文档编号: 0.1
日期: 2024-04-24

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。以两个下划线__开头，后跟任意数量的小写字母与下划线的标识符也可能被保留作关键字。

表 1 — 关键字

_	any	async	await	auto
bool	break	class	cmp	const
continue	deinit	do	dyn	else
enum	extern	false	float	for
func	if	impl	import	in
init	infer	int	internal	is
lazy	let	match	mut	never
nil	operator	partial	private	public
return	self	shl	shl_eq	shr
shr_eq	some	static	string	this
throw	true	type	typeof	uint
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

`didSet`	`get`	`infix`	`prefix`	`root`
`set`	`suffix`	`super`	`then`	`willSet`

1.4 标点符号 [lex.punc]

Punctuator:

PunctuatorPart+

PunctuatorPart: 以下之一

`~ ! @ # $ % ^ & * ( ) - + = [ ] { } | ; : ' < > , . ? /`

[1] **标点符号**由一个或数个符号组成，其中的一部分称为**运算符**，参见 11。

[2] 除了多字符标点符号外，标点符号都由单个字符组成。表 3 列出了内建的多字符标点，但不包含运算符。解析标点符号时，应尽可能长地匹配多字符标点符号。用户也可以自定义多字符运算符。

表 3 — 多字符标点符号

`{|`	`|}`	`->`	`=>`	`::`	`...`

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)*`*`

BinaryDigit:

`0`

`1`

HexadecimalLiteral:

`0x` *HexadecimalDigits*

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` 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。

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 4，其中 i 为其字面值：

表 4 — 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 5 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 5 — 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?*

Mdelim Mchar Mdelim Suffix?*

Schar:

除了\和" 以外的非换行可打印字符

EscapeSeq

TextInterpolation

EscapeSeq:

`\` *SimpleEscape*

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

TextInterpolation:

`\(` *Expression* `)`

SimpleEscape: 以下之一

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

Rchar:

非换行可打印字符

RawTextInterpolation

RawTextInterpolation:

`\ @+ (` *Expression* `)`

Mdelim:

`""" "*`

Mchar:

除了\以外的的任意可打印字符

EscapeSeq

\ 换行

TextInterpolation

1 字符串字面量表示 UTF-8 字符串，其类型为 `string`。它有两种类型：普通字符串字面量和多行字符串字面量。

2 普通字符串字面量被单个双引号包围。其中可以使用反斜杠开始的转义序列表示其他字符。普通字符串字面量也可以前后加上等量的 `@`，此时它被称为**原始字符串字面量**。原始字符串字面量中的反斜杠不会被解释为转义序列，而是字符本身。

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。

(3.4) — 如果一行末尾有反斜杠，则这个反斜杠和其后的换行符会被删除。

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"`。]

4 字符串字面量可以包含字符串插值，其形式为 `\(`*e*`)`，其中 *e* 为表达式。字符串插值会被求值后转换为字符串插入当前位置。对原始字符串字面量而言，反斜杠和括号之间需要插入等量的 `@`，否则仍然会被解释为字面符号。

1.5.4 符号字面量 [literal.symbol]

SymbolLiteral:
    `'` *Identifier*
    `'(` *UnqualID* `)`

1 符号字面量用于标识成员名称，其类型和其值为其标识符或内部 ID 的值。其内部 ID 不能为另一个符号字面量。

1.5.5 布尔字面量 [literal.boolean]

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，则它等价于以下第一个合法的表达式：

```
operator ""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。特定的语言功能可能只能在特定的作用域中生效。不同的作用域具有不同的类型，分别被不同的语言功能所引用。作用域可以互相包含。

2 *全局作用域*是整个程序文本代表的作用域，包含所有其他作用域。

2.1.1 声明作用域 [scope.decl]

1 声明作用域限制声明的范围。一个声明或绑定将会被插入到最近的声明作用域中，并且在该作用域内可以使用该名称引用被声明的实体。在离开该作用域之后，被声明的实体将不能被使用该方式引用。

2 所有的语句都具有声明作用域。全局作用域也是声明作用域。

2.1.2 函数作用域 [scope.func]

1 函数作用域限制 `return` 语句的使用。参见 5.7。

2 函数作用域也限制 `await` 运算符的使用。参见 4.2.4。

3 函数定义的块、lambda 表达式的块或表达式、`do` 表达式的块和函数调用表达式的 lambda 块具有函数作用域。

2.1.3 Lambda 作用域 [scope.lambda]

1 Lambda 作用域限制 lambda 参数的使用。参见 4.1.7.1。

2 lambda 表达式的块或表达式以及函数调用表达式的 lambda 块具有 lambda 作用域。

3 属性的访问器也具有 lambda 作用域。参见9.2。

2.1.4 序列作用域 [scope.sequence]

1 序列作用域限制 `$` 的使用，参见 4.1.5。序列作用域有一个关联的当前序列值，无论它是否实现 `core.Sequence`。

2 下标运算符 *s*`[...]` 的两个方括号之间具有序列作用域。其当前序列为 *s*。

3 函数调用表达式 *s*`(...)` 的括号之间具有序列作用域。如果 *s* 形如 *o*.*f*，且 *f* 是一个方法，则其当前序列为 *o*，否则当前序列为 *s*。[*注*：此处只能进行一次拆分，即 *a*.*b*.*c* 的当前序列不可能为 *a*。]

4 如果函数调用表达式带有一个 lambda 块，则这整个块也具有序列作用域，其当前序列确定方法同上。

[*例*：

```
let a = [1, 2, 3, 4, 5];
```

```
let o = { a };
func operator()(this: int[]) -> int[] { [] }
func v(this: int[], index: int) -> int[] { [] }

a[$ - 1] // 当前序列为 a
a($ - 1) // 当前序列为 a
o.a[$ - 1] // 成员访问，当前序列为 o.a
a.v($ - 1) // 方法调用，当前序列为 a
```

]

2.2 名称 [name]

UnqualID:

Identifier

SymbolLiteral

`init`

`deinit`

OperatorID

1 名称用于引用程序实体。一个名称可能是一个标识符、`init`、`deinit` 或一个运算符名称。

2.2.1 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

SomeType

AnyType

Type TypeQualifier

TypeQualifier:

`mut`

`const`

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

1 类型是一个集合。值是类型、类型的成员和修饰符集合的元组。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* `>`

SymbolType

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

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。

$$\texttt{int}_{l,h} \coloneqq \{x \in \mathscr{Z} \mid l \le x \le h\}$$

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

$$l \ge m$$

$$h \le M$$

$$0 \le h - l \le M$$

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

5 `int<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \ge -2^{w-1}$ 且 $h \le 2^{w-1} - 1$。`uint<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \ge 0$ 且 $h \le 2^w - 1$。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。

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

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

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

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*。

3.1.1.1 符号类型 [type.symbol]

SymbolType:
SymbolLiteral

$$\texttt{'}s \coloneqq \{\langle \texttt{'}s, \texttt{'}s \rangle\}$$

$$\texttt{core.Symbol} \coloneqq \bigcup_{\texttt{'}s} \{\texttt{'}s\}$$

1 符号字面量的类型与其值相同。`core.Symbol` 是符号字面量的公共类型。参见 18.1。

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 `[ dyn` *Expression* `]`
    Type `[` *Type* `]`
    `(` *TupleTypes*$*$ `)`
    `{` *ObjectTypes* `}`
    `(` *TupleTypes*$*$ `) ->` *Type*
    UnionType
    FuncType

TupleTypes:
    Type
    TupleTypes `,` *Type*

ObjectTypes:
    ObjectType
    ObjectTypes `,` *ObjectType*

ObjectType:
    ObjectTypeQualifier$*$ *Identifier* `:` *Type*

ObjectTypeQualifier:
    `mut`

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。如果 T 本身就是可空类型，则 T`?` 等价于 T。否则，T`?` 包含 T 的所有值和空值 `nil`。

$$T\texttt{[dyn } N\texttt{]} \coloneqq T^N$$

2 T`[dyn` N`]` 为数组类型，代表 N 个 T 的值的序列。T`[]` 为 T`[dyn _]` 的别名。

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

3 T`[`K`]` 为字典类型。

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

4 $(T_1\texttt{,}\ldots\texttt{,}T_n)$ 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。

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

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

6 $(T_1\texttt{,}\ldots\texttt{,}T_n)$ `->` 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 $T_1\texttt{|}\ldots\texttt{|}T_n$ 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。

3.1.3.1 函数类型 [type.func]

FuncType:
Parameter ReturnType
Type ReturnType

3.1.4 具名类型 [type.named]

TypeName:
`(` *Type* `)`
EntityID
FundaType
SpecialType

1 类型名称在特定的语法位置表示类型，以避免潜在的语法歧义。

3.2 子类型 [subtype]

1 类型 A 可能是类型 B 的子类型，记作 $A \preceq B$。A 可以在需要 B 的上下文中隐式转换到 B。

2 子类型关系具有自反性和传递性，即对任意类型 A、B 和 C 有 $A \preceq A$ 和 $A \preceq B \land B \preceq C \implies A \preceq C$ 成立。

$$T \preceq \texttt{void}, T \in \mathcal{T}$$
$$\texttt{never} \preceq T, T \in \mathcal{T}$$

3 任何类型都是 `void` 的子类型。`never` 是任何类型的子类型。

$$I_{l_1,h_1} \preceq J_{l_2,h_2} \text{ if } l_1 \geq l_2 \vee h_1 \leq h_2$$
$$\texttt{float<}s_1\texttt{>} \preceq \texttt{float<}s_2\texttt{>} \text{ if } s_1 \leq s_2$$
$$I_{l,h} \preceq \texttt{float<}s\texttt{>}$$
$$I, J \in \{\texttt{int}, \texttt{uint}, \texttt{int<}w\texttt{>}, \texttt{uint<}w\texttt{>}\}$$

4 范围更小的整数类型是范围更大的整数类型的子类型。长度更小的浮点类型是长度更大的浮点类型的子类型。

5 整数类型是浮点类型的子类型。当整数被隐式转换为浮点数时，其值将被转换为最接近的浮点数。[注：虽然整数转换到浮点数可能无法保持值不变，但出于习惯仍然保持这个隐式转换。] [注：浮点类型不能隐式转换到整数类型，但可以显式转换。]

$$\texttt{'}s \preceq \texttt{core.Symbol}$$

6 所有符号字面量类型都是 `core.Symbol` 的子类型。

7 `bool` 没有语言内建的子类型约束。但是，其他类型的值可以在 `if` 表达式中当作条件使用，这通过重载 `operator if` 运算符实现。

$$T \preceq T\texttt{?}$$
$$T\texttt{[]} \preceq U\texttt{[]} \text{ if } T \preceq U$$
$$T\texttt{[}K\texttt{]} \preceq U\texttt{[}L\texttt{]} \text{ if } T \preceq U \text{ and } K \preceq L$$
$$\texttt{(}T_1\texttt{,} \ldots \texttt{,}T_n\texttt{)} \preceq \texttt{(}U_1\texttt{,} \ldots \texttt{,}U_n\texttt{)} \text{ if } T_i \preceq U_i \text{ for } 1 \leq i \leq n$$

8 任意类型是其对应可空类型的子类型。如果 T_i 是 U_i 的子类型，则 T_0`[]`、T_0`[`T_1`]`、`(`T_1`,` $\ldots$ `,`T_n`)` 分别是 U_0`[]`、U_0`[`U_1`]`、`(`U_1`,` $\ldots$ `,`U_n`)` 的子类型。[注：这意味着数组、字典和元组的元素类型是协变的。]

9 对两个对象类型 T 和 U 而言，如果 U 的每个成员都有对应的 T 的成员且类型是该成员的子类型，则 T 是 U 的子类型。

10 对两个函数类型 T 和 U 而言，如果 U 的每个参数类型都是对应 T 的参数的子类型，且 T 的返回类型是 U 的返回类型的子类型，则 T 是 U 的子类型。[注：这意味着函数类型的参数类型是逆变的，返回类型是协变的。] T 可以拥有比 U 更多的顺序参数，或者 U 不包含的命名参数。

11 类类型可以定义类型转换函数。每个这样的函数定义了一个子类型关系。

12 对类型表达式而言，T `&` U 是 T 和 U 的子类型；T 和 U 是 T `|` U 的子类型。

3.3 公共类型 [type.common]

1 对两个类型 A 和 B 而言，存在一个唯一的类型 C 称为 A 和 B 的*公共类型*。C 满足：

$$
\begin{gathered}
A \preceq C \\
B \preceq C \\
\forall D \in \mathcal{T}, A \preceq D \land B \preceq D \Rightarrow C \preceq D
\end{gathered} \tag{3.1}
$$

记作 $C = A \otimes B$。公共类型满足交换律。

2 如果 $A \preceq B$，则 $A \otimes B = B$。

3 如果 A 和 B 之间没有子类型关系，则 $A \otimes B = A \mid B$。

3.4 some 类型 [type.some]

SomeType:
`some` *Type*
`some _`

1 `some` T 标识一个静态待推导类型，但保证该类型为 T 的子类型。`some _`代表一个基础类型待推导的 `some` 类型。

2 `some` 还能用于简化泛型函数声明。参见 13.2。

[例：

```
trait T { }
type A { }
type B { }

impl A : T { }
impl B : T { }

let x: some T = A { }; // x 的类型是 A
let mut y: some T = B { }; // y 的类型是 B

y = x; // 错误，B 不能赋给 A
```

]

3.5 any 类型 [type.any]

AnyType:
`any` *Type*
`any _`
`any`

1 `any` T 对 T 的子类型进行包装，保证在运行时可以接受任何为 T 的子类型的值。`any _`代表一个基础类型待推导的 `any` 类型。`any` 表示对任意类型的包装。

[例：

```
trait T { }
type A { }
type B { }

impl A : T { }
impl B : T { }

let x: any T = A { }; // x 的类型是 any T
let mut y: any T = B { }; // y 的类型是 any T

y = x; // 正确，any T 之间可以互相赋值
```

]

3.6 修饰符 [qualifier]

1 值除了总是具有类型之外，还可能带有一个或数个修饰符。修饰符指示了值的其他属性。

2 类型可以带有修饰符，指定该值需有特定的修饰符约束。

3.6.1 `mut` [qual.mut]

1 `mut` 表示该值是可变的。具有 `mut` 的值才能成为赋值操作符的左操作数。参见 4.13。

3.6.2 `const` [qual.const]

1 `const` 表示该值是一个常量值，于编译期间确定且不可变。部分语言功能只允许常量值。

2 `const` 绑定创建一个常量并插入当前作用域。该绑定的初始值必须是常量表达式。

4 表达式 [expr]

Expression:
PrimaryExpr
Operator Expression
Expression Operator
Expression Operator Expression

1 4.2到4.14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。用户也能定义新的运算符组或在当前的组中添加新的运算符，参见11.2。

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

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 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。

5 *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:
`(` *Expression* `)`
LiteralExpr
TypeLiteral
Identifier
DeductedEnumerator
`this`
`$`
StmtExpr
LambdaParameter
LambdaExpr
DoExpr

1 表达式 `(`e`)` 等价于 e，括号只作分组用途。[`注`：注意括号表达式不是一元元组。]

4.1.1 字面量表达式 [expr.lit]

LiteralExpr:
- *IntegerLiteral*
- *FloatingLiteral*
- *StringLiteral*
- *SymbolLiteral*
- *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*
- *Identifier*
- `...` *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1.5.5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 对象字面量 `{`e`}` 是 `{`e`:`e`}` 的简写。e 必须是一个标识符。

7 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 初始化字面量 [expr.lit.init]

TypeLiteral:

TypeParenLiteral

TypeArrayLiteral

TypeObjectLiteral

TypeDictLiteral

TypeParenLiteral:

TypeName `(` *Arguments?* `)` *Block**

TypeName *Block*

TypeArrayLiteral:

TypeName `[` *ExprList?* `]`

TypeObjectLiteral:

TypeName `{` *ObjectItems?* `}`

TypeDictLiteral:

TypeName `{|` *DictItems?* `|}`

1 可以使用与字面量类似的语法来创建指定类型的值。被创建的类型必须是一个类型名称。

[例：

```
let a = int[] [1, 2, 3]; // 错误，不能用 [] 初始化 int

type IntArray = int[];

let a = IntArray[1, 2, 3]; // 可以

let a = (int[])[1, 2, 3]; // 可以
```

]

2 T`(`a_1`,` ... `,`a_n`)` 以参数给定的参数创建一个类型为 T 的对象。创建对象时，会以这些参数调用给定的初始化器。以这种方式初始化的对象也可以带有 lambda 块。如果这个对象的初始化器只接受这一个参数，也可以省略括号。

3 T`[`v_1`,` ... `,`v_n`]` 以对应的数组字面量为参数创建一个类型为 T 的对象。

4 T`{|`k_1`:`v_1`,`k_2`:`v_2`,` ... `,`k_n`:`v_n`|}` 以对应的字典字面量创建一个类型为 T 的对象。

5 T`{`k_1`:`v_1`,` ... `,`k_n`:`v_n`}` 以对象方式创建一个类型为 T 的对象。

6 如果 T 是对象类型，则每个 kv 对都会用来初始化对应的成员。如果 v 不能隐式转换到成员的类型，则这是一个编译错误。如果有成员未被显式指定初始值，且其不是可空类型，则这是一个编译错误；如果是可空类型，则其初始值为 `nil`。

7 如果 T 是类类型，则选择以下方式中第一个成功的：

(7.1) — 以具名参数初始化，等价于 T`(`k_1`:`v_1`,` ... `,`k_n`:`v_n`)`；

(7.2) — 以单个对象字面量的方式初始化，等价于 T`({`k_1`:`v_1`,` ... `,`k_n`:`v_n`})`；

(7.3) — 首先默认初始化 T`()`，然后依次执行 T`.`k_1`=`v_1 到 T`.`k_n`=`v_n。

8 T`{}` 总是被解释为以对象字面量创建值。[注：如果你需要创建一个具有空 lambda 块的值，请使用 T`(do {})`。]

4.1.3 匿名静态成员表达式 [expr.enum]

DeductedEnumerator:
- `.` *Identifier*
- `.` *Identifier* `(` *Arguments?* `)` *Block**
- `.` *Identifier Block*
- `.` *Identifier* `[` *ExprList?* `]`
- `.` *Identifier* `{` *ObjectItems?* `}`

1 静态成员和枚举符可以在适当的上下文中省略类型名称而使用自动推导。参见 14.1。

4.1.4 `this` [expr.this]

1 表达式 `this` 在类方法或扩展方法中表示当前方法的调用者。如果没有在参数中显式指定 `this` 的类型，则其类型为 `self`。

4.1.5 `$` [expr.dollar]

1 表达式 `$` 只能在序列作用域（2.1.4）中使用。如果当前序列为 s，则 `$` 等价于 s`.opeartor$()`。如果 s 实现了 `core.Sequence`（例如内建数组 T`[]`），其值为 s`.size`。在其他位置使用 `$` 是一个编译错误。

4.1.6 语句表达式 [expr.stmt]

StmtExpr:
- *Block*
- *IfExpr*
- *MatchStatement*
- `break`
- `continue`
- `return` *Expression?*
- `throw` *Expression?*

IfExpr:
- *IfStatement*
- `if` *Condition* `then` *Expression*
- `if` *Condition* `then` *Expression* `else` *Expression*
- `if` *Condition* `then` *Expression* `else` *IfStatement*

1 部分语句也有对应的表达式。

2 块表达式的值是其末尾表达式的值。如果该表达式被省略，其类型为 `void`。

3 `if` 表达式的类型是其两个分支表达式的公共类型。如果 `else` 分支被省略，则视为 `void`。其值为最终执行的那个分支的值。只有一个分支会被求值。可以使用 `then` 来用表达式代替块。

4 `match` 表达式的类型为其所有分支的公共类型，值为最终匹配成功的分支的值。只有一个分支会被求值。

5 `break`、`continue`、`return` 和 `throw` 表达式的语义与其对应的语句的语义相同，类型为 `never`。

4.1.7 Lambda 表达式 [expr.lambda]

LambdaExpr:
- *LambdaQual** *LambdaParameter* *ReturnType** `=>` *LambdaBody*

LambdaQual:
- `async`

LambdaParameter:
- *ParamDecl*

LambdaBody:
- *Expression*

4.1.7.1 Lambda 参数 [expr.lambda.param]

LambdaParameter:
`$` *Digit+*
`$` *Identifier*

1 Lambda 参数只能在 lambda 作用域（2.1.3）中使用，用于引用匿名参数。其类型是待推导的。不在 lambda 作用域中使用 lambda 参数，或在显式指定参数的 lambda 表达式中使用 lambda 参数，是一个编译错误。

2 `$` 后跟数字 i 指代第 i 个参数。`$` 后跟标识符 n 指代具名参数 n。

4.1.7.2 `do` 表达式 [expr.do]

DoExpr:
*LambdaQual** `do` *Block*
*LambdaQual** `do !` *Block*

1 `do` 后跟一个块创建一个没有显式指定参数的 lambda 表达式。

2 `do!` 后跟一个块创建一个无参的 lambda 表达式并立即调用它，将其值作为整个表达式的值。

[例：

```
let arr = [1, 2, 3];

let first1 = arr.first(do { $0 > 2 }); // 获取第一个满足条件的元素

let firstFinder = do {
    for let v : $0 {
        if v > 2 { return v; }
    }
    nil
};

let first2 = firstFinder(arr); // 与上面等价

let first3 = do! {
    for let v : arr {
        if v > 2 { return v; }
    }
    nil
}; // 与上面等价
```

]

4.2 后缀运算符 [expr.suffix]

SuffixExpr:
PrimaryExpr
IndexExpr
FuncCallExpr
MemberAccessExpr
AwaitExpr
NullCheckExpr
PrevNextExpr

4.2.1 下标运算符 [expr.sub]

IndexExpr:
SuffixExpr `[` *ExprList?* `]`

1 下标运算符用于对数组进行访问。a`[`i`]` 表示数组 a 的第 i 个元素。用户自定义的下标运算符可以接受多于一个参数。

4.2.2 函数调用运算符 [expr.call]

FuncCallExpr:
SuffixExpr `(` *Arguments?* `)` *Block**
SuffixExpr Block

Arguments:
UnnamedArgs
NamedArgs
UnnamedArgs `,` *NamedArgs*

UnnamedArgs:
Expression
UnnamedArgs `,` *Expression*

NamedArgs:
Identifier `:` *Expression*
NamedArgs `,` *Identifier* `:` *Expression*

1 函数调用运算符用于调用函数。括号内的项作为参数传递给函数。

2 如果函数调用运算符的左操作数形如 $o.f$，其中 f 是一个名称且不是 o 的成员名称，则这称作*方法调用*。此时，o 将作为 f 的第一个位置参数传递给函数 f。

3 函数调用运算符可以后跟一个块。这个块将作为一个匿名 lambda 块，创建一个 lambda 表达式并作为函数的最后一个位置参数传递给函数。若此时函数没有任何其他参数，则函数调用的小括号可以省略。

4.2.3 成员访问运算符 [expr.member]

MemberAccessExpr:
SuffixExpr . *UnqualID*
SuffixExpr . *IntegerLiteral*
SuffixExpr . (*Expression*)

1 成员访问运算符用于访问对象的成员。$o.m$ 表示对象 o 的成员 m。如果 o 没有成员 m，若它后跟一个函数调用运算符，则作为方法调用处理；否则这是一个编译错误。

2 $o.i$ 用于元组成员的访问，表示 o 的第 i 个元素。i 必须是一个不包含前缀或后缀的十进制字面量。如果 i 大于元组的长度，这是一个编译错误。

3 $o.(e)$ 首先对 e 求值。如果得到一个 `symbol` 类型的值，则对 o 进行成员访问。如果得到一个整数类型的值，则对 o 进行元组成员访问。否则，这是一个编译错误。e 必须是常量表达式。

4.2.4 `await` 运算符 [expr.await]

AwaitExpr:
SuffixExpr `.await`

1 `await` 运算符用于等待一个异步表达式的结果。e`.await` 挂起当前计算，直到 e 的值可用。如果 e 的类型是 `core.Future<`T`>`，则 e`.await` 的类型是 T。只能在具有 `async` 修饰的函数作用域中使用 `await` 运算符。

4.2.5 空值检测运算符 [expr.null]

NullCheckExpr:
SuffixExpr `?`
SuffixExpr `!`

1 e`?` 对 e 进行空值检测。如果 e 的值不为 `nil`，则 e`?` 的值为 e；否则，该表达式直到空值检测运算符为止的整个表达式的值为 `nil`。e 的类型必须是 T`?`。即使空值检测运算符检测为空，表达式的其它部分仍然会被求值。

[例：

```
let a: int? = nil;

a + 1 // 编译错误，不存在接受 int? 和 int 的加法运算符
a? + 1 // nil
(a + 1)? // nil，多余的运算符
a? + 1 ?? 1 // 1，? 不会越过?? 运算符
(a? + 1) ?? 1 // 同上，但是更清晰
a? + 1 == b // 等号也会停止传播，等价于 (a? + 1) == b
```

]

2 e`!` 与 e`?` 类似，但它在 e 为 `nil` 的时候抛出 `core.Exception.checkedNil`。

3 如果空值检测运算符的操作数类型是 T`?`，则整个表达式的类型为 T。如果空值检测运算符的左操作数不是

可空类型，则空值检测运算符将被忽略。实现可以为此提出一个警告。

4.2.6 前驱后继运算符 [expr.prev-next]

PrevNextExpr:
SuffixExpr `+!`
SuffixExpr `-!`

1 e`+!` 和 e`-!` 分别表示 e 的后继和前驱。如果 e 的类型是算数类型，e`+!` 等价于 $e+1$，e`-!` 等价于 $e-1$。

4.3 前缀运算符 [expr.prefix]

PrefixExpr:
SuffixExpr
MathPrefixExpr
NotExpr
NegationExpr

4.3.1 数学前缀运算符 [expr.prefix-math]

MathPrefixExpr:
`+` *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。

4.3.2 逻辑否运算符 [expr.not]

NotExpr:
`!` *PrefixExpr*

1 逻辑否运算符`!` 用于对布尔值取反。如果操作数为 `true`，则结果为 `false`；如果操作数为 `false`，则结果为 `true`。

4.3.3 位取反运算符 [expr.neg]

NegationExpr:
`'~` *PrefixExpr*

1 位取反运算符`'~`进行按位取反。操作数的类型必须是定长整数类型。

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。

[2] 如果 a 的类型不是可空类型，则这是一个编译错误。如果 a 的类型为 A`?`，b 的类型为 B，则表达式的类型为 A 和 B 的公共类型。

4.8 区间运算符 [expr.range]

RangeExpr:
NullCoalExpr
NullCoalExpr `..` *NullCoalExpr*
NullCoalExpr `..=` *NullCoalExpr*

[1] 运算符`..` 生成左闭右开区间，结果类型是 `core.Range`。运算符`..=` 生成左闭右闭区间，结果类型是 `core.ClosedRange`。参数类型必须是整数类型。参见 18.3。

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 空值合并运算符 [expr.null-coal]

NullCoalExpr:
BitwiseExpr
BitwiseExpr `??` *NullCoalExpr*

1 a `??` b 首先对 a 求值，如果其结果不是 `nil`，则表达式的值为 a，且 b 不会被求值；否则表达式的值为 b。

4.11 比较运算符、包含运算符和匹配运算符 [expr.cmp-in]

BooleanExpr:
RangeExpr
CompareExpr
IncludeExpr
MatchExpr

1 本节中的运算符的结果都是 `bool`。

4.11.1 比较运算符 [expr.compare]

CompareExpr:
RangeExpr `!=` *RangeExpr*
RangeExpr `!<` *RangeExpr*
RangeExpr `!>` *RangeExpr*
RangeExpr `<>` *RangeExpr*
RangeExpr `cmp` *RangeExpr*
LessChainExpr
GreaterChainExpr

LessChainExpr:
RangeExpr LessChainOperator RangeExpr
LessChainExpr LessChainOperator RangeExpr

LessChainOperator: 以下之一

`<` `==` `<=`

GreaterChainExpr:

RangeExpr GreaterChainOperator RangeExpr

GreaterChainExpr GreaterChainOperator RangeExpr

GreaterChainOperator: 以下之一

`>` `==` `>=`

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

[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。

4.11.2 包含运算符 [expr.include]

IncludeExpr:

RangeExpr `in` *RangeExpr*

RangeExpr `!` `in` *RangeExpr*

[1] a `in` b 检测 a 是否在 b 中。a `!in` b 等价于 `!(`a `in` b`)`。表达式的类型为 `bool`。

4.11.3 匹配运算符 [expr.match]

MatchExpr:

RangeExpr `is` *Pattern*

RangeExpr `!is` *Pattern*

[1] a `is` p 检测 a 是否匹配模式 p。a `!is` p 等价于 `!(`a `is` p`)`。表达式的类型为 `bool`。使用的模式不能包含绑定模式。

4.12 逻辑运算符 [expr.logic]

LogicExpr:

BooleanExpr

LogicExpr `&` *BooleanExpr*

LogicExpr `|` *BooleanExpr*

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。

4.13 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 `??=` *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 `mut` 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、按位或和空值合并赋值。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。

4.14 分号运算符 [expr.semi]

Expression:

AssignExpr

AssignExpr ; *Expression*

Binding ; *Expression*

[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
BindingStatement
IfStatement
MatchStatement
WhileStatement
ForStatement
BreakStatement
ContinueStatement
ReturnStatement
ThrowStatement

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: { 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
Statement

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]

BindingStatement:
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]

IfStatement:
`if` *Condition Block*
`if` *Condition Block* `else` *Block*
`if` *Condition Block* `else` *IfStatement*

Condition:
Expression
Binding

1 如果条件为表达式 e[1]，那么这个表达式的类型必须是布尔类型（参见 11.3.9）。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 如果 `if` 语句的条件成立，执行第一个子块。否则，若有 `else` 子块，执行 `else` 子块。

5.4 `match` 语句 [stmt.match]

MatchStatement:
`match` *Expression MatchBlock*

MatchBlock:
`{` *BlockItem* MatchItem+* `}`

MatchItem:
Pattern `->` *Statement*

1 *`match` 语句*对其后跟的表达式进行模式匹配。`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。对 `match` 语句的求值将按如下顺序进行：

(1.1) — 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。

(1.2) — 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，则执行其后的语句。所有其他项都不会进行求值。

5.5 `while` 语句 [stmt.while]

WhileStatement:
`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]

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

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。

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` 且 `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`。

2 `for` 语句将被展开为如下形式：

```
{
    let mut i = e.iterator;

    while let v = i.next(); v != nil {
        p = v;
        B
    }
}
```

其中 i 和 v 是内部使用的匿名变量名称。参见 19。

5.7 控制语句 [stmt.control]

BreakStatement:
    `break` `;`

ContinueStatement:
    `continue` `;`

ReturnStatement:
    `return` *Expression?* `;`

ThrowStatement:
    `throw` *Expression?* `;`

1 控制语句包括 `break` 语句、`continue` 语句、`return` 语句和 `throw` 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，将控制流移动到该语句之后。

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。将控制流移动到该语句的下一次循环开始。

4 `return` 语句只能在函数作用域（2.1.2）中使用。它中止当前函数的执行，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 `()`。

5 `throw` 语句只能在函数作用域中使用。它终止当前函数的执行，并将后跟的表达式作为异常抛出。如果表达式被省略，除非语句当前处于 `catch` 块中，此时将会重新抛出原异常；否则这是一个编译错误。

6 模式匹配 [pattern]

Pattern:
    *PatternBody PatternAssertion**

PatternBody:
    NullPattern
    ExprPattern
    BindPattern
    ArrayPattern
    TuplePattern
    ObjectPattern
    AltPattern

PatternAssertion:
    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。本节中，v 表示值，p 表示模式。值符合特定的模式称为这个值匹配这个模式，记作 $v \parallel p$。

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:
    RangeExpr

1 v 和 e 必须可比较。$v \parallel e$ 当且仅当 v `==` e。

6.3 数组模式 [pattern.array]

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

AnyPattern:
    Pattern
    `...`
    BindKeyword `...` *Identifier*

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）。

3 如果任意项包含一个绑定，则该任意项匹配到的元素将被绑定到相应的标识符上。

6.4 元组模式 [pattern.tuple]

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:
    BindKeyword PatternBind

BindKeyword:
    `let`
    `let mut`
    `let const`

PatternBind:
Identifier PatternAssertion
ArrayPatternBind
TuplePatternBind
ObjectPatternBind

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

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

AnyPatternBind:
PatternBind
`...` *Identifier?*
NullPattern
ExprPattern

ObjectPatternBind:
`{` *ObjectPatternBodyBind* `}`

ObjectPatternBodyBind:
*ObjectItemBind (, ObjectItemBind)**

ObjectItemBind:
Identifier `:` *PatternBind*
Identifier

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。

2 如果绑定使用关键字 `const`，则这是一个常量绑定。参见 3.6.2。

3 绑定模式可以使用简写，表 6 列出了一些常见的简写形式。

表 6 — 绑定简写与其完整形式

`let [a, b]`	`[let a, let b]`
`let (v, _)`	`(let v, _)`
`let [x, ...y]`	`[let x, let... y]`
`let { x }`	`{ x: let x }`

6.6.1 选择模式 [pattern.alt]

AltPattern:
Pattern `|` *Pattern*
AltPattern `|` *Pattern*

1 选择模式同时匹配多个模式。如果其中有模式匹配成功，则整个模式匹配成功。匹配将从左到右进行。

2 选择模式中使用的模式不能包含绑定模式。

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
TraitDecl
QualifierDirective

7.1 类型声明 [decl.type]

TypeDecl:
TypeQual`*` `type` *Identifier* *TypeBody*

TypeDeclName:
Identifier
`self`

TypeQual:
`const`

TypeBody:
ObjectType
`=` *Type*

7.2 限定符指令 [qual.dir]

QualifierDirective:
DirectiveQualifiers `::`

DirectiveQualifiers:
DirectiveQualifier`+`

DirectiveQualifier:
AccessQualifier

1 可以使用限定符后跟`::`的方式为多个声明指定限定符。直到下一个限定符指令或该声明作用域结束为止，所有出现的声明都会受所指定的修饰符修饰。忽略无效的限定符。

[例：

```
public:: // 后面所有声明都是 public 的
type A = int;
```

```
const let size = 0;
```

8 函数 [func]

FuncDecl:

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

FuncQual`*` `func` *UnqualID Parameter* `=>` *Expression* `;`

FuncQual:

`async`

`const`

`extern`

Parameter:

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

ParamList:

ParamDecl

ParamList `,` *ParamDecl*

ParamDecl:

ParamName `:` *Type`?`*

`this` `:` *TypeQualifier`+`*

ParamName:

Identifier

`this`

ReturnType:

`->` *Type*

9 类 [class]

ClassDecl:
*ClassQual** `class` *Identifier ClassBody*

ClassQual:
`const`

ClassBody:
`{` *ClassMember** `}`

ClassMember:
FieldDecl
PropertyDecl
FuncDecl
TypeDecl
ClassDecl
EnumDecl
TraitDecl

1 类描述内部不透明的类型。

9.1 字段 [class.member]

FieldDecl:
*FieldQual** *BindKeyword Identifier TypeNotation Initializer*`?` `;` *FieldQual** *BindKeyword Identifier Initializer* `;`

TypeNotation:
`:` *Type*

Initializer:
`=` *Expression*

1 类中的字段表示类的内部状态，其默认访问级别为 `private`。具有 `mut` 修饰的是可变字段。类字段可以显式指定类型，也可以通过初始值推导类型。

9.2 属性 [class.property]

PropertyDecl:
*PropertyQual** *BindKeyword Identifier TypeNotation*`?` *Initializer*`?` *PropertyBody* `;`
*PropertyQual** *BindKeyword Identifier TypeNotation*`?` `=>` *Expression* `;`

PropertyQual:
AccessQual

PropertyBody:
`{` *PropertyMember+* `}`

PropertyMember:
*PropertyQual** *PropertyKeyword PropertyBlockParam?* *Block*
*PropertyQual** *PropertyKeyword PropertyExprParam?* `=>` *Expression* `;`
*PropertyQual** *PropertyKeyword* `;`

PropertyKeyword: 以下之一
`get set willSet didSet`

PropertyBlockParam:
Identifier
Identifier `,` *Identifier*

PropertyExprParam:
Identifier
`(` *Identifier* `,` *Identifier* `)`

1 类中还可以声明属性。属性是类对外暴露的接口，其默认访问级别为 `public`。属性的定义至少需要包含一个访问器。访问器的块或表达式具有 lambda 作用域。

2 属性的访问器可以以上下文关键字 `get`、`set`、`willSet` 或 `didSet` 开始。`get` 访问器不接受任何参数。`set` 访问器接受一个参数，其类型为该属性的类型。`willSet` 和 `didSet` 访问器可以接受一个或两个参数，其类型为属性的类型。

3 如果 `set` 访问器不显式写出参数，则视为其具有 lambda 参数 `$value`。如果 `willSet` 或 `didSet` 不显式写出参数，则视为其具有 lambda 参数 `$oldValue` 和 `$newValue`。

4 如果属性 p 的声明中既不包含 `get` 也不包含 `set` 访问器，则视作该类具有字段 p'，且具有访问器 `get => this.`p'。如果该属性有 `mut` 修饰，则还视为该属性具有访问器 `set => this.`p' `= $value`。如果 `get` 或 `set` 访问器后直接跟分号，则视作以上述方式生成访问器，且另一个对应的访问器若存在则必须采用这种省略方式声明。如果属性以直接后跟 `=>` e 的方式声明，则视作该属性具有访问器 `get =>` e。

5 未使用 `mut` 声明的属性不能包含 `set`、`willSet` 和 `didSet` 访问器。在考虑自动生成的访问器之后，如果一个属性缺少 `get` 访问器，或一个 `mut` 属性缺少 `set` 访问器，则这是一个编译错误。

6 属性可以具有类型提示或初始化器。如果属性未按前文所述具有对应的字段，则不能具有初始化器。如果属性省略类型提示，则其类型将从 `get` 访问器中推导。如果它的 `get` 访问器是自动生成的，这是一个编译错误。

7 当读取属性 p 时，会调用 `get` 访问器并将其返回值作为 p 的新值。

8 当给属性 p 赋值时，首先将该值隐式转换到属性的类型，令结果为 v：

(8.1) — 如果属性具有 `willSet` 访问器：

(8.1.1) — 如果它接受一个参数，将以 v 调用；

(8.1.2) — 如果它接受两个参数，将以 p（旧值）和 v 调用。

(8.2) — 以 v 调用属性的 `set` 访问器。

(8.3) — 如果属性具有 `didSet` 访问器：

(8.3.1) — 如果它接受一个参数，将以 p（旧值）调用；

(8.3.2) — 如果它接受两个参数，将以 p 和 v 调用。

(8.3.3) — 如果函数体内没有使用新值，则不会调用 `get` 访问器并使用第一种形式调用。

9.3 方法 [method]

1 类中的函数声明称作*方法*。方法隐含了一个 `this` 参数，其为调用该方法的对象。

2 如果类内方法显式指定了 `this` 参数，则它的类型必须是这个类。在这种情况下，可以省略 `this` 的类型，而仅包含修饰符。如果省略 `this` 参数，则 `this` 不包含任何修饰符。

[例：

```
class A {
    f() { } // #1
    f(this: A) { } // 与 #1 等价
    f(this: int) { } // 错误，类方法不能指定另外的 this 类型
    f(this: A mut) { } // #2，可以修改 A 的成员
    f(this: mut) { } // 与 #2 等价
}

f(this: A) { } // 可以，与 #1 等价
f(this: int) { } // 可以，类外方法能任意指定 this 类型
```

]

9.3.1 方法查找 [method.lookup]

1

9.4 构造器 [class.init]

1 构造器是函数名称为关键字 `init` 的方法。构造器可以接受任意类型的参数并且不返回值。

2 在构造器内部可以修改 `this` 成员的值，即使 `this` 并未显式被标记为 `mut`。不能在成员被赋值之前使用它们的值，或者调用使用它们值的方法。这是一个编译错误。

3 泛型类的构造器可以显式指定返回类型，这称为参与推导的构造器。这类构造器可以单独指定泛型参数。

9.5 析构器 [class.deinit]

1 析构器是函数名称为关键字 `deinit` 的方法。析构器不接受任何非 `this` 参数且不返回值。不能在类外部定义析构器。析构器不能抛出异常。

2 当一个类对象被销毁时，将会调用析构器。

10 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"` 。]

10.1 传统枚举类型 [enum.trad]

1 只包含单独枚举符的枚举类型称作*传统枚举类型*。传统枚举类型可以指定基底类型 B，也可以为其枚举符指定值。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`。

2 在传统枚举类型中，每个枚举符都有对应的值。其确定如下：

(2.1) — 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；

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

(2.3) — 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式转换为对应的基底类型。

11 运算符 [op]

11.1 运算符名称 [op.name]

OperatorID:

`operator` *OperatorType*`?` *OperatorName*

`operator` *StringLiteral*

`operator $`

`operator if`

`operator is` *Type*

`operator as` *Type*

OperatorName:

Operator

Identifier

OperatorKeyword

`( )`

`[ ]`

OperatorKeyword: 以下之一

`shl shr cmp in`

OperatorType: 以下之一

`infix prefix suffix`

Operator:

CustomOperator

`;`

CustomOperator:

OperatorSymbol`+`

OperatorSymbol: 以下之一

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

[1] 运算符名称指代对应的运算符函数的名称，用来重载运算符。运算符名称由关键字 `operator` 后跟运算符组成。`operator` 后面可以跟 `infix`、`prefix` 或 `suffix` 以消歧义。

[2] `operator` 后跟的运算符可以是内建运算符或自定义运算符。表 7 列出了全部内建运算符，其中上半部分为可以重载的运算符，下半部分为不能重载的运算符。

[3] `operator` 后面还能跟有一些特殊符号组合，表示特殊含义。

[4] `opeartor()` 重载函数调用（4.2.2）。`operator[]` 重载下标运算符（4.2.1）。`opeartor$` 重载 `$`（4.1.5）。`opeartor if` 允许其它值被用于条件中。`operatro is`T 创建自定义隐式转换。`opeartor as` T 创建自定义显式转换。

表 7 — 内建运算符

+!	-!	前缀 +	前缀-	前缀!
'~	*	/	%	二元 +
二元-	shl	shr	'&	'^
'|	..	..=	~	==
!=	<	<=	>	>=
!<	!>	<>	cmp	in
&	|			
.	await	?	后缀!	??
!in	is	!is	=	+=
-=	*=	/=	%=	shl_eq
shr_eq	'&=	'^=	'|=	??=
++	--	~>	<~	;

[5] `operator` 后跟一个字符串字面量用于自定义后缀。这个字符串字面量必须是一个空字符串，且不包含任何前缀。

11.2 自定义运算符 [op.user]

OpeartorDeclaration:

`operator` *CustomOperator OperatorSpecifier* `;`

`operator` *Identifier OperatorSpecifier* `;`

OperatorSpecifier:

`prefix`

`suffix`

`infix`

`infix :` *OperatorName*

`infix : (` *OperatorPrecedence* `,` *OperatorPrecedence* `)`

OperatorPrecedence:

OperatorName

`-`

[1] 用户可以声明自己的运算符。自定义运算符的名称由关键字 `operator` 后跟一个标识符或标点符号组成。

[2] 自定义的运算符不能与内建的运算符相同，也不能是`->`、`=>` 或`...`。如果自定义的运算符是一个标识符，则它不能在任何作用域中与另一个标识符相同，否则这是一个编译错误。自定义的运算符也不能是 `prefix`、`suffix` 或 `infix`。

[3] 自定义的运算符可以是前缀、后缀或者中缀的，分别使用 `prefix`、`suffix` 和 `infix` 指定。后缀与前缀运算符不能指定优先级。后缀运算符的优先级总是高于前缀运算符，并高于所有类型的中缀运算符。中缀运算符可以指定其优先级。表 8 展示了可选择的中缀运算符优先级。

[4] 中缀运算符可以指定自己的优先级与某个内建的运算符组相同，或指定两个相邻的运算符组，创建一个自定义的优先级。不能在 `==` 到 `=` 之间创建新的优先级。此外，还可以使用 `(_, *)` 或 `(=, _)` 指定比表中运算

表 8 — 中缀运算符优先级

*	/	%
+	-	
shl	shr	
'&	'^	'|
..	..=	
~		
??		
== 等		
&	|	
= 等		

符更低或更高的优先级。[注：自定义的优先级无法指定结合性。然而，可以通过自定义运算符的重载函数自行处理优先级。参见11.3.5。] 中缀运算符也可以不指定优先级。这类运算符视为比 == 具有更高的优先级。除了未指定优先级的中缀运算符，运算符的优先级关系是线性的。

5 如果中缀运算符的优先级组处于 == 组，则其不能与任何内建运算符混用，且其返回值必须为布尔类型。如果中缀运算符处于 & 组，则其不能与任何内建运算符混用，且其参数和返回值必须为布尔类型。如果中缀运算符处于 = 组，则其不能与任何内建运算符混用，且其返回值必须是 void。自定义的与?? 或 & 优先级相同的运算符不具有短路求值特性。。如果两个自定义运算符具有相同等级的优先级，且不是内建的优先级，则在一个表达式中将其混用是一个编译错误。如果一个自定义运算符未指定优先级，则它与任何优先级高于 == 的中缀运算符混用是一个编译错误。

[例：

```
operator >> infix : +;
operator << infix : +;
operator !! infix : (*, +);
operator ~~ infix : (*, +);

a >> b + c; // OK，等价于 (a >> b) + c
a >> b << c; // OK，等价于 (a >> b) << c
a !! b * c; // OK，等价于 a !! (b * c)
a !! b ~~ c; // 错误，不能在同一个表达式中混用具有新优先级的运算符

operator ** infix;
operator && infix;

a ** b * c; // 错误，未指定优先级的运算符不能与 * 混用
a = a ** b; // 可以，允许赋值运算符
a ** b && c; // 错误，相互之间也不能混用
```

]

6 一个运算符可以同时定义为前缀、后缀和中缀，但不能定义同一个类别的相同运算符多于一次。如果自定义

运算符产生了语法歧义，则这是一个编译错误。

[例：

```
operator ** suffix;
operator ** infix;
operator && prefix;
operator && infix;

a ** && b; // 错误，无法确定谁为中缀运算符
(a**) && b; // 可以
a ** (&&b); // 可以
```

]

7 解析表达式的流程如下：

(7.1) — 构造基本表达式和运算符。运算符将尽可能长地构造，请在必要的时候使用空格分隔。

(7.2) — 如果有运算符`.`，将它及后面的必需成分一起替换成一个基本表达式。

(7.3) — 如果有运算符 `as` 或 `is`，将它及后面的必需成分一起替换成一个后缀运算符。[注：`is` 运算符之后的模式将会尽可能长地构造。请在需要的情况下加上括号。]

(7.4) — 对每一组连续且多于一个的基本表达式，如果除了第一个以外都是`.`后缀运算、元组字面量、数组字面量或块，将它们替换为后缀运算符。否则，这是一个编译错误。

(7.5) — 确定每个运算符是前缀、后缀还是中缀的。

(7.6) — 对每组连续的表达式-运算符-表达式组，如果存在唯一一个运算符满足：

(7.6.1) — 它可以作为中缀运算符；

(7.6.2) — 它前面的所有运算符都可以作为后缀运算符；

(7.6.3) — 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，之前的所有运算符是后缀运算符，之后的所有运算符是前缀运算符。如果存在多于一个或不存在这样的运算符，这是一个编译错误。

(7.7) — 第一个子表达式之前的所有运算符都是前缀运算符。最后一个子表达式之后的所有运算符都是后缀运算符。如果有运算符不是这个类别，这是一个编译错误。

(7.8) — 将基本表达式与其后的后缀运算符替换成一个后缀表达式。

(7.9) — 将后缀表达式与其前的前缀运算符替换成一个前缀表达式。

(7.10) — 此时，表达式将成为子表达式与中缀运算符交替的链，且总以子表达式开头和结尾。

(7.11) — 如果表达式中同时包含未指定优先级的运算符和另一个优先级高于 `==` 的运算符，这是一个编译错误。

(7.12) — 运算符按照优先级组依次选择其操作数，并结合成为子表达式。如果此时存在非法的运算符混用，这是一个编译错误。

(7.13) — 如果此时仍然剩余多于一个子表达式，这是一个编译错误。否则，表达式解析完成。

11.3 运算符重载 [op.over]

1 运算符可以被重载，以为其他类似提供类似于内建类型的行为。当声明一个重载运算符函数时，视同内建的运算符具有一个对应的重载函数签名来进行重载检测。

[例：

```
func operator+(lhs: int?, rhs: int?) { lhs? + rhs? } // 可以
func operator+(lhs: int, rhs: int) { lhs + rhs } // 错误，与内建运算符冲突
```

]

11.3.1 函数调用运算符 [op.over.call]

1 函数调用运算符可以接受任意数量的参数并返回任意值。重载了这个运算符的类型可以隐式转换到对应的函数类型。

2 函数调用运算符只能作为类方法被重载。

11.3.2 下标运算符 [op.over.sub]

11.3.3 前缀运算符 [op.over.prefix]

11.3.4 后缀运算符 [op.over.suffix]

11.3.5 中缀运算符 [op.over.infix]

1 对中缀运算符 `@` 而言，e_1 `@` e_2 `@` ... `@` e_n 将会调用 `opeartor infix@(`e_1`,`e_2`,...,`e_n`)` 或 `opeartor infix@([`e_1`,`e_2`,...,`e_n`])`，取先匹配的那一个。如果没有找到匹配的重载函数，且该运算符所处的优先级组未定义其结合性，则这是一个编译错误。否则，将会按照其结合性拆分为多次以两个参数调用的形式。若仍然无法找到匹配的重载函数，则这是一个编译错误。

11.3.6 比较运算符 [op.over.cmp]

1 比较运算符的重载解析与一般的中缀运算符相同。`cmp` 的返回类型必须为 `core.Order` 或 `core.Order?`。

```
func operator==(lhs: T, rhs: T) { (lhs cmp rhs) is .equal }
func operator!=(lhs: T, rhs: T) { (lhs cmp rhs) !is .equal }
func operator<(lhs: T, rhs: T) { (lhs cmp rhs) is .less }
func operator!<(lhs: T, rhs: T) { (lhs cmp rhs) !is .less }
func operator<=(lhs: T, rhs: T) { (lhs cmp rhs) is .less | .equal }
func operator>(lhs: T, rhs: T) { (lhs cmp rhs) is .greater }
func operator!>(lhs: T, rhs: T) { (lhs cmp rhs) !is .greater }
func operator>=(lhs: T, rhs: T) { (lhs cmp rhs) is .greater | .equal }
func operator<>(lhs: T, rhs: T) { (lhs cmp rhs) is .less | .greater }
```

2 如果类型 T 重载了 `operator==`，则会自动生成 `operator!=`:

```
func operator!=(lhs: T, rhs: T) { !(lhs == rhs) }
```

`operator<` 与 `opeartor!<`、`operator>` 与 `operator!>` 也有类似的自动生成规则。[`注`：虽然`!in` 不能被重载，但是它也有类似的语义。]

11.3.7 赋值运算符 [op.over.assign]

1 赋值运算符，及与其相同优先级的运算符的重载函数必须返回 `void`。

11.3.8 $ 运算符 [op.over.dollar]

1 `$` 表达式将被转换为对当前序列 s 调用 s.`operator$()`。它只能接受一个 `this` 参数。

11.3.9 条件运算符 [op.over.if]

1 `operator if` 允许类型在条件中被使用。重载了 `operator if`、`operator perfix!`、`opeartor&` 和 `opeartor|` 的类型和 `bool` 称为*布尔类型*。它只能接受一个 `this` 参数，且其返回值类型必须为布尔类型。

2 当布尔类型的表达式 e 被用于条件中时，会不断调用 e.`operator if()`，直到返回值为 `bool` 为止。然后这个值将被用于条件判断。

12 概念 [trait]

TraitDecl:
    *TraitQual** `trait` *Identifier* *TraitBody*

TraitQual:
    `const`

TraitBody:
    `{` *TraitMember** `}`

TraitMember:
    ClassMember
    TypeConstraint

TypeConstraint:
    `type` *Identifier* `;`
    `type` *Identifier* `:` *Type* `;`

[1] 概念对类型进行约束，也可以为满足条件的类型提供额外的功能。

12.1 实现 [impl]

ImplDecl:
    `impl` *Type* *TraitSpec*? *ImplBody*

TraitSpec:
    `:` *Type*

ImplBody:
    `{` *ImplMember** `}`

ImplMember:
    ClassMember

[1] 实现用于为类型添加额外的特性。实现可以显式指定以令类型满足某个概念。

13 泛型 [generic]

GenericSpecification:
`<` *GenericParameters* `>`

GenericParameters:
GenericParameter
GenericParameters `,` *GenericParameter*

GenericParameter:
`dyn?` *Identifier* `...?` *GenericConstraint?* *GenericIfConstraint?*

GenericConstraint:
`: type`
`:` *Type*
GenericTraitConstraint

GenericTraitConstraint:
`impl` *Type*

GenericIfConstraint:
`if` *Expression*

GenericArguments:
GenericArgument
GenericArguments `,` *GenericArgument*

GenericArgument:
Type
`dyn?` *Expression*
`dyn _`

1 *X* 中的实体可以带有编译器的类型或非类型参数进行泛化。

13.1 高阶类型 [kind]

13.2 some 泛型 [generic.some]

1 如果 some 被用于一个函数的参数类型中，则其相当于一个匿名的泛型参数。

[例：

```
func f(a: some A) { }

// 等价于
func<T impl A> f(a: T) { }
```

]

13.3 动态泛型 [generic.dynamic]

1 非类型的泛型参数可以使用关键字 `dyn` 修饰。以这种方式修饰的参数在运行时可变，但可以对其使用静态分析。其对应的泛型实参也必须使用 `dyn` 修饰。

[例：

```
type<dyn I: int> T = { };

let a: T<dyn 0> = { };
let b: T<dyn 1> = a; // 错误，a 和 b 的类型不匹配

func externFunc() -> int;

let n: int = externFunc();

let c: T<dyn n> = a; // 正确：n 是运行时值，编译时值 0 与其兼容
```

]

2 动态参数可以特定方式进行约束。

[例：

```
type<dyn I: int> T = { };

func<dyn I: int> f(a: T<dyn I>, b: T<dyn I>) => a;

func externFunc() -> T<dyn _>; // 未知的动态类型

let a = externFunc();
let b = externFunc();

f(a, b); // 错误，无法保证 a 和 b 的动态类型参数相同

func<dyn I: int> extract(a: T<dyn I>) => I;

if extract(a) == extract(b) {
    f(a, b); // 正确，能发现 a 和 b 的动态类型参数相同
}
```

]

14 类型推导 [deduct]

1 X 可以在适当的地方不提供显式类型，而是让类型自动推导。

14.1 自动推导的静态成员 [deduct.static]

1 使用类型的静态成员和枚举符时，可以省略类型名称。

2 匿名静态成员表达式 e 按照如下方法匹配类型 T：

(2.1) — 如果 T 是枚举类型，且具有枚举符 e，且枚举符参数与 e 匹配，则匹配成功。

(2.2) — 否则，如果 T 具有类型为 T 的静态成员 e，且 e 不含有枚举符参数，则匹配成功。

(2.3) — 否则，如果 e 是函数类型的表达式，且 T 具有名称为 e、返回类型为 T 的静态方法，且 $T.e$ 的参数类型与 e 的兼容，则匹配成功。

(2.4) — 其他情况匹配失败。

[例：

```
enum E { A, B(int), C };

class X {
    static let x = X();
    static func v(i: int) => X();
    init() { }
}

func f(e: E) { }
func g(x: X) { }

f(.A); // 可以
f(.B); // 错误，枚举符参数不一致
f(.B(0)); // 可以
g(.x); // 可以，等价于 g(X.x)
g(.v(0)); // 可以，等价于 g(X.v(0))
```

]

3 匿名静态成员表达式匹配可以跨越方法调用。对方法调用 $o.f$ 而言，如果 T 有方法 f 其类型与 $o.f$ 兼容，且返回类型为 T，则 $o.f$ 匹配 T 当且仅当 o 匹配 T。

[例：

```
enum E { A, B(int), C };
```

```
  enum F { X, Y, Z };

  impl E {
      func f() => self.C;
      func g() => F.X;
  }

  impl F {
      func f() => E.A;
      func g() => E.C;
  }

  func f(e: E) { }

  f(.C.f()); // 可以，推导可以穿过方法调用，且只会检查 E.f
  f(.Y.g()); // 错误，E 上的方法 g 不返回 E
]
```

15 模块 [module]

15.1 导入指令 [import]

ImportDirective:
`import` *ImportPath* `;`
`import` *ImportPath* `:` *ImportItems* `;`

ImportPath:
ExternalImportPath
InternalImportPath
RelativeImportPath

ExternalImportPath:
ImportPathPart
ExternalImportPath `.` *ImportPathPart*

InternalImportPath:
`root` `.` *ImportPathPart*
InternalImportPath `.` *ImportPathPart*

RelativeImportPath:
`.` *ImportPathPart*
RelativeImportPath `.` *ImportPathPart*

ImportPathPart:
Identifier
StringLiteral
`super`

ImportItem:
`*`
`operator`
Identifier
Identifier `as` *Identifier*

1 导入指令用于将其他模块的内容引入到当前模块中。

2 被导入的模块可以通过三种方式指定：外部路径、内部路径或相对路径。

3 外部路径以模块名开始，用于导入外部模块。

4 内部路径以 `root.` 开始，从当前模块根目录开始寻找模块。

5 相对路径以 `.` 开始，从当前模块文件开始寻找模块。

6 模块的路径各部分可以使用一个标识符或者字符串标识。上下文关键字 `root` 指代当前模块的根目录，但只

能用于开头；上下文关键字 `super` 指代当前目录的父目录。模块路径组成可以与 `root` 或 `super` 相同，但此时必须使用字符串表示。

[例：

```
import root.foo.bar; // 导入/foo/bar 模块
import .baz; // 导入当前目录下的 baz 模块
import .super.qux; // 导入父目录下的 qux 模块

import "some-hypen"; // 导入外部模块 some-hypen
import "super"; // 导入外部模块 super
```

]

7 `*` 导入指定模块的所有内容，但是不包括运算符定义。

15.2 访问控制 [access]

AccessQualifier: 以下之一

`public internal private`

1 模块的每个顶层声明都具有**访问可见性**。访问可见性有以下三种类型：公开（`public`）、内部（`internal`）、私有（`private`），其级别依次降低。类成员和实现成员也具有访问可见性。

2 导入一个外部模块的内容时，只有公开的内容才能被导入。导入一个内部模块的内容时，只有公开和内部的内容才能被导入。私有模块只能被该模块内部访问。

3 顶层声明的默认可见性为 `internal`。类字段的默认可见性为 `private`。其他类成员和实现成员的默认可见性为 `public`。如果一个成员没有显式指定可见性，且其所在实体的可见性比其默认可见性要更低，则使用其所在实体的可见性。

[例：

```
class X {
    let x: int; // 默认为 private
    public let y: int; // 显式指定为 public
    func f() {} // 默认为 public
    private func g() {} // 显式指定为 private
}

internal class Y {
    let x: int; // 默认为 private
    public let y: int; // 显式指定为 public
    func f() {} // 默认为 internal
    public func g() {} // 显式指定为 public
}
```

16 特性与修饰符 [attr]

Attribute:

- `@` *Identifier*
- `@` *Identifier* `(` *Arguments?* `)`
- `@` *Identifier* `[` *ExprList?* `]`
- `@` *Identifier* `{` *ObjectItems?* `}`
- `@` *Identifier* `{|` *DictItems?* `|}`

1 特性可以修饰特定的程序实体，以对其添加特定的描述或限制。

16.1 `noreturn` [attr.noreturn]

1 `noreturn` 修饰的函数将不会返回。函数的返回类型必须是 `never`。实现可以对实际控制流能到达函数结尾的 `noreturn` 函数提出一个警告或编译错误。

17 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。

18 杂项 [core.misc]

18.1 类型 [core.type]

1 本节包含了若干类型，它们均为语言内建，但并不是单个关键字。

```
type Symbol = __intrinsic;
```

2 core.Symbol 是所有符号字面量的公共类型。它只能通过内建的符号字面量来获得值。

18.2 序 [core.order]

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

1 core.Order 定义了序关系。它是内建 cmp 运算符的返回值。自定义类型可以通过实现 cmp 运算符来支持序关系，参见 11.3.6。

2 .less 代表左操作数小于右操作数。.equal 代表左操作数等于右操作数。.greater 代表左操作数大于右操作数。

3 cmp 也可以返回 Order?。如果返回值为 nil，则代表两个操作数之间没有序关系。

18.3 范围 [core.range]

```
class Range<T, dyn S: T, dyn E: T>;

class ClosedRange<T, dyn S: T, dyn E: T>;
```

1 core.Range 表示左闭右开区间。它是内建.. 的结果类型。

2 core.ClosedRange 表示闭区间。它是内建..= 的结果类型。

18.3.1 构造器 [core.range.init]

```
func init<T, dyn S: T, dyn E: T>(start: T, end: T) -> Range<T, dyn S, dyn E>;

func init<T, dyn S: T, dyn E: T>(start: T, end: T) -> ClosedRange<T, dyn S, dyn E>;
```

1 Range 和 ClosedRange 可以使用两个参数 start 与 end 构造，分别表示开头与结尾。如果 start > end 则会抛出.InvalidBounds。

18.3.2 实现 [core.range.impl]

```
impl<T, dyn S, dyn E> Range<T, dyn S, dyn E> : Sequence<T, dyn E - S> {
    type Iterator = RangeIterator<T>;
}

impl<T, dyn S, dyn E> ClosedRange<T, dyn S, dyn E> : Sequence<T, dyn (E - S)+!> {
    type Iterator = ClosedRangeIterator<T>;
}
```

1 Range 和 ClosedRange 是序列，其迭代器类型是 core.RangeIterator。

18.3.3 迭代器 [core.range.iter]

```
class RangeIterator<T>;

class ClosedRangeIterator<T>;
```

1 core.RangeIterator 是 Range 的迭代器类型。

2 core.ClosedRangeIterator 是 ClosedRange 的迭代器类型。

19 序列库 [core.seq]

1 core.seq 库定义了序列概念，并提供了一些操作序列的函数。

19.1 概念 [core.seq.traits]

```
trait Sequence<T> {
    type Item = T;
    type Iterator : core.Iterator<T>;

    let size: uint;
    let iterator: Iterator;

    func map(f: T -> U) -> self<U>;

    func operator$() { this.size }
}
```

1 Sequence 表示序列。

2 Item 是序列的元素类型，其始终为 T。

3 Iterator 是序列的迭代器类型，其必须实现了 core.Iterator<T>。

4 size 是序列的大小。序列重载了 operator$，返回 size。

5 iterator 是序列的迭代器。

6 map 将序列中的每个元素应用函数 f，返回一个新的序列。

```
trait Iterator<T> {
    type Item = T;

    func next(this: mut) -> T?;
}
```

7 Iterator 表示迭代器。

8 next 返回迭代器的下一个元素，如果迭代器已经到达末尾，则返回 nil。

19.2 辅助函数 [core.seq.helper]

索引

语法产生式索引

库名称索引