文档编号: 0.1
日期: 2024-04-07

编程语言 X

目录

表格列表

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，并以 UTF-8 编码读取。

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*。注释可以嵌套。在将程序文本分割为标记以后，注释和空白一起被删除。

1.2 标识符 [lex.identifier]

Identifier:
*IdentifierHead IdentifierTail**

IdentifierHead:
Unicode(XID_Start)
-

IdentifierTail:
IdentifierHead
Unicode(XID_Continue)

1 *标识符*以具有 Unicode XID_Start 属性的字符或_开始，后跟零个或数个具有 Unicode XID_Continue 属性的字符，但不能与关键字相同。标识符区分大小写。

1.3 关键字 [lex.keyword]

1 表 1 中的标识符被保留做关键字。关键字不能作为标识符使用。

表 1 — 关键字

_	async	await	bool	cmp
const	deinit	else	enum	false
float	for	if	init	int
let	match	mut	never	nil
return	self	shl	shl_eq	shr
shr_rq	string	then	this	throw
true	type	typeof	uint	var
void	while			

2 表 2 中的标识符称为*上下文关键字*。在特定的语法结构中它将被解析为关键字，在其他位置可以当作一般标识符使用。

表 2 — 上下文关键字

then

1.4 运算符 [lex.op]

Operator: 以下之一

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

CustomOperator:

CustomOperatorPart+

CustomOperatorPart: 以下之一

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

[1] `.`, `...` 和 `=` 被保留不能重载为运算符。

1.5 字面量 [lex.literal]

Literal:

IntegerLiteral

FloatingLiteral

StringLiteral

SymbolLiteral

BooleanLiteral

1.5.1 整数字面量 [literal.integer]

IntegerLiteral:

DecimalLiteral Suffix?

BinaryLiteral Suffix?

HexadecimalLiteral Suffix?

DecimalLiteral:

Digits

Digits:

Digit

Digits `_?` *Digit*

Digit: 以下之一

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

BinaryLiteral:

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

`0B` *BinaryDigit* *(*`_?` *BinaryDigit)**

BinaryDigit:

`0`

`1`

HexadecimalLiteral:

HexadecimalPrefix HexadecimalDigits

HexadecimalPrefix: 以下之一

`0x` `0X`

HexadecimalDigits:

HexadecimalDigit

HexadecimalDigits `_*` *HexadecimalDigit*

HexadecimalDigit: 以下之一

`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。其中的下划线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。**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` 或 `0B` 后跟若干二进制数字构成。*X* 不支持八进制字面量。

[2] 整数字面量的值为其数字序列表示的值，依不同前缀分别为十进制、十六进制或二进制。最左侧的数字为最高位。字面量的类型参见表格 3，其中 i 为其字面值：

表 3 — 整数字面量后缀

后缀	对应的类型	后缀	对应的类型
无	`int`$_i$	`u`	`uint`$_i$
`i8`	`int<8>`$_i$	`u8`	`uint<8>`$_i$
`i16`	`int<16>`$_i$	`u16`	`uint<16>`$_i$
`i32`	`int<32>`$_i$	`u32`	`uint<32>`$_i$
`i64`	`int<64>`$_i$	`u64`	`uint<64>`$_i$
`f` 或 `f64`	`float<64>`	`f32`	`float<32>`

如果字面量的字面值超出了其类型的约束范围，则这是一个编译错误。

1.5.2 浮点字面量 [literal.floating]

FloatingLiteral:

DecimalFloatingLiteral Suffix?

HexadecimalFloatingLiteral Suffix?

DecimalFloatingLiteral:

Digits `.` *Digits ExponentPart?*

Digits ExponentPart

HexadecimalFloatingLiteral:

HexadecimalPrefix HexadecimalDigits `.` *HexadecimalDigits BinaryExponentPart?*

HexadecimalPrefix HexadecimalDigits BinaryExponentPart

ExponentPart:

`e` *Sign?* *Digits*

`E` *Sign?* *Digits*

BinaryExponentPart:

`p` *Sign?* *Digits*

`P` *Sign?* *Digits*

Sign: 以下之一

`+ -`

[1] 浮点字面量用于表示浮点数, 其中的下划线用作分隔并且不影响字面量的值。浮点字面量的小数点前后不允许省略数字。

[2] 浮点字面量的类型按表 4 确定。其值依如下方式确定：如果它包含指数部分，则命 e 为指数部分按十进制数字解析得到的数；否则，e 为 0。对十进制浮点字面量而言，命 s 为除去指数的部分按十进制数字解析得到的数，则令 $f = s \times 10^e$。对十六进制浮点字面量而言，命 s 为除去指数的部分按十六进制数字解析得到的数，则令 $f = s \times 2^e$。浮点字面量的值为其类型中最接近 f 的值。如果 f 太大，则值为对应的正无限大；如果 f 太小，则值为 0。

表 4 — 浮点字面量后缀

后缀	对应的类型
无或 `f64`	`float<64>`
`f32`	`float<32>`

1.5.3 字符串字面量 [literal.string]

StringLiteral:

`"` *Schar** `"` *Suffix?*

`@"` *Rchar** `"` *Suffix?*

`$"` *SIchar** `"` *Suffix?*

`$@"` *RIchar** `"` *Suffix?*

Mdelim *Mchar** *Suffix?* *Mdelim*

Schar:

除了\和" 以外的非换行可打印字符

EscapeSeq

EscapeSeq:

`\` *SimpleEscape*

`\u{` *HexadecimalDigit+* `}`

SimpleEscape: 以下之一

`' " ? \a b f n r t v`

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 字符串字面量表示 UTF-8 字符串，其类型为 `string`。它有三种种类：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。插值字符串中除了原有的转义序列外，还有 {{和}} 作为单个大括号的转义序列。

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:
`'` *Identifier*

1 符号字面量后跟的标识符可以与关键字相同。

1.5.5 布尔字面量 [literal.boolen]

BooleanLiteral:
`true`
`false`

$$\texttt{true} \coloneqq \langle \text{true}, \texttt{bool} \rangle$$

$$\texttt{false} \coloneqq \langle \text{false}, \texttt{bool} \rangle$$

1 布尔字面量的类型为 `bool`。`true` 和 `false` 分别对应其真值与假值。

1.5.6 字面量后缀 [literal.suffix]

Suffix:
`_?` *SuffixIdentifier*

SuffixIdentifier:
*SuffixIdentifierHead SuffixIdentifierTail**

SuffixIdentifierHead:
Unicode(XID_Start)

SuffixIdentifierTail:
SuffixIdentifierHead
Unicode(XID_Continue)

1 整数、浮点数与字符串能带有内建或用户自定义的后缀。后缀由字母开始，后跟任意数量的字母或数字，字面量与后缀之间可以添加_分隔。用户自定义的后缀不能与内建后缀相同，否则这是一个编译错误。

2 假设字面量 l 带有自定义后缀 s。如果 l 是整数字面量，则它等价于以下第一个合法的表达式：

```
operator ""\var{s}($l$)
```

其中 i 是 l 移除后缀的形式；

```
operator ""$s$($t$)
```

其中 t 是包含 i 所有字符的字符串字面量。

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 类型系统 [typesystem]

3.1 类型、值和对象 [type]

Type:
FundaType
SpecialType
CompType

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

1 类型是一个集合。值是类型和这个类型的成员的元组。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* `>`
`uint <` *Expression* `>`
`float`
`float <` *Expression* `>`

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

1 `void` 标识只有唯一一个值的类型。

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

2 `never` 标识没有值的类型。

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

3 `bool` 标识具有真或假两个值的类型。

$$\texttt{int}_{l,h} \coloneqq \{x \in \mathfrak{X} \mid l \le x \le h\}$$

4 $\texttt{int}_{l,h}$ 称作*整数类型*，其中 l 和 h 为待推导常数。在本规范中，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 $l \geq 0$。

5 `int<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \geq -2^{w-1}$ 且 $h \leq 2^{w-1} - 1$。`uint<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \geq 0$ 且 $h \leq 2^w - 1$。其中 w 可以取 8、16、32、64 或 128。它们称作*定长整数类型*，表示长度固定的整数。只能在定长整数类型上进行位运算。

$$\texttt{float<}s\texttt{>}^* \subset \mathscr{R}$$

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

$$\texttt{float<}s\texttt{>} \coloneqq \texttt{float}^* \cup \texttt{float}^\dagger$$

6 `float<`s`>` 称作*浮点类型*。其中 s 为 32 或 64。`float` 为 `float<64>` 的别名。

7 整数类型、定长整数类型和浮点类型称为*算术类型*。

3.1.2 特殊类型 [type.special]

SpecialType:
`self`

1 `self` 用于在方法或概念中指示类型自身。

3.1.3 复合类型 [type.comp]

CompType:
Type `?`
Type `[ ]`
Type `[` *Type* `]`
`(` *TupleTypes*`*` `)`
`{` *ObjectTypes* `}`
`(` *TupleTypes*`*` `)` `->` *Type*
UnionType

TupleTypes:
Type
TupleTypes `,` *Type*

ObjectTypes:
ObjectType
ObjectTypes `,` *ObjectType*

ObjectType:

*ObjectTypeQualifier** *Identifier* `:` *Type*

ObjectTypeQualifier:

`let`

`var`

UnionType:

Type `|` *Type*

UnionType `|` *Type*

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

1 T`?` 为可空类型。

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

2 T`[]` 为数组类型。

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

3 T`[`K`]` 为字典类型。

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

$$\texttt{()} \coloneqq \texttt{void}$$

4 `(`T_1`,`$\ldots$`,`T_n`)` 称作元组类型。特别地，只有一个元素的元组与其内部元素类型等价；没有元素的元组与 `void` 等价。

$$\texttt{\{}K_1\texttt{:}T_1\texttt{,}\ldots\texttt{,}K_n\texttt{:}T_n\texttt{\}} \coloneqq \prod_{i=1}^{n} T_i$$

5 `{`K_1`:`T_1`,`$\ldots$`,`K_n`:`T_n`}` 称作对象类型。

6 `(`T_1`,`$\ldots$`,`T_n`)` `->` R 称作函数类型。

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

7 T_1`|`$\ldots$`|`T_n 称作联合类型。其中各个 T_i 是无序的。相同的 T_i 将被视为一个。

3.1.4 具名类型 [type.named]

NamedType:
EntityID
FundaType
SpecialType

3.2 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) — $\texttt{void} \otimes T \coloneqq \texttt{void}$

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

(2.6) — $\texttt{float<}s_1\texttt{>} \otimes \texttt{float<}s_2\texttt{>} \coloneqq \texttt{float<}\max\{s_1, s_2\}\texttt{>}$

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

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

(2.9) — $T \otimes T_1\texttt{|}T_2\texttt{|}\ldots\texttt{|}T_n \coloneqq T\texttt{|}T_1\texttt{|}\ldots\texttt{|}T_n$

(2.10) — $T \otimes U \coloneqq T\texttt{|}U$

3.3 修饰符 [qualifier]

3.3.1 `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 4.2 到 4.13 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。

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

2 $\mathrm{e} \triangleright \omega$ 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*。$v = *$ 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。

3 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。

4 下文的数学定义式中，$:=$ 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。

5 对含有子表达式的表达式求值时，总是先对其子表达式按出现次序从左到右求值。

6 *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
LambdaParameter
(*Expression*)
LambdaExpr

$$(e) := e$$

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*
- *FloatingLiteral*
- *StringLiteral*
- *BooleanLiteral*
- `nil`
- *ArrayLiteral*
- *TupleLiteral*
- *ObjectLiteral*
- *DictLiteral*

ArrayLiteral:
- `[` *ExprList?* `]`

TupleLiteral:
- `(` *TupleExprList?* `)`

ExprList:
- *ExprItem*
- *ExprList* `,` *ExprItem*

TupleExprList:
- *ExprItem* `,` *ExprItem*
- *TupleExprList* `,` *ExprItem*

ExprItem:
- *Expression*
- `...` *Expression*

ObjectLiteral:
- `{` *ObjectItems* `}`

ObjectItems:
- *ObjectItem*
- *ObjectItems* `,` *ObjectItem*

ObjectItem:
- *Identifier* `:` *Expression*
- `...` *Expression*

DictLiteral:
- `{|` *DictItems?* `|}`

DictItems:
- *DictItem*
- *DictItems* `,` *DictItem*

DictItem:

Expression `:` *Expression*

`...` *Expression*

1 字面量本身是基本表达式。整数字面量、浮点字面量、字符串字面量和布尔字面量的值分别参见1.5.1、1.5.2、1.5.3和**??**节的定义。

2 `nil` 的类型为 T`?`，其中 T 为待推导的类型参数。

3 数组字面量 `[`e_1`,`...`,`e_n`]` 表示一个显式写出其各元素的数组值。其类型为 T`[]`，其中 T 为各表达式的公共类型。如果其中包含形如 `...`e 的项，则视同将 e 的各元素显式插入在该位置。e 必须实现 `core.Sequence`。如果数组字面量不包含任何成员，则 T 是一个待推导的类型。

4 元组字面量 `(`e_1`,`...`,`e_n`)` 表示一个显式写出其各元素的元组值。其类型为 `(`T_1`,`...`,`T_n`)`，其中 T_i 为 e_i 的类型。如果其中包含形如 `...`e 的项，则视同将 e 的各元素显式插入在该位置。e 必须也是一个元组。特别的，`()` 的类型为 `void`，是 `void` 的唯一值。

5 对象字面量 `{`k_1`:`e_1`,`...`,`x_n`:`e_n`}` 表示一个显式写出其各元素的对象值。其类型为 `{`k_1`:`T_1`,`...`,`x_n`:`T_n`}`，其中 T_i 为 e_i 的类型。如果其中包含形如 `...`e 的项，则视同将 e 的各成员以相同标签显式插入在该位置。e 必须是对象类型。[`注:` 对象字面量必须至少包含一个键值对。`{}` 将被解析为一个块。]

6 字典字面量 `{|`k_1`:`v_1`,`k_2`:`v_2`,`...`,`k_n`:`v_n`|}` 表示一个显式写出其各元素的字典值。其类型为 T`[`K`]`，其中 T 为各 v_i 的类型，K 为各 k_i 的公共类型。如果其中包含形如 `...`e 的项，则视同将 e 的各元素对显式插入在该位置。e 必须是字典类型。如果字典字面量不包含任何类型，则 T 和 K 是待推导的类型。

4.1.2 Lambda 表达式 [expr.lambda]

LambdaExpr:

FuncQual`*` *LambdaParameter* *ReturnType*`*` `=>` *LambdaBody*

LambdaParameter:

ParamDecl

LambdaBody:

Expression

4.1.3 Lambda 参数 [expr.lambda-param]

LambdaParameter:

`$` *Digit*`+`

`$` *Identifier*

1 Lambda 参数是一个特殊的标识符，用于在 Lambda 表达式中引用参数。Lambda 参数的类型是待推导的。

4.2 后缀运算符 [expr.suffix]

SuffixExpr:

PrimaryExpr

SuffixExpr `[` *ExprList?* `]`

SuffixExpr `(` *ExprList?* `)` *Block**

SuffixExpr `.` *Identifier*

SuffixExpr `.` *IntegerLiteral*

SuffixExpr `.` `init`

SuffixExpr `.` `deinit`

SuffixExpr `.` `await`

SuffixExpr `as` *Type*

SuffixExpr `+?`

SuffixExpr `-?`

4.2.1 下标运算符 [expr.sub]

[1]

4.3 前缀运算符 [expr.prefix]

PrefixExpr:

SuffixExpr

`+` *PrefixExpr*

`-` *PrefixExpr*

[1] 前缀运算符 `+` 和 `-` 分别表示正号和负号。其中 `+` 的值为其操作数的值，而-的值为其相反数。操作数类型必须为算术类型。

4.4 乘法运算符 [expr.mul]

MulExpr:

PrefixExpr

MulExpr `*` *PrefixExpr*

MulExpr `/` *PrefixExpr*

MulExpr `%` *PrefixExpr*

[1] 运算符 `*`、`/`和`%` 分别表示乘法、除法和余数。乘除法只对整数类型进行溢出检查，而不对定长整数类型和浮点类型进行。除零检测对整数类型和定长整数类型都生效。

4.5 加法运算符 [expr.add]

AddExpr:

MulExpr

AddExpr `+` *MulExpr*

AddExpr `-` *MulExpr*

[1] 运算符 `+` 和 `-` 分别表示加法和减法。其操作必须为算术类型。加减法只对整数类型进行溢出检查，而不对定长整数类型和浮点类型进行。

4.6 移位运算符 [expr.shift]

ShiftExpr:
AddExpr
ShiftExpr `shl` *AddExpr*
ShiftExpr `shr` *AddExpr*

[1] 运算符 `shl` 和 `shr` 表示按位左移和右移。其操作数必须为定长整数类型。在同一个表达式中混合使用 `shl` 和 `shr` 是一个编译错误。

4.7 位运算符 [expr.bit]

BitwiseExpr:
ShiftExpr
BitwiseExpr `'&` *ShiftExpr*
BitwiseExpr `'^` *ShiftExpr*
BitwiseExpr `'|` *ShiftExpr*

[1] 运算符`'&`、`'^` 和`'|` 分别表示按位与、按位异或和按位或。其操作数必须为定长整数类型。在同一个表达式中混合使用`'&`、`'^` 和`'|` 是一个编译错误。

4.8 区间运算符 [expr.range]

RangeExpr:
BitwiseExpr
BitwiseExpr `..` *BitwiseExpr* *BitwiseExpr* `..=` *BitwiseExpr*

[1] 运算符`..` 用于生成左闭右开区间。运算符`..=` 用于形成闭区间。

4.9 连接运算符 [expr.connect]

ConnectExpr:
RangeExpr
ConnectExpr `~` *RangeExpr*

[1] 运算符`~`用于连接字符串或集合。其操作数的类型必须满足以下条件之一：

(1.1) — 两个操作数都是 `string`；

(1.2) — 一个操作数满足 `core.Sequence<`T`>`，另一个是 T；

(1.3) — 两个操作数都满足 `core.Sequence<`T`>`。

对第一种情况，结果等于将两个字符串左右连接得到的结果；对第二种情况，x `~` y 等于 `[`x`, ...`y`]` 或 `[...`x`,` y`]`，取决于哪个操作数是序列；对第三种情况，x `~` y 等于 `[...`x`, ...`y`]`。

4.10 比较运算符和包含运算符 [expr.cmp-in]

CompareExpr:

RangeExpr

CompareExpr `==` *RangeExpr*

CompareExpr `!=` *RangeExpr*

CompareExpr `<` *RangeExpr*

CompareExpr `!<` *RangeExpr*

CompareExpr `>` *RangeExpr*

CompareExpr `!>` *RangeExpr*

CompareExpr `<=` *RangeExpr*

CompareExpr `>=` *RangeExpr*

CompareExpr `<>` *RangeExpr*

CompareExpr `cmp` *RangeExpr*

CompareExpr `in` *RangeExpr*

CompareExpr `! in` *RangeExpr*

$$
\begin{aligned}
a \texttt{ == } b &\iff a \texttt{ cmp } b = \texttt{.equal} \\
a \texttt{ != } b &\iff a \texttt{ cmp } b \neq \texttt{.equal} \\
a \texttt{ < } b &\iff a \texttt{ cmp } b = \texttt{.less} \\
a \texttt{ !< } b &\iff a \texttt{ cmp } b \neq \texttt{.less} \\
a \texttt{ > } b &\iff a \texttt{ cmp } b = \texttt{.greater} \\
a \texttt{ !> } b &\iff a \texttt{ cmp } b \neq \texttt{.greater} \\
a \texttt{ <= } b &\iff a \texttt{ cmp } b = \texttt{.less} \text{ or } \texttt{.equal} \\
a \texttt{ >= } b &\iff a \texttt{ cmp } b = \texttt{.greater} \text{ or } \texttt{.equal} \\
a \texttt{ <> } b &\iff a \texttt{ cmp } b = \texttt{.less} \text{ or } \texttt{.greater}
\end{aligned}
$$

[1] a `cmp` b 比较两个表达式，其结果类型为 `core.Order`。其余比较运算符的结果类型为 `bool`。

[2] `<`、`<=` 和 `==` 可以连续使用。a `<` b `<=` c 等价于 a `<` b `&` b `<=` c。`>`、`>=` 和 `==` 也可以用类似方式混合。以其他方式在一个表达式中使用超过一个比较运算符是一个编译错误。

[3] a `in` b 检测 a 是否在 b 中。a `!in` b 检测 a 是否不在 b 中。b 必须实现 `core.Sequence`。

4.11 逻辑运算符 [expr.logic]

LogicExpr:

CompareExpr

LogicExpr `&` *CompareExpr*

LogicExpr `|` *CompareExpr*

1 `&` 和 `|` 是逻辑运算符。两者的操作数都必须实现 `core.Boolean`。它们都使用短路求值。在同一个表达式中混合使用两个运算符是一个编译错误。

4.12 赋值运算符 [expr.assign]

AssignExpr:

LogicExpr

SuffixExpr `=` *LogicExpr*

SuffixExpr `+=` *LogicExpr*

SuffixExpr `-=` *LogicExpr*

SuffixExpr `*=` *LogicExpr*

SuffixExpr `/=` *LogicExpr*

SuffixExpr `%=` *LogicExpr*

SuffixExpr `shl_eq` *LogicExpr*

SuffixExpr `shr_eq` *LogicExpr*

SuffixExpr `'&=` *LogicExpr*

SuffixExpr `'^=` *LogicExpr*

SuffixExpr `'|=` *LogicExpr*

SuffixExpr `++`

SuffixExpr `--`

SuffixExpr `<~` *LogicExpr*

LogicExpr `~>` *SuffixExpr*

1 赋值表达式的结果类型是 `void`。

2 `=` 将左操作数的值更新为右操作数的值。左操作数必须是可变的，且右操作数必须能隐式转换到左操作数。

3 复合赋值运算符 `+=`、`-=`、`*=`、`/=`、`%=`、`shl_eq`、`shr_eq`、`'&=`、`'^=` 和`'|=` 分别表示加、减、乘、除、取余、左移、右移、按位与、按位异或和按位或赋值。对这些运算符而言，a *op*`=` b 或 a *op*`_eq` b 等价于 a `=` a *op* b，但 a 只被求值一次。

4 自增运算符 e`++` 等价于 e `=` e`+?`，自减运算符 e`--` 等价于 e `=` e`-?`，但 e 只被求值一次。

5 追加运算符 e `<~` v 等价于 e `=` e `~` v，v `~>` e 等价于 e `=` v `~` e，但 e 只被求值一次。

4.13 分号运算符 [expr.semi]

Expression:

AssignExpr

AssignExpr `;` *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 语句是块的构成部分。如果语句包含子块，则这个块将优先作为语句的构成部分而不是其中的表达式的一部分。[例：

```
// 错误： \tcode{\{ true \}} 被认为是 \tcode{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[1]，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。

(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。

1) 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。

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_k) 而言，v`.`k 匹配 p_k 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*
`let` *PatternBind*

PatternBind:
Identifier PatternAssterion
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:
    `const`

TypeBody:
    ObjectType
    `=` *Type*

7.2 类声明 [decl.class]

8 函数 [func]

FuncDecl:

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

FuncQual:

`async`

`const`

FuncName:

Identifier

`init`

`deinit`

`operator` *Operator*

`operator` *StringLiteral*

Parameter:

`(` *ParamList`?`* `)`

ParamList:

ParamDecl

ParamList `,` *ParamDecl*

ParamDecl:

ParamName TypeNotation`?`

ParamName:

Identifier

`this`

ReturnType:

`->` *TypeNotation*

9 概念 [concept]

10 类 [class]

ClassDecl:

ClassQual`*` `class` *Identifier* *ClassBody*

ClassQual:

`const`

1 类描述内部不透明的类型。

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。[2]

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。

2) 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 顺序库 [core.order]

1 `core.order` 对各种序关系进行了定义。

15.1 类型 [core.order.type]

```
enum Order {
    less,
    equal,
    greater,
    unordered
}
```

16 序列库 [core.seq]

1 `core.seq` 库包含所有与序列相关的库。

索引

语法产生式索引