行模式匹配。整个语句的类型为每个匹配项表达式类型的公共类型。

2 `match` 语句的各项中的模式必须覆盖被匹配表达式的所有可能值，否则这是一个编译错误。

3 对 `match` 语句的求值将按如下顺序进行：

(3.1) — 如果语句匹配块之前有项，执行这些项。他们的作用域是整个块。

2) 因为赋值表达式的类型是 `void`，形如 `p = e` 的程序文本将始终被看做一个模式匹配而不是赋值表达式。

(3.2) — 按出现顺序对每个项进行匹配。如果某个项的模式匹配成功，则对其后的表达式进行求值，将其作为整个 match 表达式的值。所有其他项的表达式都不会进行求值。

5.5 `while` 语句 [stmt.while]

WhileStmt:

`while` *Expression? Block*

1 *`while` 语句*处理循环，其中的表达式必须实现 `core.Boolean`。整个语句的类型为 `void`。

2 `while` 语句每次循环都会对控制表达式进行求值。如果求值为真，则继续循环，否则终止循环。如果表达式被省略，则等价于表达式为 `true`。

5.6 `for` 语句 [stmt.for]

ForStmt:

`for` *Pattern* `:` *Expression Block*

1 *`for` 语句*进行明确的范围循环。形如 `for` p `:` e B 的 `for` 语句需满足:e 实现了 `core.Sequence;typeof(`e`).Item` 匹配 p 不会失败，否则这是一个编译错误。p 中注入的变量在整个 `for` 语句的范围内生效。整个语句的类型为 `void`。

5.7 控制语句 [stmt.control]

BreakStmt:

`break ;`

ContinueStmt:

`continue ;`

ReturnStmt:

`return` *Expression?* `;`

1 控制语句包括 `break` 语句、`continue` 语句和 `return` 语句。

2 `break` 语句只能在 `while` 或 `for` 语句中使用。它终止最内层的循环语句。

3 `continue` 语句只能在 `while` 或 `for` 语句中使用。它终止最内侧循环语句的本次循环。

4 `return` 语句只能在函数块中使用。它中止函数块的执行，并将后跟的表达式作为整个函数的返回值。如果表达式被省略，则等价于 `()`。

6 模式匹配 [pattern]

Pattern:

*PatternBody PatternAssterion**

PatternBody:

NullPattern
ExprPattern
BindPattern
ArrayPattern
TuplePattern
ObjectPattern

PatternAssterion: TypeAssertion IncludeAssertion CondAssertion

1 模式匹配用于检验一个值是否符合特定的模式，以及在符合特定的模式时从中提取某些成分。值符合特定的模式称为这个值匹配这个模式。本节中，v 表示待匹配的值，p 表示待匹配的模式。

2 模式 p 由模式主体和模式断言构成。模式主体规定匹配的结构与操作，模式断言则对值的特征进行断言。一个主体可以带有任意数量的断言。

6.1 空模式 [pattern.null]

NullPattern:

_

1 空模式能够匹配任意值。匹配成功后，v 的值将被丢弃。

6.2 表达式模式 [pattern.expr]

ExprPattern:

Expression

1 表达式模式中的表达式必须实现了 `core.Equtable`。v 匹配表达式模式 p 当且仅当 v==p。

6.3 数组模式 [pattern.array]

ArrayPattern:

`[` *AnyPattern (`,` AnyPattern)** `]`

AnyPattern:
    Pattern
    ...

1 数组模式匹配序列中的元素。其中`...`项（称作任意项模式）只能出现至多一次，否则这是一个编译错误。v 必须实现 `core.Sequence`，否则这是一个编译错误。

1. 如果模式不包含任意项，且 v`.size` 与模式中项的数量不相等，则匹配失败。
2. 如果模式包含任意项，且 v`.size` 小于模式中非任意项的数量，则匹配失败。

2 在那之后，将按如下规则依次对 v 的元素进行匹配。如果每个匹配都成功，则整个模式 p 匹配 v。

1. 对任意项模式之前的模式（如果不存在任意项则对每个子模式），p_i 匹配 v`[`i`]`，其中 i 是子模式的索引（从 0 开始）。
2. 对任意项模式之后的模式，p_r 匹配 v`[$-`r`]`，其中 r 是子模式从后向前数的索引（从 0 开始）。

6.4 元组模式 [pattern.type]

TuplePattern:
    `(` *AnyPattern (*`,` *AnyPattern)** `)`

1 与数组模式类似，元组模式匹配元组。

6.5 对象模式 [pattern.object]

ObjectPattern
    `{` *ObjectPatternBody* `}`

ObjectPatternBody
    ObjectItem (`,` *ObjectItem)**

ObjectItem
    Identifier `:` *Pattern*

1 对象模式对对象进行匹配。如果对于每个对 (k, p) 而言，$v.k$ 匹配 p 都成立，则整个模式匹配成功。

2 与数组和元组匹配不同，对象匹配是开放的，即 v 可以包含未在模式中列出的项。

6.6 绑定模式 [pattern.bind]

BindPattern:
    `var` *PatternBind TypeNotation?*
    `let` *PatternBind TypeNotation?*

PatternBind:
Identifier
ArrayPatternBind
TuplePatternBind
ObjectPatternBind

ArrayPatternBind:
`[` *AnyPatternBind (*, *AnyPatternBind)** `]`

TuplePatternBind:
`(` *AnyPatternBind (*, *AnyPatternBind)** `)`

AnyPatternBind:
PatternBind
`...`
NullPattern
ExprPattern

ObjectPatternBind
`{` *ObjectPatternBodyBind* `}`

ObjectPatternBodyBind
ObjectItemBind (, *ObjectItemBind)**

ObjectItemBind
Identifier `:` *PatternBind*

1 绑定模式可以匹配任意值。匹配成功后，该标识符将作为一个变量插入到当前作用域中。如果绑定使用的是 `var`，则该变量具有 `mut` 修饰。

2 绑定模式可以使用简写：`let [a, b]` 等价于 `[let a, let b]`；`let [i, _]` 等价于 `[let i, _]`。

6.7 类型断言 [pattern.type]

TypeAssertion:
`is` *Type*
`:` *Type*
`as` *Type*

1 类型断言对值的类型进行约束。它包括以下类型：

(1.1) — `is T` 要求值的类型与 `T` 完全一致。

(1.2) — `: T` 要求值的类型是 `T` 的子类型。

(1.3) — `as T` 要求值的类型能够转换到 `T`，无论显式或隐式。

6.8 包含断言 [pattern.include]

IncludeAssertion:

`in` *Expression*

1 包含断言要求值包含在某个集合 e 中。如果 v `!in` e，则匹配失败。

6.9 条件断言 [pattern.cond]

CondAssertion:

`if` *Expression*

1 条件断言要求值满足某个条件。

7 声明 [decl]

Declaration:

BlockDecl

BlockDecl:

FuncDecl

TypeDecl

ClassDecl

EnumDecl

7.1 类型声明 [decl.type]

TypeDecl:

TypeQual`*` `type` *Identifier TypeBody*

TypeQual:

TypeBody:

ObjectType `=` *Type*

7.2 类声明 [decl.class]

8 函数 [func]

FuncDecl:

FuncQual`*` `func` *FuncName ParamList ReturnType*`?` *Block*

FuncName:

Identifier

`init`

`operator` *Operator*

9 概念 [concept]

10 类 [class]

11 枚举 [enum]

EnumDecl:
`enum` *Identifier EnumBaseType?* `{` *Enumerators* `}`

EnumBaseType:
`:` *Type*

Enumerators:
Enumerator `(` `,` *Enumerator* `)*` `,` `?`

Enumerator:
Attribute? Identifier EnumeratorTail?

EnumeratorTail:
`=` *Expression*
`[` *Type* `]`
TupleType
ObjectType

1 枚举类型用来表示一组孤立值，在其定义中使用枚举符表示。枚举符还可以带有参数，以表示同一枚举符下的一系列值。

[例：

```
enum E {
    A,
    B(int),
    C[int],
    D{ name: string }
}
```

上述代码定义了一个枚举类型 `E`，它包含四个枚举符，可以以如下方式访问：`E.A`、`E.B(0)`、`E.C[1, 2, 3]` 及 `E.D  name: "Hello"` 。]

2 枚举类型可以指定基底类型 B，也可以为其枚举符指定值，这类枚举类型称为*传统枚举类型*。如果一个传统枚举类型没有显式指定基底类型，则 B 为 `int`。B 必须实现 `core.Equtable`。

3 在传统枚举类型中，每个枚举符都有对应的值。其确定如下：

(3.1) — 如果该枚举符被指定值，则其值为被指定的表达式隐式转换到 B 的结果；

(3.2) — 否则，如果该枚举符是第一个值，则其值为 B`.init();`

(3.3) — 否则，假设该枚举符的前一个值为 v，则其值为 v`+?`。

每个枚举符都可以显示转换为对应的基底类型。

12 属性与修饰符 [attr]

12.1 `noreturn` [attr.noreturn]

1 `noreturn` 修饰的函数将不会返回。函数的返回类型必须省略或者是 `never`。

13 访问控制 [access]

1 实体 x 可以访问实体 y 意味着，x 的声明中指代 y 的 *IDExpr*（无论是显式还是隐式）是合法的。[3]

2 变量、函数、类型、模块的声明 e 可以包含以下的访问控制符：

(2.1) — `public`，任意实体 f 都可以访问它；

(2.2) — `internal`，只有与 e 处于同一个包中的实体才能访问它；

(2.3) — `private`，只有与 e 定义在同一个文件中的实体才可以访问它。

块作用域的声明不能具有访问控制符。如果一个非块作用域的声明不包含访问控制符，那么

(2.4) — 如果它对应的 *UnqualID* 不是标识符，它是 `public` 的；

(2.5) — 如果它的标识符以下划线开头，那么它是 `private` 的；否则

(2.6) — 它是 `public` 的。

扩展不能具有访问控制符（因为它不能被使用第二次）。声明块的声明控制符应用到其内部的声明。

3) 某些情况可能不存在这样的 *IDExpr*，这并不意味着 x 能访问或不能访问 y。

14 core 库介绍 [core]

1 core 库是唯一与语言相互作用的库。实现了解 core 库的所有组件的接口以及（需要的话）内部细节。一个实现必须提供 core 库。

15 序列库 [core.seq]

1 `core.seq` 库包含所有与序列相关的库。

索引

作用域, 6
    lambda, 6
    序列, 6
    枚举, 6
    模块, 6
    类型, 6
    语句, 6
修饰符, 10, 28
    mut, 10

函数, 23
名称查找, 6
声明, 22
    类, 22
    类型, 22
属性, 28
    noreturn, 28
库, 30
    序列库, 31

模式匹配, 18
    元组, 19
    包含断言, 21
    对象, 19
    数组, 18
    条件断言, 21
    空, 18
    类型断言, 20
    绑定, 19
    表达式, 18
求值, 12

类型, 7
    公共类型, 9
    基本类型, 7
    复合类型, 8
    特殊类型, 8

表达式, 11
    Lambda, 13
    基本, 12
    字面量, 13
访问控制, 29
语句, 15
    for, 17
    if, 16
    match, 16
    while, 17
    块, 15
    控制, 17
    绑定, 15
运算符
    分号, 14

语法产生式索引

Alnum, 1
AnyPattern, 19
AnyPatternBind, 20
ArrayPattern, 18
ArrayPatternBind, 20

BinaryDigit, 3
BinaryLiteral, 3
Binding, 15
BindingStmt, 15
BindPattern, 19
Block, 15
BlockDecl, 22
BlockItem, 15
BreakStmt, 17

CompType, 8
CondAssertion, 21
Condition, 16
ContinueStmt, 17
CustomOperator, 2
CustomOperatorPart, 2

DecimalLiteral, 2
Declaration, 22
Digit, 2

EnumBaseType, 26
EnumDecl, 26
Enumerator, 26
Enumerators, 26
EnumeratorTail, 26
Expression, 11
ExprPattern, 18

ForStmt, 17

FuncDecl, 23
FuncName, 23
FundaType, 7

HexadecimalDigit, 3
HexadecimalLiteral, 3

Identifier, 1
IdentifierHead, 1
IdentifierTail, 1
IfStmt, 16
IncludeAssertion, 21
IntegerLiteral, 2

LambdaExpr, 13
LambdaIdentifier, 1
LambdaParameter, 13
Literal, 2
LiteralExpression, 13

MatchBlock, 16
Matcher, 16
MatchItem, 16
MatchStmt, 16
Mchar, 4
Mdelim, 4

NamedTypes, 9
NormalIdentifier, 1
NullPattern, 18

Operator, 2

Pattern, 18
PatternAssterion, 18
PatternBind, 20
PatternBody, 18

PrimaryExpr, 12

RawOperator, 2
Rchar, 3
ReturnStmt, 17
RIchar, 4

Schar, 3
SemicolonExpression, 14
SIchar, 4
SimpleLambdaExpr, 13
SpecialType, 8
Statement, 15
StringLiteral, 3
Suffix, 5
SymbolLiteral, 5

TuplePattern, 19
TuplePatternBind, 20
Type, 7
TypeAssertion, 20
TypeBody, 22
TypeDecl, 22
TypeQual, 22

UnnamedTypes, 8

WhileStmt, 17