仓颉编程语言语言规约

版本 0.22.17

华为技术有限公司
编译器与编程语言实验室

目录

第一章 词法结构 **1**

1.1 标识符和关键字 1

1.2 分号和换行符 3

1.3 字面量 3

1.3.1 整数类型字面量 4

1.3.2 浮点数类型字面量 5

1.3.3 布尔类型字面量 7

1.3.4 字符串类型字面量 7

1.3.5 Rune 类型字面量 8

1.4 操作符 9

1.5 注释 11

第二章 类型 **13**

2.1 不可变类型 13

2.1.1 数值类型 13

2.1.2 Rune 类型 17

2.1.3 Bool 类型 18

2.1.4 Unit 类型 19

2.1.5 Nothing 类型 19

2.1.6 String 类型 19

2.1.7 Tuple 类型 23

2.1.8 Range 类型 25

2.1.9 Function 类型 26

2.1.10 enum 类型 27

2.2 可变类型 34

2.2.1 Array 类型 34

2.2.2 VArray 类型 38

2.2.3 struct 类型 40

2.2.4 class 类型和 interface 类型 47

2.3 类型转换 47

2.3.1 Value Types 之间的类型转换 47

2.3.2 class/interface 之间的类型转换 50

2.4 类型别名 . . . 51
2.4.1 类型别名定义的规则: . . . 51
2.4.2 类型别名的使用 . . . 53
2.5 类型间的关系 . . . 54
2.5.1 类型相等 . . . 54
2.5.2 子类型 . . . 54
2.5.3 最小公共父类型 . . . 55
2.5.4 最大公共子类型 . . . 55
2.6 类型安全 . . . 55

第三章 名字、作用域、变量、修饰符 **57**
3.1 名字 . . . 57
3.2 作用域 . . . 58
3.2.1 块 . . . 58
3.2.2 作用域级别 . . . 58
3.2.3 作用域原则 . . . 59
3.2.4 遮盖 . . . 62
3.3 变量 . . . 65
3.3.1 变量的定义 . . . 65
3.3.2 变量的初始化 . . . 67
3.4 修饰符 . . . 69
3.4.1 访问修饰符 . . . 69
3.4.2 非访问修饰符 . . . 69

第四章 表达式 **71**
4.1 字面量 . . . 71
4.2 变量名和函数名 . . . 72
4.2.1 泛型函数名作为表达式 . . . 72
4.3 条件表达式 . . . 73
4.4 模式匹配表达式 . . . 76
4.4.1 模式 . . . 77
4.4.2 模式的分类 . . . 83
4.4.3 字符串、字节和 Rune 的匹配规则 . . . 83
4.4.4 Pattern Guards . . . 83
4.5 循环表达式 . . . 84
4.5.1 for-in 表达式 . . . 84
4.5.2 while 表达式 . . . 85
4.5.3 do-while 表达式 . . . 86
4.5.4 循环表达式总结 . . . 87
4.6 try 表达式 . . . 87
4.7 控制转移表达式 . . . 88
4.7.1 break 表达式 . . . 89

4.7.2 continue 表达式 . . . 89
4.7.3 return 表达式 . . . 90
4.7.4 throw 表达式 . . . 91
4.8 数值类型转换表达式 . . . 91
4.9 this 和 super 表达式 . . . 92
4.10 spawn 表达式 . . . 92
4.11 synchronized 表达式 . . . 92
4.12 括号表达式 . . . 92
4.13 后缀表达式 . . . 92
4.13.1 成员访问表达式 . . . 93
4.13.2 函数调用表达式 . . . 93
4.13.3 索引访问表达式 . . . 94
4.13.4 问号操作符 . . . 95
4.14 自增自减表达式 . . . 98
4.15 算术表达式 . . . 99
4.16 关系表达式 . . . 104
4.17 type test 和 type cast 表达式 . . . 105
4.17.1 is 操作符 . . . 105
4.17.2 as 操作符 . . . 106
4.18 位运算表达式 . . . 107
4.19 区间表达式 . . . 109
4.20 逻辑表达式 . . . 109
4.21 coalescing 表达式 . . . 110
4.22 流表达式 . . . 111
4.22.1 pipeline 操作符 . . . 112
4.22.2 composition 操作符 . . . 113
4.23 赋值表达式 . . . 114
4.24 Lambda 表达式 . . . 117
4.25 Quote 表达式 . . . 118
4.26 宏调用表达式 . . . 118
4.27 引用传值表达式 . . . 118
4.28 操作符的优先级和结合性 . . . 118
4.29 表达式的求值顺序 . . . 121

第五章 函数 **123**
5.1 函数定义 . . . 123
5.1.1 函数修饰符 . . . 124
5.1.2 参数 . . . 124
5.1.3 函数体 . . . 128
5.1.4 函数的返回类型 . . . 129
5.1.5 函数声明 . . . 131
5.1.6 函数重定义 . . . 131

5.2 函数类型 . . . 132
5.3 函数调用 . . . 133
5.3.1 命名实参 . . . 133
5.3.2 函数调用类型检查 . . . 133
5.3.3 尾随 Lambda . . . 135
5.3.4 变长参数 . . . 136
5.4 函数作用域 . . . 138
5.5 Lambda 表达式 . . . 139
5.6 闭包 . . . 140
5.7 函数重载 . . . 144
5.7.1 函数重载定义 . . . 144
5.8 mut 函数 . . . 146
5.8.1 定义 . . . 146
5.8.2 接口中的 mut 函数 . . . 147
5.8.3 访问规则 . . . 148

第六章 类和接口 **151**
6.1 类 . . . 151
6.1.1 类的定义 . . . 151
6.1.2 类的成员 . . . 156
6.1.3 Object 类 . . . 171
6.1.4 This 类型 . . . 171
6.2 接口 . . . 172
6.2.1 接口定义 . . . 172
6.2.2 接口成员 . . . 173
6.2.3 接口继承 . . . 176
6.2.4 实现接口 . . . 178
6.2.5 Any 接口 . . . 182
6.3 覆盖、重载、遮盖、重定义 . . . 183
6.3.1 覆盖 . . . 183
6.3.2 重载 . . . 184
6.3.3 遮盖 . . . 185
6.3.4 重定义 . . . 185
6.3.5 访问控制等级限制 . . . 187
6.4 非顶层成员的访问修饰符 . . . 188
6.5 泛型在类与接口中使用的限制 . . . 188
6.5.1 实例化类型导致函数签名重复 . . . 188
6.5.2 类与接口的泛型成员函数 . . . 189

第七章 属性 **191**
7.1 属性的语法 . . . 191
7.2 属性的定义 . . . 193

7.3 属性的实现 . . . 196
7.4 属性的修饰符 . . . 197

第八章 扩展 **201**
8.1 扩展语法 . . . 201
8.1.1 直接扩展 . . . 203
8.1.2 接口扩展 . . . 203
8.2 扩展的成员 . . . 206
8.2.1 函数 . . . 206
8.2.2 属性 . . . 208
8.3 泛型扩展 . . . 210
8.4 扩展的访问和遮盖 . . . 213
8.5 扩展的继承 . . . 215
8.6 扩展的导入导出 . . . 217
8.6.1 直接扩展的导出 . . . 217
8.6.2 接口扩展的导出 . . . 218
8.6.3 导入扩展 . . . 220

第九章 泛型 **223**
9.1 类型形参与类型变元 . . . 223
9.2 泛型约束 . . . 223
9.3 类型型变 . . . 224
9.3.1 定义 . . . 224
9.3.2 泛型不型变 . . . 224
9.3.3 函数类型的型变 . . . 225
9.3.4 元组类型的协变 . . . 225
9.3.5 型变的限制 . . . 225
9.4 泛型约束上界中导出的约束 . . . 225
9.5 泛型函数与泛型类型的定义 . . . 226
9.5.1 泛型函数 . . . 226
9.5.2 泛型类型 . . . 227
9.6 泛型类型检查 . . . 227
9.6.1 泛型声明的检查 . . . 227
9.6.2 泛型声明使用的检查 . . . 229
9.6.3 泛型实例化的深度 . . . 229
9.7 泛型实例化 . . . 230
9.7.1 泛型函数的实例化 . . . 230
9.7.2 类与接口的实例化 . . . 231
9.7.3 struct 的实例化 . . . 232
9.7.4 Enum 的实例化 . . . 232
9.8 泛型函数重载 . . . 232

第十章 重载 **235**
10.1 函数重载 . . . 235
10.1.1 函数重载的定义 . . . 235
10.1.2 重载函数候选集 . . . 236
10.1.3 函数重载决议 . . . 238
10.2 操作符重载 . . . 243
10.2.1 定义操作符函数 . . . 244
10.2.2 操作符函数的作用域以及调用时的搜索策略 . . . 245
10.2.3 可以被重载的操作符 . . . 245
10.2.4 索引操作符重载 . . . 248

第十一章 包和模块管理 **249**
11.1 包 . . . 249
11.1.1 包的声明 . . . 249
11.1.2 包的成员 . . . 250
11.1.3 访问修饰符 . . . 251
11.1.4 包的导入 . . . 253

第十二章 异常 **263**
12.1 Try 表达式 . . . 263
12.1.1 普通 Try 表达式 . . . 264
12.1.2 Try-With-Resources 表达式 . . . 268
12.2 Throw 表达式 . . . 271

第十三章 语言互操作 **273**
13.1 C 语言互操作 . . . 273
13.1.1 unsafe 上下文 . . . 273
13.1.2 调用 C 函数 . . . 273
13.1.3 类型映射 . . . 277
13.1.4 CType 接口 . . . 282

第十四章 元编程 **285**
14.1 quote 表达式和 Tokens 类型 . . . 285
14.1.1 代码插值 . . . 287
14.1.2 quote 表达式求值规则 . . . 288
14.2 宏 . . . 289
14.2.1 宏定义 . . . 289
14.2.2 宏调用 . . . 290
14.2.3 宏作用域和包的导入 . . . 294
14.2.4 嵌套宏和递归宏 . . . 295
14.2.5 限制 . . . 297
14.2.6 内置编译标记 . . . 297

第十五章 并发 **303**
15.1 仓颉线程 . . . 303
15.1.1 创建线程 . . . 303
15.1.2 Future<T> 泛型类 . . . 304
15.1.3 Thread 类 . . . 306
15.1.4 线程睡眠 . . . 307
15.1.5 线程终止 . . . 307
15.2 线程上下文 . . . 308
15.2.1 接口 ThreadContext . . . 308
15.2.2 线程上下文关闭 . . . 308
15.2.3 线程局部变量 . . . 308
15.3 同步机制 . . . 309
15.3.1 原子操作 . . . 309
15.3.2 IllegalSynchronizationStateException . . . 311
15.3.3 IReentrantMutex . . . 312
15.3.4 ReentrantMutex . . . 312
15.3.5 synchronized . . . 313
15.3.6 Monitor . . . 314
15.3.7 MultiConditionMonitor . . . 315
15.4 内存模型 . . . 319
15.4.1 数据竞争 Data Race . . . 319
15.4.2 Happens-Before . . . 319

第十六章 常量求值 **321**
16.1 const 变量 . . . 321
16.2 const 表达式 . . . 323
16.3 const 上下文 . . . 323
16.4 const 函数 . . . 324
16.4.1 接口中的 const 函数 . . . 325
16.5 const init . . . 325

第十七章 注解 **327**
17.1 自定义注解 . . . 327

附录 **A** 仓颉语法 **331**
A.1 词法 . . . 331
A.1.1 注释 . . . 331
A.1.2 空白和换行 . . . 331
A.1.3 符号 . . . 331
A.1.4 关键字 . . . 333
A.1.5 字面量 . . . 335
A.1.6 标识符 . . . 339
A.2 语法 . . . 340

A.2.1 编译单元 340
A.2.2 包定义和包导入 340
A.2.3 top-level 定义 341
A.2.4 类型 354
A.2.5 表达式语法 356

第一章　词法结构

本章主要介绍仓颉编程语言的词法结构，完整的词法和语法的 BNF 表示请参见附录 A。

注：为了增加文档的可读性，正文中的语法定义会和附录中的语法定义略有不同，正文中会将符号和关键字替换为它们的字面表示（而非词法结构中的名字）。

1.1　标识符和关键字

标识符可以用作变量、函数、类型、package 和 module 等等的名字。将标识符分为普通标识符和原始标识符（raw identifier），普通标识符是除了关键字以外，由字母（大写和小写的 ASCII 编码的拉丁字母 A-Z 和 a-z）或下划线（ASCII 编码的 _）开头，后接任意长度的字母、数字（ASCII 编码的数字 0-9）或下划线（ASCII 编码的 _）组合而成的字符串，原始标识符是在普通标识符的外面加上一对反引号（“`..`”），并且反引号内可以使用关键字。标识符的语法定义为：

```
identifier
    : Identifier
    | PUBLIC
    | PRIVATE
    | PROTECTED
    | OVERRIDE
    | ABSTRACT
    | SEALED
    | OPEN
    | REDEF
    | GET
    | SET
    ;

Identifier
    : '_'* Letter (Letter | '_' | DecimalDigit)*
    | '`' '_'* Letter (Letter | '_' | DecimalDigit)* '`'
    ;

Letter
    : [a-zA-Z]
```

```
    ;
DecimalDigit
    : [0-9]
    ;
```

关键字是不能作为标识符使用的特殊字符串，仓颉语言的关键字如下表所示：

Keyword		
as	break	Bool
case	catch	class
const	continue	Rune
do	else	enum
extend	for	from
func	false	finally
foreign	Float16	Float32
Float64	if	in
is	init	inout
import	interface	Int8
Int16	Int32	Int64
IntNative	let	mut
main	macro	match
Nothing	operator	prop
package	quote	return
spawn	super	static
struct	synchronized	try
this	true	type
throw	This	unsafe
Unit	UInt8	UInt16
UInt32	UInt64	UIntNative
var	VArray	where
while		

上下文关键字是可以作为标识符使用的特殊字符串，它们在部分语法中作为关键字存在，但也可以作为普通标识符使用。

Contextual Keyword		
abstract	open	override
private	protected	public
redef	get	set
sealed		

1.2 分号和换行符

有两个符号可以表示表达式或声明的结束：分号（;）和换行符。其中，; 的含义是固定的，无论出现在什么位置都表示表达式或声明的结束，可以将多个使用 ; 分隔的表达式或声明写在同一行。但换行符的含义并不固定，根据它出现的位置不同，既可以像空格符一样作为两个 token 的分隔符，也可以像 ; 一样作为表达式或声明的结束符。

换行符可以出现在任意两个 token 之间，但除了下面列出的禁止使用换行符作为两个 token 之间的分隔符的场景、字符串字面值和多行注释之外，其他情况下，将依据“最长匹配”原则（尽可能地将更多的 token 组成一条合法的表达式或声明）确定将换行符处理为 token 间的分隔符拟或表达式或声明的结束符。“最长”匹配结束前遇到的换行符会被处理为 token 间的分隔符，匹配结束后的换行符被处理为表达式或声明的结束符。举例如下：

```
let width1: Int32 = 32 // The newline character is treated as a terminator.
let length1: Int32 = 1024 // The newline character is treated as a terminator.
let width2: Int32 = 32; let length2: Int32 = 1024 // The newline character is
 ↪  treated as a terminator.
var x = 100 + // The newline character is treated as a connector.
200 * 300 - // The newline character is treated as a connector.
50 // The newline character is treated as a terminator.
```

禁止使用换行符作为两个 token 之间的分隔符的场景：

- 一元操作符和操作数之间禁止使用换行符做分隔符；
- 调用表达式中，(和它之前的 token 之间禁止使用换行符做分隔符；
- 索引访问表达式中，[和它之前的 token 之间禁止使用换行符做分隔符；
- constant pattern 中，$ 和它之后的标识符之间禁止使用换行符做分隔符；

注：上述场景只是禁止了换行符作为两个 token 之间的分隔符的功能，并不代表这些场景中不能使用换行符（如果使用了换行符，会被直接处理为表达式或声明的结束符）。

字符串字面值和多行注释也不适用于“最长匹配”原则：

- 对于单行字符串，当遇到第一个匹配的非转义双引号，即停止匹配；
- 对于多行字符串，当遇到第一个匹配的非转义三个双引号，即停止匹配；
- 对于多行原始字符串，当遇到第一个匹配的非转义双引号以及和开头相同个数的井号（#），即停止匹配；
- 对于多行注释，当遇到第一个匹配的 */，即停止匹配；

1.3 字面量

字面量是一种表达式，它表示的是一个不能被修改的值。

字面量也有类型，仓颉中拥有字面量的类型包括：整数类型、浮点数类型、Rune 类型、布尔类型和字符串类型。字面量的语法为：

```
literalConstant
    : IntegerLiteral
    | FloatLiteral
    | RuneLiteral
    | booleanLiteral
    | stringLiteral
    ;

stringLiteral
    : lineStringLiteral
    | multiLineStringLiteral
    | MultiLineRawStringLiteral
    ;
```

1.3.1 整数类型字面量

整数类型字面量有 4 种进制表示形式：二进制（使用 `0b` 或 `0B` 前缀）、八进制（使用 `0o` 或 `0O` 前缀）、十进制（没有前缀）、十六进制（使用 `0x` 或 `0X` 前缀）。同时可以加可选的后缀来指定整数类型字面量的具体类型。

整数类型字面量的语法定义为：

```
IntegerLiteralSuffix
   : 'i8' |'i16' |'i32' |'i64' |'u8' |'u16' |'u32' | 'u64'
   ;

IntegerLiteral
   : BinaryLiteral IntegerLiteralSuffix?
   | OctalLiteral IntegerLiteralSuffix?
   | DecimalLiteral '_'* IntegerLiteralSuffix?
   | HexadecimalLiteral IntegerLiteralSuffix?
   ;

BinaryLiteral
    : '0' [bB] BinDigit (BinDigit | '_')*
    ;

BinDigit
    : [01]
    ;

OctalLiteral
    : '0' [oO] OctalDigit (OctalDigit | '_')*
```

```
    ;

OctalDigit
    : [0-7]
    ;

DecimalLiteral
    : ([1-9] (DecimalDigit | '_')*) | DecimalDigit
    ;

DecimalDigit
    : [0-9]
    ;

HexadecimalLiteral
    : '0' [xX] HexadecimalDigits
    ;
HexadecimalDigits
    : HexadecimalDigit (HexadecimalDigit | '_')*
    ;

HexadecimalDigit
    : [0-9a-fA-F]
    ;
```

其中 `IntegerLiteralSuffix` 后缀与类型的对应为:

Suffix	Type	Suffix	Type
i8	Int8	u8	UInt8
i16	Int16	u16	UInt16
i32	Int32	u32	UInt32
i64	Int64	u64	UInt64

1.3.2 浮点数类型字面量

浮点数类型字面量有 2 种进制表示形式：十进制（没有前缀）和十六进制（使用 `0x` 或 `0X` 前缀）。十进制浮点数中，整数部分和小数部分（包含小数点）需要至少包含其一，并且无小数部分时必须包含指数部分（使用 `e` 或 `E` 前缀）。十六进制浮点中，整数部分和小数部分（包含小数点）需要至少包含其一，并且必须包含指数部分（使用 `p` 或 `P` 前缀）。同时可以加可选的后缀来指定浮点数类型字面量的具体类型。

浮点数数类型字面量的语法定义为:

```
FloatLiteralSuffix
    : 'f16' | 'f32' | 'f64'
```

```
    ;

FloatLiteral
    : (DecimalLiteral DecimalExponent | DecimalFraction DecimalExponent? |
     ↪  (DecimalLiteral DecimalFraction) DecimalExponent?)  FloatLiteralSuffix?
    | (Hexadecimalprefix (HexadecimalDigits | HexadecimalFraction |
     ↪  (HexadecimalDigits HexadecimalFraction)) HexadecimalExponent)

DecimalFraction
    : '.' DecimalFragment
    ;

DecimalFragment
    : DecimalDigit (DecimalDigit | '_')*
    ;

DecimalExponent
    : FloatE Sign? DecimalFragment
    ;

HexadecimalFraction
    : '.' HexadecimalDigits
    ;

HexadecimalExponent
    : FloatP Sign? DecimalFragment
    ;

FloatE
    : [eE]
    ;

FloatP
    : [pP]
    ;

Sign
    : [-]
    ;

Hexadecimalprefix
```

```
    : '0' [xX]
    ;
```

其中 FloatLiteralSuffix 后缀与类型的对应为:

Suffix	Type
f16	Float16
f32	Float32
f64	Float64

1.3.3 布尔类型字面量

布尔类型字面量只有两个：true 和 false。

```
booleanLiteral
    : 'true'
    | 'false'
    ;
```

1.3.4 字符串类型字面量

字符串字面量可以分为三类：单行字符串字面量，多行字符串字面量，多行原始字符串字面量。

单行字符串字面量使用一对单引号或双引号定义。引号中的内容可以是任意数量的任意字符。如果想要将引号或反斜杠 (\) 作为字符串本身的字符包括在内，需要在其前面加上 (\)。单行字符串字面量不能通过包含换行符来跨越多行。

单行字符串字面量的语法定义为:

```
lineStringLiteral
    : '"' (lineStringExpression | lineStringContent)* '"'
    ;

lineStringExpression
    : '${' SEMI* (expressionOrDeclaration (SEMI+ expressionOrDeclaration?)*) SEMI*
    ↪  '}'
    ;

lineStringContent
    : LineStrText
    ;

LineStrText
    : ~["\\\r\n]
    | EscapeSeq
    ;
```

多行字符串字面量开头结尾需各存在三个双引号（“ “ “）或三个单引号（’’’）。引号中的内容可以是任意数量的任意字符。如果要将用于括起字符串的三个引号（" 或 '）或反斜杠（\）作为字符串本身的字符，则必须在它们前面加上反斜杠（\）。如果在开头的三个双引号后没有换行符，或在当前文件结束之前没有遇到非转义的三个双引号，则编译报错。不同于单行字符串字面量，多行字符串字面量可以跨越多行。

多行字符串字面量的语法定义为：

```
multiLineStringLiteral
    : '"""' NL (multiLineStringExpression | multiLineStringContent)* '"""'
    ;

multiLineStringExpression
    : '${' end* (expressionOrDeclaration (end+ expressionOrDeclaration?)*) end* '}'
    ;

multiLineStringContent
    : MultiLineStrText
    ;

MultiLineStrText
    : ~('\\')
    | EscapeSeq
    ;
```

多行原始字符串字面量以一个或多个井号（#）和一个单引号或双引号（'或"）开头，后跟任意数量的合法字符，直到出现与字符串开头相同的引号和与字符串开头相同数量的井号为止。在当前文件结束之前，如果还没遇到匹配的双引号和相同个数的井号，则编译报错。与多行字符串字面量一样，原始多行字符串字面量可以跨越多行。不同支持在于，转义规则不适用于多行原始字符串字面量，字面量中的内容会维持原样（转义字符不会被转义）。

多行原始字符串字面量的语法定义为：

```
MultiLineRawStringLiteral
    : MultiLineRawStringContent
    ;

fragment MultiLineRawStringContent
    : '#' MultiLineRawStringContent '#'
    | '#' '"' .*? '"' '#'
    ;
```

1.3.5 Rune 类型字面量

一个 Rune 字面量由字符 r 开头，后跟一个（单引号或双引号）单行字符串字面量。字符串字面量内必须恰好包含一个字符。语法为：

```
RuneLiteral
    : 'r' '\'' (SingleChar | EscapeSeq) '\''
    : 'r' '"' (SingleChar | EscapeSeq) '"'
    ;

fragment SingleChar
    :   ~['\\\r\n]
    ;

EscapeSeq
    : UniCharacterLiteral
    | EscapedIdentifier
    ;

fragment UniCharacterLiteral
    : '\\' 'u' '{' HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
     ↪  HexadecimalDigit '}'
    ;

fragment EscapedIdentifier
    : '\\' ('t' | 'b' | 'r' | 'n' | '\'' | '"' | '\\' | 'f' | 'v' | '0' | '$')
    ;
```

1.4 操作符

下表列出了仓颉支持的所有操作符（越靠近表格顶部，操作符的优先级越高），关于每个操作符的详细介绍，请参考表达式。

Operator	Description
`@`	Macro call expression
`.`	Member access
`[]`	Index access
`()`	Function call
`++`	Postfix increment
`--`	Postfix decrement
`?`	Question mark
`!`	Logic NOT
`-`	Unary negative
`**`	Power
`*`	Multiply
`/`	Divide
`%`	Remainder
`+`	Add
`-`	Subtract
`<<`	Bitwise left shift
`>>`	Bitwise right shift
`..`	Range operator
`..=`	
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`is`	Type test
`as`	Type cast
`==`	Equal
`!=`	Not equal
`&`	Bitwise AND

Operator	Description
^	Bitwise XOR
|	Bitwise OR
&&	Logic AND
||	Logic OR
??	coalescing
|>	Pipeline
~>	Composition
=	Assignment
**=	Compound assignment
*=	
/=	
%=	
+=	
-=	
<<=	
>>=	
&=	
^=	
|=	
&&=	
||=	

1.5 注释

仓颉支持如下注释方式:

单行注释，以 // 开头，语法定义为:

```
LineComment
    : '//' ~[\n\r]*
    ;
```

多行注释，注释内容写在 /* 和 */ 之内，支持嵌套，语法定义为:

```
DelimitedComment
    : '/*' ( DelimitedComment | . )*? '*/'
    ;
```

第二章　类型

仓颉编程语言是一种静态类型（statically typed）语言：大部分保证程序安全的类型检查发生在编译期。同时，仓颉编程语言是一种强类型（strongly typed）语言：每个表达式都有类型，并且表达式的类型决定了它的取值空间和它支持的操作。静态类型和强类型机制可以帮助程序员在编译阶段发现大量由类型引发的程序错误。

仓颉中的类型可分为两类：不可变类型（immutable type）和可变类型（mutable type）。其中，不可变类型包括数值类型（分为整数类型和浮点数类型，详见数值类型）、Rune 类型、Bool 类型、Unit 类型、Nothing 类型、String 类型、元组（Tuple）类型、Range 类型、函数（Function）类型、enum 类型；可变类型包括 Array 类型、VArray 类型、struct 类型、class 类型和 interface 类型。

不可变类型和可变类型的区别在于：不可变类型的值，其数据值一经初始化后就不会发生变化；可变类型的值，其数据值初始化后仍然有可以修改的方法。

接下来将逐一介绍每种类型、类型的取值以及它们支持的操作。

2.1　不可变类型

下面依次介绍仓颉编程语言中的不可变类型。

2.1.1　数值类型

数值类型包括整数类型与浮点数类型，分别用于表示整数和浮点数。整数类型包含有符号（signed）整数类型和无符号（unsigned）整数类型，其中，有符号整数类型包括 Int8、Int16、Int32、Int64 和 IntNative，分别用于表示编码长度为 8-bit、16-bit、32-bit、64-bit 和平台相关大小的有符号整数值的类型；无符号整数类型包括 UInt8、UInt16、UInt32、UInt64 和 UIntNative，分别用于表示编码长度为 8-bit、16-bit、32-bit、64-bit 和平台相关大小的无符号整数值的类型。浮点数类型包括 Float16、Float32 和 Float64，分别用于表示编码长度为 16-bit、32-bit 和 64-bit 的浮点数的类型。

IntNative/UIntNative 的长度与当前系统的位宽一致。

下表列出了所有的数值类型以及它们的表示范围：

Type	Range
Int8	$-2^7 \sim 2^7-1$ (-128 to 127)
Int16	$-2^{15} \sim 2^{15}-1$ (-32768 to 32767)
Int32	$-2^{31} \sim 2^{31}-1$ (-2147483648 to 2147483647)
Int64	$-2^{63} \sim 2^{63}-1$ (-9223372036854775808 to 9223372036854775807)
IntNative	platform dependent

Type	Range
UInt8	$0 \sim 2^8 - 1$ (0 to 255)
UInt16	$0 \sim 2^{16} - 1$ (0 to 65535)
UInt32	$0 \sim 2^{32} - 1$ (0 to 4294967295)
UInt64	$0 \sim 2^{64} - 1$ (0 to 18446744073709551615)
UIntNative	platform dependent
Float16	See IEEE 754 binary16 format
Float32	See IEEE 754 binary32 format
Float64	See IEEE 754 binary64 format

其中，为了方便使用，仓颉编程语言中特别提供了一些类型别名（关于类型别名详见下文中的类型别名小节）：

1. Byte 类型作为 UInt8 的类型别名，Byte 与 UInt8 完全等价。
2. Int/UInt 类型分别作为 Int64/UInt64 的类型别名，Int 与 Int64 完全等价，UInt 与 UInt64 完全等价。

```
let a: UInt8 = 128
let b: Byte = a // ok

let c: Int64 = 9223372036854775807
let d: Int = c // ok

let e: UInt64 = 18446744073709551615
let f: UInt = e // ok
```

下面首先介绍数值类型字面量和数值类型支持的操作。

2.1.1.1 数值类型字面量

我们称 5、24、2.9、3.14 等这样的数值为数值类型字面量，字面量只能用它们的值称呼它们，并且它们的值不能被修改。

下面分别介绍整数类型字面量和浮点类型字面量。

整数类型字面量可以使用 4 种进制表示形式：二进制、八进制、十进制、十六进制。其中，二进制以 0b（或 0B）开头，八进制以 0o（或 0O）开头，十六进制以 0x（或 0X）开头，十进制没有前缀。例如，可以将十进制值 24 定义成下列 4 种形式的任意一种，其中下划线 _ 充当分隔符的作用，方便识别数值的位数：

```
0b0001_1000     // Binary.
0o30            // Octal.
24              // Decimal.
0x18            // Hexadecimal.
```

整数类型字面量的语法定义参见整数类型字面量。

浮点数类型符合 IEEE 754 格式：Float16 使用 IEEE 754 中的半精度格式（binary 16），Float32 使用 IEEE 754 中的单精度格式（binary32），Float64 使用 IEEE 754 中的双精度格式（binary64）。一个浮点数通常包含整数部分，小数部分（包含小数点）和指数部分，并且有两种进制表示形式：十进制、十六进制。在十进制表示中，一个浮点数至少要包含一个整数部分或一个小数部分，并且当没有小数部分时必须包含指数部分（以 e 或 E 为前缀，底数为 10）。在十六进制表示中，一个浮点数至少要包含一个整数部分或小数部分（以 0x 开头），同时必须包含指数部分（以 p 或 P 为前缀，底数为 2）。浮点类型量中有几个特殊的值需要注意：正无穷（POSITIVE_INFINITY），负无穷（NEGATIVE_INFINITY），Not a Number（NaN），正 0（+0.0），负 0（-0.0）。其中，无穷表示操作结果超出了表示范围，例如将两个很大的浮点数相乘，或将一个非零浮点数除以浮点零时，其结果都是无穷，无穷的符号由操作数的符号决定，符合“正负得负，负负得正”的规律。NaN 表示既不是实数也不是无穷的情况，例如计算正无穷乘以 0 的结果就是 NaN。浮点类型字面量举例如下：

```
3.14        // decimal Float64 3.14.
2.4e-1      // decimal Float64 0.24.
2e3         // decimal Float64 2000.0.
.8          // decimal Float64 0.8.
.123e2      // decimal Float64 12.3.
0x1.1p0     // hexadecimal Float64 1.0625 (decimal value).
0x1p2       // hexadecimal Float64 4 (decimal value).
0x.2p4      // hexadecimal Float64 2 (decimal value).
```

浮点类型字面量的语法定义参见浮点数类型字面量章节。

仓颉编程语言中的浮点小数类型的最小正值为非正规浮点小数 (subnormal floating point number)。

对于 Float32 类型最小可表示的正浮点数为 2^{-149}，大约为 1.4e-45；而最大值可表示的数为 $(2 - 2^{-23}) \times 2^{127}$，大约为 3.40282e38。

对于 Float64 类型最小可表示的正浮点数为 2^{-1074}，大约为 4.9e-324；而最大可表示的数为 $(2 - 2^{-52}) \times 2^{1023}$，大约为 1.79769e308。

当非 0 的浮点小数字面值因过小或者过大而舍入得到 0 或者无穷，那么编译器会告警。

2.1.1.2 数值类型支持的操作符

数值类型支持的操作符包括：算术操作符、位操作符、关系操作符、自增（减）操作符、一元负号操作符和（复合）赋值操作符。由于整数类型支持所有以上操作符，而浮点类型除了不支持位操作符外，其余操作符均支持，因此接下来简要介绍整数类型上的这些操作符（详细介绍参考表达式章节）。

算术操作符包括加（+）、减（-）、乘（*）、除（/）、取余（%）、取幂（**），默认算术操作符没有被重载（操作符重载详见参见操作符重载章节）的情况下，要求算术操作符的两个操作数的类型必须相同，如果要将两个不同类型的操作数进行操作，必须先进行强制类型转换。关于算术操作符的优先级和结合性，请参考算术表达式中的介绍。下面举例说明整数类型上的算术操作：

```
2 + 3                       // add result: 5
3 - 1                       // sub result: 2
3 * 9                       // mul result: 27
24 / 8                      // div result: 3
```

```
7 % 3                       // mod result: 1
2 ** 3                      // power result: 8
5 + 15 - 2 * 10 / 4         // result: 15
5 + 10 - 3 * 4 ** 2 / 3 % 5  // result: 14
```

位操作符包括位求反（!）、左移（<<）、右移（>>）、位与（&）、位异或（^）、位或（|）。关于位操作符的详细说明，请参考第 4 章中关于位运算表达式的介绍。下面简单举例说明整数类型上的位操作：

```
!10         // bitwise logical NOT operator: -11
10 << 1     // left shift operator: 10*2=20
10 >> 1     // right shift operator: 10/2=5
10 & 15     // bitwise logical AND operator: 10
10 ^ 15     // bitwise XOR operator: 5
10 | 15     // bitwise logical OR operator: 15
0b1010 & 0b1110 ^ 0b1111 | 0b0101   // result: 0b0101
```

关系操作符包括小于（<）、大于（>）、小于等于（<=）、大于等于（>=）、相等（==）、不等（!=），关系表达式的结果是一个 Bool 类型的值（true 或 false）。关于关系操作符的详细说明，请参考第 4 章中关于关系操作符的介绍。下面简单举例说明整数类型上的关系操作：

```
2 == 3      // result: false
2 != 3      // result: true
8 < 10      // result: true
9 <= 9      // result: true
24 > 28     // result: false
24 >= 23    // result: true
```

自增（减）操作符包括自增（++）和自减（--），可看做是一种特殊的赋值操作符（见下），用于实现将变量值加（减）1。自增（减）操作符只能作为后缀操作符使用，且只能作用于整数类型可变变量，因为不可变变量和整数字面量的值不允许修改。关于自增（减）操作符的详细说明，请参考第 4 章中关于自增和自减操作符的介绍。下面简单举例说明整数类型上的自增（减）操作：

```
main() {
    var i: Int32 = 5
    var j: Int32 = 6
    i++ // i=6
    j-- // j=5
    return 0
}
```

一元负号操作符使用 - 表示，结果是对其操作数取负。关于 - 的详细说明，请参考第 4 章中关于一元负号操作符的介绍。下面简单举例说明整数类型上的一元负号操作：

```
var i: Int32 = 5
var j: Int64 = 100
```

```
var k: Int32 = -i   // k = -5
var l: Int64 = -j   // l = -100
```

（复合）赋值操作符包括赋值（=）和复合赋值操作（op=），其中 op 可以是算术操作符、逻辑操作符和位操作符中的任意二元操作符。（复合）赋值操作可以实现一个值为数值类型的表达式到数值类型变量的赋值。关于（复合）赋值操作符的详细说明，请参考第 4 章中关于赋值表达式的介绍。下面简单举例说明整数类型上的（复合）赋值操作：

```
main() {
    var x: Int64 = 5
    var y: Int64 = 10
    x = y   // assignment: x = 10
    x += y  // compound assignment: x = 20
    x -= y  // x = 10
    x *= y  // x = 100
    x /= y  // x = 10
    x %= y  // x = 0
    x = 5   // x = 5
    x **= 2 // x = 25
    x <<= 1 // x = 50
    x >>= 2 // x = 12
    x &= y  // x = 8
    x ^= y  // x = 2
    x |= y  // x = 10
    return 0
}
```

浮点数类型支持的操作符

浮点类型支持的操作包括（注意没有位操作）：算术操作、关系操作、自增（减）操作、一元正（负）号操作、条件操作和（复合）赋值操作。浮点数的计算可能使用与其本身类型不同精度的操作。浮点操作中有几种特殊情况需要注意：在关系表达式中，如果有操作数的值为 NaN，则表达式的结果为 `false`，除了 `NaN != x` 与 `x != NaN` 的结果为 `true`（x 可以是包括 NaN 在内的任意浮点数）。

```
let x = NaN
var y = 3.14
var z = x < y  // z = false
var v = x != x // v = true
var w = (x < y) == !(x >= y) // w = false
```

在算术表达式中，包含 NaN 或无穷的表达式的值遵循 IEEE 754 标准：

2.1.2 Rune 类型

仓颉编程语言使用 Rune 类型表示 Unicode code point。这些值可分为可打印字符、非打印字符（不可显示的字符，如控制符）、特殊字符（如单引号和反斜线）。

- Rune 类型字面量（Rune Literals）

  Rune 类型字面量的语法定义参见Rune 类型字面量。

  Rune 类型的字面量由一对单引号和字符组成，如：

```
'S'
'5'
' ' // blank
```

  非打印和特殊的字符型字面量使用转义字符（\）定义：

escape character	character
\0	Empty character
\\	backslash \
\b	Backspace
\f	Writer
\n	newline
\r	Enter
\t	Horizontal tab
\v	Vertical tab
\’	single quotation mark ’
\”	double quotation marks ”
\u{X}	Any Unicode code point, where X is a 1-8 digit hexadecimal number

- Rune 类型支持的操作

  Rune 类型仅支持关系操作符（根据 Unicode code point 编码进行比较）。

```
main() {
    'A'=='A'    // result: true
    'A'!='A'    // result: false
    'A'<'a'     // result: true
    'A'<='A'    // result: true
    'A'>'a'     // result: false
    'A'>='A'    // result: true
    return 0
}
```

2.1.3 Bool 类型

仓颉编程语言使用 Bool 表示布尔类型。

- Bool 类型字面量（Bool Literal）

  Bool 类型的字面量只有两个：true 和 false，分别表示“真”和“假”。

```
let bool1: Bool = true
let bool2: Bool = false
```

- Bool 类型支持的操作

  仓颉编程语言不支持数值类型与 Bool 之间的类型转换，也禁止非 Bool 的值作为 Bool 值来使用，否则会报编译错误。Bool 类型支持的操作有：赋值操作、部分复合赋值操作（&&=, ||=）、部分关系操作（==, !=）和所有逻辑操作（!, &&, ||）。下面举例说明：

```
main() {
    let bool1: Bool = true
    var bool2: Bool = false
    bool2 = true    // assignment
    bool2 &&= bool1 // bool2=true
    bool2 ||= bool1 // bool2=true
    true == false   // return false
    true != false   // return true
    !false          // logical NOT, return true
    true && false   // logical AND, return false
    false || false  // logical OR, return false
    return 0
}
```

2.1.4 Unit 类型

Unit 类型只有一个值：()。

2.1.5 Nothing 类型

Nothing 是一种特殊的类型，它不包含任何值，并且 Nothing 是所有类型的子类型。Nothing 暗示着死代码，例如如果变量为 Nothing 类型，则其永远不会被使用，因此也不需要为其创建内存；如果函数调用的某个参数为 Nothing 类型，则该参数后面的参数不会被求值，函数调用本身也不会被执行；如果块内某条表达式是 Nothing 类型，则其后的所有表达式和声明均不会被执行到。仓颉编译器会对检测到的死代码进行编译告警。

关于哪些表达式具有 Nothing 类型，参见控制转移表达式。

2.1.6 String 类型

仓颉编程语言使用 String 表示字符串类型，用于表达文本数据。由一串 Unicode 字符组合而成。

2.1.6.1 String 字面量

字符串字面量可以分为三类：单行字符串字面量，多行字符串字面量，多行原始字符串字面量。

字符串字面量的语法定义参见字符串类型字面量章节。

单行字符串字面量使用一对双引号定义（例如，"singleLineStringLiteral"），双引号中的内容可以是任意数量的任意字符（除了非转义的双引号以及单独出现的 \）。单行字符串字面量只能写在同一行中，不能跨越多行。

```
let s1 = ""     // empty string
let s2: String = "Hello Cangjie Lang"       // define string s2
var s3 = "\"Hello Cangjie Lang\""           // define string s3 containing
 ↪  character "
var s4: String = "Hello Cangjie Lang\n" // define string s4 containing character \n
// illegal string with character \
let s5 = "hello\"
// illegal string with character "
let s6 = "Hello "Cangjie Lang"
```

多行字符串字面量以三个双引号（"""）开头，并以三个双引号（"""）结尾。开头的双引号后需要换行，否则报错。字面量从开头的双引号换行后的第一行开始，到遇到的第一个非转义的三个双引号为止，之间的内容可以是任意数量的任意合法字符（除了非转义的三个双引号 """ 和单独出现的 \）。在当前文件结束之前，如果还没遇到非转义的三个双引号，则编译报错。不同于单行字符串字面量，多行字符串字面量必须跨越多行。

```
// empty multi-line string
let s1 = """
"""

// Error: there must be a line terminator after the beginning double quotations.
let errorStr = """abc
"""

/*
The result of s2 is:
This
    is a multi-line string
*/
let s2 = """
This
    is a multi-line string"""

/*
The result of s3 is:
    This
  is a multi-line string
*/
let s3 = """
```

```
    This
  is a multi-line string"""

/*
The result of s4 is:
This
  is a
multi-line string
*/
let s4 = """
This
    is a
multi-line string
"""

/*
The result of s5 is:
This is a

    multi-line string
*/
let s5 = """
This is a\n
    multi-line string
"""
```

多行原始字符串字面量以一个或多个井号（#）和一个双引号开始，并以一个双引号以及和开始相同个数的井号（#）结束。需要注意的是，字符串字面值遇到第一个匹配的双引号以及和开始相同个数的井号即结束匹配。开始的双引号和结束的双引号之间的内容可以是任意数量的任意合法字符（除了一个双引号以及和开始相同个数的井号）。在当前文件结束之前，如果还没遇到匹配的双引号和相同个数的井号，则编译报错。不同于（普通）多行字符串字面量，多行原始字符串字面量中的内容会维持原样（转义字符不会被转义）。

```
// empty multi-line raw string
let empty1 = #""#
// empty multi-line raw string
let empty2 = ##""##

/*
The result of s2 is:

    This
  is a multi-line string
*/
```

```
let s2 = ##"
    This
  is a multi-line raw string"##

/*
The result of s3 is:
This is a\n
    multi-line string
*/
let s3 = #"This is a\n
    multi-line string"#

/*
The result of s4 is:
 This is a "#
*/
let s4 = ##" This is a "#
"##

// Error: the beginning and ending numbers of `#` are not matched.
let errorStr1 = ##" error string "#

// Error: the beginning and ending numbers of `#` are not matched.
let errorStr2 = ##" error string "###

// Error: the string literal is terminated when meeting the first matched `"##`.
let errorStr3 = ##" error string "## error "##
```

String 类型支持使用 == 进行判等（使用 != 进行判不等），两个字符串相等，当且仅当它们对应的 code point sequences 完全相同。

2.1.6.2 插值字符串

插值字符串是一种包含一个或多个插值表达式的字符串字面量（不适用于多行原始字符串字面量）。当插值字符串解析为结果字符串时，每个插值表达式所在位置会被对应表达式的结果替换。

每个插值表达式必须用 {} 括号包起来，并加上 $ 前缀。大括号中支持和块中一样的表达式声明序列，但不能为空。多行字符串中的插值表达式支持换行。

```
let obj = "apples"
let count = 10
let interps1 = "There are ${count * count} ${obj}."
// The result of "interps1" is: There are 100 apples.
let error = "Empty sequence is not allowed ${}" // Error
```

如果字符串中的 $ 符号后面没有 {，则当成普通 $ 符号处理。如果在 $ 前添加了转义字符 \，那么无论 $ 符号后面是不是 {，都不会被当成插值表达式处理。

```
let d1 = "The $ sign."
// The result of "d1" is: The $ sign.
let d2 = "The \${v}."
// The result of "d2" is: The ${v}.
```

插值表达式也支持自定义类型，只要该类型满足 ToString interface 约束。上述已介绍的类型都满足 ToString interface。

2.1.7 Tuple 类型

仓颉编程语言中元组（Tuple）类型是一个由多种不同类型组合而成的不可变（immutable）类型，使用 (Type1, Type2, ..., TypeN) 表示，其中 N 代表元组的维度。关于元组类型，说明如下：

1. Type1 到 TypeN 可以是任意类型（要求 N 不小于 2，即 Tuple 至少是二元的）。
2. 对于元组的每一维度，例如第 K 维，可以存放任何 TypeK 子类型的实例。
3. 当 (Type1, Type2, ..., TypeN) 中的 Type1 到 TypeN 均支持使用 == 进行值判等（使用 != 进行值判不等）时，此 Tuple 类型才支持使用 == 进行值判等（使用 != 进行值判不等）；否则，此 Tuple 类型不支持 == 和 !=（如果使用 == 和 !=，编译报错）。两个同类型的 Tuple 实例相等，当且仅当相同位置（index）的元素全部相等（意味着它们的长度相等）。

Tuple 类型的语法定义为：

```
tupleType
    : '(' type (',' type)+ ')'
    ;
```

2.1.7.1 元组字面量

元组字面量使用格式 (expr1, expr2, ... , exprN) 表示，其中多个表达式之间使用逗号分隔，并且每个表达式可以拥有不同的类型。Tuple literal 的语法定义为：

```
tupleLiteral
    : '(' expression (',' expression)+ ')'
    ;
```

元组字面量举例：

```
(3.14, "PIE")
(2.4, 3.5)
(2, 3, 4)
((2, 3), 4)
```

2.1.7.2 使用元组做解构

元组也可以用来对另一个元组值进行解构：将一个元组中不同位置处的元素分别绑定到不同的变量。下面的例子展示了如何对多返回值函数的返回值进行解构。

```
func multiValues(a: Int32, b: Int32): (Int32, Int32) {
    return (a + b, a - b) // The type of the return value of the function
     ↪  multiValues is (Int32, Int32).
}

main() {
    var (x, y) = multiValues(8,24) // Define an anonymous tuple who has two
     ↪  elements, i.e., x and y.
    print("${x}") // output: 32
    print("${y}") // output: -16
    return 0
}
```

2.1.7.3 元组的下标访问

元组支持通过 tupleName[index] 的方式访问其中某个具体位置的元素（index 表示从 0 开始的下标，并且只能是一个 Int64 类型的字面量）。

```
main() {
    var z = multiValues(8, 24) // the type of z is inferred to be (Int32, Int32)
    print("${z[0]}") // output: 32
    print("${z[1]}") // output: -16
    return 0
}
```

2.1.7.4 定义元组类型的变量

在定义元组类型的变量时，可以省略类型标注，由编译器根据程序上下文推断出具体的类型。

```
let tuplePIE = (3.14, "PIE")     // The type of tuplePIE is inferred to be
 ↪  (Float64, String).
var pointOne = (2.4, 3.5)       // The type of pointOne is inferred to be (Float64,
 ↪  Float64).
var pointTwo = (2, 3, 4)        // The type of pointTwo is inferred to be (Int64,
 ↪  Int64, Int64).
var pointThree = ((2, 3), 4)    // The type of pointThree is inferred to be
 ↪  ((Int64, Int64), Int64).
```

2.1.8 Range 类型

仓颉编程语言使用 Range<T> 表示 Range 类型，并要求 T 只能实例化为实现了 Comparable interface 和 Countable interface 的类型。Range 类型用于表示一个拥有固定步长的序列，并且是一个不可变（immutable）类型。

每个 Range<T> 类型的实例都会包含 start、end 和 step 值。其中，start 和 end 分别表示序列的起始值和终止值，step 表示序列中前后两个元素之间的差值。

Range 类型支持使用 == 进行判等（使用 != 进行判不等），两个相同类型的 Range 实例相等，当且仅当它们同时为“左闭右开”或“左闭右闭”，并且它们的 start 值、end 值和 step 值均对应相等。

2.1.8.1 创建 Range 实例

存在两种 Range 操作符：.. 和 ..=，分别用于创建“左闭右开”和“左闭右闭”的 Range 实例。它们的使用方式分别为 start..end:step 和 start..=end:step（其中 start、end 和 step 均是表达式）。关于这两种表达式，说明如下。

1. 要求 start 和 end 的类型相同，step 的类型为 Int64。

2. 表达式 start..end:step 中，当 step > 0 且 start >= end，或者 step < 0 且 start <= end 时，start..end:step 返回一个空 Range 实例（不包含任何元素的空序列）；如果 start..end:step 的结果是非空 Range 实例，则 Range 实例中元素的个数等于 (end-start)/step 向上取整（即大于等于 (end-start)/step 的最小整数）。

3. 表达式 start..=end:step 中，当 step > 0 且 start > end，或者 step < 0 且 start < end 时，start..=end:step 返回一个空 Range 实例；如果 start..=end:step 的结果是非空 Range 实例，则 Range 实例中元素的个数等于 ((end-start)/step)+1 向下取整（即小于等于 ((end-start)/step)+1 的最大整数）。

```
let range1 = 0..10:1    // Define an half-open range [0,10) (with step = 1) which
 ↪  contains 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.
let range2 = 0..=10:2   // Define a closed range [0,10] (with step = 2) which
 ↪  contains 0, 2, 4, 6, 8 and 10.
let range3 = 10..0:-2   // Define an half-open range [10,0) (with step = -2) which
 ↪  contains 10, 8, 6, 4 and 2.
let range4 = 10..=0:-1  // Define a closed range [10,0] (with step = -1) which
 ↪  contains 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 and 0.
let range5 = 10..0:1    // Define an empty range.
let range6 = 0..0:1     // Define an empty range.
let range7 = 0..=0:1    // Define a closed range [0,0] (with step = 1) which only
 ↪  contains 0.
let range8 = 0..=10:-1  // Define an empty range.
```

4. 表达式 start..end:step 和 start..=end:step 中，step 的值不能等于 0；

```
let range9 = 0..10: 0               // error: the value of the step should not be
↪ zero.
let range10 = 0..=10: 0             // error: the value of the step should not be
↪ zero.
```

5. 除了下节提到的 Range<Int64> 类型的实例用在下标操作符 [] 中时可以省略 start 或 end 外，其他场景下，start 和 end 均是必选的，只有 step 是可选的（默认 step=1）。

```
let range11 = 1..10          // Define an half-open range [1, 10) with step = 1.
let range12 = 1..=10         // Define a closed range [1, 10] with step = 1.
let range13 = ..10           // error: the start value is required.
let range14 = 1..            // error: the end value is required.
let range15 = ..             // error: the start and end value are required.
```

2.1.8.2 Range 类型的使用

Range 类型的表达式主要用在 for-in 表达式中（参见循环表达式），以及作为下标用于截取片段。需要注意的是：

- 当 Range<Int64> 类型的实例用在下标操作符 [] 中时，start 和 end 均是可选的。它们的值由使用的上下文决定：在自定义类型上重载下标操作符 [] 且参数类型是 Range<Int64> 时，使用过程中省略 start 时 start 的值等于 0，省略 end 时 end 的值等于 Int64 的最大值。

2.1.8.3 定义 Range 类型的变量

在定义 Range<T> 类型的变量时，可以显式添加类型标注，也可以省略类型标注（此时由编译器根据程序上下文推断）。

```
let range16: Range<Int64> = 0..10       // Define an half-open range [0,10) with
↪ step = 1
let range17: Range<Int64> = -10..10:2   // Define an half-open range [-10,10) with
↪ step = 2
let range18 = 0..=10                  // Define a closed range [0,10] with step = 1
let range19 = -10..=10:2               // Define a closed range [-10,10] with step
↪ = 2
```

2.1.9 Function 类型

函数类型（function type）表示一个具体函数的类型，它同样是 immutable 的。函数类型的语法定义为：

```
arrowType
    : parameterTypes '->' type
    ;
parameterTypes
    : '(' (type (',' type)*)? ')'
    ;
```

参数类型列表 parameterTypes 和返回类型 type 使用 -> 连接。其中，参数类型列表的一对 () 是必选的，() 内可以包含 0 个或多个类型（多个类型使用 , 分隔）：

```
() -> Int32     // A function type has an empty parameter type list, and its return
 ↪  value type is 'Int32'.
() -> Unit      // A function type has an empty parameter type list, and its return
 ↪  value type is 'Unit'.
(Float32) -> Int32  // A function type has one parameter whose type is 'Float32',
 ↪  and its return value type is 'Int32'
(Int32, Int32, Float32) -> Int32 // A function type has three parameters, and its
 ↪  return value type is 'Int32'
(Int32, Int32, Float32) -> (Int32, Int32) // A function type has three parameters,
 ↪  and its return value type is '(Int32, Int32)'
(Int32, Int32, Float32) -> Unit // A function type has three parameters, and its
 ↪  return value type is 'Unit'.
```

仓颉编程语言中，函数是一等公民（first-class citizens），意味着函数可以作为其他函数的参数，可以作为其他函数的返回值，也可以直接赋值给变量。

函数类型不支持使用 ==（!=）进行判等（判不等）。更多关于函数的介绍，参看第 5 章函数部分。

2.1.10 enum 类型

enum 类型是一种 immutable 类型，用于将一组相关的值（称为 constructor）组织在一起。声明为 enum 类型的值，其在任何时刻只能取 enum 类型定义中的某个 constructor。

2.1.10.1 定义 enum 类型

定义 enum 类型的语法为：

```
enumDefinition
    : enumModifier? 'enum' identifier typeParameters? ('<:' superInterfaces)?
     ↪  genericConstraints? '{' enumBody '}'
    ;

enumBody
    : '|'? caseBody ('|' caseBody)* (functionDefinition |
     ↪  operatorFunctionDefinition | propertyDefinition | macroExpression)*
    ;

caseBody
    : identifier ('(' type (',' type)* ')')?
    ;
```

其中 enumModifier 是 enum 类型的修饰符（即 public），enum 是关键字，identifier 是 enum 类型的名字，typeParameters 和 genericConstraints 分别是类型变元列表和类型变元的约束（参考第 9 章泛

型相关内容)。enumBody 中可以定义若干 caseBody（即 constructor），每个 constructor 可以没有参数，也可以有一组不同类型的参数，多个 constructor 之间使用 | 分隔，第一个 constructor 之前可以存在一个可选的 |。caseBody 后可以定义 enum 的其它成员，包含成员函数、操作符重载函数、成员属性。

enum 类型定义举例:

```
/*
Define an enum type 'TimeUnit1' who has four constructors: 'Year', 'Month', 'Day'
 ↪ and 'Hour'.
*/
enum TimeUnit1 {
    Year | Month | Day | Hour
}

/*
'TimeUnit2' has four constructors: 'Year(Int32)', 'Month(Int32, Float32)',
 ↪ 'Day(Int32, Float32, Float32)' and 'Hour(Int32, Float32, Float32, Float32)'.
*/
enum TimeUnit2 {
    | Year(Float32)
    | Month(Float32, Float32)
    | Day(Float32, Float32, Float32)
    | Hour(Float32, Float32, Float32, Float32)
}

/*
Define a generic enum type 'TimeUnit3<T1, T2>' who has four constructors:
 ↪ 'Year(T1)', 'Month(T1, T2)', 'Day(T1, T2, T2)' and 'Hour(T1, T2, T2, T2)'.
*/
enum TimeUnit3<T1, T2> {
    | Year(T1)
    | Month(T1, T2)
    | Day(T1, T2, T2)
    | Hour(T1, T2, T2, T2)
}
```

关于 enum 类型，需要注意的是:

1. enum 类型只能定义在 top-level。
2. 不支持空括号无参的 constructor 定义，且无参 constructor 的类型为被定义的 enum 类型本身，不视作函数类型。有参的 constructor 具有函数类型，但不能作为一等公民使用。例如:

```
enum E {
    | mkE // OK. The type of mkE is E but not () -> E.
```

```
    | mkE() // Syntax error.
}

enum E1 {
    | A
}
let a = A // ok, a: E1

enum E2 {
    | B(Bool)
}
let b = B // error

enum E3 {
    | C
    | C(Bool)
}
let c = C // ok, c: E3
```

3. 作为一种自定义类型，enum 类型默认不支持使用 ==（!=）进行判等（判不等）。当然，可以通过重载 ==（!=）操作符（参见操作符重载）使得自定义的 enum 类型支持 ==（!=）。

4. 同一个 enum 中支持 constructor 的重载，但是只有参数个数参与重载，参数类型不参与重载，也就是说，允许同一个 enum 中定义多个同名 constructor，但是要求它们的参数个数不同（没有参数的 constructor 虽不为函数类型，也可以与其它 constructors 重载），例如：

```
enum TimeUnit4 {
    | Year
    | Year(Int32)                          // ok
    | Year(Float32)                        // error: redeclaration of 'Year'
    | Month(Int32, Float32)                // ok
    | Month(Int32, Int32)                  // error: redeclaration of 'Month'
    | Month(Int32)                         // ok
    | Day(Int32, Float32, Float32)         // ok
    | Day(Float32, Float32, Float32)       // error: redeclaration of 'Day'
    | Day(Float32, Float32)                // ok
    | Hour(Int32, Float32, Float32, Float32) // ok
    | Hour(Int32, Int32, Int32, Int32)     // error: redeclaration of 'Hour'
    | Hour(Int32, Int32, Int32)            // ok
}
```

5. enum 类型支持递归和互递归定义，例如：

```
// recursive enum
enum TimeUnit5 {
```

```
    | Year(Int32)
    | Month(Int32, Float32)
    | Day(Int32, Float32, Float32)
    | Hour(Int32, Float32, Float32, Float32)
    | Twounit(TimeUnit5, TimeUnit5)
}

// mutually recursive enums
enum E1 {
    A | B(E2)
}
enum E2 {
    C(E1) | D(E1)
}
```

6. enum 不可以被继承。
7. enum 可以实现接口（接口参看第 6 章）。

2.1.10.2 Enum 值的访问

第一种是通过 enum 名加上 constructor 名的访问方式，例如：

```
let time1 = TimeUnit1.Year
let time2 = TimeUnit2.Month(1.0, 2.0)
let time3 = TimeUnit3<Int64, Float64>.Day(1, 2.0, 3.0)
```

第二种是省略 enum 名的方式，例如：

```
let time4 = Year                                // syntax sugar of 'TimeUnit1.Year'
let time5 = Month(1.0, 2.0)                     // syntax sugar of
 ↪  'TimeUnit2.Month(1.0, 2.0)'
let time6 = Day<Int64, Float64>(1, 2.0, 3.0) // syntax sugar of 'TimeUnit3<Int64,
 ↪  Float64>.Day(1, 2.0, 3.0)'
```

注：由于仓颉语言支持对泛型参数的类型推导，所以在使用上述方式一与方式二访问时，泛型参数 <T> 均可省略。

关于第二种使用方式需要满足以下规则。

当不存在其它变量名、函数名、类型名、包名冲突的时候，可以省略类型前缀使用。否则优先选择变量名、函数名、类型名或包名。

```
package p
}

func g() {
```

```
    let a = A // ok, find p.A
    let b = E.A // ok
    let c = p.A // ok
    F(1) // ok, find p.F
    E.F(1) // ok
    p.F(1) // ok
    let x: C // ok, find p.C
    let y: p.C // ok
}
```

enum 构造器在省略类型前缀使用时，可以在构造器名称后面声明 enum 类型的泛型参数。

```
enum E<T> {
    | A(T)
    | B(T)
}

func f() {
    let a = A(1)        // ok, a: E<Int64>
    let b = A<Int32>(2) // ok, b: E<Int32>
}
```

2.1.10.3 enum 解构

使用 match 表达式和 enum pattern 可以实现对 enum 的解构（详见 **4.4.1** 模式章节）。
对前文中定义的 TimeUnit1 和 TimeUnit2 的解构，举例如下：

```
let time12 = TimeUnit1.Year
let howManyHours = match (time12) {
    case Year => 365 * 24 // matched
    case Month => 30 * 24
    case Day => 24
    case Hour => 1
}

let time13 = TimeUnit2.Month(1.0, 1.5)
let howManyHours = match (time13) {
    case Year(y) => y * 365.0 * 24.0
    case Month(y, m) => y * 365.0 * 24.0 + m * 30.0 * 24.0 // matched
    case Day(y, m, d) => y * 365.0 * 24.0 + m * 30.0 * 24.0 + d * 24.0
    case Hour(y, m, d, h) => y * 365.0 * 24.0 + m * 30.0 * 24.0 + d * 24.0 + h
}
```

2.1.10.4 enum 的其它成员

enum 中也可以定义成员函数、操作符函数和成员属性。

- 定义普通成员函数参见函数定义。
- 定义操作符函数的语法参见操作符重载。
- 定义成员属性的语法参见属性的定义。

以下是一些 enum 定义的简单示例。

```
enum Foo {
    A | B | C
    func f1() {
        f2(this)
    }
    static func f2(v: Foo) {
        match (v) {
            case A => 0
            case B => 1
            case C => 2
        }
    }
    prop item: Int64 {
        get() {
            f1()
        }
    }
}

main() {
    let a = Foo.A
    a.f1()
    Foo.f2(a)
    a.item
}
```

enum 中的 constructor、静态成员函数、实例成员函数之间不能重载。因此，enum 的 constructor、静态成员函数、实例成员函数、成员属性之间均不能重名。

2.1.10.5 Option 类型

Option type 是一种泛型 enum 类型：

```
enum Option<T> { Some(T) | None }
```

Some 和 None 是两个 constructor，Some(T) 表示一种有值的状态，None 表示一种无值的状态。类型变元 T 被实例化为不同的类型，会得到不同的 Option type，例如：Option<Int32>、Option<String> 等。

Option 类型的另一种写法是在类型名前加 ?，即对于任意类型 Type，?Type 等价于 Option<Type>。例如，?Int32 等价于 Option<Int32>，?String 等价于 Option<String> 等等。

关于 Option 类型，需要注意：

1. 虽然 T 和 Option<T> 是不同的类型，但是当明确知道某个位置需要的是 Option<T> 类型的值时，可以直接传一个 T 类型的值，编译器会将 T 类型的值封装成 Option<T> 类型的 Some constructor。例如，下面的定义是合法的。

```
let v1: ?Int64 = 100
let v2: ??Int64 = 100
```

2. 对于两个不相等的类型 T1 和 T2，即使它们之间拥有子类型关系，Option<T1> 和 Option<T2> 之间也不会构成子类型关系。

Option 类型的使用遵循泛型 enum 类型的使用规则。例如，可以定义一系列不同 Option 类型的变量：

```
let opInt32_1 = Some(100)           // The type of 'opInt32_1' is 'Option<Int32>'
let opInt32_2 = Option<Int32>.None  // The type of 'opInt32_2' is 'Option<Int32>'
let opChar = Some('m')              // The type of 'opChar' is 'Option<Rune>'
let opBool = Option<Bool>.None      // The type of 'opBool' is 'Option<Bool>'
let opEnum = Some(TimeUnit1.Year)   // The type of 'opEnum' is 'Option<TimeUnit1>'
```

Option 值的解构 提供若干方式以方便对 Option 值的解构：模式匹配、getOrThrow 函数、coalescing 操作符（??）和问号操作符（?）。

1. Option 类型是一种 enum 类型，因此可以使用模式匹配表达式中的 enum pattern（详见enum 模式）实现 Option 值的解构。

```
let number1 = match (opInt32_1) {
    case Some(num) => num // matched
    case None => 0
}

let number2 = match (opInt32_2) {
    case Some(num) => num
    case None => 0        // matched
}

let enumValue = match (opEnum) {
    case Some(tu) => match (tu) {
                         case Year => "Year"   // matched
                         case Month => "Month"
                         case Day => "Day"
```

```
                        case Hour => "Hour"
                    }
    case None => "None"
}
```

2. 对于 Option<T> 类型的表达式 e，通过调用函数 getOrThrow() 或 getOrThrow(exception: ()->Exception) 实现对 e 的解构：如果 e 的值等于 Option<T>.Some(v)，则 e.getOrThrow()（或 e.getOrThrow(lambdaArg))的值等于 v 的值;如果 e 的值等于 Option<T>.None,则 e.getOrThrow() 在运行时抛出 NoneValueException 异常（e.getOrThrow(lambdaArg) 在运行时抛出 lambdaArg 中指定的异常)。

```
let number1 = opInt32_1.getOrThrow()                    // number1 = 100
let number2 = opInt32_2.getOrThrow()                    // throw NoneValueException
let number3 = opInt32_2.getOrThrow{ MyException("Get None value") } // throw
↪ MyException
```

3. 对于 Option<T> 类型的表达式 e1 和 T 类型的表达式 e2，表达式 e1 ?? e2 的类型为 T。当 e1 的值等于 Option<T>.Some(v) 时，e1 ?? e2 的值等于 v 的值；当 e1 的值等于 Option<T>.None 时，e1 ?? e2 的值等于 e2 的值。关于 ?? 操作符的详细介绍，参见coalescing 表达式。

```
let number1 = opInt32_1 ?? 0  // number1 = 100
let number2 = opInt32_2 ?? 0  // number2 = 0
```

1. 问号操作符（? ）是一元后缀操作符。? 需和后缀操作符 .，(),{} 或 [] 一起使用，以实现 Option<T> 对这些后缀操作符的支持。对于一个 Option 类型的表达式 e，e?.b 表示当 e 为 Some 时得到 Option<T>.Some(b)，否则得到 Option<T>.None（T 为 b 的类型），其它操作符同理。关于 ? 操作符的详细介绍，参见问号操作符。

```
class C {
    var item = 100
}
let c = C()
let c1 = Some(c)
let c2 = Option<C>.None
let r1 = c1?.item   // r1 = Option<Int64>.Some(100)
let r2 = c2?.item   // r2 = Option<Int64>.None
```

2.2 可变类型

下面依次介绍仓颉编程语言中的可变类型。

2.2.1 Array 类型

仓颉编程语言使用泛型类型 Array<T> 表示 Array 类型，Array 类型用于存储一系列相同类型（或者拥有公共父类）的元素。关于 Array<T>，说明如下：

1. Array<T> 中的元素是有序的，并且支持通过索引（从 0 开始）访问其中的元素。
2. Array 类型的长度是固定的，即一旦建立一个 Array 实例，其长度是不允许改变的。
3. Array 是引用类型，定义为引用类型的变量，变量名中存储的是指向数据值的引用，因此在进行赋值或函数传参等操作时，Array 拷贝的是引用。
4. 类型变元 T 被实例化成不同的类型，会得到不同的 Array 类型，例如，Array<Int32> 和 Array<Float64> 分别表示 Int32 类型的 Array 和 Float64 类型的 Array。
5. 当 Array<T> 中的 Type 类型支持使用 == 进行值判等（使用 != 进行值判不等）时，Array<T> 类型支持使用 == 进行值判等（使用 != 进行值判不等）；否则，Array<T> 类型不支持 == 和 !=（如果使用 == 和 !=，编译报错）。两个同类型的 Array<T> 实例值相等，当且仅当相同位置（index）的元素全部相等（意味着它们的长度相等）。
6. 多维 Array 定义为 Array 的 Array，使用 Array<Array<...>> 表示。例如，Array<Array<Int32>> 表示 Int32 类型的二维 Array。
7. 当 ElementType 拥有子类型时，Array<ElementType> 中可以存放任何 ElementType 子类型的实例，例如 Array<Object> 中可以存放任何 Object 子类的实例。

2.2.1.1 创建 Array 实例

存在 2 种创建 Array 实例的方式：

构造函数

```
// Array<T>()
let emptyArr1 = Array<Int64>()  // create an empty array whose type is Array<Int64>
let emptyArr2 = Array<String>() // create an empty array whose type is
 ↪  Array<String>

// Array<T>(size: Int64, initElement: (Int64)->T)
let array3 = Array<Int64>(3) { i => i * 2 } // 'array3' has 3 elememts: 0, 2, 4
let array4 = Array<String>(2) { i => "$i" } // 'array4' has 2 elememts: "0", "1"
```

Array 字面量 Array 字面量使用格式 [element1, element2, ... , elementN] 表示，其中多个 element 之间使用逗号分隔，每个 element 可以是一个 expressionElement（普通表达式）。Array 字面量的语法定义为：

```
arrayLiteral : '[' (elements)? ']' ;
elements     : element (',' element)* ;
element      : expression ;
```

Array 字面量每个元素都是一个表达式：

```
let emptyArray: Array<Int64> = []    // empty Array<Int64>
let array0 = [1, 2, 3, 3, 2, 1]      // array0 = [1, 2, 3, 3, 2, 1]
let array1 = [1 + 3, 2 + 3, 3 + 3]   // array1 = [4, 5, 6]
```

- 在上下文没有明确的类型要求时，若所有 element 的最小公共父类型是 T，Array 字面量的类型是 Array<T>。
- 在上下文有明确的类型要求时，此时要求所有 element 的类型都是上下文所要求的 element 类型的子类型。

2.2.1.2 访问 Array 中的元素

对于 Array 类型的实例 arr，支持以下方式访问 arr 中的元素：

访问某个位置处的元素：通过 arr[index1]...[indexN] 的方式访问具体位置处的元素（其中 index1,...,indexN 是索引值的表达式，它们的类型均为 Int64），例如：

```
let array5 = [0, 1]
let element0 = array5[0]      // element0 = 0
array5[1] = 10                // change the value of the second element of 'array5'
 ↪ through index

let array6 = [[0.1, 0.2], [0.3, 0.4]]
let element01 = array6[0][1]  // element01 = 0.2
array6[1][1] = 4.0            // change the value of the last element of 'array6'
 ↪ through index
```

迭代访问：通过 for-in 表达式（见表达式章节）迭代访问 arr 中的元素。例如：

```
func access() {
    let array8 = [1, 8, 0, 1, 0]
    for (num in array8) {
        print("${num}")   // output: 18010
    }
}
```

2.2.1.3 访问 Array 的大小

支持通过 arr.size 的方式返回 arr 中元素的个数。

```
let array9 = [0, 1, 2, 3, 4, 5]
let array10 = [[0, 1, 2], [3, 4, 5]]
let size1 = array9.size  // size1 = 6
let size2 = array10.size // size2 = 2
```

2.2.1.4 Array 的切片

当 Range<Int64> 类型的值用作 Array 下标时，用于截取 Array 的一个片段（称之为 slicing）。需要注意的是：

- step 必须是 1，否则运行时会抛出异常。
- slicing 返回的类型仍然是相同的 Array 类型，并且是原 Array 的引用。对切片中元素的修改会影响到原数组。
- 当使用 start..end 作为 Array 下标时，如果省略 start，则将 start 的值设置为 0，如果省略 end，则将 end 的值设置为 Array 的长度值。
- 当使用 start..=end 作为 Array 下标时，如果 start 被省略，则将 start 的值设置为 0。start..=end 形式的 end 不能省略。
- 如果下标是一个空 range 值，那么返回的是一个空的 Array 。

```
let array7 = [0, 1, 2, 3, 4, 5]

func slicingTest() {
    array7[0..5]      // [0, 1, 2, 3, 4]
    array7[0..5:1]    // [0, 1, 2, 3, 4]
    array7[0..5:2]    // runtime exception
    array7[5..0:-1]   // runtime exception
    array7[5..0:-2]   // runtime exception
    array7[0..=5]     // [0, 1, 2, 3, 4, 5]
    array7[0..=5:1]   // [0, 1, 2, 3, 4, 5]
    array7[0..=5:2]   // runtime exception
    array7[5..=0:-1]  // runtime exception
    array7[5..=0:-2]  // runtime exception
    array7[..4]       // [0, 1, 2, 3]
    array7[2..]       // [2, 3, 4, 5]
    array7[..]        // [0, 1, 2, 3, 4, 5]
    array7[..4:2]     // error: the 'range step' is not allowed here.
    array7[2..:-2]    // error: the 'range step' is not allowed here.
    array7[..:-1]     // error: the 'range step' is not allowed here.

    array7[..=4]      // [0, 1, 2, 3, 4]
    array7[..=4:2]    // error: the 'range step' is not allowed here.

    array7[0..5:-1]   // runtime exception
    array7[5..=0]     // []
    array7[..0]       // []
    array7[..=-1]     // []
    let temp: Array<Int64> = array7[..]
    temp[0] = 6 // temp == array7 == [6, 1, 2, 3, 4, 5]
```

```
}
```

当使用 slicing 进行赋值时，支持两种不同的用法：

- 如果 = 右侧表达式的类型是数组的元素类型，会将这个表达式的值作为元素覆盖切片范围的元素。
- 如果 = 右侧表达式的类型与数组的类型相同，会将这个数组拷贝覆盖当前切片范围，这时要求右侧表达式的 size 必须与切片范围相同，否则运行时会抛出异常。

```
let arr = [1, 2, 3, 4, 5]
arr[..] = 0
// arr = [0, 0, 0, 0, 0]
arr[0..2] = 1
// arr = [1, 1, 0, 0, 0]
arr[0..2] = [2, 2]
// arr = [2, 2, 0, 0, 0]
arr[0..2] = [3, 3, 3] // runtime exception
arr[0..2] = [4] // runtime exception

let arr2 = [1, 2, 3, 4, 5]
arr[0..2] = arr2[0..2] // ok
// arr = [1, 2, 0, 0, 0]
arr[0..2] = arr2 // runtime exception
arr[..] = arr2
```

2.2.2 VArray 类型

仓颉编程语言使用泛型类型 `VArray<T, $N>` 表示 `VArray` 类型，`VArray` 类型用于存储一系列相同类型（或者拥有公共父类型）的元素。关于 `VArray<T, $N>`，说明如下：

1. `VArray<T, $N>` 中的元素是有序的，并且支持通过索引（从 `0` 开始）访问其中的元素。如果提供的索引大于或等于其长度，在编译期间能够推导出下标的则编译报错，否则在运行时抛出异常。

2. `VArray<T, $N>` 类型的长度是类型的一部分。其中 `N` 表示 `VArray` 的长度，它必须是一个整数字面量，通过固定语法 `$` 加数字进行标注。当提供的整数字面量为负数时，编译期间进行报错。

3. `VArray` 是值类型，定义为值类型的变量，变量名中存储的是数据值本身，因此在进行赋值或函数传参等操作时，`VArray` 类型拷贝的是值。

4. 当类型变元 `T` 被实例化成不同的类型，或 `$N` 标注长度不相等时，会得到不同的 `VArray` 类型。例如 `VArray<Int32, $5>` 和 `VArray<Float64, $5>` 是不同类型的 `VArray`。`VArray<Int32, $2>` 和 `VArray<Int32, $3>` 也是不同类型的 `VArray`。

5. `VArray` 的泛型变元 `T` 是 `VArray` 类型时，表示一个多维的 `VArray` 类型。

2.2.2.1 创建 VArray 实例

VArray 同样可以使用 Array 字面量来创建新的实例。这种方式只能在上下文中能明确该字面量是 VArray 类型时才可以使用，否则仍然会优先推断为 Array 类型。

VArray 的长度必须为 Int64 的字面量类型，且必须与提供的字面量 Array 长度一致，否则会编译报错。

```
let arr1: VArray<Int64, $5> = [1,2,3,4,5]
let arr2: VArray<Int16, $0> = []
let arr3: VArray<Int16, $0> = [1] // error
let arr4: VArray<Int16, $-1> = [] // error
```

除此以外 VArray 也可以使用构造函数的形式创建实例。

```
// VArray<T, $N>(initElement: (Int64)->T)
let arr5: VArray<Int64, $5> = VArray<Int64, $5> { i => i } // [0, 1, 2, 3, 4]
// VArray<T, $N>(item!: T)
let arr6: VArray<Int64, $5> = VArray<Int64, $5>(item: 0) // [0, 0, 0, 0, 0]
```

VArray 总是不允许缺省类型参数 <T, $N> 。

2.2.2.2 访问 VArray 中的元素

对于 VArray 类型的实例 arr，支持以下方式访问 arr 中的元素：

通过 arr[index1]...[indexN] 的方式访问具体位置处的元素（其中 index1,...,indexN 是索引值的表达式，它们的类型均为 Int64），例如：

需要注意的是，VArray 的下标取值操作会返回指定元素的拷贝。这意味着如果元素是值类型，那么下标取值会得到一个不可修改的新实例。对于多维 VArray，我们也不能通过 arr[n][m] = e 的方式来修改内层的 VArray，因为通过 arr[n] 所获取的内层 VArray 是一个经过拷贝的新的 VArray 实例。

```
var arr7: VArray<Int64, $2> = [0, 1]
let element0 = arr7[0]      // element0 = 0
arr7[1] = 10                // change the value of the second element of 'array5'
 ↪  through index

// Get and Set of multi-dimensional VArrays.
var arr8: VArray<VArray<Int64, $2>, $2> = [[1, 2], [3, 4]]
let arr9: VArray<Int64, $2> = [0, 1]
let element1 = arr8[1][0]   // element1 = 3
arr8[1][1] = 5              // error: function call returns immutable value
arr8[1] = arr9              // arr8 = [[1, 2], [0, 1]]
```

2.2.2.3 获取 VArray 的长度

支持通过 arr.size 的方式返回 arr 中元素的个数。

```
let arr9: VArray<Int64, $6> = [0, 1, 2, 3, 4, 5]
let size = arr9.size  // size = 6
```

2.2.2.4 VArray 在函数签名中时

当 VArray 作为函数的参数或返回值时，需要标注 VArray 的长度：

```
func mergeArray<T>(a: VArray<T,$2>, b: VArray<T, $3>): VArray<T, $5>
```

2.2.3 struct 类型

struct 类型是一种 mutable 类型，在其内部可定义一系列的成员变量和成员函数，定义 struct 类型的语法为：

```
structDefinition
    : structModifier? 'struct' identifier typeParameters? ('<:' superInterfaces)?
     ↪  genericConstraints? structBody
    ;

structBody
    : '{'
        structMemberDeclaration*
        structPrimaryInit?
        structMemberDeclaration*
      '}'
    ;

structMemberDeclaration
    : structInit
    | staticInit
    | variableDeclaration
    | functionDefinition
    | operatorFunctionDefinition
    | macroExpression
    | propertyDefinition
    ;
```

其中 structModifier 是 struct 的修饰符，struct 是关键字，identifier 是 struct 类型的名字，typeParameters 和 genericConstraints 分别是类型变元列表和类型变元的约束（参考第 9 章泛型相关内容）。structBody 中可以定义成员变量（variableDeclaration），成员属性（propertyDefinition），主构造函数（structPrimaryInit），构造函数（structInit）和成员函数（包括普通成员函数和操作符函数）。

关于 struct 类型，需要注意的是：

1. struct 是值类型，定义为值类型的变量，变量名中存储的是数据值本身，因此在进行赋值或函数传参等操作时，struct 类型拷贝的是值。
2. struct 类型只能定义在 top-level。

3. 作为一种自定义类型，struct 类型默认不支持使用 ==（!=）进行判等（判不等）。当然，可以通过重载 ==（!=）操作符（参见操作符重载）使得自定义的 struct 类型支持 ==（!=）。
4. struct 不可以被继承。
5. struct 可以实现接口。
6. 如果一个 struct 类型中的某个（或多个）非静态成员变量的类型中引用了 struct 自身，则称此 struct 为递归 struct 类型。对于多个 struct 类型，如果它们的非静态成员变量的类型之间构成了循环引用，则称这些 struct 类型间构成了互递归。递归（或互递归）定义的 struct 类型是非法的，除非每条递归链 T_1，T_2，...，T_N 上都存在至少一个 T_i 被封装在 Class、Interface、Enum 或函数类型中，也就是说，可以使用 Class、Interface、Enum 或函数类型使递归（或互递归）的 struct 定义合法化。

struct 类型定义举例：

```
struct Rectangle1 {
    let width1: Int32
    let length1: Int32
    let perimeter1: () -> Int32

    init (width1: Int32, length1: Int32) {
        this.width1 = width1
        this.length1 = length1
        this.perimeter1 = { => 2 * (width1 + length1) }
    }
    init (side: Int32) {
        this(side, side)
    }
    func area1(): Int32 { width1 * length1 }
}

// Define a generic struct type.
struct Rectangle2<T> {
    let width2: T
    let length2: T

    init (side: T) {
      this.width2 = side
      this.length2 = side
    }

    init (width2!: T, length2!: T) {
        this.width2 = width2
        this.length2 = length2
    }
}
```

递归和互递归 struct 类型定义举例：

```
struct R1 { // error: 'R1' cannot have a member that recursively contains it
    let other: R1
}
struct R2 { // ok
    let other: Array<R2>
}

struct R3 { // error:  'R3' cannot have a member that recursively contains it
    let other: R4
}
struct R4 { // error:  'R4' cannot have a member that recursively contains it
    let other: R3
}

struct R5 { // ok
    let other: E1
}
enum E1 { // ok
    A(R5)
}
```

2.2.3.1 struct 成员变量

定义成员变量的语法为：

```
variableDeclaration
    : variableModifier* NL* (LET | VAR) NL* patternsMaybeIrrefutable ((NL* COLON
    ↪  NL* type)? (NL* ASSIGN NL* expression) | (NL* COLON NL* type))
    ;
```

在定义成员变量的过程中，需要注意的是：

1. 在主构造函数之外定义的成员变量可以有初始值，也可以没有初始值。如果有初始值，初始值表达式中仅可以引用定义在它之前的已经初始化的成员变量，以及 struct 中的静态成员函数。

2.2.3.2 构造函数

在仓颉编程语言中，有两种构造函数：主构造函数和 init 构造函数 (简称构造函数)。

主构造函数　主构造函数的语法定义如下：

```
structPrimaryInit
    : (structNonStaticMemberModifier)? structName '('
```

```
      structPrimaryInitParamLists? ')'
      '{'
          expressionOrDeclarations?
      '}'
    ;

structName
    : identifier
    ;

structPrimaryInitParamLists
    : unnamedParameterList  (','  namedParameterList)?
       (',' structNamedInitParamList)?
    | unnamedParameterList (',' structUnnamedInitParamList)?
       (',' structNamedInitParamList)?
    | structUnnamedInitParamList (',' structNamedInitParamList)?
    | namedParameterList (',' structNamedInitParamList)?
    | structNamedInitParamList
    ;

structUnnamedInitParamList
    : structUnnamedInitParam (',' structUnnamedInitParam)*
    ;

structNamedInitParamList
    : structNamedInitParam (',' structNamedInitParam)*
    ;

structUnnamedInitParam
    : structNonStaticMemberModifier?  ('let' | 'var')  identifier ':' type
    ;

structNamedInitParam
    : structNonStaticMemberModifier?   ('let' | 'var')  identifier '!' ':' type
      ('=' expression)?
    ;
```

主构造函数的定义包括以下几个部分：

1、修饰符：可选。可以被所有访问修饰符修饰，默认的可访问性为 `internal`。详细内容请参考包和模块管理章节访问修饰符。

2、主构造函数名：与类型名一致。主构造函数名前不允许使用 `func` 关键字。

3、形参列表：主构造函数与 `init` 构造函数不同的是，前者有两种形参：普通形参和成员变量形参。普

通形参的语法和语义与函数定义中的形参一致。

引入成员变量形参是为了减少代码冗余。成员变量形参的定义，同时包含形参和成员变量的定义，除此之外还表示了通过形参给成员变量赋值的语义。省略的定义和表达式会由编译器自动生成。

- 成员变量形参的语法和成员变量定义语法一致，此外，和普通形参一样支持使用 ! 来标注是否为命名形参；
- 成员变量形参的修饰符有：public, private, protected, internal；详细内容请参考包和模块管理章节访问修饰符
- 成员变量形参只允许非静态成员变量，即不允许使用 static 修饰；
- 成员变量形参不能与主构造函数外的成员变量同名；
- 成员变量形参可以没有初始值。这是因为主构造函数会由编译器生成一个对应的构造函数，将在构造函数体内完成将形参给成员变量的赋值；
- 成员变量形参也可以有初始值，初值表达式中可以引用该成员变量定义之前已经定义的其他形参或成员变量（不包括定义在主构造函数外的实例成员变量），但不能修改这些形参和成员变量的值。需要注意的是，成员变量形参的初始值只在主构造函数中有效，不会在成员变量定义中包含初始值；
- 成员变量形参后不允许出现普通形参，并且要遵循函数定义时的参数顺序，命名形参后不允许出现非命名形参。

4、主构造函数体：主构造函数不允许调用本 struct 中其它构造函数。主构造函数体内允许写声明和表达式，其中声明和表达式需要满足 init 构造函数的要求。

主构造函数定义的例子如下：

```
struct Test{
    static let counter: Int64 = 3
    let name: String = "afdoaidfad"
    private Test(
        name: String,              // regular parameter
        annotation!: String = "nnn",    // regular parameter
        var width!: Int64 = 1,     // member variable parameter with initial value
        private var length!: Int64,    // member variable parameter
        private var height!: Int64 = 3   // member variable parameter
    ) {}
}
```

主构造函数是 init 构造函数的语法糖，编译器会自动生成与主构造函数对应的构造函数和成员变量的定义。自动生成的构造函数形式如下：

- 其修饰符与主构造函数修饰符一致；
- 其形参从左到右的顺序与主构造函数形参列表中声明的形参一致；
- 构造函数体内形式如下：
  - 首先是对成员变量的赋值，语法形式为 this.x = x, 其中 x 为成员变量名；

- 然后是主构造函数体的其它代码；

```
struct B<X,Y> {
    B(
        x: Int64,           // primary constructor, it's name is the same as the
         ↪ struct
        y: X,
        v!: Int64 = 1,      // regular parameter
        private var z!: Y   // member variable parameter
    ) {}

    /* The corresponding init constructor with primary constructor auto-generated
    by compiler.

    private var z: Y    // auto generated member variable definition
    init( x: Int64, y: X, v!: Int64 = 1, z!: Y) { // auto generated named parameter
     ↪ definition
        this.z = z // auto generated assign expression of member variable
    }
    */
}
```

一个 struct 最多可以定义一个主构造函数，除了主构造函数之外，可以照常定义其他构造函数，但要求其他构造函数必须和主构造函数所对应的构造函数构成重载。

init 构造函数　除了主构造函数，还可以自定义构造函数，构造函数使用关键字 init 加上参数列表和函数体的方式定义。一个 struct 中可以定义多个构造函数，但要求它们和主构造函数所对应的构造函数必须构成重载（关于函数重载，请参见函数重载定义）。另外：

1. 构造函数的参数可以有默认值。禁止使用实例成员变量 this.variableName 及其语法糖 variableName 作为构造函数参数的默认值；
2. 构造函数在所有实例成员变量完成初始化之前，禁止使用隐式传参或捕获了 this 的函数或 lambda，但允许使用 this.variableName 或其语法糖 variableName 来访问已经完成初始化的成员变量 variableName；
3. 构造函数中的 lamdba 和嵌套函数不能捕获 this ， this 也不能作为表达式单独使用。
4. 构造函数中允许通过 this 调用其他构造函数。如果调用，必须在构造函数体内的第一个表达式处，在此之前不能有任何表达式或声明。在构造函数体外，不允许通过 this 调用表达式来调用构造函数。
5. 若构造函数没有显式调用其他构造函数，则需要确保 return 之前本 struct 声明的所有实例成员变量均完成初始化，否则编译报错；
6. 编译器会对所有构造函数之间的调用关系进行依赖分析，循环的调用将编译报错。

7. 构造函数的返回类型为 Unit。

如果一个 struct 既没有定义主构造函数，也没有定义 init 构造函数，则会尝试生成一个（public 修饰的）无参构造函数。如果存在本 struct 的实例成员变量没有初始值，则编译报错。

2.2.3.3 struct 的其它成员

struct 中也可以定义成员函数、操作符函数、成员属性和静态初始化器。

- 定义普通成员函数参见函数定义。
- 定义操作符函数的语法参见操作符重载。
- 定义成员属性的语法参见属性的定义。
- 定义静态初始化器的语法参见静态初始化器。

2.2.3.4 struct 中的修饰符

struct 及其成员可以使用访问修饰符进行修饰，详细内容请参考包和模块管理章节访问修饰符。

成员函数和变量可以使用 static 修饰，这些成员是静态成员，静态成员属于 struct 类型的成员，而不是 struct 实例的成员。在 struct 定义外部，实例成员变量和实例成员函数只能通过 struct 实例访问，静态成员变量和静态成员函数只能通过 struct 类型的名字访问。

另外函数还可以被 mut 修饰，mut 函数是一种特殊的实例成员函数。mut 函数详细介绍参考函数章节mut 函数。

struct 构造函数以及主构造函数内部定义的成员变量只能使用访问修饰符修饰，不能使用 static 修饰。

2.2.3.5 struct 的实例化

定义完 struct 类型之后，就可以创建对应的 struct 实例。struct 实例的定义方式按照是否包含类型变元可分为两种:

1. 定义非泛型 struct 的实例：StructName(arguments)。其中 StructName 为 struct 类型的名字，arguments 为实参列表。StructName(arguments) 会根据重载函数的调用规则（参见函数重载决议）调用对应的构造函数，然后生成 StructName 的一个实例。举例如下:

```
let newRectangle1_1 = Rectangle1(100, 200) // Invoke the first custom constructor.
let newRectangle1_2 = Rectangle1(300)      // Invoke the second custom constructor.
```

1. 定义泛型 struct 的实例：StructName<Type1, Type2, ... , TypeK>(labelValue1, labelValue2, ... , labelValueN)。与定义非泛型 struct 的实例的差别仅在于需要对泛型参数进行实例化。举例如下:

```
let newRectangle2_1 = Rectangle2<Int32>(100) // Invoke the custom constructor.
let newRectangle2_1 = Rectangle2<Int32>(width2: 10, length2: 20) // Invoke another
↪  custom constructor.
```

最后，需要注意的是：对于 struct 类型的变量 structInstance，如果它使用 let 定义，不支持通过 structInstance.varName = expr 的方式修改成员变量 varName 的值（即使 varName 使用 var 定义）；如果 structInstance 使用 var 定义，并且 varName 同样使用 var 定义，支持通过 structInstance.varName = expr 的方式修改成员变量 varName 的值。

2.2.4 class 类型和 interface 类型

class 和 interface 是引用类型，定义为引用类型的变量，变量名中存储的是指向数据值的引用，因此在进行赋值或函数传参等操作时，class 和 interface 拷贝的是引用。

请参见类和接口

2.3 类型转换

作为一种强类型（strongly typed）语言，仓颉编程语言仅支持显式类型转换（亦称强制类型转换）。对于值类型，支持使用 valueType(expr) 实现将表达式 expr 的类型转换成 valueType；对于 class 和 interface，通过使用 as 操作符（见as 操作符）实现静态类型转换。对于值类型，我们说 valueTypeA 到 valueTypeB 是可转换的，是指定义了将 valueTypeA 转换成 valueTypeB 的转换规则，对于没有定义转换规则的两个值类型，我们称它们是不可转换的。对于 class 和 interface，如果两个类型在继承关系图上存在父子类型关系，那么这两个类型之间就是可转换的（当然，转换的结果也有可能是失败的），否则，这两个类型之间就是不可转换的。

下面分别介绍值类型之间的类型转换规则和 class/interface 之间的类型转换规则。对于其它未提及的类型，仓颉不支持通过上述两种方式实现它们之间的类型转换。

2.3.1 Value Types 之间的类型转换

对于数值类型，支持如下类型转换（未列出即表示不支持）：

1. Rune 类型到 UInt32 类型的转换；
2. 整数类型（包括 Int8, Int16, Int32, Int64, IntNative, UInt8, UInt16, UInt32, UInt64, UIntNative）到 Rune 类型的转换。
3. 所有数值类型（包括 Int8，Int16，Int32，Int64，IntNative，UInt8，UInt16，UInt32，UInt64，UIntNative，Float16，Float32，Float64）之间的双向转换。

Rune 到 UInt32 的转换使用 UInt32(e) 的方式，其中 e 是一个 Rune 类型的表达式，UInt32(e) 的结果是 e 的 Unicode scalar value 对应的 UInt32 类型的整数值。

整数类型到 Rune 的转换使用 Rune(num) 的方式，其中 num 的类型可以是任意的整数类型，且仅当 num 的值落在 [0x0000, 0xD7FF] 或 [0xE000, 0x10FFFF]（即 Unicode scalar value）中时，返回对应的 Unicode scalar value 表示的字符，否则，编译报错（编译时可确定 num 的值）或运行时抛异常。

```
main() {
    var c: Rune = 'a'
    var num: UInt32 = 0
```

```
    num = UInt32(c) // num = 97
    num -= 32       // num = 65
    c = Rune(num)   // c = `A`
    return 0
}
```

为了保证类型安全，仓颉编程语言不支持数值类型之间的隐式类型转换（数值字面量的类型由上下文推断得到，这种情形并不是隐式类型转换），要实现一种数值类型到另外一种数值类型的转换，必须使用显式的方式：NumericType(expr)，表示将 expr 的类型强制转换为 NumericType 类型（NumericType 表示任意一种数值类型），如转换成功，会返回一个新的从 expr 构造而来的类型为 NumericType 的值。数值类型转换的语法定义为：

```
numericTypeConvExpr
    : numericTypes '(' expression ')'
    ;
numericTypes
    : 'Int8' | 'Int16' | 'Int32' | 'Int64' | 'IntNative' | 'UInt8' | 'UInt16' |
    ↪ 'UInt32' | 'UInt64' | 'UIntNative' | 'Float16' | 'Float32' | 'Float64'
    ;
```

如果根据数值类型所占 bit 数来定义数值类型间的“大小关系”（所占 bit 数越多，类型越“大”，所占 bit 数越少，类型越“小”），则仓颉编程语言支持以下类型转换：

a) 有符号整数类型之间的双向转换：小转大时数值结果不变，大转小时若超出小类型的表示范围，则根据上下文中的属性宏确定溢出处理策略（默认使用抛出异常的策略），不同溢出策略详见算术表达式。下面以 Int8 和 Int16 之间的转换为例进行说明（溢出时，使用抛异常的处理策略）：

```
main() {
    var i8Number: Int8 = 127
    var i16Number: Int16 = 0
    i16Number = Int16(i8Number)     // ok: i16Number = 127
    i8Number = Int8(i16Number)      // ok: i8Number = 127
    i16Number = 128
    i8Number = Int8(i16Number)      // throw an ArithmeticException
    return 0
}
```

b) 无符号整数类型之间的双向转换：规则同上。以 UInt16 和 UInt32 之间的转换为例进行说明（其他情况遵循一样的规则）：

```
main() {
    var u16Number: UInt16 = 65535
    var u32Number: UInt32 = 0
    u32Number = UInt32(u16Number)   // ok: u32Number = 65535
    u16Number = UInt16(u32Number)   // ok: u16Number = 65535
```

```
    u32Number = 65536
    u16Number = UInt16(u32Number)   // throw an ArithmeticException
    return 0
}
```

c) 浮点类型之间的双向转换：使用 round-to-nearest 模式。举例说明 Float32 和 Float64 之间的转换：

```
main() {
    var f32Number: Float32 = 1.1
    var f64Number: Float64 = 0.0
    f64Number = Float64(f32Number)      // f64Number = 1.100000023841858
    f32Number = Float32(f64Number)      // f32Number = 1.1
    f64Number = 1.123456789
    f32Number = Float32(f64Number)      // f32Number = 1.1234568
    f32Number = 4.4E38                  // f32Number = POSITIVE_INFINITY
    f64Number = Float64(f32Number)      // f64Number = POSITIVE_INFINITY
    f64Number = 4.4E38
    f32Number = Float32(f64Number)      // f32Number = POSITIVE_INFINITY
    f64Number = Float64(f32Number*0.0)
    f32Number = Float32(f64Number)      // f32Number = NaN
    return 0
}
```

d) 有符号整数类型和无符号整数类型之间的双向转换：因为任何有符号整数类型的表示范围均不能包含长度相同的无符号整数类型的表示范围（反之亦然），因此它们之间进行转换时，只要待转换表达式的值落在目标整数类型的表示范围之内则转换成功，否则根据上下文中的属性宏确定溢出处理策略（默认使用抛出异常的策略），不同溢出策略详见算术表达式。下面以 Int8 和 UInt8 之间的转换为例进行说明（溢出时，使用抛异常的处理策略）：

```
main() {
    var i8Number: Int8 = 127
    var u8Number: UInt8 = 0
    u8Number = UInt8(i8Number)  // ok: u8Number = 127
    u8Number = 100
    i8Number = Int8(u8Number)   // ok: i8Number= 100
    i8Number= -100
    u8Number = UInt8(i8Number)  // throw an ArithmeticException
    u8Number = 255
    i8Number = Int8(u8Number)   // throw an ArithmeticException
    return 0
}
```

e) 整数转换为浮点数：结果为尽可能接近原整数的浮点数。超出目标类型的表示范围时，返回 POSITIVE_INFINITY 或 NEGTIVE_INFINITY。

e) Converting from an integer to floating point types: it will produce the closest possible float value. POSITIVE_INFINITY or NEGTIVE_INFINITY is returned when the value exceeds the range of the target type.

f) 浮点数转换为整数：浮点类型到有符号整数类型的转换使用 round-toward-zero 模式，即保留整数部分舍弃小数部分。当整数部分超出目标整数类型的表示范围，则根据上下文中的整数溢出策略处理。如果是 throwing 策略，那么抛出异常；否则按如下规则转换：

- NaN 返回 0
- 小于整数取值范围下界时（包括负无穷），返回整数的取值范围下界
- 大于整数取值范围上界时（包括正无穷），返回整数的取值范围上界

f) Converting from a floating point to integer types: it follows the round-toward-zero mode, i.e. the fractional part is discarded. When the integer part is beyond the range of the target type, the result depends on the integer overflow strategy. If the strategy is throwing, an exception is thrown; otherwise, the result is as follows: * NaN will return 0 * Values larger than the maximum integer value, including POSITIVE_INFINITY, will saturate to the maximum value of the integer type. * Values smaller than the minimum integer value, including NEGTIVE_INFINITY, will saturate to the minimum value of the integer type.

```
main() {
    var i32Number: Int32 = 1024
    var f16Number: Float16 = 0.0
    var f32Number: Float32 = 0.0
    f16Number = Float16(i32Number)  // ok: f16Number = 1024.0
    f32Number = Float32(i32Number)  // ok: f32Number = 1024.0
    i32Number = 2147483647
    f16Number = Float16(i32Number)  // f16Number = POSITIVE_INFINITY
    f32Number = Float32(i32Number)  // precision lost: f32Number = 2.14748365E9

    f32Number = 1024.1024
    i32Number = Int32(f32Number)    // ok: i32Number = 1024
    f32Number = 1024e10
    i32Number = Int32(f32Number)    // throw an Exception
    f32Number = 3.4e40              // f32Number = POSITIVE_INFINITY
    i32Number = Int32(f32Number)    // throw an Exception
    f32Number = 3.4e40 * 0.0        // f32Number = NaN
    i32Number = Int32(f32Number)    // throw an Exception
    return 0
}
```

2.3.2 class/interface 之间的类型转换

对于一个 class/interface 类型的实例 obj，如果需要将它的（静态）类型转换到另一个 class/interface 类型 TargetType，可使用：obj as TargetType。

关于 as 操作符的使用，以及 class/interface 之间的类型转换规则，参见as 操作符。

2.4 类型别名

当某个类型的名字比较复杂或者在特定场景中不够直观时，可以选择使用类型别名的方式为此类型取一个简单并且直观的别名。定义类型别名的语法为：

```
typeAlias
    : typeModifier? `type` identifier typeParameters? `=` type
    ;
```

其中，typeModifier 是可选的可访问性修饰符（即 public），type 是关键字，identifier 是任意的合法标识符，type 是任意的在 top-level 可见的类型，identifier 和 type 之间使用 = 进行连接。另外，也可通过在 identifier 之后添加类型参数（typeParameters）的方式定义泛型（参考第 9 章泛型）别名。通过以上声明，即为类型 type 定义了一个名字为 identifier 的别名，并且 identifier 和 type 被视作同一种类型。例如：

```
type Point2D = (Float64, Float64)
type Point3D = (Float64, Float64, Float64)

let point1: Point2D = (0.5, 0.8)
let point2: Point3D = (0.5, 0.8, 1.1)
```

上述 type 定义并不会定义一个新的类型，它的作用仅仅是为某个已有类型定义另外一个名字而已，别名和原类型被视作同一个类型，并且别名不会对原类型的使用带来任何影响。

2.4.1 类型别名定义的规则：

1. 类型别名的定义只能出现在 top-level。

```
func test() {
    type Point2D = (Float64, Float64)           // error: type alias can only be
     ↪  defined at top-level
    type Point3D = (Float64, Float64, Float64)  // error: type alias can only be
     ↪  defined at top-level
}
```

2. 使用 type 定义类型别名时，原类型必须在 type 定义的位置可见。

```
class LongNameClassA {

}
type ClassB = LongNameClassB    // error: use of undeclared type 'LongNameClassB'
```

3. 定义泛型别名时，如果泛型别名中引入了原类型中没有使用的泛型参数，则编译器会告警。

```
type Class1<V> = GenericClassA<Int64, V>        // ok. ClassA is a generic class
type Class2<Value, V> = GenericClassB<Int64, V> // warning: the type parameter
 ↪  'Value' in 'Class2<Value, V>' is not used in 'GenericClassB<Int64, V>'
type Int<T> = Int32                             // warning: the type parameter 'T'
 ↪  in 'Int<T>' is not used in `Int32`
```

4. 定义泛型别名时，不允许为别名和原类型中的类型参数添加泛型约束，在用到泛型别名时，可以按需为其添加泛型约束。另外，原类型中已有的泛型约束会“传递”至别名。

```
type Class1<V> where V <: MyTrait = GenericClassA<Int64, V> // error: generic
 ↪  constraints are not allowed here
type Class2<V> = GenericClassB<Int64, V> where V <: MyTrait // error: generic
 ↪  constraints are not allowed here

type Class3<V> = GenericClassC<Int64, V>
func foo<V> (p: Class3<V>) where V <: MyTrait {             // add generic
 ↪  constraints when 'Class3<V>' is used
    functionBody
}

class ClassWithLongName<T> where T<:MyTrait {
    classBody
}
type Class<T> = ClassWithLongName<T>                      // Class<T> also has the
 ↪  constraint 'where T<:MyTrait'
```

5. 一个（或多个）type 定义中禁止出现循环引用（无论是直接的或是间接的）。其中，判断循环引用的方式是通过名字判断是否存在循环引用，并不是使用类型展开的方式。

```
type Value = GenericClassAp<Int64, Value>   // error: 'Value' references itself
type Type1 = (Int64)->Type1                 // error: 'Type1' references itself
type Type2 = (Int64, Type2)                    // error: 'Type2' references itself
type Type3 = Type4                          // error: 'Type3' indirectlly
 ↪  references itself
type Type4 = Type3
```

6. 类型别名被视为与原类型等价的类型。例如，在下面的例子中，可以将 Int 类型的参数和 Int32 类型的参数直接相加（Int 定义为 Int32 的别名）。注意，不能通过使用别名达到函数重载的目的：

```
type Int = Int32
let numOne: Int32 = 10
let numTwo: Int = 20
let numThree = numOne + numTwo
```

```
func add(left: Int, right: Int32): Int { left + right }

func add(left: Int32, right: Int32): Int32 { left + right }     // error: invalid
 ↪  redeclaration of 'add : (Int32, Int32)->Int32'
```

7. type 定义的默认可见性为 default。如果需要在其他 package 内使用本 package 中定义的类型别名，需要同时满足：(1) 原类型在本 package 中的可见性修饰符为 public；(2) type 定义使用 public 修饰符。另外，需要注意的是：别名可以与原类型拥有不同的可见范围，但是别名的的可见范围不能大于原类型的可见范围。

```
// a.cj
package A
public class ClassWithLongNameA {

}
class ClassWithLongNameB {

}

public type classA = ClassWithLongNameA          // ok
type classAInter = ClassWithLongNameA   // ok

/* error: classB can not be declared with modifier 'public', as
 ↪  'ClassWithLongNameB' is internal */
public type classB = ClassWithLongNameB

// b.cj
package B
import A.*
let myClassA: A.classA = ClassWithLongNameA()
```

2.4.2 类型别名的使用

类型别名可以用在任何等号右手边它指向的原类型能够使用的位置：

1. 作为类型使用，例如：

```
type A = B
class B {}
var a: A = B() // Use typealias A as type B
```

2. 当类型别名实际指向的类型为 class、struct 时，可以作为构造器名称使用

```
type A = B
class B {}
func foo() { A() }  // Use type alias A as constructor of B
```

3. 当类型别名实际指向的类型为 class、interface、struct 时，可以作为访问内部静态成员变量或函数的类型名

```
type A = B
class B {
 static var b : Int32 = 0;
 static func foo() {}
}
func foo() {
 A.foo() // Use A to access static method in class B
 A.b
}
```

4. 当类型别名实际指向的类型为 enum 时，可以作为 enum 声明的构造器的类型名

```
enum TimeUnit {
 Day | Month | Year
}
type Time = TimeUnit
var a = Time.Day
var b = Time.Month   // Use type alias Time to access constructors in TimeUnit
```

2.5 类型间的关系

类型间的关系有两种：相等和子类型。

2.5.1 类型相等

对于任意两个类型 T1 和 T2，如果它们满足以下任一条件，则称 T1 和 T2 相等（记为 $T1 \equiv T2$）：

- 存在类型别名定义 `type T1 = T2`；
- 在 class 定义的内部和 class 的 extend 内部，T1 是 class 的名字，T2 是 This；
- T1 和 T2 的名字完全相同（自反性）；
- $T2 \equiv T1$（对称性）；
- 存在类型 Tk，满足 $T1 \equiv Tk$ 且 $Tk \equiv T2$（传递性）；

2.5.2 子类型

对于任意两个类型 T1 和 T2，如果它们满足以下任一条件，则称 T1 是 T2 的子类型（记为 $T1 <: T2$）：

- $T1 \equiv T2$；
- T1 是 Nothing 类型；

- T1 和 T2 均是 Tuple 类型，并且 T1 每个位置处的类型都是 T2 对应位置处类型的子类型；
- T1 和 T2 均是 Function 类型，并且 T2 的参数类型是 T1 参数类型的子类型，T1 的返回类型是 T2 返回类型的子类型；
- T1 是任意 class 类型，T2 是 Object 类型；
- T1 和 T2 均是 interface 类型，并且 T1 继承了 T2；
- T1 和 T2 均是 class 类型，并且 T1 继承了 T2；
- T2 是 interface 类型，并且 T1 实现了 T2；
- 存在类型 Tk，满足 $T1 <: Tk$ 且 $Tk <: T2$（传递性）；

2.5.3 最小公共父类型

在有子类型的类型系统里，有时会遇到需要求两个类型的最小公共父类型的情形，例如 if 表达式的类型便是其两个分支的类型的最小公共父类型，match 表达式类似。

两个类型的最小公共父类型，是其公共父类型中最小的一个。最小意味着它是其他所有公共父类型的子类型

最小公共父类型定义如下：对于任意两个类型 T1 和 T2，如果类型 LUB 满足如下规则，则 LUB 是 T1 和 T2 的最小公共父类型：

- 对于同时满足 T1 <: T 和 T2 <: T 的任意类型 T，LUB <: T 也成立。

注意，如果公共父类型中的某个类型不比其他类型大，它只是极小的，并不一定是最小的。

2.5.4 最大公共子类型

因为子类型关系中存在逆变（定义参考泛型章节下的类型型变）的情形，如函数类型的参数类型是逆变的，此时会需要求两个类型的最大公共子类型。

两个类型的最大公共子类型，是其公共子类型中最大的一个。最大意味着它是其他所有公共子类型的父类型

最大公共子类型定义如下：对于任意两个类型 T1 和 T2，如果类型 GLB 满足如下规则，则 GLB 是 T1 和 T2 的最大公共子类型：

- 对于同时满足 T <: T1 和 T <: T2 的任意类型 T，T <: GLB 也成立。

注意，如果公共子类型中的某个类型不比其他类型小，它只是极大的，并不一定是最大的。

2.6 类型安全

在没有数据竞争的情况下，编译器保证内存安全和类型安全。

下面是一个类型安全和内存安全都得不到保证的例子：

```
class C {
    var x = 1
    var y = 2
    var z = 3
```

```
}

enum E {
    A(Int64) | B(C)
}

var e = A(1)

main() {
    spawn {
        while (true) {
            e = B(C())      // writing to `e`
            e = A(0)
        }
    }
    while (true) {
        match (e) {         // reading from `e`
            case A(n) =>
                println(n+1)
            case B(c) =>
                c.x = 2
                c.x = 3
                c.x = 4
        }
    }
}
```

不保证是因为对变量 e 赋值的线程和读取该变量的线程之间存在数据竞争。有关数据竞争的更多信息，请参见第 15 章 “并发”。

第三章　名字、作用域、变量、修饰符

本章首先介绍名字、作用域、遮盖，然后介绍其中一种名字——变量，包括变量的定义和初始化，最后是修饰符。

3.1　名字

仓颉编程语言中，我们用名字（names）标识变量、函数、类型、package、module 等实体（entities）。

名字必须是一个合法的标识符。

仓颉编程语言的关键字、变量、函数、类型（包括：class、interface、struct、enum、type alias）、泛型参数、package 名、module 名共用同一个命名空间，即，在同一个 scope 声明或定义的实体，不允许同名（除了构成重载的名字）。不同 scope 声明或定义的实体允许同名，但可能产生 shadow。

```
let f2 = 77

func f2() { // Error,  function name is the same as the variable f2
    print("${f2}")
}

// Variable, function, type names cannot be the same as keywords
let Int64 = 77   // Error: 'Int64' is a keyword

func class() {  // Error: class is a keyword
    print("${f2}")
}

main(): Int64 {
    print("${f2}")   // Print 77

    let f2 = { => // Shadowed the global variable f2
       print("10")
    }
    f2()
    return 1
}
```

3.2 作用域

对一个实体，在其名字所在的作用域（scope）中可以直接使用名字访问，不需要通过前缀限定词访问。作用域是可嵌套的，一个作用域包含自身以及自身包含的嵌套的作用域。名字在其嵌套作用域中如果没有被遮盖或覆盖，也可以直接使用名字访问。

3.2.1 块

在仓颉语言中，由一对匹配的大括号及其中可选的表达式声明序列组成的结构被称之为块（block）。块在仓颉语言中无处不在，例如函数定义的函数体、if 表达式的两个分支、while 表达式的循环体，都是块。块会引出新的作用域。

块的语法定义为：

```
block
    : '{' expressionOrDeclarations '}'
    ;
```

块拥有值。块的值由其中的表达式与声明序列确定。对块进行求值时，会按表达式和变量声明在块中的顺序进行。

若块的最后一项是表达式，当该表达式求值完毕后，该表达式的值即为该块的值：

```
{
    let a = 1
    let b = 2
    a + b
} // The value of the block is a + b
```

若块的最后一项是声明，当该声明处理完毕后，该块的值为 ()：

```
{
    let a = 1
} // The value of the block is ()
```

若块中不含有任何表达式或声明，该块的值为 ()：

```
{ } // The value of this empty block is ()
```

3.2.2 作用域级别

如果嵌套的多层作用域中存在相同的名字，内层作用域引入的名字会遮盖外层作用域的名字，我们称内层作用域级别比外层作用域级别更高。

在嵌套的作用域中，作用域级别比较定义如下：

1. 通过 `import` 引入的名字，作用域级别最低；
2. 包内 top-level 的名字，其作用域级别比 1 中的名字高；

3. 类型内部、函数定义或表达式中引入的名字，其定义通常被包围在一对花括号 {}（即块）中，其作用域级别相较于花括号 {} 外层的名字要高；

4. 对于类和接口，子类中的名字比父类中的名字的作用域级别更高，子类中的名字可能会 shadow 或 override 父类中的名字。

```
import p1.a   // a has the lowest scope level

var x = 1   // x 's scope level is higher than p1.a

open class A {    // A's scope level the same as x at line 3
    var x = 3     // This x has higher scope level than the x at line 3
    var y = 'a'
    func f(a: Int32, b: Float64): Unit {

    }
}

class B <: A {
    var x = 5   // This x has higher scope level than the x at line 6

    func f(x!: Int32 = 7) { // This x's scope level is higher than the x at line 14

    }
}
```

3.2.3 作用域原则

根据名字引入的位置分为三种：top-level，local-level，类型内部，这三种类型的 scope 原则各有不同，以下分别进行介绍。

3.2.3.1 Top-level

Top-level 引入的名字遵守如下作用域原则：

- Top-level 函数、类型，其作用域为整个 package，名字对整个 package 可见，其中类型包括：class、interface、enum 、struct、type 引入的名字。
- Top-level 变量，即由 let，var 和 const 引入的名字，其作用域为从定义（包括赋初值）完成之后开始，不包括从本文件开头到变量声明之间的区间，名字对 package 的其它文件可见。但由于变量的初始化过程可能有副作用，必须先声明且初始化后再使用；

```
/* Global variables can be accessed only after defined. */
let x = y         //Error: y is used before being defined
let y = 4
```

```
let a = b          //Error: b is used before being defined
let b = a
let c = c          //Error: unresolved identifier 'c' (in the right of '=')

//Function names are visible in the entire package
func test() { test2() }   // OK

func f() { test() } // OK

func test2(): Int64 {
    var x = 99
    return x
}
```

3.2.3.2 Local-level

在函数定义或表达式内部声明或定义的名字具有 local-level 作用域。定义在块内的变量比块外的有更高的作用域级别。

- 局部变量，其作用域从声明之后开始到 scope 结束，必须先定义和初始化后使用；局部变量的遮盖从引入变量名的声明或定义之后开始；
- 局部函数，其作用域从声明的位置之后到 scope 结束，支持递归定义，不支持互递归定义；
- 函数的参数和泛型参数的作用域从参数名声明后开始到函数体结束，其作用域级别与函数体内定义的变量等同。
  - 函数定义 `func f(x : Int32, y! : Int32 = x) { }` 是合法的；
  - 函数定义 `func f(x! : Int32 = x) { }` 是不合法的。
- 泛型类型声明或者扩展泛型类型时引入的泛型参数从参数名声明后开始到类型体或扩展体结束，其作用域级别与类型内定义的名字等同。
  - 泛型类型定义 `class C<T> {}` 中 `T` 的作用域从 `T` 出现到整个 `class C` 的声明结束；
  - 泛型类型的扩展 `extend C<T> {}` 中 `T` 的作用域从 `T` 出现到整个扩展定义结束。
- lambda 表达式的参数名的作用域与函数的相同，为 lambda 表达式的函数体部分，其作用域级别可视为与 lambda 表达式的函数体内定义的变量等同。
- main、构造函数的形参名字被视为由函数体块引入，在函数体块中再次引入与形参同名的名字会触发重定义错。
- `if-let` 表达式条件中引入的名字被视为由 `if` 块引入，在 `if` 块中再次引入相同名字会触发重定义错。
- match 表达式的 match case 中 pattern 引入的名字，其作用域级别比所在的 match 表达式更高，从引入处开始到该 match case 结束。每个 match case 均有独立的作用域。match case 绑定模式中引入的名字被视作由胖箭头 => 之后的作用域引入，在 => 之后再次引入相同名字会触发重定义错。
- 对于所有三种循环，循环条件与循环块的作用域级别相同，即其中引入的名字互相不能遮盖。且额外规定循环条件无法引用循环体中定义的变量。则有如下推论：

  - 对于 for-in 表达式，其循环体可以引用循环条件中引入的变量名。
  - 对于 while 和 do-while 表达式，它们的循环条件都无法引用其循环体中引入的变量名，即便 do-while 的循环条件在循环体后。

- 对于 for-in 循环，额外规定其循环条件处引入的变量不能在 in 关键字之后的表达式中使用。
- 对于 try 异常处理，try 后面紧跟的块以及每个 catch 块的的作用域互相独立。catch pattern 引入的名字被视作由 catch 后紧跟的块引入，在 catch 块中再次引入相同名字会触发重定义错。
- try-with-resources 表达式，try 关键字和 {} 之间引入的名字，被视作由 try 块引入，在 try 块中再次引入相同名字会触发重定义错。

```
// a: The scope of a local variable begins after the declaration
let x = 4
func f(): Unit {
    print("${x}") // Print 4

    let x = 99
    print("${x}") // Print 99
}

let y = 5
func g(): Unit {
    let y = y     // 'y' in the right of '=' is the global variable 'y'
    print("${y}") // Print 5

    let z = z     // Error: unresolved identifier 'z' (in the right of '=')
}

// b: The scope of a local function begins after definition
func test1(): Unit {
    func test2(): Unit {
        print("test2")
        test3()  // Error, test3's scope is begin after definition
    }
    func test3(): Unit {
        test2()
    }

    test2()
}

let score: Int64 = 90
let good = 70
var scoreResult: String = match (score) { // binding pattern
    case 60 => "pass" // constant pattern.
```

```
        case 100 => "Full" // constant pattern.
        case good => "good" // This good has higher scope level than the good at line 2
    }
```

3.2.3.3 类型内部引入

- class/interface/struct/enum 的成员的 scope 为整个 class/interface/struct/enum 定义内部;
- enum 定义中的 constructor 的作用域为整个 enum 定义内部，关于 enum 中 constructor 名字访问的具体规则，见enum 类型。

3.2.4 遮盖

通常来说，若两个相互重叠的拥有不同级别的作用域中引入了相同的名字，则会发生遮盖（shadowing）：其中作用域级别高的名字会遮盖作用域级别低的名字。这导致作用域级别低的名字要么需要加上前缀限定词访问，要么无法访问。当作用域级别高的名字的作用域结束时，遮盖消除。具体说来，作用域级别不同时：

- 若作用域级别高的名字 C 为一个类型，则直接发生遮盖。

```
// == in package a ==
public class C {} // ver 1

// == in package b ==
import a.*

class C {} // ver 2

let v = C() // will use ver 2
```

- 若作用域级别高的名字 x 为一个变量，则直接发生遮盖。对于成员变量的遮盖规则，请参见“类和接口”章节。

```
let x = 1

func foo() {
    let x = 2
    println(x) // will print 2
}
```

- 若作用域级别高的名字 p 为 package 名字，则直接发生遮盖。

```
// == in package a ==
public class b {
    public static let c = 1
}
```

```
// == in package a.b ==
public let c = 2

// == in package test ==
import a.*
import a.b.*

let v = a.b.c // will be 1
```

- 若作用域级别高的名字 f 为一个成员函数，则按重载的规则判断 f 是否发生重载，如无重载，则可能发生覆盖或重定义；如无重载且不能覆盖/重定义，则报错。具体规则，见覆盖、重定义。

```
open class A {
    public open func foo() {
        println(1)
    }
}

class B <: A {
    public override func foo() {  // override
        println(2)
    }

    public func foo() {  // error, conflicting definitions
        println(3)
    }
}
```

- 若作用域级别高的名字 f 为非成员函数，则按重载的规则判断 f 是否发生重载。如无重载，则视为遮盖。

```
func foo() { 1 }

func test() {
    func foo() { 2 } // shadows
    func foo(x: Int64) { 3 } // overloads

    println(foo()) // will print 2
}
```

下面的例子展示了函数内变量的作用域级别和之间的遮盖关系，以及项与类型在同一命名空间。

```
func g(): Unit {
    f(1, 2)  // OK. f is a top-level definition, which is visible in the whole file
```

```
}

func f(x: Int32, y: Int32): Unit {
    // Error. Term Maze shadows the type Maze, the Maze cannot be used as a type
    // let Maze : Maze
    // var x = 1    // Error, x was introduced by parameter
    var i = 1       // OK

    for (i in 1..3) { // OK, a new i in the block
        let v = 1
    }              // i and v disappear

    print("${i}")     // OK. i is defined at line 9
    // print("${v}")  // Error, v disappeared and cannot be found
}

enum Maze {
    C1 | C2
    // | C1 // Error. The C1 has been used
}
```

下面的例子展示了不同包之间定义的变量和类的作用域级别关系和之间遮盖关系。

```
// File a.cj
package p1

class A {}
var v: Int32 = 10

// File b.cj
package p2
import p1.A        // A belongs to the package p1
import p1.v        // v belongs to the package p1

// p2 has defined its own v, whose scope level is higher than the v from package p1
var v: Int32 = 20

func displayV1(): Unit {
    // According to the scope level, p2.v shadows p1.v, therefore access p2.v here
    print("${v}")          // output: 20
}

var a = A()         // Invoke A in p1
```

下面的例子展示了继承中的遮盖关系。

```
var x = 1

open class A {
    var x = 3   // this x shadows the top level x
}

class B <: A {
    var x = 5   // error: a member of the subtype must not shadow a member of the
     ↪  supertype.

    func f(x!: Int32 = 7): Unit { // this x shadows all the previous x

    }
}
```

3.3 变量

仓颉编程语言作为一种静态类型（statically typed）语言，要求每个变量的类型必须在编译时确定。

根据是否可进行修改，可将变量分为 3 类：不可变变量（一旦初始化，值不可改变）、可变变量（值可以改变）、const 变量（必须编译期初始化，不可修改）。

3.3.1 变量的定义

变量定义的语法定义为：

```
variableDeclaration
    : variableModifier* ('let' | 'var' | 'const') patternsMaybeIrrefutable (((':'
     ↪  type)? ('=' expression)) | (':' type))
    ;

patternsMaybeIrrefutable
    : wildcardPattern
    | varBindingPattern
    | tuplePattern
    | enumPattern
    ;
```

变量的定义均包括四个部分：修饰符、let/var/const 关键字、patternsMaybeIrrefutable 和变量类型。其中：

1. 修饰符

- top-level 变量的修饰符包括：public, protected, private, internal
- 局部变量不能用修饰符修饰
- class 类型的成员变量的修饰符包括：public, protected, private, internal, static
- struct 类型的成员变量的修饰符包括：public, private, internal, static

2. 关键字 let/var/const

- let 用于定义不可变变量，let 变量一旦初始化就不能再改变。
- var 用于定义可变变量。
- const 用于定义 const 变量。

3. patternsMaybeIrrefutable

- let（或 var/const）之后只能是那些一定或可能为 irrefutable 的 pattern（见模式的分类）。在语义检查阶段，会检查 pattern 是否真的是 irrefutable，如果不是 irrefutable pattern，则编译报错。
- let（或 var/const）之后的 pattern 中引入的新的变量，全部都是 let 修饰（或 var 修饰）的变量。
- 在 class 和 struct 中定义成员变量时，只能使用 binding pattern（见绑定模式）。

4. 变量类型是可选的，不声明变量类型时需要给变量初始值，编译器将尝试根据初始值推断变量类型；

5. 变量可以定义在 top-level, 表达式内部，class/struct 类型内部。

需要注意的是：
（1）pattern 和变量类型之间需要使用冒号（:）分隔，pattern 的类型需要和冒号后的类型相匹配。
（2）关键字 let/var/const 和 pattern 是必选的；
（3）局部变量除了使用上述语法定义之外，还有如下几种情形会引入局部变量：

- for-in 循环表达式中，for 和 in 中间的 pattern，详见for-in 表达式
- 函数、lambda 定义中的形参，详见参数
- try-with-resource 表达式中 ResourceSpecifications，详见异常
- match 表达式中，case 后的 pattern，详见模式匹配表达式

（4）可以使用一对反引号（`）将关键字变为合法的标识符（例如，`open`，`throw`等)。
下面给出变量定义的一些实例：

```
let b: Int32                // Define read-only variable b with type Int32.
let c: Int64                // Define read-only variable c with type Int64.
var bb: String              // Define writeable variable bb with type String.
var (x, y): (Int8, Int16)   // Define two writeable variable: x with type Int8, x
 ↪  with type Int16.
var `open` = 1              // Define a variable named `open` with value 1.
var `throw` = "throw"       // Define a variable named `throw` with value "throw".
const d: Int64 = 0      // Define a const variable named d with value 0.
```

3.3.2 变量的初始化

3.3.2.1 不可变变量和可变变量

不可变变量和可变变量的初始化均有两种方式：定义时初始化和先定义后初始化。需要注意的是，每个变量在使用前必须进行初始化，否则会报编译错误。关于变量的初始化，举例如下：

```
func f() {
    let a = 1               // Define and initialize immutable variable a.
    let b: Int32            // Define immutable variable b without initialization.
    b = 10                  // Initialize variable b.
    var aa: Float32 = 3.14  // Define and initialize mutable variable aa.
}
```

使用 let 定义的不可变变量，只能被赋值一次（即初始化），如果被多次赋值，会报编译错误。使用 var 定义的可变变量，支持多次赋值。

```
func f() {
    let a = 1             // Define and initialize immutable variable a.
    a = 2                 // error: immutable variable a cannot be reassigned.
    var b: Float32 = 3.14 // Define and initialize mutable b.
    b = 3.1415            // ok: mutable variable b can be reassigned.
}

class C {
    let m1: Int64
    init(a: Int64, b: Int64) {
        m1 = a
        if (b > 0) {
            m1 = a * b  // OK: immutable variable can be reassigned in constructor.
        }
    }
}
```

3.3.2.2 全局变量及静态变量初始化

全局变量指定义在 top-level 的变量，静态变量包含定义在 class 或 struct 中的静态变量。
全局变量和静态变量的初始化必须满足以下规则：

- 全局变量在声明时必须立即对其进行初始化，否则报错。即，声明必须提供一个初始化表达式。
- 静态变量在声明时必须立即对其进行初始化，可以采用与全局变量的初始化相同的形式，也可以在静态初始化器中进行初始化（更多细节见静态初始化器部分）。
  - 注意，静态变量不能在其他静态变量中初始化：

```
class Foo {
    static let x: Int64
    static let y = (x = 1) // it's forbidden
}
```

- 初始化表达式 e 不能依赖未初始化的全局变量或静态变量。编译器会进行保守的分析，如果 e 可能会访问到未初始化的全局变量或静态变量，则报错。详细的分析取决于编译器的实现，在规范中没有指定。

全局/静态变量的初始化时机和初始化顺序规则如下:

- 所有的全局/静态变量都在 main (程序入口) 之前完成初始化;
- 对于同一个文件中声明的全局/静态变量，初始化顺序根据变量的声明顺序从上到下进行；如果使用了静态初始化器，则根据静态初始化器的初始化顺序规则执行（详见静态初始化器部分）
- 同一个包里不同文件或不同包中声明的全局/静态变量的初始化顺序取决于文件或包之间的依赖关系。如果文件 B.cj 依赖文件 A.cj 且 A.cj 不依赖 B.cj，则 A.cj 中的全局/静态变量的初始化在 B.cj 中的全局/静态变量的初始化之前;
- 如果文件或包之间存在循环依赖或者不存在任何依赖，那么它们之间的初始化顺序不确定，由编译器实现决定。

```
/* The initialization of the global variable cannot depend on the global
   variables defined in other files of the same package. */
// a.cj
let x = 2
let y = z       // OK, b.cj does not depend on this file directly or indirectly.
let a = x       // OK.
let c = A()

/* c.f is an open function, the compiler cannot statically determine whether the
   function meets the initialization rules of global variables, and an error may
   be reported. */
let d = c.f()

open class A {
    // static var x = A.z     // Error, A.z is used before its initialization.
    // static var y = B.f     // Error, B.f is used before its initialization.
    static var z = 1
    public open func f(): Int64 {
        return 77
    }
}
```

```
class B {
    static var e = A.z    // OK.
    static var f = x      // OK.
}

// b.cj
let z = 10
// let y = 10   // Error, y is already defined in a.cj.

// main.cj
main(): Int64 {
    print("${x}")
    print("${y}")
    print("${z}")
    return 1
}
```

3.3.2.3 const 变量

详见 const 章节。

3.4 修饰符

仓颉提供了很多修饰符，主要分以下两类:

- 访问修饰符
- 非访问修饰符

修饰符通常放在定义处的最前端，用来表示该定义具备某些特性。

3.4.1 访问修饰符

详细内容请参考包和模块管理章节访问修饰符。

3.4.2 非访问修饰符

仓颉提供了许多非访问修饰符以支持其它丰富的功能

- **open** 表示该实例成员可被子类覆盖，或者该类能被子类继承，详见类
- **sealed** 表示该 class 或 interface 只能在当前包内被继承或实现，详见类
- **override** 表示覆盖父类的成员，详见类
- **redef** 表示重新定义父类的静态成员，详见类
- **static** 表示该成员是静态成员，静态成员不能通过实例对象访问，详见类
- **abstract** 表示该 class 是抽象类，详见类

- **foreign** 表示该成员是外部成员，详见语言互操作
- **unsafe** 表示与 C 语言进行互操作的上下文，详见语言互操作
- **sealed** 表示该 class 或 interface 只能在当前包内被继承或实现，详见类
- **mut** 表示该成员是具有可变语义的，详见函数

这些修饰符的具体功能详见对应的章节。

第四章　表达式

表达式通常由一个或多个操作数（operand）构成，多个操作数之间由操作符（operator）连接，每个表达式都有一个类型，计算表达式值的过程称为对表达式的求值（evaluation）。

在仓颉编程语言中，表达式几乎无处不在，有表示各种计算的表达式（如算术表达式、逻辑表达式等），也有表示分支和循环的表达式（如 if 表达式、循环表达式等）。对于包含多个操作符的表达式，必须明确每个操作符的优先级、结合性以及操作数的求值顺序。优先级和结合性规定了操作数与操作符的结合方式，操作数的求值顺序规定了二元和三元操作符的操作数求值顺序，它们都会对表达式的值产生影响。

下面将依次介绍仓颉中的表达式。

注：本章中对于各操作符的操作数类型的规定，均建立在操作符没有被重载的前提下。

4.1　字面量

字面量是一种拥有固定语法的表达式。对于内部不包含其他表达式的字面量（参见 1.3 字面量），它的值就是字面量自身的值，它的类型可由其语法或所在的上下文决定。当无法确定字面量类型时，整数字面量具有 `Int64` 类型，浮点数字面量具有 `Float64` 类型。对于可在内部包含其他表达式的集合类字面量和元组字面量（参见 [值类型]），它的值等于对其内部所有表达式求值后得到的字面量的值，它的类型由其语法确定。

字面量举例：

```
main(): Int64 {
    10u8                        // UInt8 literal
    10i16                       // Int16 literal
    1024u32                     // UInt32 literal
    1024                        // Int64 literal
    1024.512_f32                // Float32 literal
    1024.512                    // Float64 literal
    'a'                         // Rune literal
    true                        // Bool literal
    "Cangjie"                   // String literal
    ()                          // Unit literal
    [1, 2, 3]                   // Array<Int64> literal
    (1, 2, 3)                   // (Int64, Int64, Int64) literal
    return 0
}
```

4.2 变量名和函数名

变量名和函数名（这里的变量名和函数名也包括通过包名指向的变量或函数）本身也是表达式。对于变量名，它的值等于变量求值后的值，它的类型为变量的类型；对于函数名，它的值是一个闭包（见 5.7 节），它的类型为对应的函数类型。

变量名和函数名举例：

```
let intNum: Int64 = 100 // 'intNum' is the name of a variable, whose value and
 ↪  type are '100' and 'Int64', respectively.

/* 'add' is the name of a function, whose value and type are '(p1: Int64, p2:
 ↪  Int64) => {p1 + p2}' and '(Int64, Int64) -> Int64', respectively. */
func add(p1: Int64, p2: Int64) {
    p1 + p2
}

let value = p1.x // x is a variable defined in package p1.
```

对于变量名，规定 var 声明的变量始终是可变的，let 声明的变量只可以被赋一次值（声明时或声明之后），赋值前是可变的，赋值后是不可变的。

4.2.1 泛型函数名作为表达式

仓颉语言中支持函数（见第 5 章）做为第一成员，同时也支持声明类型参数的泛型函数（见第 9 章）。当函数为泛型函数时，函数名作为表达式使用时必须给出泛型函数的类型实参。例如：

```
func identity<T>(a: T) { // identity is a generic function
    return a
}

var b = identity // error: generic function 'identity' needs type arguments

var c = identity<Int32> // ok: Int32 is given as a type argument
```

identity 是一个泛型函数，所以 identity 不是合法的表达式，只有给出了类型实参的 identity<Int32> 才是一个合法的表达式。

若一个函数在当前作用域中被重载了，当重载函数中存在多个类型完备的可选函数，那么直接使用该类型名作为表达式是有歧义错误的，例如：

```
interface Add<T> {
    operator func +(a: T): T
}

func add<T>(i: T, j: Int64): Int64 where T <: Add<Int64> { // f1
```

```
    return i + j;
}

func add<T>(i: T, j: T): T where T <: Add<T> { // f2
    return i + j;
}

main(): Int64 {
    var plus = add<Int64> // error: ambiguous use of 'add'
    return 0
}
```

4.3 条件表达式

条件表达式即 if 表达式，可以根据判定条件是否成立来决定执行哪条代码分支，实现分支控制逻辑。
if 表达式的语法定义为：

```
ifExpression
    : 'if' '(' ('let' deconstructPattern '<-')? expression ')' block ('else' (
    ↪ ifExpression | block))?
    ;
```

其中 if 是关键字，if 之后是一个包围在小括号内的表达式，接着是一个块，块之后是可选的 else 分支。else 分支以 else 关键字开始，后接新的 if 表达式或一个块。
if 表达式举例：

```
main(): Int64 {
    let x = 100
    // if expression without else branch
    if (x > 0) {
        print("x is larger than 0")
    }

    // if expression with else branch
    if (x > 0) {
        print("x is larger than 0")
    } else {
        print("x is not larger than 0")
    }

    // if expression with nested if expression
    if (x > 0) {
        print("x is larger than 0")
```

```
        } else if (x < 0) {
            print("x is lesser than 0")
        } else {
            print("x equals to 0")
        }

        return 0
    }
```

if 表达式首先对 if 之后的表达式进行求值（要求表达式的类型为 Bool），如果表达式的值等于 true，则执行它之后的块，否则，执行 else 之后的 if 表达式或块（如果存在）。if 表达式的值等于被执行到的分支中的表达式的值。

对于包含 let 的 if 表达式，我们称之为 if-let 表达式。我们可以用 if-let 表达式来做一些简单的解构操作。

一个基础的 if-let 表达式举例：

```
main(): Int64 {
    let x: Option<Int64> = Option<Int64>.Some(100)
    // if-let expression without else branch
    if (let Some(v) <- x) {
        print("x has value")
    }
    // if-let expression with else branch
    if (let Some(v) <- x) {
        print("x has value")
    } else {
        print("x has not value")
    }
    return 0
}
```

if-let 表达式首先对 <- 之后的表达式进行求值（表达式的类型为可以是任意类型），如果表达式的值能匹配 let 之后的 pattern，则执行它之后的块，否则，执行 else 之后的 if 表达式或块（如果存在）。if-let 表达式的值等于被执行到的分支中的表达式的值。

let 之后的 pattern 支持常量模式、通配符模式、绑定模式、Tuple 模式、enum 模式。

if 表达式的类型

对于没有 else 分支的 if 表达式，它的类型为 Unit，它的值等于 ()。因为 if expr1 {expr2} 是 if expr1 {expr2; ()} else {()} 的语法糖。

对于包含 else 分支的 if 表达式，

- 如果 if 表达式的值没有被读取或者返回，那么 if 表达式的类型为 Unit，两个分支不要求存在公共父类型；否则，按如下规则检查；
- 在上下文没有明确的类型要求时，如果 if 的两个分支类型，设它们为 T1 和 T2，则 if 表达式的类型是 T1 和 T2 的最小公共父类型 T。如果不存在最小公共父类型 T ，则编译报错；

- 在上下文有明确的类型要求时，此类型即为 `if` 表达式的类型。此时要求 `if` 的两个分支的类型都是上下文所要求的类型的子类型。

  举例如下：

```
struct S1 { }
struct S2 { }

interface I1 {}
class C1 <: I1 {}
class C2 <: I1 {}

interface I2{}
class D1 <: I1 & I2 {}
class D2 <: I1 & I2 {}

func test12() {
    if (true) {  // OK. The type of this if expression is Unit.
        S1()
    } else {
        S2()
    }

   if (true) {  // OK. The type of this if expression is Unit.
        C1()
    } else {
        C2()
    }

    return if (true) {  // Error. The `if` expression is returned. There is no
     ↪ least common supertype of `D1` and `D2`.
        D1()
    } else {
        D2()
    }
}
```

注意：为了保持代码格式的规整以及提高代码的可维护性，同时为了避免悬垂 else（dangling-else）问题，仓颉编程语言要求每个 `if` 分支和 `else` 分支中的执行部分（即使只有一条表达式）必须使用一对花括号括起来成为一个块。（悬垂 else 问题是指无法确定形如 `if cond1 if cond2 expr1 else expr2` 的代码中的 `else expr2` 是归属于内层 `if` 还是外层 `if` 的问题。如果其归属于内层 `if`，则代码应解读为 `if cond1 (if cond2 expr1 else expr2)`；如果其归属于外层 `if`，则代码应解读为 `if cond1 (if cond2 expr1) expr2`。但如果强制分支使用大括号，则无此问题。）

4.4 模式匹配表达式

仓颉编程语言中支持使用模式匹配表达式（match 表达式）实现模式匹配（pattern matching），允许开发者使用更精简的代码描述复杂的分支控制逻辑。直观上看，模式描述的是一种结构，这个结构定义了一个与之匹配的实例集合，模式匹配就是去判断给定的实例是否属于模式定义的实例集合。显然，匹配的结果只有两种：匹配成功和匹配失败。match 表达式可分为两类：带 selector 的 match 表达式和不带 selector 的 match 表达式。

match 表达式的语法定义为：

```
matchExpression
    : 'match' '(' expression ')' '{' matchCase+ '}'
    | 'match' '{' ('case' (expression | '_') '=>' expressionOrDeclaration+)+ '}'
    ;
matchCase
    :  'case' pattern ('|' pattern)* patternGuard? '=>' expressionOrDeclaration+
    ;
patternGuard
    : 'where' expression
    ;
```

对于带 selector 的 match 表达式，关键字 match 之后的 expression 即为待匹配的 selector。selector 之后的 {} 内可定义若干 matchCase，每个 matchCase 以关键字 case 开头，后跟一个 pattern 或者多个由 | 分隔的相同种类的 pattern（关于不同 pattern 的详细定义，见模式节），一个可选的 pattern guard，一个胖箭头 => 和一系列（至少一个）声明或表达式（多个声明或表达式之间使用分号或换行符分隔）。

在执行 match 表达式的过程中，匹配顺序即 case 定义的顺序，selector 按照匹配顺序依次和 case 中定义的 pattern 进行匹配，一旦 selector 和当前 pattern 匹配成功（且满足 pattern guard），则执行 => 之后的代码，且无需再与它之后的 pattern 进行匹配；否则（与当前 pattern 不匹配），继续与下一个 pattern 进行匹配判断，依次类推。下面的例子展示了 match 表达式中使用常量模式进行分支控制：

```
let score: Int64 = 90
var scoreResult: String = match (score) {
    case 0 => "zero"
    case 10 | 20 | 30 | 40 | 50 => "fail"
    case 60 => "pass"
    case 70 | 80 => "good"
    case 90 | 100 => "excellent" // matched
    case _ => "not a valid score"
}
```

出于安全和完备的考虑，仓颉编程语言要求 case 表达式中定义的所有 pattern 和其对应的 patternGuard（如果存在）组合起来要覆盖 selector 的所有可能取值（即 exhaustive），如果编译器判断出未实现完全覆盖，则会报错。为实现完全覆盖，通常可以在最后一个 case 中使用通配符 _ 来处理其他 case 未覆盖到的情况。另外，不要求每个 pattern 定义的空间是互斥的，即不同 pattern 之间覆盖的空间可以有重叠。

对于不带 selector 的 match 表达式，关键字 match 之后没有 expression，并且 {} 中的每条 case 中，关键字 case 和 => 之间只能为一个 Bool 类型的表达式（或者通配符 _，表示永远为 true）。

在执行的过程中，依次判断 case 之后的表达式的值，一旦表达式的值等于 true，则执行 => 之后的代码，且无需再判断它之后的所有 case。事实上，不带 selector 的 match 表达式其实是一连串嵌套 if-else if 表达式的简洁表达。

同样地，要求不带 selector 的 match 表达式满足 exhaustive（即任何情况下至少有一个 case 是满足的）。编译器会尽量做 exhaustive 检查，如果无法判断，则报错并提示添加 case _。

```
let score = 80
var result: String = match {
    case score < 60 => "fail"
    case score < 70 => "pass"
    case score < 90 => "good" // matched
    case _ => "excellent"
}
```

类似于 if 表达式，对于 match 表达式，无论其是否有 selector，它的类型遵循如下规则：

- 如果 match 表达式的值没有被读取或者返回，那么 match 表达式的类型为 Unit，所有分支不要求存在公共父类型；否则，按如下规则检查；
- 在上下文没有明确的类型要求时，假设 match 的所有分支类型分别为 T1，...，Tn，则 match 表达式的类型是 T1，...，Tn 的最小公共父类型 T，如果不存在最小公共父类型 T 则报错。
- 在上下文有明确的类型要求时，此类型即为 match 表达式的类型。此时要求每条 case 中 => 之后的表达式的类型都是上下文所要求的类型的子类型。

4.4.1 模式

仓颉编程语言提供了丰富的模式种类，包括：

1. 常量模式（constant patterns）；
2. 通配符模式（wildcard patterns）；
3. 绑定模式（binding patterns）；
4. tuple 模式（tuple patterns）；
5. 类型模式（type patterns）；
6. enum 模式（enum patterns）；

pattern 的语法定义为：

```
pattern
    : constantPattern
    | wildcardPattern
```

```
    | varBindingPattern
    | tuplePattern
    | typePattern
    | enumPattern
    ;
```

4.4.1.1 常量模式

常量模式可以是整数字面量、字节字面量、浮点数字面量、Rune 字面量、布尔字面量、字符串字面量（不支持字符串插值）、Unit 字面量。常量模式中字面量的类型需要和 selector 的类型一致，selector 和一个常量模式匹配成功的条件是 selector 与常量模式中的字面量相等（这里指值相等）。

常量模式的语法定义为：

```
constantPattern
    : literalConstant
    ;
```

使用常量模式的例子如下：

```
func f() {
    let score: Int64 = 90
    var scoreResult: String = match (score) {
        case 0 => "zero"
        case 10 | 20 | 30 | 40 | 50 => "fail"
        case 70 | 80 => "good"
        case 90 => "excellent" // matched
        case _ => "not a valid score"
    }
}
```

需要注意的是，浮点数字面量匹配在常量模式中遵循浮点数的判等规则，会和判等存在一样的精度问题。

4.4.1.2 通配符模式

通配符模式使用下划线 _ 表示，它可以匹配任意值，常用于部分匹配（例如作为占位符）或作为 match 表达式的最后一个 pattern 来匹配其它 case 未覆盖到的情况。

通配符模式的语法定义为：

```
wildcardPattern
    : '_'
    ;
```

使用通配符模式的例子如下：

```
let score: Int64 = 90
var scoreResult: String = match (score) {
    case 60 => "pass"
    case 70 | 80 => "good"
    case 90 | 100 => "excellent" // matched
    case _ => "fail"             // wildcard pattern: used for default case
}
```

4.4.1.3 绑定模式

绑定模式同样可以匹配任意值，但与通配符模式不同的是，绑定模式会将匹配到的值绑定到 binding pattern 中定义的变量，以便在 => 之后的表达式中进行访问。

绑定模式的语法定义为：

```
varBindingPattern
    : identifier
    ;
```

绑定模式中定义的变量是不可变的。

使用绑定模式的例子如下：

```
let score: Int64 = 90
var scoreResult: String = match (score) {
    case 60 => "pass"
    case 70 | 80 => "good"
    case 90 | 100 => "excellent" // matched
    case failScore =>            // binding pattern
        let passNeed = 60 - failScore
        "failed with ${failScore}, and ${passNeed} need to pass"
}
```

绑定模式中定义的变量，作用域从第一次出现的位置处开始至下一个 case 之前。需要注意，使用 | 连接多个模式时不能使用绑定模式，也不可嵌套出现在其它模式中，否则会报错。

```
let opt = Some(0)
match (opt) {
    case x | x => {}               // error: variable cannot be introduced in
     ↪ patterns connected by '|'
    case Some(x) | Some(x) => {}   // error: variable cannot be introduced in
     ↪ patterns connected by '|'
    case x: Int64 | x: String => {} // error: variable cannot be introduced in
     ↪ patterns connected by '|'
}
```

4.4.1.4 Tuple 模式

tuple pattern 用于匹配 Tuple 值，tuple pattern 定义为由圆括号括起来的多个 pattern，每个 pattern 之间使用逗号分隔：(pattern_1, pattern_2, ··· pattern_k)。例如，(x, y, z) 是由三个 binding pattern 组成的一个 tuple pattern，(1, 0, 0) 是由三个 constant pattern 组成的一个 tuple pattern。tuple pattern 中的子 pattern 个数需要和 selector 的维度相同，并且如果子 pattern 是 constant pattern 或 enum pattern 时，其类型要和 selector 对应维度的类型相同。

tuple 模式的语法定义为：

```
tuplePattern
    : '(' pattern (',' pattern)+ ')'
    ;
```

使用 tuple 模式的例子如下：

```
let scoreTuple = ("Allen", 90)
var scoreResult: String = match (scoreTuple) {
    case ("Bob", 90) =>  "Bob got 90"
    case ("Allen", score) =>  "Allen got ${score}" // matched
    case ("Allen", 100) | ("Bob", 100) =>  "Allen or Bob got 100"
    case (_, _) =>  ""
}
```

4.4.1.5 类型模式

使用类型模式可以很方便地实现 type check 和 type cast。类型模式的语法定义为：

```
typePattern
  : (wildcardPattern | varBindingPattern) ':' type
  ;
```

对于类型模式 varBindingPattern : type（或 wildcardPattern : type）。首先判断要匹配的值的运行时类型是否是 : 右侧 type 定义的类型或它的子类，若类型匹配成功则将值的类型转换为 type 定义的类型，然后将新类型的值与 : 左侧的 varBindingPattern 进行绑定（对于 wildcardPattern : type，不存在绑定）。只有类型匹配，才算成功匹配，否则匹配失败，因此，varBindingPattern : type（或 wildcardPattern : type）可以同时实现 type test 和 type cast。

使用类型模式匹配的例子如下：

```
open class Point {
    var x: Int32 = 1
    var y: Int32 = 2
    init(x: Int32, y: Int32) {
        this.x = x
        this.y = y
    }
```

```
}
class ColoredPoint <: Point {
    var color: String = "green"
    init(x: Int32, y: Int32, color: String) {
        super(x, y)
        this.color = color
    }
}
let normalPt = Point(5,10)
let colorPt = ColoredPoint(8,24,"red")
var rectangleArea1: Int32 = match (normalPt) {
    case _: Point => normalPt.x * normalPt.y  // matched
    case _ => 0
}
var rectangleArea2: Int32 = match (colorPt) {
    case cpt: Point => cpt.x * cpt.y          // matched
    case _ => 0
}
```

4.4.1.6 enum 模式

enum 模式主要和 enum 类型配合使用。

enum pattern 用于匹配 enum constructor，格式是 constructorName（无参构造器）或 constructorName(pattern_1, pattern_2, ..., pattern_k)（有参构造器), 圆括号内用逗号分隔的若干 pattern（可以是其它任何类型的 pattern，并允许嵌套）依次对每个参数进行匹配。

enum 模式的语法定义为:

```
enumPattern
   : (userType '.')? identifier enumPatternParameters?
   ;
enumPatternParameters
   : '(' pattern (',' pattern)* ')'
   ;
```

使用 enum 模式匹配的例子如下:

```
enum TimeUnit {
    | Year(Float32)
    | Month(Float32, Float32)
    | Day(Float32, Float32, Float32)
    | Hour(Float32, Float32, Float32, Float32)
}
let oneYear = TimeUnit.Year(1.0)
```

```
var howManyHours: Float32 = match (oneYear) {
    case Year(y) => y * Float32(365 * 24) // matched
    case Month(y, m) => y * Float32(365 * 24) + m * Float32(30 * 24)
    case Day(y, m, d) => y * Float32(365 * 24) + m * Float32(30 * 24) + d *
     ↪ Float32(24)
    case Hour(y, m, d, h) => y * Float32(365 * 24) + m * Float32(30 * 24) + d *
     ↪ Float32(24) + h
}

let twoYear = TimeUnit.Year(2.0)
var howManyYears: Float32 = match (twoYear) {
    case Year(y) | Month(y, m) => y // error: variable cannot be introduced in
     ↪ patterns connected by '|'
    case Year(y) | Month(x, _) => y // error: variable cannot be introduced in
     ↪ patterns connected by '|'
    case Year(y) | Month(y, _) => y // error: variable cannot be introduced in
     ↪ patterns connected by '|'
    ...
}
```

如果模式匹配所在的作用域中，某个 pattern 中的 identifier 是 enum 构造器时，该 identifier 总是会被当成 enum pattern 进行匹配，否则才会作为 binding pattern 匹配。

```
enum Foo {
    A | B | C
}

func f() {
    let x = Foo.A
    match (x) {
        case A => 0 // enum pattern
        case B => 1 // enum pattern
        case C => 2 // enum pattern
        case D => 3 // binding pattern
    }
}
```

需要注意的是，对 enum 进行匹配时，要求 enum pattern 的类型和 selector 的类型相同，同时编译器会检查 enum 类型的每个 constructor（包括 constructor 的参数的值）是否被完全覆盖，如果未做到全覆盖，则编译器会报错。

```
enum TimeUnit {
    | Year(Float32)
```

```
        | Month(Float32, Float32)
        | Day(Float32, Float32, Float32)
        | Hour(Float32, Float32, Float32, Float32)
    }

    let oneYear = TimeUnit.Year(1.0)
    var howManyHours: Float32 = match (oneYear) { // error: match must be exhaustive
        case Year(y) => y * Float32(365 * 24)
        case Month(y, m) => y * Float32(365 * 24) + m * Float32(30 * 24)
    }
```

4.4.2 模式的分类

一般地，在类型匹配的前提下，当一个 pattern 有可能和它所要匹配的值不匹配时，称此 pattern 为 refutable pattern；反之，当一个 pattern 总是可以和它所要匹配的值匹配时，称此 pattern 为 irrefutable pattern。对于上述介绍的各类 pattern，规定：

- constant pattern 总是 refutable pattern；
- wildcard pattern 总是 irrefutable pattern；
- binding pattern 总是 irrefutable pattern；
- tuple pattern 是 irrefutable pattern，当且仅当其包含的每个 pattern 都是 irrefutable pattern；
- type pattern 总是 refutable pattern；
- enum pattern 是 irrefutable pattern，当且仅当其对应的 enum 类型中只有一个带参 constructor，且 enum pattern 中包含的其他 pattern（如果存在）都是 irrefutable pattern。

4.4.3 字符串、字节和 Rune 的匹配规则

在模式匹配的目标是静态类型为 Rune 的值时，Rune 字面量和单字符字符串字面量都可用于表示 Rune 类型字面量的常量 pattern。

在模式匹配的目标是静态类型为 Byte 的值时，一个表示 ASCII 字符的字符串字面量可用于表示 Byte 类型字面量的常量 pattern。

4.4.4 Pattern Guards

为了对匹配出来的值做进一步的判断，仓颉支持使用 pattern guard。pattern guard 可以在 match 表达式中使用，也可以在 for-in 表达式中使用。本节主要介绍 pattern guard 在 match 表达式中的使用，关于其在 for in 表达式中的使用请参见for-in 表达式。

match 表达式中，为了提供更加强大和精确的匹配模式，支持 pattern guard，即在 pattern 与 => 之间加上 where boolExpression（boolExpression 是值为布尔类型的表达式）。匹配的过程中，只有当值与 pattern 匹配并且满足 where 之后的 boolExpression 时，整个 case 才算匹配成功，否则匹配失败。

pattern guard 的语法定义为：

```
patternGuard
    : 'where' expression
    ;
```

使用 pattern guards 的例子如下：

```
let oneYear = Year(1.0)
var howManyHours: Float32 = match (oneYear) {
    case Year(y) where y > 0.0f32 => y * Float32(365 * 24) // matched
    case Year(y) where y <= 0.0f32 => 0.0
    case Month(y, m) where y > 0.0f32 && m > 0.0f32 => y * Float32(365 * 24) + m *
     ↪  Float32(30 * 24)
    case Day(y, m, d) where y > 0.0f32 && m > 0.0f32 && d > 0.0f32 => y *
     ↪  Float32(365 * 24) + m * Float32(30 * 24) + d * Float32(24)
    case Hour(y, m, d, h) where y > 0.0f32 && m > 0.0f32 && d > 0.0f32 && h > 0.0f32
     ↪  => y * Float32(365 * 24) + m * Float32(30 * 24) + d * Float32(24) + h
    case _ => 0.0
}
```

4.5 循环表达式

仓颉编程语言支持三种循环表达式：for-in 表达式、while 表达式和 do-while 表达式。
循环表达式的语法定义为：

```
loopExpression
  : forInExpression
  | whileExpression
  | doWhileExpression
  ;
```

4.5.1 for-in 表达式

一个完整的 **for-in** 表达式具有如下形式：

```
for (p in e where c) {
    s
}
```

其中 pattern guard where c 是非必须的，因此更简易的 **for-in** 表达式具有如下形式：

```
for (p in e) {
    s
}
```

for-in 表达式的语法定义为：

```
forInExpression
    : 'for' '(' patternsMaybeIrrefutable 'in' expression patternGuard? ')' block
    ;

patternsMaybeIrrefutable
    : wildcardPattern
    | varBindingPattern
    | tuplePattern
    | enumPattern
    ;

patternGuard
    : 'where' expression
    ;
```

上述语法定义中，关键字 for 之后只能是那些一定或可能为 irrefutable 的 pattern（见模式的分类）。在语义检查阶段，会检查 for 之后的 pattern 是否真的是 irrefutable，如果不是 irrefutable pattern，则编译报错。另外，如果 for 之后的 pattern 中存在 binding pattern，相当于新声明了一个（或多个）let 变量，每个变量的作用域从它第一次出现的位置到循环体结束。

for-in 会先对 expression 求值，再调用其 iterator() 函数，获取一个类型为 Iterator<T> 的值。程序通过调用 Iterator<T> 的 next() 函数开始执行循环，我们可以使用 pattern 匹配迭代的元素，如果匹配成功（如果存在 patternGuard，也必须同时满足 patternGuard 的条件），则执行循环体 block，然后在开始处重新调用 next() 继续循环，当 next() 返回 None 时循环终止。

Iterable<T> 和 Iterator<T> 可在标准库中查阅。

```
main(): Int64 {
    let intArray: Array<Int32> = [0, 1, 2, 3, 4]
    for (item in intArray) {
        print(item)          // output: 01234
    }

    let intRange = 0..5
    for (number in intRange where number > 2) {
        print(number)        // output: 34
    }

    return 0
}
```

4.5.2 while 表达式

while 表达式的语法定义为：

```
whileExpression
    : 'while' '(' ('let' deconstructPattern '<-')? expression ')' block
    ;
```

其中 while 是关键字，while 之后是一个小括号，小括号内可以是一个表达式或者一个 let 声明的解构匹配，接着是一个块。

一个基础的 while 表达式举例:

```
main(): Int64 {
    var hundred = 0
    while (hundred < 100) { // until hundred = 100
        hundred++
    }
    return 0
}
```

while 表达式首先对 while 之后的表达式进行求值（要求表达式的类型为 Bool），如果表达式的值等于 true，则执行它之后的块，接着重新计算表达式的值并判断是否重新执行一次循环；如果表达式的值等于 false，则终止循环。

对于包含 let 的 while 表达式，我们称之为 while-let 表达式。我们可以用 while-let 表达式来做一些简单的解构操作。

一个基础的 while-let 表达式举例:

```
main(): Int64 {
    var x: Option<Int64> = Option<Int64>.Some(100)
    // while-let expression
    while (let Some(v) <- x) {
        print("x has value")
        x = ...
    }
    return 0
}
```

while-let 表达式首先对 <- 之后的表达式进行求值（表达式的类型为可以是任意类型），如果表达式的值能匹配 let 之后的 pattern，则执行它之后的块，接着重新计算表达式的值然后再次匹配并判断是否重新执行一次循环；如果匹配失败，则终止当前的 while 循环。

let 之后的 pattern 支持常量模式、通配符模式、绑定模式、Tuple 模式、enum 模式。

4.5.3 do-while 表达式

do-while 表达式的语法定义为:

```
doWhileExpression
    : 'do' block 'while' '(' expression ')'
    ;
```

与 while 表达式不同的是：while 表达式在第一次循环迭代时，如果表达式 expression 的值为 false，则循环体不会被执行；然而对于 do-while 表达式，第一次循环迭代时，先执行循环体 block，然后再根据表达式 expression 的值决定是否再次执行循环体，也就是说 do-while 表达式中的循环体会至少执行一次。例如：

```
main(): Int64 {
    var hundred = 0
    do {
        hundred++
    } while (hundred < 100)
    return 0
}
```

4.5.4 循环表达式总结

for-in、while 和 do-while 这三种循环表达式的表达能力是等价的，通常，在知道循环的次数或遍历一个序列中的所有元素时使用 for-in 表达式；在不知道循环的次数，但知道循环终止条件时使用 while 或 do-while 表达式。

三种循环表达式的类型均为 Unit。

由于 break、continue 表达式必须有包围着它们的循环体，所以对于三种循环表达式，其循环条件中出现的 break 或 continue 均会绑定到其最近的外层循环；如外层不存在循环，则报错。例如

```
while (true) {
    println("outer") // printed once
    do {
        println("inner") // printed once
    } while (break)  // stop the execution of the outer loop
    println("unreached") // not printed
}
```

4.6 try 表达式

根据是否涉及资源的自动管理，将 try 表达式分为两类：不涉及资源自动管理的普通 try 表达式，以及会进行资源自动管理的 try-with-resources 表达式。

try 表达式的语法为：

```
tryExpression
    : 'try' block 'finally' block
    | 'try' block ('catch' '(' catchPattern ')' block)+ ('finally' block)?
    | 'try' '(' resourceSpecifications ')' block ('catch' '(' catchPattern ')'
    ↪  block)* ('finally' block)?
    ;
```

普通 try 表达式的主要目的是错误处理，详见异常章节。
try-with-resources 表达式的主要目的是自动释放非内存资源，详见异常章节。

4.7 控制转移表达式

控制转移表达式会改变程序的执行顺序。控制转移表达式的类型是 Nothing 类型，该类型是任何类型的子类型。仓颉编程语言提供如下控制转移表达式：

- break
- continue
- return
- throw

控制转移表达式可以像其他表达式一样，作为子表达式成为复杂表达式的一部分，但是有可能会导致产生不可达代码（不可达部分会编译告警）：

```
main(): Int64 {
    return return 1 // warning: the left return expression is unreachable
}
```

控制转移表达式中，break 和 continue 必须有包围着它们的循环体，且该循环体无法穿越函数边界；return 必须有包围着它的函数体，且该函数体无法穿越；对 throw 不作要求。

“包围着的循环体”无法穿越“函数边界”。在下面的例子中，break 出现在函数 f 中，外层的 while 循环体不被视作包围着它的循环体；continue 出现在 lambda 表达式中，外层的 while 循环体不被视作包围着它的循环体。

```
while (true) {
    func f() {
        break // Error: break must be used directly inside a loop
    }
    let g = { =>
        continue // Error: continue must be used directly inside a loop
    }
}
```

控制转义表达式的语法为：

```
jumpExpression
  : 'break'
  | 'continue'
  | 'return' expression?
  | 'throw'  expression
  ;
```

4.7.1 break 表达式

break 表达式只能出现在循环表达式的循环体中，并将程序的执行权交给被终止循环表达式之后的表达式。例如下面的代码通过在 while 循环体内使用 break 表达式，实现在区间 [1,49] 内计算 4 和 6 的最小公倍数。

```
main(): Int64 {
    var index: Int32 = 0
    while (index < 50) {
        index = index + 1
        if ((index % 4 == 0) && (index % 6 == 0)) {
            print("${index} is divisible by both 4 and 6") // output: 12
            break
        }
    }
    return 0
}
```

需要注意的是，当 break 出现在嵌套的循环表达式中时，只能终止直接包围它的循环表达式，外层的循环并不会受影响。例如下面的程序将输出 5 次 12 is divisible by both 4 and 6，且每次同时会输出 i 的值：

```
main(): Int64 {
    var index: Int64 = 0
    for (i in 0..5) {
        index = i
        while (index < 20) {
            index = index + 1
            if ((index % 4 == 0) && (index % 6 == 0)) {
                print("${index} is divisible by both 4 and 6")
                break
            }
        }
        print("${i}th test")
    }
    return 0
}
```

4.7.2 continue 表达式

continue 表达式只能出现在循环表达式的循环体中，用于提前结束离它最近循环表达式的当前迭代，然后开始新一轮的循环（并不会终止循环表达式）。例如下面的代码输出区间 [1,49] 内所有可以同时被 4 和 6 整除的数（12、24、36、48），对于其他不满足要求的数，同样会显式地输出。

```
main(): Int64 {
    var index: Int32 = 0
    while (index < 50) {
        index = index + 1
        if ((index % 4 == 0) && (index % 6 == 0)) {
            print("${index} is divisible by both 4 and 6")
            continue
        }
        print("${index} is not what we want")
    }
    return 0
}
```

4.7.3 return 表达式

return 表达式只能出现在函数体中，它可以在任意位置终止函数的执行并返回，实现控制流从被调用函数到调用函数的转移。return 表达式有两种形式：return 和 return expr（expr 是一个表达式）。

- 若为 return expr 的形式，我们将 expr 的值作为函数的返回值，所以要求 expr 的类型与函数定义中的返回类型保持一致。

```
// return expression
func larger(a: Int32, b: Int32): Int32 {
    if (a >= b) {
        return a
    } else {
        return b
    }
}
```

- 若为 return 的形式，我们将其视为 return () 的语法糖，所以要求函数的返回类型也为 Unit。

```
// return expression
func equal(a: Int32, b: Int32): Unit {
    if (a == b) {
        print("a is equal to b")
        return
    } else {
        print("a is not equal to b")
    }
}
```

需要说明的是，return 表达式作为一个整体，其类型并不由后面跟随的表达式决定（return 后面跟随的表达式为 ()），而是 Nothing 类型。

4.7.4 throw 表达式

throw 表达式用于抛出异常，在调用包含 throw 表达式的代码块时，如果 throw 表达式被执行到，就会抛出相应的异常，并由事先定义好的异常处理逻辑进行捕获和处理，从而改变程序的执行流程。

下面的例子中，当除数为 0 时，抛出算术异常：

```
func div(a: Int32, b: Int32): Int32 {
    if (b != 0) {
        return a / b
    } else {
        throw ArithmeticException()
    }
}
```

关于 return 和 throw 表达式，本节只做了最简单的使用举例，有关它们的详细介绍，请分别参见函数和异常章节。

4.8 数值类型转换表达式

数值类型转换表达式用于实现数值类型间的转换，它的值是类型转换后的值，它的类型是转换到的目标类型（但原表达式的类型不受目标类型影响），详细的转换规则可参见类型转换。

数值类型转换表达式的语法定义为：

```
numericTypeConvExpr
    : numericTypes '(' expression ')'
    ;

numericTypes
    : 'Int8'
    | 'Int16'
    | 'Int32'
    | 'Int64'
    | 'UInt8'
    | 'UInt16'
    | 'UInt32'
    | 'UInt64'
    | 'Float16'
    | 'Float32'
    | 'Float64'
    ;
```

4.9 this 和 super 表达式

this 和 super 表达式分别使用 this 和 super 表示，this 可以出现在所有实例成员函数和构造函数中，表示当前实例，super 只能出现在 class 类型定义中，表示当前定义的类型的直接父类的实例（详见类）。禁止使用单独的 super 表达式。

this 和 super 表达式的语法定义为：

```
thisSuperExpression
    : 'this'
    | 'super'
    ;
```

4.10 spawn 表达式

spawn 表达式用于创建并启动一个 thread，详见并发章节。

4.11 synchronized 表达式

synchronized 表达式用于同步机制中，详见并发章节。

4.12 括号表达式

括号表达式是指使用圆括号括起来的表达式。圆括号括起来的子表达式被视作一个单独的计算单元被优先计算。

括号表达式的语法定义为：

```
parenthesizedExpression
    : '(' expression ')'
    ;
```

括号表达式举例：

```
1 + 2 * 3 - 4    // The result is 3.
(1 + 2) * 3 - 4  // The result is 5.
```

4.13 后缀表达式

后缀表达式由表达式加上后缀操作符构成。根据后缀操作符的不同，分为：成员访问表达式、函数调用表达式、索引表达式。在成员访问表达式、函数调用表达式、索引表达式中的后缀操作符的前面，可以使用可选的 ? 操作符，以实现 Option 类型对这些后缀操作符的支持。关于 ? 操作符，详见下文介绍。

后缀表达式的语法定义为：

```
postfixExpression
    : atomicExpression
    | type '.' identifier
    | postfixExpression '.' identifier typeArguments?
    | postfixExpression callSuffix
    | postfixExpression indexAccess
    | postfixExpression '.' identifier callSuffix? trailingLambdaExpression
    | identifier callSuffix? trailingLambdaExpression
    | postfixExpression ('?' questSeperatedItems)+
    ;
```

4.13.1 成员访问表达式

成员访问表达式的语法定义为上述后缀表达式语法的第 3 条:

```
postfixExpression '.' identifier typeArguments?
```

成员访问表达式可以用于访问 class、interface、struct 等的成员。

成员访问表达式的形式为 `T.a`。T 可以表示为特定的实例或类型名，我们将其称为成员访问表达式的主体。a 表示成员的名字。

- 如果 T 是类的实例化对象，通过这种方式可以访问类或接口中的非静态成员。
- 如果 T 是 `struct` 的实例，允许通过实例名访问 `struct` 内的非静态成员。
- 如果 T 是类名、接口名或 `struct` 名，允许直接通过类型名访问其静态成员。

需要注意的是：类、接口和 `struct` 的静态成员的访问主体只能是类型名。

- T 是 `this`：在类或接口的作用域内，可以通过 `this` 关键字访问非静态成员。
- T 是 `super`：在类或接口作用域内，可以通过 `super` 关键字访问当前类对象直接父类的非静态成员。

对于成员访问表达式 `e.a`，如果 e 是类型名:

- 当 a 是 e 的可变静态成员变量时，`e.a` 是可变的，其他情况下 `e.a` 是不可变的。

如果 e 是表达式（假设 e 的类型是 T）:

- 当 T 是引用类型时，如果 a 是 T 的可变实例成员变量，则 `e.a` 是可变的，否则 `e.a` 是不可变的;
- 当 T 是值类型时，如果 e 可变且 a 是 T 的可变实例成员变量，则 `e.a` 是可变的，否则 `e.a` 是不可变的。

4.13.2 函数调用表达式

函数调用表达式的语法定义为上述后缀表达式语法的第 4 条，其中 `callSuffix` 和 `valueArgument` 的语法定义为:

```
callSuffix
    : '(' (valueArgument (',' valueArgument)*)? ')'
    ;

valueArgument
    : identifier ':' expression
    | expression
    | refTransferExpression
    ;

refTransferExpression
    : 'inout' (expression '.')? identifier
    ;
```

函数调用表达式用于调用函数，函数详见第 5 章。

对于函数调用表达式 f(x)，假设 f 的类型是 T。如果 T 是函数类型，则调用名为 f 的函数，否则，如果 T 重载了函数调用操作符 ()，则 f(x) 会调用其 () 操作符重载函数（参见可以被重载的操作符）。

4.13.3 索引访问表达式

索引访问表达式的语法定义为上述后缀表达式语法的第 5 条，其中 indexAccess 的语法定义为：

```
indexAccess
    : '[' (expression | rangeElement) ']'
    ;

rangeElement
    :  '..'
    | ('..=' | '..' ) expression
    | expression '..'
    ;
```

索引访问表达式用于那些支持索引访问的类型（包括 Array 类型和 Tuple 类型）通过下标来访问其具体位置的元素，详见第 2 章中关于 Array 类型和 Tuple 类型的介绍。

对于索引访问表达式 e[a]（假设 e 的类型是 T）：

- 当 T 是元组类型时，e[a] 是不可变的；
- 当 T 不是元组类型时，如果 T 重载了 set 形式的操作符 []（参见可以被重载的操作符），则 e[a] 是可变的，否则 e[a] 是不可变的。

对于索引访问表达式 e1[e2]，仓颉语言总是先求值 e1 至 v1，再将 e2 求值至 v2，最后根据下标 v2 选取对应的值或调用相应的重载了的 [] 操作符。

4.13.4 问号操作符

问号操作符 ? 为一元后缀操作符，它必须和上文介绍的后缀操作符 .，()，{} 或 [] 一起使用（出现在后缀操作符之前），实现 Option 类型对这些后缀操作符的支持，例如：a?.b，a?(b)，a?[b] 等等。其中 () 是函数调用，当函数调用最后一个实参是 lambda 时，可以使用尾闭包语法 a?{b}。将包含 ?.、?()、?{} 或 ?[] 的表达式称为 optional chaining 表达式。

optional chaining 表达式的语法定义为上述后缀表达式语法的最后一条，其中 questSeperatedItems 的语法定义为：

```
questSeperatedItems
    : questSeperatedItem+
    ;

questSeperatedItem
    : itemAfterQuest (callSuffix | callSuffix? trailingLambdaExpression |
    ↪  indexAccess)?
    ;

itemAfterQuest
    : '.' identifier typeArguments?
    | callSuffix
    | indexAccess
    | trailingLambdaExpression
    ;
```

关于 optional chaining 表达式，规定：

1. 对于表达式 e，将 e 中的所有 ? 删除，并且将紧邻 ? 之前的表达式的类型由 Option<T> 替换为 T 之后，得到表达式 e1。如果 e1 的类型是 Option 类型，则在 e 之后使用 .、()、{} 或 [] 时，需要在 e 和这些操作符之间加上 ?；否则，不应该加 ?；
2. Optional chaining 表达式的类型是 Option<T>（即无论其中有几个 ?，类型都只有一层 Option），类型 T 为 optional chaining 中最后一个表达式（变量或函数名、函数调用表达式、下标访问表达式）的类型；
3. 一旦 optional chaining 中的某个 Option 类型的表达式的值为 None，则整个 optional chaining 表达式的值为 None；如果 optional chaining 中每个 Option 类型的表达式的值都等于某个 Some 值，则整个表达式的值为 Some(v)（v 的类型是最后一个表达式的类型）。

以表达式 a?.b，c?(d) 和 e?[f] 为例，说明如下：

1. 表达式 a 的类型需要是某个 Option<T1> 且 T1 包含实例成员 b；表达式 c 的类型需要是某个 Option<(T2)->U2> 且 d 的类型为 T2；表达式 e 的类型需要是某个 Option<T3> 且 T3 支持下标操作符；
2. 表达式 a?.b，c?(d) 和 e?[f] 的类型分别为 Option<U1>，Option<U2> 和 Option<U3>，其中 U1 是 T1 中实例成员 b 的类型，U2 是函数类型 (T2)->U2 的返回值类型，U3 是 T3 执行下标操作的返回类型；

3. 当 a，c 和 e 的值分别等于 Some(v1)，Some(v2) 和 Some(v3) 时，a?.b，c?(d) 和 e?[f] 的值分别等于 Option<U1>.Some(v1.b)，Option<U2>.Some(v2(d)) 和 Option<U3>.Some(v3[f])；当 a，c 和 e 的值分别等于 None 时，a?.b，c?(d) 和 e?[f] 的值分别等于 Option<U1>.None，Option<U2>.None 和 Option<U3>.None（注意这里的 b，d 和 f 都不会被求值）。

事实上，表达式 a?.b，c?(d) 和 e?[f] 分别等价于如下 match 表达式：

```
// a?.b is equivalent to the following match expression.
match (a) {
    case Some(v) => Some(v.b)
    case None => None<U1>
}

// c?(d) is equivalent to the following match expression.
match (c) {
    case Some(v) => Some(v(d))
    case None => None<U2>
}

// e?[f] is equivalent to the following match expression.
match (e) {
    case Some(v) => Some(v[f])
    case None => None<U3>
}
```

再来看一个包含多个 ? 的多层访问的例子 a?.b.c?.d（以 ?. 为例，其他操作类似，不再赘述）：

1. 表达式 a 的类型需要是某个 Option<Ta> 且 Ta 包含实例成员 b，b 的类型中包含实例成员变量 c 且 c 的类型是某个 Option<Tc>，Tc 包含实例成员 d；

2. 表达式 a?.b.c?.d 的类型为 Option<Td>，其中 Td 是 Tc 的实例成员 d 的类型；

3. 当 a 的值等于 Some(va) 且 va.b.c 的值等于 Some(vc) 时，a?.b.c?.d 的值等于 Option<Td>.Some(vc.d)；当 a 的值等于 Some(va) 且 va.b.c 的值等于 None 时，a?.b.c?.d 的值等于 Option<Td>.None（d 不会被求值）；当 a 的值等于 None 时，a?.b.c?.d 的值等于 Option<Td>.None（b，c 和 d 都不会被求值）。

表达式 a?.b.c?.d 等价于如下 match 表达式：

```
// a?.b.c?.d is equivalent to the following match expression.
match (a) {
    case Some(va) =>
        let x = va.b.c
        match (x) {
            case Some(vc) => Some(vc.d)
```

```
                case None => None<Td>
            }
        case None =>
            None<Td>
    }
```

Optional chaining 也可以作为左值表达式（参见赋值表达式），例如 `a?.b = e1`, `a?[b] = e2`, `a?.b.c?.d = e3` 等。

赋值表达式的左侧是 optional chaining 表达式时，要求 optional chaining 表达式是可变的（参见赋值表达式）。因为函数类型是不可变的，所以只需要考虑 `?.` 和 `?[]` 这两种情况，并且它们都可以归纳到 `a?.b = c` 和 `a?[b] = c` 这两种基本的场景（假设 a 的类型为 `Option<T>`），规定：

- 对于 `a?.b`，仅当 T 是引用类型且 b 可变时，`a?.b` 是可变的，其他情况下 `a?.b` 是不可变的。
- 对于 `a?[b]`，仅当 T 是引用类型且重载了 set 形式的操作符 `[]` 时，`a?[b]` 是可变的，其他情况下 `a?[b]` 是不可变的。

当 `a?.b`（或 `a?[b]`）可变时，如果 a 的值等于 `Option<T>.Some(v)`，将 c 的值赋值给 `v.b`（或 `v[b]`）；如果 a 的值等于 `Option<T>.None`，什么也不做（b 和 c 也不会被求值）。

类似地，表达式 `a?.b = e1`，`a?[b] = e2` 和 `a?.b.c?.d = e3` 分别等价于如下 match 表达式。

```
// a?.b = e1 is equivalent to the following match expression.
match (a) {
    case Some(v) => v.b = e1
    case None => ()
}

// a?[b] = e2 is equivalent to the following match expression.
match (a) {
    case Some(v) => v[b] = e2
    case None => ()
}

// a?.b.c?.d = e3 is equivalent to the following match expression.
match (a) {
    case Some(va) =>
        match (va.b.c) {
            case Some(vc) => vc.d = e3
            case None => ()
        }
    case None =>
        ()
}
```

操作符 ? 应用举例：

```
// The usuage of ?.
class C {
    var item: Int64 = 100
}
let c = C()
let c1 = Option<C>.Some(c)
let c2 = Option<C>.None
let r1 = c1?.item              // r1 = Option<Int64>.Some(100)
let r2 = c2?.item              // r2 = Option<Int64>.None
func test1() {
    c1?.item = 200             // c.item = 200
    c2?.item = 300             // no effect
}

// The usuage of ?()
let foo = {i: Int64 => i + 1}
let f1 = Option<(Int64) -> Int64>.Some(foo)
let f2 = Option<(Int64) -> Int64>.None
let r3 = f1?(1)                // r3 = Option<Int64>.Some(2)
let r4 = f2?(1)                // r4 = Option<Int64>.None

// The usuage of ?[] for tuple access
let tuple = (1, 2, 3)
let t1 = Option<(Int64, Int64, Int64)>.Some(tuple)
let t2 = Option<(Int64, Int64, Int64)>.None
let r7 = t1?[0]                // r7 = Option<Int64>.Some(1)
let r8 = t2?[0]                // r8 = Option<Int64>.None
func test3() {
    t1?[0] = 10                // error: 't1?[0]' is immutable
    t2?[1] = 20                // error: 't2?[0]' is immutable
}
```

4.14 自增自减表达式

自增自减表达式是包含自增操作符（++）或自减操作符（--）的表达式；a++ 与 a-- 分别是 a+=1 与 a-=1 的语法糖。自增和自减操作符实现对值的加 1 和减 1 操作，且只能作为后缀操作符使用。++ 和 --是非结合的，所以类似于 a++--这样的在同一个表达式中同时包含两个及以上 ++/--但未使用圆括号规定计算顺序的表达式是被（从语法上）禁止的。

自增（自减）表达式的语法定义为：

```
incAndDecExpression
    : postfixExpression ('++' | '--' )
```

```
    ;
```

对于表达式 expr++（或 expr--），规定如下：

1. expr 的类型必须是整数类型；
2. 因为 expr++（或 expr--）是 expr += 1 （或 expr -= 1）的语法糖，所以此 expr 同时必须也是可被赋值的（见赋值表达式）；
3. expr++（或 expr--）的类型为 Unit。

自增（自减）表达式举例：

```
var i: Int32 = 5
i++     // i = 6
i--     // i = 5
i--++   // syntax error
var j = 0
j = i-- // semantics error
```

4.15 算术表达式

算术表达式是包含算术操作符的表达式。仓颉编程语言支持的算术操作符包括：一元负号（-）、加（+）、减（-）、乘（*）、除（/）、取余（%）、求幂（**）。除了一元负号是一元前缀操作符，其他操作符均是二元中缀操作符。它们的优先级和结合性参见下文。

算术表达式的语法定义为：

```
prefixUnaryExpression
    : prefixUnaryOperator* incAndDecExpression
    ;

prefixUnaryOperator
    : '-'
    | ...
    ;

additiveExpression
    : multiplicativeExpression (additiveOperator multiplicativeExpression)*
    ;

multiplicativeExpression
    : exponentExpression (multiplicativeOperator exponentExpression)*
    ;
```

```
exponentExpression
    : prefixUnaryExpression (exponentOperator prefixUnaryExpression)*
    ;

additiveOperator
    : '+' | '-'
    ;

multiplicativeOperator
    : '*' | '/' | '%'
    ;

exponentOperator
    : '**'
    ;
```

一元负号（-）的操作数只能是数值类型的表达式。一元前缀负号表达式的值等于操作数取负的值，类型和操作数的类型相同：

```
let num1: Int64 = 8
let num2 = -num1      // num2 = -8, with 'Int64' type
let num3 = -(-num1)   // num3 =  8, with 'Int64' type
```

对于二元操作符 *，/，%，+ 和 -，要求两个操作数的类型相同。其中 % 的操作数只支持整数类型，*，/，+ 和 - 的操作数可以是任意数值类型。

```
let a = 2 + 3       // add:       5
let b = 3 - 1       // sub:       2
let c = 3 * 4       // multi:    12
let d = 6.6 / 1.1   // division:  6
let e = 4 % 3       // mod:       1
```

特别地，除法（/）的操作数为整数时，将非整数值向 0 的方向舍入为整数。整数取余运算 a % b 的值定义为 a - b * (a / b)。

```
let q1 =  7 /  3    // integer division:   2
let q2 = -7 /  3    // integer division:  -2
let q3 =  7 / -3    // integer division:  -2
let q4 = -7 / -3    // integer division:   2
let r1 =  7 %  3    // integer remainder:  1
let r2 = -7 %  3    // integer remainder: -1
let r3 =  7 % -3    // integer remainder:  1
let r4 = -7 % -3    // integer remainder: -1
```

** 表示求幂运算（如 x**y 表示计算底数 x 的 y 次幂）。** 的左操作数只能为 Int64 类型或 Float64 类型，并且：

1. 当左操作类型为 Int64 时，右操作数只能为 UInt64 类型，表达式的类型为 Int64；
2. 当左操作类型为 Float64 时，右操作数只能为 Int64 类型或 Float64 类型，表达式的类型为 Float64。

```
let p1 = 2 ** 3               // p1 = 8
let p2 = 2 ** UInt64(3 ** 2)  // p2 = 512
let p3 = 2.0 ** 3.0           // p3 = 8.0
let p4 = 2.0 ** 3 ** 2        // p4 = 512.0
let p5 = 2.0 ** 3.0           // p5 = 8.0
let p6 = 2.0 ** 3.0 ** 2.0    // p6 = 512.0
```

当左操作类型为 Float64，右操作数类型为 Int64 时，存在一些特殊情况需要明确求幂表达式的值，具体罗列为：

```
x ** 0 = 1.0                                  // for any x
0.0 ** n = POSITIVE_INFINITY                  // for odd n < 0
-0.0 ** n = NEGATIVE_INFINITY                 // for odd n < 0
0.0 ** n = POSITIVE_INFINITY                  // for even n < 0
-0.0 ** n = POSITIVE_INFINITY                 // for even n < 0
0.0 ** n = 0.0                                // for even n > 0
-0.0 ** n = 0.0                               // for even n > 0
0.0 ** n = 0.0                                // for odd n > 0
-0.0 ** n = -0.0                              // for odd n > 0
POSITIVE_INFINITY ** n = POSITIVE_INFINITY // for n > 0
NEGATIVE_INFINITY ** n = NEGATIVE_INFINITY // for odd n > 0
NEGATIVE_INFINITY ** n = POSITIVE_INFINITY // for even n > 0
POSITIVE_INFINITY ** n = 0.0                  // for n < 0
NEGATIVE_INFINITY ** n = -0.0                 // for odd n < 0
NEGATIVE_INFINITY ** n = 0.0                  // for even n < 0.
```

注：在上述所列特殊情况之外，当左操作数的值为 NaN 时，无论右操作数取何值，求幂表达式的值均等于 NaN。

当左操作类型为 Float64，右操作数类型为 Float64 时，同样存在一些特殊情况需要明确求幂表达式的值。具体罗列为：

```
x ** 0.0 = 1.0                                 // for any x
x ** -0.0 = 1.0                                // for any x
0.0 ** y = POSITIVE_INFINITY                   // for the value of y is equal to an
 ↪  odd integer < 0
-0.0 ** y = NEGATIVE_INFINITY                  // for the value of y is equal to an
 ↪  odd integer < 0
0.0 ** NEGATIVE_INFINITY = POSITIVE_INFINITY
-0.0 ** NEGATIVE_INFINITY = POSITIVE_INFINITY
0.0 ** POSITIVE_INFINITY = 0.0
```

```
-0.0 ** POSITIVE_INFINITY = 0.0
0.0 ** y = 0.0                              // for finite y > 0.0 and its value is
 ↪ equal to an odd integer
-0.0 ** y = -0.0                            // for finite y > 0.0 and its value is
 ↪ equal to an odd integer
-1.0 ** POSITIVE_INFINITY = 1.0
-1.0 ** NEGATIVE_INFINITY = 1.0
1.0 ** y = 1.0                              // for any y
x ** POSITIVE_INFINITY = 0.0                //  for -1.0 < x < 1.0
x ** POSITIVE_INFINITY = POSITIVE_INFINITY // for any x < -1.0 or for any x > 1.0
x ** NEGATIVE_INFINITY = POSITIVE_INFINITY // for -1.0 < x < 1.0
x ** NEGATIVE_INFINITY = 0.0                // for any x < -1.0 or for any x > 1.0
POSITIVE_INFINITY ** y = 0.0                // for y < 0.0
POSITIVE_INFINITY ** y = POSITIVE_INFINITY // for y > 0.0
NEGATIVE_INFINITY ** y = -0.0               // for finite y < 0.0 and its value is
 ↪ equal to an odd integer
NEGATIVE_INFINITY ** y = NEGATIVE_INFINITY // for finite y > 0.0 and its value is
 ↪ equal to an odd integer
NEGATIVE_INFINITY ** y = 0.0                // for finite y < 0.0 and its value is
 ↪ not equal to an odd integer
NEGATIVE_INFINITY ** y = POSITIVE_INFINITY // for finite y > 0.0 and its value is
 ↪ not equal to an odd integer
0.0 ** y = POSITIVE_INFINITY                // for finite y < 0.0 and its value is
 ↪ not equal to an odd integer
-0.0 ** y = POSITIVE_INFINITY               // for finite y < 0.0 and its value is
 ↪ not equal to an odd integer
0.0 ** y = 0.0                              // for finite y > 0.0 and its value is
 ↪ not equal to an odd integer
-0.0 ** y = 0.0                             // for finite y > 0.0 and its value is
 ↪ not equal to an odd integer
x ** y = NaN                               // for finite x < 0.0 and finite y whose
 ↪ value is not equal to an integer
```

注：在上述所列特殊情况之外，一旦有操作数的值为 NaN，则求幂表达式的值等于 NaN。

算术表达式结果为整数类型时存在整数溢出的可能。仓颉提供了三种属性宏及编译选项来控制整数溢出的处理行为（以下简称为行为）。如下表格所示：

attributes	Options	behavior	explaining for behavior
@OverflowThrowing	-int-overflow throwing	throwing	throwing an exception
@OverflowWrapping	-int-overflow wrapping	wrapping	wrapping around at the numeric bounds of the type

attributes	Options	behavior	explaining for behavior
@OverflowSaturating	int-overflow saturating	saturating	saturating at the numeric bounds of the type

注：1. 默认的行为为 throwing。2. 假设属性宏与编译选项同时作用在某一范围，且代表的行为不相同，则该范围以属性宏代表的行为为准。

行为示例：

```
@OverflowThrowing
func test1(x: Int8, y: Int8) { // if x equals to 127 and y equals to 3
    let z = x + y // throwing OverflowException
}

@OverflowWrapping
func test2(x: Int8, y: Int8) { // if x equals to 127 and y equals to 3
    let z = x + y // z equals to -126
}

@OverflowSaturating
func test3(x: Int8, y: Int8) { // if x equals to 127 and y equals to 3
    let z = x + y // z equals to 127
}
```

属性宏与编译选项作用范围冲突示例：

```
// Compile with cjc --int-overflow saturating test.cj
// this file's name is test.cj

@OverflowWrapping
func test2(x: Int8, y: Int8) {
    let z = x + y // the behavior is wrapping
}

func test3(x: Int8, y: Int8) {
    let z = x + y // the behavior is saturating
}
```

特别地，对于 `INT_MIN * -1`, `INT_MIN / -1` 和 `INT_MIN % -1`，规定的行为如下：

Expression	Throwing	Wrapping	Saturating
INT_MIN * -1 or -1 * INT_MIN	throwing OverflowException	INT_MIN	INT_MAX

Expression	Throwing	Wrapping	Saturating
INT_MIN / -1	throwing OverflowException	INT_MIN	INT_MAX
INT_MIN % -1	0	0	0

需要注意的是，对于整数溢出行为是 throwing 的场景，若整数溢出可提前在编译期检测出来，则编译器会直接给出报错。

4.16 关系表达式

关系表达式是包含关系操作符的表达式。关系操作符包括 6 种：相等（==）、不等（!=）、小于（<）、小于等于（<=）、大于（>）、大于等于（>=）。关系操作符都是二元操作符，并且要求两个操作数的类型是一样的。关系表达式的类型是 Bool 类型，即值只可能是 true 或 false。关系操作符的优先级和结合性见下文。

关系表达式的语法定义为：

```
equalityComparisonExpression
    : comparisonOrTypeExpression (equalityOperator comparisonOrTypeExpression)?
    ;

comparisonOrTypeExpression
    : shiftingExpression (comparisonOperator shiftingExpression)?
    | ...
    ;

equalityOperator
    : '!=' | '=='
    ;

comparisonOperator
    : '<' | '>' | '<=' | '>='
    ;
```

关系表达式举例：

```
main(): Int64 {
    3 < 4         // return true
    3 <= 3        // return true
    3 > 4         // return false
    3 >= 3        // return true
    3.14 == 3.15  // return false
    3.14 != 3.15  // return true
    return 0
}
```

需要注意的是，关系运算符是非结合（non-associative）运算符，即无法写出类似于 a < b < c 这样的表达式。

```
main(): Int64 {
    3 < 4 < 5      // error: `<` is non-associative
    3 == 3 != 4    // error: `==` and `!=` are non-associative
    return 0
}
```

4.17 type test 和 type cast 表达式

type test 表达式是包含操作符 is 的表达式，type cast 表达式是包含操作符 as 的表达式。is 和 as 的优先级和结合性参见下文。

type test 和 type cast 表达式的语法定义为：

```
comparisonOrTypeExpression
    : ...
    | shiftingExpression ('is' type)?
    | shiftingExpression ('as' userType)?
    ;
```

4.17.1 is 操作符

e is T 是一个用于类型检查的表达式，e is T 的类型是 Bool。其中 e 可以是任何类型的表达式，T 可以是任何类型。

当 e 的运行时类型 R 是 T 的子类型时，e is T 的值为 true，否则值为 false。

is 操作符举例：

```
open class Base {
    var name: String = "Alice"
}
class Derived1 <: Base {
    var age: UInt8 = 18
}
class Derived2 <: Base {
    var gender: String = "female"
}

main(): Int64 {

    var testVT = 1 is Int64            // testVT = true
    testVT = 1 is String               // testVT = false
    testVT = true is Int64             // testVT = false
```

```
    testVT = [1, 2, 3] is Array<Int64> // testVT = true

    let base1: Base = Base()
    let base2: Base = Derived1()
    let base3: Base = Derived2()

    let derived1: Derived1 = Derived1()
    let derived2: Derived2 = Derived2()

    var test = base1 is Base          // test = true
    test = base1 is Derived1          // test = false
    test = base1 is Derived2          // test = false
    test = base2 is Base              // test = true
    test = base2 is Derived1          // test = true
    test = base2 is Derived2          // test = false
    test = base3 is Base              // test = true
    test = base3 is Derived1          // test = false
    test = base3 is Derived2          // test = true

    test = derived1 is Base           // test = true
    test = derived1 is Derived1       // test = true
    test = derived1 is Derived2       // test = false
    test = derived2 is Base           // test = true
    test = derived2 is Derived1       // test = false
    test = derived2 is Derived2       // test = true

    return 0
}
```

4.17.2 as 操作符

e as T 是一个用于类型转换的表达式，e as T 的类型是 Option<T>。其中 e 可以是任何类型的表达式，T 可以是任何具体类型。

当 e 的运行时类型 R 是 T 的子类型时，e as T 的值为 Some(e)，否则值为 None。

as 操作符举例：

```
open class Base {
    var name: String = "Alice"
}
class Derived1 <: Base {
    var age: UInt8 = 18
}
```

```
class Derived2 <: Base {
    var gender: String = "female"
}

main(): Int64 {
    let base1: Base = Base()
    let base2: Base = Derived1()
    let base3: Base = Derived2()

    let derived1: Derived1 = Derived1()
    let derived2: Derived2 = Derived2()

    let castOP1 = base1 as Base         // castOP = Option<Base>.Some(base1)
    let castOP2 = base1 as Derived1     // castOP = Option<Derived1>.None
    let castOP3 = base1 as Derived2     // castOP = Option<Derived2>.None
    let castOP4 = base2 as Base         // castOP = Option<Base>.Some(base2)
    let castOP5 = base2 as Derived1     // castOP = Option<Derived1>.Some(base2)
    let castOP6 = base2 as Derived2     // castOP = Option<Derived2>.None
    let castOP7 = base3 as Base         // castOP = Option<Base>.Some(base3)
    let castOP8 = base3 as Derived1     // castOP = Option<Derived1>.None
    let castOP9 = base3 as Derived2     // castOP = Option<Derived2>.Some(base3)

    let castOP10 = derived1 as Base     // castOP = Option<Base>.Some(derived1)
    let castOP11 = derived1 as Derived1 // castOP = Option<Derived1>.Some(derived1)
    let castOP12 = derived1 as Derived2 // castOP = Option<Derived2>.None
    let castOP13 = derived2 as Base     // castOP = Option<Base>.Some(derived2)
    let castOP14 = derived2 as Derived1 // castOP = Option<Derived1>.None
    let castOP15 = derived2 as Derived2 // castOP = Option<Derived2>.Some(derived2)

    return 0
}
```

4.18 位运算表达式

位运算表达式是包含位运算操作符的表达式。仓颉编程语言支持 1 种一元前缀位运算操作符：按位求反（!），以及 5 种二元中缀位运算操作符：左移（<<）、右移（>>）、按位与（&）、按位异或（^）和按位或（|）。位运算操作符的操作数只能为整数类型，通过将操作数视为二进制序列，然后在每一位上进行逻辑运算（0 视为 false，1 视为 true ）或移位操作来实现位运算。&、^ 和 | 的操作中，位与位之间执行的是逻辑操作（参见逻辑表达式）。位运算操作符的优先级和结合性参见下文。

位运算表达式的语法定义为：

```
prefixUnaryExpression
```

```
    : prefixUnaryOperator* incAndDecExpression
    ;

prefixUnaryOperator
    : '!'
    | ...
    ;

bitwiseDisjunctionExpression
    : bitwiseXorExpression ( '|' bitwiseXorExpression)*
    ;

bitwiseXorExpression
    : bitwiseConjunctionExpression ( '^' bitwiseConjunctionExpression)*
    ;

bitwiseConjunctionExpression
    : equalityComparisonExpression ( '&' equalityComparisonExpression)*
    ;

shiftingExpression
    : additiveExpression (shiftingOperator additiveExpression)*
    ;

shiftingOperator
    : '<<' | '>>'
    ;
```

位运算表达式举例:

```
func foo(): Unit {
    !10             // The result is -11
    !20             // The result is -21
    10 << 1         // The result is 20
    10 << 1 << 1    // The result is 40
    10 >> 1         // The result is 5
    10 & 15         // The result is 10
    10 ^ 15         // The result is 5
    10 | 15         // The result is 15
    1 ^ 8 & 15 | 24 // The result is 25
}
```

对于移位操作符，要求其操作数必须是整数类型（但两个操作数的类型可以不一样），并且无论左移还是

右移，右操作数都不允许为负数（对于编译时可检查出的此类错误，编译报错，如果运行时发生此错误，则抛出异常）。

对于无符号数的移位操作，移位和补齐规则是：左移低位补 0 高位丢弃，右移高位补 0 低位丢弃。对于有符号数的移位操作，移位和补齐规则是：

1. 正数和无符号数的移位补齐规则一致;
2. 负数左移低位补 0 高位丢弃;
3. 负数右移高位补 1 低位丢弃。

```
let p: Int8 = -30
let q = p << 2      // q = -120
let r = p >> 2      // r = -8
let r = p >> -2     // error
let x: UInt8 = 30
let b = x << 3      // b = 240
let b = x >> 1      // b = 15
```

另外，如果右移或左移的位数（右操作数）等于或者大于操作数的宽度，则为 overshift，如果编译时可以检测到则报错，否则运行时抛出异常。

```
let x1 : UInt8 = 30 // 0b00011110
let y1 = x1 >> 11   // compilation error
```

4.19 区间表达式

区间表达式是包含区间操作符的表达式。区间表达式用于创建 Range 实例。区间表达式的语法定义为：

```
rangeExpression
    : bitwiseDisjunctionExpression ('..=' | '..') bitwiseDisjunctionExpression
    ↪  (':' bitwiseDisjunctionExpression)?
    | bitwiseDisjunctionExpression
    ;
```

区间操作符有两种：.. 和..=，分别用于创建“左闭右开”和“左闭右闭”的 Range 实例。关于它们的介绍，请参见Range 类型。

4.20 逻辑表达式

逻辑表达式是包含逻辑操作符的表达式。逻辑操作符的操作数只能为 Bool 类型的表达式。仓颉编程语言支持 4 种逻辑操作符：逻辑非（!）、逻辑与（&&）、逻辑或（||）。它们的优先级和结合性参见下文。

逻辑表达式的语法定义为：

```
prefixUnaryExpression
    : prefixUnaryOperator* incAndDecExpression
```

```
    ;

prefixUnaryOperator
    : '!'
    | ...
    ;

logicDisjunctionExpression
    : logicConjunctionExpression ( '||' logicConjunctionExpression)*
    ;

logicConjunctionExpression
    : rangeExpression ( '&&' rangeExpression)*
    ;
```

逻辑非（!）是一元操作符，它的作用是对其操作数的布尔值取反：!false 的值等于 true，!true 的值等于 false。

逻辑与（&&）和逻辑或（||）均是二元操作符。对于表达式 expr1 && expr2，只有当 expr1 和 expr2 的值均等于 true 时，它的值才等于 true；对于表达式 expr1 || expr2，只有当 expr1 和 expr2 的值均等于 false 时，它的值才等于 false。

&& 和 || 采用短路求值策略：计算 expr1 && expr2 时，当 expr1=false 则无需对 expr2 求值，整个表达式的值为 false；计算 expr1 || expr2 时，当 expr1=true 则无需对 expr2 求值，整个表达式的值为 true。

```
main(): Int64 {
    let expr1 = false
    let expr2 = true
    !true                           // Logical NOT, return false.
    1 > 2 && expr1                  // Logical AND, return false without computing
     ↪ the value of expr1.
    1 < 2 || expr2                  // Logical OR, return true without computing
     ↪ the value of expr2.
    return 0
}
```

4.21 coalescing 表达式

coalescing 表达式是包含 coalescing 操作符的表达式。coalescing 操作符使用 ?? 表示，?? 是二元中缀操作符，其优先级和结合性参见下文。

coalescing 表达式的语法定义为：

```
coalescingExpression
    : logicDisjunctionExpression ('??' logicDisjunctionExpression)*
```

```
    ;
```

coalescing 操作符用于 Option 类型的解构。假设表达式 e1 的类型是 Option<T>，对于表达式 e1 ?? e2，规定：

1. 表达式 e2 具有类型 T。
2. 表达式 e1 ?? e2 具有类型 T。
3. 当 e1 的值等于 Option<T>.Some(v) 时，e1 ?? e2 的值等于 v 的值（此时，不会再去对 e2 求值，即满足“短路求值”）；当 e1 的值等于 Option<T>.None 时，e1 ?? e2 的值等于 e2 的值。

表达式 e1 ?? e2 是如下 match 表达式的语法糖：

```
// when e1 is Option<T>
match (e1) {
    case Some(v) => v
    case None => e2
}
```

coalescing 表达式使用举例：

```
main(): Int64 {
    let v1 = Option<Int64>.Some(100)
    let v2 = Option<Int64>.None
    let r1 = v1 ?? 0
    let r2 = v2 ?? 0
    print("${r1}") // output: 100
    print("${r2}") // output: 0

    return 0
}
```

4.22 流表达式

流表达式是包含流操作符的表达式。流操作符包括两种：表示数据流向的中缀操作符 |>（称为 pipeline）和表示函数组合的中缀操作符 ~>（称为 composition）。|> 和 ~> 的优先级相同，并介于 || 和赋值操作符 = 之间。|> 和 ~> 的结合性均为左结合，详情参考下文。流表达式的语法定义为：

```
flowExpression
    : logicDisjunctionExpression (flowOperator logicDisjunctionExpression)*
    ;

flowOperator
    : '|>' | '~>'
    ;
```

4.22.1 pipeline 操作符

pipeline 表达式是单个参数函数调用的语法糖，即 e1 |> e2 是 let v = e1; e2(v) 的语法糖（即先对 |> 操作符左边的 e1 求值）。这里 e2 是函数类型的表达式，e1 的类型是 e2 的参数类型的子类型；或者 e2 的类型重载了函数调用操作符 ()（参见可以被重载的操作符）。

注意：这里的 f 不能是 init 和 super 构造函数。

```
func f(x: Int32): Int32 { x + 1 }

let a: Int32 = 1
var res = a |> f // ok
var res1 = a |> {x: Int32 => x + 1} // ok

func h(b: Bool) { b }
let res3 = a < 0 || a > 10  |> h // Equivalence: (a < 0 || a > 10)  |> h

func g<T>(x: T): T { x }
var res4 = a |> g<Int32> // ok

class A {
    let a: Int32
    let b: Int32
    init(x: Int32) {
        a = x
        b = 0
    }
    init(x: Int32, y: Int32) {
        x |> init // error: `init` is not a valid expression
        b = y
    }
}

// PIPELINE with operator `()` overloading
class A {
    operator func ()(x: Int32) {
        x
    }
}
let obj = A()
let a: Int32 = 1
let res = a |> obj // Equivalence: obj(a)
```

4.22.2 composition 操作符

composition 表达式表示两个单参函数的组合。也就是说，composition 表达式 e1 ~> e2 是 let f = e1; let g = e2; {x => g(f(x))} 的语法糖（即先对 ~> 操作符左边的 e1 求值）。这里的 f，g 均为函数类型的表达式或者其类型重载了单参的函数调用操作符 ()（参见可以被重载的操作符），则会有以下四种情况：

e1 ~> e2	The lambda expression
e1 and e2 are function types, and the return value type of e1 is a subtype of the argument type of e2	`let f = e1; let g = e2; {x => g(f(x))}`
The type f implements the single-parameter operator () overloading function, and g is a function type, and the return value type of f.operator() is a subtype of the argument type of g	`let f = e1; let g = e2; {x => g(f.operator()(x))}`
f is a function type, and the type of g implements the the single-parameter operator () overloading function, and the return value type of f is a subtype of the argument type of g.operator().	`let f = e1; let g = e2; {x => g.operator()(f(x))}`
The types of f and g both implement the single-parameter operator () overloading function, and the return value type of f.operator() is a subtype of the argument type of g.operator()	`let f = e1; let g = e2; {x => g.operator()(f.operator()(x))}`

注意：这里的 e1，e2 求值后不能是 init 和 super 构造函数。

```
func f(x: Int32): Float32 { Float32(x) }
func g(x: Float32): Int32 { Int32(x) }

var fg = f ~> g // Equivalence: {x: Int32 => g(f(x))}

let lambdaComp = {x: Int32 => x} ~> f // ok

func h1<T>(x: T): T { x }
func h2<T>(x: T): T { x }
var hh = h1<Int32> ~> h2<Int32> // ok

// COMPOSITION with operator `()` overloading
class A {
    operator func ()(x: Int32): Int32 {
        x
    }
```

```
}
class B {
    operator func ()(x: Float32): Float32 {
        x
    }
}
let objA = A()
let objB = B()
let af = objA ~> f // ok
let fb = f ~> objB // ok
let aa = objA ~> objA // ok
```

4.23 赋值表达式

赋值表达式是包含赋值操作符的表达式，用于将左操作数的值修改为右操作数的值，要求右操作数的类型是左操作数类型的子类型。对赋值表达式求值时，总是先计算 = 右边的表达式，再计算 = 左边的表达式，最后进行赋值。

对于复合赋值表达式求值时，总是先计算 = 左边的表达式的左值，然后根据这个左值取右值，然后将该右值与 = 右边的表达式做计算（若有短路规则会继续遵循短路规则），最后赋值。

除了子类型允许的赋值外，如果右操作数是字符串字面量，而左操作数的类型是 Byte 或 Rune，则字符串值将分别被强制赋值为 Byte 或 Rune，并对强制赋值进行赋值。

赋值操作符分为普通赋值操作符和复合赋值操作符，赋值表达式的语法定义为：

```
assignmentExpression
    : leftValueExpressionWithoutWildCard assignmentOperator flowExpression
    | leftValueExpression '=' flowExpression
    | tupleLeftValueExpression `=` flowExpression
    | flowExpression
    ;

tupleLeftValueExpression
    : `(` (leftValueExpression | tupleLeftValueExpression) (`,`
    ↪  (leftValueExpression | tupleLeftValueExpression))+ `,`? `)`
    ;

leftValueExpression
    : leftValueExpressionWithoutWildCard
    | '_'
    ;

leftValueExpressionWithoutWildCard
    : identifier
```

```
    | leftAuxExpression '?'? assignableSuffix
    ;

leftAuxExpression
    : identifier typeArguments?
    | type
    | thisSuperExpression
    | leftAuxExpression ('?')? '.' identifier typeArguments?
    | leftAuxExpression ('?')? callSuffix
    | leftAuxExpression ('?')? indexAccess
    ;

assignableSuffix
    : fieldAccess
    | indexAccess
    ;

fieldAccess
    : '.' identifier
    ;

assignmentOperator
    : '=' | '+=' | '-=' | '**=' | '*=' | '/=' | '%=' | '&&=' | '||='
    | '&=' | '|=' | '^=' | '<<=' | '>>='
    ;
```

出现在（复合）赋值操作符左侧的表达式称为*左值表达式*（上述定义中的 leftValueExpression）。

语法上，*左值表达式*可以是一个 identifier 或 _，或者一个 leftAuxExpression 后接 assignableSuffix（包含 fieldAccess 和 indexAccess 两类），leftAuxExpression 和 assignableSuffix 之间可以有可选的 ? 操作符（对 Option Type 的实例进行赋值的语法糖）。leftAuxExpression 可以是以下语法形式：1、一个包含可选类型实参（typeArguments）的 identifier；2、this 或 super；3、一个 leftAuxExpression 后接一个 .（二者之间可以有可选的 ? 操作符）和一个存在可选类型实参的 identifier；4、一个 leftAuxExpression 后接一个函数调用后缀 callSuffix 或索引访问后缀 indexAccess（callSuffix 或 indexAccess 之前可以有可选的 ? 操作符）。

语义上，*左值表达式*只能是如下形式的表达式：

1. identifier 表示的变量（参见变量名和函数名）；
2. 通配符 _，意味着忽略 = 右侧表达式的求值结果（复合赋值表达式禁止使用通配符）；
3. 成员访问表达式 e1.a 或者 e2?.a（参见成员访问表达式）；
4. 索引访问表达式 e1[a] 或者 e2?[a]（参见索引访问表达式）。

注：其中 e1 和 e2 必须是满足 leftAuxExpression 语法的表达式。

*左值表达式*是否合法，取决于*左值表达式*是否是可变的：仅当*左值表达式*可变时，它才是合法的。关于上述表达式的可变性，可参见对应章节。

赋值表达式的类型是 Unit，值是 ()，这样做的好处是可以避免类似于错误地将赋值表达式当做判等表达式使用等问题的发生。在下面的例子中，如果先执行 (a = b)，则返回值是 ()，而 () 不能出现在 = 的左侧，所以执行 ()=0 时就会报错。同样地，由于 if 之后的表达式必须为 Bool 类型，所以下例中的 if 表达式也会报错。另外，= 是非结合的，所以类似于 a = b = 0 这样的同一个表达式中同时包含两个以上 = 的表达式是被禁止的（无法通过语法检查）。

```
main(): Int64 {
    var a = 1
    var b = 1
    a = (b = 0) // semantics error
    if (a = 5) {  // semantics error
    }
    a = b = 0   // syntax error

    return 0
}
```

复合赋值表达式 a op= b 不能简单看做赋值表达式与其他二元操作符的组合 a = a op b（其中 op 可以是算术操作符、逻辑操作符和位操作符中的任意二元操作符，操作数 a 和 b 的类型为操作符 op 所要求的类型）。在仓颉语言中，a op= b 中的 a 只会被求值一次（副作用也只发生一次），而 a = a op b 中的 a 会被求值两次（副作用也发生两次）。因为复合赋值表达式也是一个赋值表达式，所以复合赋值操作符也是非结合的。复合赋值表达式同样要求两个操作数的类型相同。

下面举例说明复合赋值表达式的使用：

```
a **= b
a *= b
a /= b
a %= b
a += b
a -= b
a <<= b
a >>= b
a &&= b
a ||= b
a &= b
a ^= b
a |= b
```

最后，如果用户重载了 **、*、/、%、+、-、<<、>>、&、^、| 操作符，那么仓颉语言会提供其对应的复合赋值操作符 **=、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、|= 的默认实现。但有些额外的要求，否则无法为 a = a op b 提供赋值语义：

1. 重载后的操作符的返回类型需要与左操作数的类型一致或是其子类型，即对于 `a op= b` 中的 a，b，op，它们需要能通过 `a = a op b` 的类型检查。例如当有子类型关系 `A <: B <: C` 时，若用户重载的 + 的类型是 `(B, Int64) -> B` 或 `(B, Int64) -> A`，则仓颉语言可以提供默认实现；若用户重载的 + 的类型是 `(B, Int64) -> C`，则仓颉语言不会为其提供默认实现。
2. 要求 `a op= b` 中的 a 必须是可被赋值的，例如是一个变量。

多赋值表达式是一种特殊的赋值表达式，多赋值表达式等号左边必须是一个 tuple，这个 tuple 里面的元素必须都是左值，等号右边的表达式也必须是 tuple 类型，右边 tuple 每个元素的类型必须是对应位置左值类型的子类型。注意：当左侧 tuple 中出现 _ 时，表示忽略等号右侧 tuple 对应位置处的求值结果（意味着这个位置处的类型检查总是可以通过的）。

多赋值表达式可以将右边的 tuple 类型的值，一次性赋值给左边 tuple 内的对应左值，省去逐个赋值的代码。

```
main(): Int64 {
    var a: Int64
    var b: Int64

    (a, b) = (1, 2) // a == 1, b == 2
    (a, b) = (b, a) // swap, a == 2, b == 1
    (a, _) = (3, 4) // a == 3
    (_, _) = (5, 6) // no assignment

    return 0
}
```

多赋值表达式可以看成是如下形式的语法糖。赋值表达式右侧的表达式会优先求值，再对左值部分从左往右逐个赋值。

```
main(): Int64 {
    var a: Int64
    var b: Int64
    (a, b) = (1, 2)
    // desugar
    let temp = (1, 2)
    a = temp[0]
    b = temp[1]
    return 0
}
```

4.24 Lambda 表达式

Lambda 表达式是函数类型的值，详见第 5 章函数。

4.25 Quote 表达式

Quote 表达式用于引用代码，并将其表示为可操作的数据对象，主要用于元编程，详见第 14 章元编程。

4.26 宏调用表达式

宏调用表达式用于调用仓颉定义的宏，主要用于元编程，详见第 14 章元编程。

4.27 引用传值表达式

引用传值表达式只可用于 C 互操作中调用 CFunc 场景中，详见第 13 章互操作中 inout 参数一节。

4.28 操作符的优先级和结合性

对于包含两个或两个以上操作符的表达式，它的值由操作符和操作数的分组结合方式决定，而分组结合方式取决于操作符的优先级和结合性。简单来讲，优先级规定了不同操作符的求值顺序，结合性规定了具有相同优先级的操作符的求值顺序。

如果一个表达式中包含多个不同优先级的操作符，那么它的计算顺序是：先计算包含高优先级操作符的子表达式，再计算包含低优先级操作符的子表达式。在包含多个同一优先级操作符的子表达式中，计算次序由操作符的结合性决定。

下表列出了各操作符的优先级、结合性、功能描述、用法以及表达式的类型。其中越靠近表格顶部，操作符的优先级越高：

Operator	Associativity	Description	Usage	Expression type
@	Right associative	macro call expression	@expr1 @expr2	Unit
.	Left associative	Member access	Name.name	The type of name
[]		Index access	varName[expr]	The type of the element of varName
()		Function call	funcName(expr)	Return type of funcName
++	None	Postfix increment	varName++	Unit
--		Postfix decrement	varName--	Unit
?		Question mark	expr1?.expr2 etc.	Option<T> (T is the type of expr2)

Operator	Associativity	Description	Usage	Expression type
!	Right associative	Bitwise Logic NOT	`!expr`	The type of `expr`
-		Unary negative	`-expr`	
**	Right associative	Power	`expr1 ** expr2`	The type of `expr1`
*	Left associative	Multiply	`expr1 * expr2`	The type of `expr1` or `expr2`, since `expr1` and `expr2` have the same type
/		Divide	`expr1 / expr2`	
%		Remainder	`expr1 % expr2`	
+	Left associative	Add	`expr1 + expr2`	The type of `expr1` or `expr2`, since `expr1` and `expr2` have the same type
-		Subtract	`expr1 - expr2`	
<<	Left associative	Bitwise left shift	`expr1 << expr2`	The type of `expr1`, where `expr1` and `expr2` can have different types
>>		Bitwise right shift	`expr1 >> expr2`	
..	None	Range operator	`expr1..expr2:expr3`	Range type
..=			`expr1..=expr2:expr3`	
<	None	Less than	`expr1 < expr2`	Except the type of 'expr as userType' is `Option<userType>`, other expressions have Bool type
<=		Less than or equal	`expr1 <= expr2`	
>		Greater than	`expr1 > expr2`	
>=		Greater than or equal	`expr1 >= expr2`	

Operator	Associativity	Description	Usage	Expression type
`is`		Type check	`expr is type`	
`as`		Type cast	`expr as userType`	
`==`	None	Equal	`expr1 == expr2`	Bool
`!=`		Not equal	`expr1 != expr2`	
`&`	Left associative	Bitwise AND	`expr1 & expr2`	The type of `expr1` or `expr2`, since `expr1` and `expr2` have the same type
`^`	Left associative	Bitwise XOR	`expr1 ^ expr2`	The type of `expr1` or `expr2`, since `expr1` and `expr2` have the same type
`|`	Left associative	Bitwise OR	`expr1 | expr2`	The type of `expr1` or `expr2`, since `expr1` and `expr2` have the same type
`&&`	Left associative	Logic AND	`expr1 && expr2`	Bool
`||`	Left associative	Logic OR	`expr1 || expr2`	Bool
`??`	Right associative	coalescing	`expr1 ?? expr2`	The type of `expr2`
`|>`	Left associative	Pipeline	`expr1 |> expr2`	The type of `expr1 |> expr2` is the type of `expr2(expr1)`
`~>`		Composition	`expr1 ~> expr2`	The type of `expr1 ~> expr2` is the type of the lambda expression `{x=>expr2(expr1(x))}`
`=`	None	Assignment	`leftValue = expr`	Unit
`**=`		Compound assignment	`leftValue **= expr`	
`*=`			`leftValue *= expr`	

Operator	Associativity	Description	Usage	Expression type
/=			leftValue /= expr	
%=			leftValue %= expr	
+=			leftValue += expr	
-=			leftValue -= expr	
<<=			leftValue <<= expr	
>>=			leftValue >>= expr	
&=			leftValue &= expr	
^=			leftValue ^= expr	
|=			leftValue |= expr	
&&=			leftValue &&= expr	
||=			leftValue ||= expr	

注：? 和 .、()、{}、[] 一起使用时，是一种语法糖形式，不会严格按照它们固有的优先级和结合性进行求值，详见问号操作符。

4.29 表达式的求值顺序

表达式的求值顺序规定了计算操作数的值的顺序，显然只有包含二元操作符的表达式才存在求值顺序的概念。仓颉编程语言的默认求值顺序为：

1. 对于包含逻辑与（&&）、逻辑或（||）和 coalescing（??）的表达式，仅当操作符的右操作数的值会影响整个表达式的值时，才计算右操作数的值，否则只计算左操作数的值。因此，&&、|| 和 ?? 的求值顺序为：先计算左操作数的值，再计算右操作数的值。
2. 对于 optional chaining 表达式，其中的 ? 会将表达式分隔成若干子项，按从左到右的顺序对各子项依次求值（子项内按使用到的操作符的求值顺序进行求值）。
3. 对于其他表达式（如算术表达式、关系表达式、位运算表达式等），同样按从左往右的顺序求值。

第五章　函数

函数是一段完成特定任务的独立代码片段，可以通过函数名字来标识，这个名字可以被用来调用函数。仓颉编程语言中函数是一等公民，函数可以赋值给变量，或作为参数传递，或作为返回值返回。

5.1　函数定义

仓颉编程语言中，函数的定义遵循以下语法规则：

```
functionDefinition
        : functionModifierList? 'func'
        identifier
        typeParameters?
        functionParameters
        (':' type)?
        (genericConstraints)?
        block
        ;
```

可以总结为如下：

- 通过关键字 `func` 来定义一个函数
- `func` 前可以用函数修饰符进行修饰
- `func` 后需要带函数名
- 可选的类型形参列表，类型形参列表由 `<>` 括起，多个类型形参之间用，分隔
- 函数的参数由 `()` 括起，多个参数用，分隔，并且需要给定每个参数的参数类型。详见 [函数参数]。
- 可缺省的函数返回类型，由`:type` 表示
- 函数定义时必须有函数体，函数体是一个块（不包含函数形参）。

以下示例是一个完整的函数定义具备的所有要素，它没有使用访问修饰符 `public` 修饰表示其在包内部可访问，函数名为 `foo`，有一个 `Int64` 类型的参数 `a`，有返回类型 `Int64`，有函数体。

```
func foo(a: Int64): Int64 { a }
```

函数名不能被进行赋值，即函数名不能以表达式左值的形式在程序中出现。
如以下示例中的操作是禁止的。

```
func f(a: Int64): Int64 { a }

// f = {a: Int64 => a + 1}      // compile-time error
```

5.1.1 函数修饰符

5.1.1.1 全局函数的修饰符

全局函数可以被所有访问修饰符修饰，默认的可访问性为 internal。详细内容请参考包和模块管理章节访问修饰符。

5.1.1.2 局部函数的修饰符

局部函数无可用修饰符。

5.1.1.3 成员函数的修饰符

类成员函数可用修饰符有：public, protected, private, internal, static, open, override, redef 详见类的成员以及包和模块管理章节访问修饰符;

接口成员函数可用修饰符有：static，mut, 详见接口成员;

struct 成员函数可用修饰符有：mut，public，private，internal，static，详见struct 类型。

enum 成员函数可用修饰符有：public，private，internal，static，详见enum 类型。

如果不提供可访问修饰符，接口成员函数以外的成员函数可以在当前包及子包内被访问，接口成员函数默认是 public 语义。

5.1.2 参数

函数定义时参数列表中参数的顺序：非命名参数，命名参数（包括：不带默认值命名参数和带默认值的参数），参数列表的语法定义如下：

```
functionParameters
    : ('(' (unnamedParameterList (',' namedParameterList)? )? ')')
      | ('(' (namedParameterList)? ')')
    ;

nondefaultParameterList
    : unnamedParameter (',' unnamedParameter)* (',' namedParameter)*
    | namedParameter (',' namedParameter)*
    ;

namedParameterList
    : (namedParameter | defaultParameter) (',' (namedParameter |
    ↪  defaultParameter))*
    ;
```

```
namedParameter
    : identifier '!' ':' type
    ;

defaultParameter
    : identifier '!' ':' type '=' expression
    ;

unnamedParameterList
    : unnamedParameter (',' unnamedParameter)*
    ;

unnamedParameter
    : (identifier | '_') ':' type
    ;
```

对于非命名参数，可以使用一个 _ 代替一个函数体中不会使用到的参数。

```
func bar(a: Int64 , b!: Float64 = 1.0, s!: String) {}   // OK
func bar2(a: Int64 = 1, b!: Float64 = 1.0, s: String = "Hello") {}   // Error
func foo(a: Int64, b: Float64)(s: String = "Hello") {}  // Error

func f1(a: Int64, _: Int64): Int64 {
    return a + 1
}

func f2(_: String): Unit {
    print("Hello Cangjie")
}
```

函数参数均为不可变变量，即均有 let 修饰，在函数定义内不能对其进行赋值。

```
func foo(a: Int64, b: Float64) {
    a = 1       // Error: the parameter 'a' is immutable, and cannot be assigned.
    b = 1.0     // Error: the parameter 'b' is immutable, and cannot be assigned.
}
```

函数的参数类型不受下述“是否是命名形参”、“是否有默认值”的影响；且讨论参数类型时，一个类型与它的 alias 被视为相同的类型。

5.1.2.1 命名形参

函数定义时通过在形参名后添加 ! 来定义命名形参。

```
namedParameter
    : identifier '!' ':' type
    ;
```

- 函数定义时，命名形参后不允许有非命名形参；

```
func add1(a: Int32, b!: Int32): Int32 { a + b } // ok

func add2(a!: Int32, b: Int32): Int32 { a + b } // error
```

- 当形参被定义成命名形参后，调用这个函数时，则必须在实参值前使用参数名：前缀来指定这个实参对应的形参，否则编译报错。

```
func add(a: Int32, b!: Int32): Int32 { a + b }

add(1, b: 2) // ok
add(1, 2)    // error
```

- 如果抽象函数或 open 修饰的函数有命名形参，那么实现函数或 override 修饰的函数也需要保持同样的命名形参。

```
open class A {
  public open func f(a!: Int32): Int32 {
      return a + 1
  }
}
class B <: A {
  public override func f(a!: Int32): Int32 { // ok
      return a + 1
  }
}
class C <: A {
  public override func f(b!: Int32): Int32 { // error
      return b + 1
  }
}
```

5.1.2.2 参数的默认值

函数的参数可以有默认值，通过 = 来为参数定义默认值。当默认值被定义后，调用这个函数可以忽略这个参数，函数体会使用默认值。为了便于理解，有默认值的参数称为可选参数。

函数定义时，可选参数是一种命名形参，可选参数名后必须添加!，否则编译报错；

```
defaultParameter
    : identifier '!' ':' type '=' expression
    ;
```

如下示例，我们定义了一个 add 函数，它的参数类型列表 (Int32, Int32)。函数定义时，b 具有默认值 1。因此，当我们调用 add 函数，并且只传递一个值 3 时，3 会被赋值给 a，从而返回结果 4。

如果传入两个值 3 和 2，那么 b 的值为 2。

```
func add(a: Int32, b!: Int32 = 1): Int32 { a + b }

add(3)          // invoke add(3, 1), return 4
add(3, b: 2)    // return 5
```

- 类或接口中被 open 关键字修饰的函数不允许有可选参数。
- 操作符函数不允许有可选参数。
- 匿名函数（lambda 表达式）不允许有可选参数。
- 函数参数默认值中引入的名字从静态作用域中获得，即引入的名字为函数定义时可访问到的名字。
- 函数参数默认值中引入的名字在函数定义时可访问即可，无需和函数本身的可访问性一致。
- 函数参数和其默认值不属于该函数的函数体。
- 参数默认值在函数调用时求值，而非在函数定义时求值。
- 规定函数调用时参数求值顺序是按照定义时顺序从左到右，函数参数的默认值可以引用定义在该参数之前的形参。
- 函数调用时，通过函数名调用可以使用默认值，通过变量名调用不支持使用默认值。

```
// Compile-time error.
// func f(a: Int32, b!: Int32 = 2, c: Int32): Int32 { ... }

// OK.
func f1(a: Int32, b: Int32, c!: Int32 = 3, d!: Int32 = 4): Int32 {
    a + b + c + d
}

func test() {
    f1(1, 2)      // 10,  f1(1, 2, 3, 4)
    f1(1, 2, c: 5)        // 12,  f1(1, 2, 5, 4)
}

/* The names introduced in the default value, does not need to have the same or
 ↪  least restrictive  accessibility of the function. */
var x = 10
var y = 10
func g() {}
public func f2(a!: Int64 = x * 2 + y, b!: ()->Unit = g) {}  // OK.
```

```
class AAA {
    static private var x = 10

    func f(a!: Int64 = x) { // OK, public method can use private static field.
        print("${a}")
        x = x + 10
        print("${x}")
    }
}

/*
When a function is called, the name in the function declaration can use the
 ↪ default value. When a function is called by using a variable name, arguments
 ↪ cannot be optional.
*/
func f1(): (Int64) -> Unit { g1 }

func g1(a!: Int64 = 42) {
    print("g1: ${a}")
}

let gg1 = f1()
let x = gg1()    // Error, cannot ommit the argument.
let gg3 = g1
let a = gg3()    // Error, cannot ommit the argument.
```

由于函数的形参和其默认值不属于该函数的函数体，所以下面例子中的 return 表达式缺少包围它的函数体——它既不属于外层函数 f（因为内层函数定义 g 已经开始），也不在内层函数 f 的函数体中：

```
func f() {
    func g(x! :Int64 = return) { // Error: return must be used inside a function
     ↪ body
        0
    }
    1
}
```

5.1.3 函数体

函数体由一个 block 组成。

5.1.3.1 局部变量

在仓颉编程语言中，允许在函数体内定义变量，将其称为局部变量。
变量可以用 var 修饰为可变变量或 let 修饰为不可变变量。

```
func foo(): Unit {
    var a = 1
    let b = 1
}
```

5.1.3.2 嵌套函数

在仓颉编程语言中，允许在函数体内定义函数，将其称为嵌套函数。嵌套函数中可以捕获外部函数中定义的变量或其他嵌套函数。嵌套函数支持递归。

```
func foo(): Unit {
    func add(a: Int32, b: Int32) { a + b }

    let c = 1

    func nest(): Unit {
        print("${c}")           // 1
        var b = add(1, 2)   // b = 3
    }
}
```

5.1.4 函数的返回类型

函数的返回类型有以下情况:

- 任何类型（见第 2 章类型）
- 返回值为元组的函数: 可以使用元组类型作为函数的返回类型，将多个值作为一个复合返回值返回。如以下例子，它的返回是一个元组 (a, b)，返回类型是 (Int32, Int32)

  ```
  func returnAB(a: Int32, b: Int32): (Int32, Int32) { (a, b) }
  ```

- 函数类型作为返回类型: 可以使用函数类型作为另一个函数的返回类型，如下示例。在: 后紧跟的是 add 函数的类型 (Int32, Int32) -> Int32。

  ```
  func returnAdd(a: Int32, b: Int32): (Int32, Int32) -> Int32 {
      return {a, b => a + b}    // Return a lambda expression.
  }
  ```

如果指定函数的返回类型，则在函数定义的参数列表后使用 :Type 指定，此时要求函数体的类型、函数体中所有 return e 表达式中 e 的类型是标注的类型（Type）的子类型，否则编译报错。

```
class A {}
class B <: A {}
// Return is not written.
func add(a: Int32, b: Int32): B {
    var c = a + b
    if (c > 100) {
        return B()
    } else {
        return A() // Compilation error since A is not a subtype of B.
    }
}
```

特别地，指定返回类型为 **Unit** 时（如 `func f(x:Bool):Unit { x }`），则编译器会在函数体中所有可能返回的地方自动插入表达式 `return ()`，使得函数的返回类型总是为 `Unit` 。示例如下：

```
func Column(c: (Data) -> Unit): Unit { 2 } // return () is automatically inserted
func Column(c: (Data) -> Unit) { 2 }   // return type is Int64
```

如果不指定函数的返回类型，则编译器会根据函数体的类型以及函数体中的所有 `return` 表达式来共同推导出函数的返回类型。此过程不是完备的，如遇到（互）递归函数而无法推导它们的返回类型时，编译报错。（注意：不能为没有函数体的函数推导其返回类型。）

函数的返回类型推导规则如下：函数体是表达式和声明的序列，我们将序列的最后一项的类型记为 `T0`（若块的最后一项是表达式，则为表达式的类型；若最后一项为声明，则 `T0 = Unit`），再将函数体中所有 `return e` （包括所有子表达式中的 `return e`）表达式中 `e` 的类型记为 `T1 ... Tn`，则函数的返回类型是 `T0, T1, ..., Tn` 的最小公共父类型。如果不存在最小公共父类型，则产生一个编译错误。示例如下：

```
open class Base {}
class Child <: Base {}

func f(a: Rune) {
    if (false) {
        return Base()
    }
    return Child()
}
```

- 函数体的类型是块的最后一项的类型，即 `return Child()` 的类型，其类型为 `Nothing`。
- 第一个 `return e` 表达式 `return Base()` 中 `e` 的类型是 `Base`。
- 第二个 `return e` 表达式 `return Child()` 中 `e` 的类型为 `Child`。
- 由于 `Nothing`, `Base`, `Child` 三者的最小公共父类型是 `Base`，所以该函数的返回类型为 `Base`。

注：函数的返回值具有 `let` 修饰的语义。

5.1.5 函数声明

仓颉编程语言中，函数声明和函数定义的区别是，前者没有函数体。函数声明的语法如下：

```
functionDeclaration
        : functionModifierList? 'func'
        identifier
        typeParameters?
        functionParameters
        (':' type)?
        genericConstraints?
        ;
```

函数声明可以出现在抽象类，接口中。

5.1.6 函数重定义

对于非泛型函数，在同一个作用域中，参数类型完全相同的同名函数被视为重定义，将产生一个编译错误。以下几种情况要格外注意：

- 同名函数即使返回类型不同也构成重定义。
- 同名的泛型与非泛型函数永不构成重定义。
- 在继承时，对于子类中的一个与父类同名且参数类型完全相同的函数，若其返回类型是父类中的版本的子类型，则构成覆盖，也不构成重定义。（这是因为子类与父类作用域不同。）

对于两个泛型函数，如果重命名一个函数的泛型形参后，其非泛型部分与另一个函数的非泛型部分构成重定义，则这两个泛型函数构成重定义。举例如下：

1. 下面这两个泛型函数构成重定义，因为存在一种 `[T1 |-> T2]` 的重命名（作用到第一个函数上），使得两个函数的非泛型部分构成重定义。

   ```
   func f<T1>(a: T1) {}
   func f<T2>(b: T2) {}
   ```

2. 下面这两个泛型函数不构成重定义，因为找不到上述的一种重命名。

   ```
   func f<X,Y>(a:X, b:Y) {}
   func f<Y,X>(a:X, b:Y) {}
   ```

3. 下面的这两个泛型函数构成重定义，因为存在一种 `[X |-> Y, Y |-> X]` 的重命名使得两个函数的非泛型部分构成重定义。

   ```
   func f<X,Y>(a:X, b:Y) {}
   func f<Y,X>(a:Y, b:X) {}
   ```

5.2 函数类型

函数类型由函数的参数类型和返回类型组成，其中参数类型与返回类型之间用-> 分隔。

```
functionType:
    : '(' (type (, type)*)? ')' '->' type
```

以下是一些示例:

- 示例 1: 没有参数、返回类型为 Unit

```
func hello(): Unit { print("Hello!") }

// function type: () -> Unit
```

- 示例 2: 参数类型为 Int32，返回类型为 Unit

```
func display(a: Int32): Unit { print("${a}") }

// function type: (Int32) -> Unit
```

- 示例 3: 两个参数类型均为 Int32，返回类型为 Int32

```
func add(a: Int32, b: Int32): Int32 { a + b }

// function type: (Int32, Int32) -> Int32
```

- 示例 4: 参数类型为 (Int32, Int32) -> Int32, Int32 和 Int32，返回类型为 Unit

```
func printAdd(add: (Int32, Int32) -> Int32, a: Int32, b: Int32): Unit {
    print("${add(a, b)}")
}
// function type: ((Int32, Int32) -> Int32, Int32, Int32) -> Unit
```

- 示例 5: 两个参数类型均为 Int32，返回类型为函数类型 (Int32, Int32) -> Int32

```
func returnAdd(a: Int32, b: Int32): (Int32, Int32) -> Int32 {
  {a, b => a + b}
}

// function type: (Int32, Int32) -> (Int32, Int32) -> Int32
```

- 示例 6: 两个参数类型均为 Int32, 返回为一个元组类型为: (Int32, Int32)。

```
func returnAB(a: Int32, b: Int32): (Int32, Int32) { (a, b) }

// function type: (Int32, Int32) -> (Int32, Int32)
```

5.3 函数调用

有关函数调用表达式的语法，请参考函数调用表达式。

5.3.1 命名实参

命名实参是指在函数调用时，在实参值前使用 形参名 ： 前缀来指定这个实参对应的形参。只有在函数定义时使用! 定义的命名形参，才可以在调用时使用命名实参的语法。

函数调用时，所有的命名形参均必须使用命名实参来传参，否则报错。

在函数调用表达式中，命名实参后不允许出现非命名实参。使用命名实参指定实参值时，可以不需要和形参列表的顺序保持一致。

```
func add(a!: Int32, b!: Int32): Int32 {
    a + b
}

var sum1 = add(1, 2)        // error
var sum2 = add(a: 1, b: 2)  // OK, 3
var sum3 = add(b: 2, a: 1)  // OK, 3
```

5.3.2 函数调用类型检查

本节介绍的是，给定一个调用表达式，对调用表达式所需要进行的类型检查。如果被调用的函数涉及重载，则需要根据函数重载的规则进行类型检查和重载决议。

1、如果函数调用表达式中指定了类型参数，只有类型实参的个数与类型形参的个数相同才可能通过类型检查，即假设函数调用表达式为：`f<T1, ..., Tm>(A1, ..., An)`，其中给定了 m 个类型实参，则函数类型形参数量须为 m；

```
open class Base {}
class Sub <: Base {}
func f<X, Y>(a: X, b: Y) {}  // f1
func f<X>(a: Base, b: X) {}  // f2
f<Base>(Base(), Sub()) // f2 may pass the type checking
```

2、根据调用表达式中的实参和调用表达式所在的类型检查上下文中指定的返回类型 R 对函数进行类型检查。

假设函数定义为：

$$f_i < T_{i1}, ..., T_{ip} > (A_{i1}, ..., A_{ik}) : R_i \ \ where \ \ C_{i1}, ..., C_{iq_i}$$

1) 如果调用表达式带了类型实参：`fi<T1, ..., Tp>(A1, ..., Ak)`，那么对于函数 `fi` 的类型检查规则如下：

a) 类型实参约束检查：类型实参 `<T1, ..., Tp>` 需要满足函数 `fi` 的类型约束。

$$\sigma = [T_1 \mapsto T_{i1}, ..., T_p \mapsto T_{ip}]$$

$$\Delta \vdash \sigma \ \ solves \ \ C_{i1}, ..., C_{iq_i}$$

b) 参数类型检查：将类型实参代入函数 `fi` 的形参后，满足实参类型 `(A1, ..., Ak)` 是类型实参代入形参后类型的子类型。

$$\sigma = [T_1 \mapsto T_{i1}, ..., T_p \mapsto T_{ip}]$$

$$\Delta \vdash (A1, ..., Ak) <: \sigma(A_{i1}, ..., A_{ik})$$

c) 返回类型检查：如果调用表达式的上下文对其有明确类型要求 `R`，则需要根据返回类型进行类型检查，将类型实参代入函数 `fi` 的返回类型 `Ri` 后，满足类型实参代入后的返回类型是 `R` 的子类型。

$$\sigma = [T_1 \mapsto T_{i1}, ..., T_p \mapsto T_{ip}]$$

$$\Delta \vdash \sigma R_i <: R$$

2）如果调用表达式不带类型实参：`f(A1, ..., An)`，那么对于函数 `fi` 的类型检查规则如下：

a) 如果`fi`是非泛型函数，则按如下规则进行类型检查：

i. 参数类型检查：实参类型 `(A1, ..., Ak)` 是形参类型的子类型。

$$\Delta \vdash (A1, ..., Ak) <: (A_{i1}, ..., A_{ik})$$

ii. 返回类型检查：如果调用表达式的上下文对其有明确类型要求 `R`，则检查函数 `fi` 的返回类型 `Ri` 是 `R` 的子类型。

$$\Delta \vdash R_i <: R$$

```
open class Base {}
class Sub <: Base {}
func f(a: Sub) {1}  // f1
func f(a: Base) {Base()}  // f2
let x: Base = f(Sub())  // f2 can pass the type checking
```

b) 如果`fi`是泛型函数，则按如下规则进行类型检查 ：

i. 参数类型检查：存在代换使得实参类型`(A1, ..., Ak)`是形参类型代换后的类型的子类型。

$$\sigma = [T_1 \mapsto T_{i1}, ..., T_p \mapsto T_{ip}]$$
$$\Delta \vdash (A1, ..., Ak) <: \sigma(A_{i1}, ..., A_{ik})$$

ii. 返回类型检查：如果调用表达式的上下文对其有明确类型要求 `R`，则需要根据返回类型进行类型检查，将 i) 中的代换代入函数 `fi` 的返回类型 `Ri` 后，满足代换后的返回类型是 `R` 的子类型。

$$\sigma = [T_1 \mapsto T_{i1}, ..., T_p \mapsto T_{ip}]$$
$$\Delta \vdash \sigma R_i <: R$$

需要注意的是：

1. 如果函数有缺省值，在类型检查时会补齐缺省值之后再进行类型检查；
2. 如果函数有命名参数，命名参数的顺序可能会和形参顺序不一致，在类型检查时，命名实参要与名字匹配的命名形参对应。

5.3.3 尾随 Lambda

当函数最后一个形参是函数类型，并且函数调用对应的实参是 lambda 时，我们可以使用尾随 lambda 语法，将 lambda 放在函数调用的尾部，括号外面。

```
func f(a: Int64, fn: (Int64)->Int64) { fn(a) }

f(1, { i => i * i }) // normal function call
f(1) { i => i * i } // trailing lambda

func g(a!: Int64, fn!: (Int64)->Int64) { fn(a) }

g(a: 1, fn: { i => i * i }) // normal function call
g(a: 1) { i => i * i } // trailing lambda
```

当函数调用有且只有一个 lambda 实参时，我们还可以省略 `()`，只写 lambda。

```
func f(fn: (Int64)->Int64) { fn(1) }

f{ i => i * i }
```

如果尾随 lambda 不包含形参，`=>` 可以省略。

```
func f(fn: ()->Int64) { fn() }

f{ i * i }
```

需要注意的是，尾随 lambda 语法只能用在具有 函数名/变量名的函数调用上，并且尾随 lambda 的 lambda 表达式只会解释为 函数名/变量名对应的函数的参数。这意味着以下两种调用例子是不合法的：

```
func f(): (()->Unit)->Unit {
    {a => }
}

f() {} // error, the lambda expression is not argument for f.

func g(a: ()->Unit) {}

if (true) { g } else { g } () {} // error, illegal trailing lambda syntax
```

必须先将以上的表达式赋值给变量，使用变量名调用时才可以使用尾随 lambda 语法。如下面的代码所示:

```
let f2 = f()
f2 {} // ok
let g2 = if (true) { g } else { g }
g2() {} // ok
```

普通函数调用和构造函数调用都可以使用这个语法，包含 this() 和 super()。

```
this(1, { i => i * i } )
this(1) { i => i * i }

super(1, { i => i * i } )
super(1) { i => i * i }
```

5.3.4 变长参数

变长参数是一种特殊的函数调用语法糖，当形参最后一个非命名参数是 Array 类型时，实参中对应位置可以直接传入参数序列代替 Array 字面量。

1. 变长参数没有特殊的声明语法，只要求函数声明处最后一个非命名参数是 Array 类型。
2. 变长参数在函数调用时可以使用普通参数列表的形式逐个传入 Array 的多个元素。
3. 非命名参数中，只有最后一个位置的参数可以使用变长参数。命名参数不能使用这个语法糖。
4. 变长参数对全局函数、静态成员函数、实例成员函数、局部函数、构造函数、函数变量、lambda、函数调用操作符重载、索引操作符重载的调用都适用，不支持其它操作符重载、compose、pipeline 这几种调用方式。
5. 变长参数的个数可以是 0 个或以上。
6. 变长参数只有在函数重载所有情况都不匹配的情况下，才判断是否可以应用语法糖，优先级最低。

```
func f1(arr: Array<Int64>) {}
func f2(a: Int64, arr: Array<Int64>) {}
func f3(arr: Array<Int64>, a: Int64) {}
func f4(arr1!: Array<Int64>, a!: Int64, arr2!: Array<Int64>) {}
```

```
func g() {
    let li = [1, 2, 3]
    f1(li)
    f1(1, 2, 3) // using variable length argument
    f1() // using variable length argument
    f2(4, li)
    f2(4, 1, 2, 3) // using variable length argument

    f3(1, 2, 3) // error, Array is not the last parameter
    f4(arr1: 1,2,3, a: 2, arr2: 1,2,3) // error, named parameters cannot use
    ↪ variable length argument
}
```

函数重载决议总是会优先考虑不使用变长参数就能匹配的函数，只有在所有函数都不能匹配，才尝试使用变长参数解析。

当编译器无法决议时会报错。

```
open class A {
    func f(v: Int64): Unit { // f1
    }
}

class B <: A {
    func f(v: Array<Int64>): Unit { // f2
    }
}

func p1() {
    let x = B()
    x.f(1) // call the f1
}

func g<T>(arg: T): Unit { // g1
}

func g(arg: Array<Int64>): Unit { // g2
}

func p2() {
    g(1) // call the g1
}
```

```
func h(arg: Any): Unit { // h1
}
func h(arg: Array<Int64>): Unit { // h2
}

func p3() {
    h(1) // call the h1
}
```

5.4 函数作用域

在仓颉编程语言中，函数可以在源程序顶层定义或者在函数内部定义：

- 全局函数

  在源程序顶层定义函数称为全局函数，它的作用域是全局的。如下示例，函数 globalFunction 是全局函数。它的作用域是全局作用域。

  ```
  func globalFunction() {}
  ```

- 嵌套函数

  在函数体内部定义的函数成为嵌套函数，它的作用域是局部的，详见作用域。如下示例，函数 nested-Function 是嵌套函数，它的作用域是从定义之后开始，到 globalFunction 函数体结束。

  ```
  func globalFunction() {
    func nestedFunction() {}
  }
  ```

- 成员函数

  在类型定义中可以声明或定义成员函数。成员函数的作用域是整个类型及其扩展。

  ```
  interface Myinterface {
    func foo(): Unit
    static func bar(): Unit
  }

  class MyClass {
    func foo() {}
    static func bar() {}
  }
  ```

- 扩展成员函数

  在扩展中可以声明额外的成员函数。它的作用域是被扩展类型的所有扩展，同时受访问修饰符限制。

```
extend MyType {
  func foo(): Unit {}
}
```

5.5 Lambda 表达式

一个 Lambda 表达式是一个匿名的函数。
Lambda 表达式的语法如下:

```
lambdaExpression
        : '{' NL* lambdaParameters? '=>' NL* expressionOrDeclarations '}'
        ;

lambdaParameters
    : lambdaParameter (',' lambdaParameter)* ','?
    ;

lambdaParameter
    : (identifier | '_') (':' type)?
    ;
```

lambda 表达式有两种形式，一种是有形参的 `{a: Int64 => e1; e2 }`，另一种是无形参的 `{ => e1; e2 }` (e1 和 e2 是表达式或声明序列)。

```
let f1: (Int64, Int64)->Int64 = {a: Int64, b: Int64 => a + b}

var f2: () -> Int32 = { => 123 }
```

对于有形参的 lambda 表达式，可以使用一个 _ 代替一个形参，_ 代表在 lambda 的函数体中不会使用到的参数。

```
let f2 = {n: Int64, _: Int64 => return n * n}
let f3: (Int32, Int32) -> Int32 = {n, _ => return n * n}
let f4: (String) -> String = {_ => return "Hello"}
```

Lambda 表达式中不支持声明返回类型。

- 若上下文明确指定了 lambda 表达式的返回类型，则其返回类型为上下文指定的类型。如果指定的返回类型是 Unit，则仓颉编译器还会在 lambda 表达式的函数体中所有可能返回的地方插入 `return ()`，使其总是返回 Unit 类型。指定了 Unit 返回类型的示例如下:

```
func column(c: (Data) -> Unit) {...}
func row(r: (Data) -> Unit) {...}
```

```
func build():Unit {
    column { _ =>
        row { _ =>
            buildDetail()
            buildCalendar()
        } // OK. Well typed since 'return' is inserted.
        width(750)
        height(700)
        backgroundColor("#ff41444b")
    }  // OK. Well typed since 'return' is inserted.
}
```

- 若不存在这样的上下文，则 => 右侧的表达式的类型会被视为 lambda 表达式的返回类型。同函数一样，若无法推导出返回类型，则会编译报错。

Lambda 表达式中参数的类型标注可缺省，编译器会尝试从上下文进行类型推断，当编译器无法推断出类型时会编译报错。

```
var sum1: (Int32, Int32) -> Int32 = {a, b => a + b}

var sum2: (Int32, Int32) -> Int32 = {a: Int32, b => a + b}

var display = { => print("Hello") }

var a = { => return 1 }
```

=> 右侧的内容与普通函数体的规则一样，同样可以省略 return。若 => 的右侧为空，返回值为 ()。

```
sum1 = {a, b => a + b}

sum2 = {a, b => return a + b}       // Same as that in the previous line.
```

Lambda 表达式支持原地调用，例如：

```
let r1 = {a: Int64, b: Int64 => a + b}(1, 2) // r1 = 3
let r2 = { => 123 }()                        // r2 = 123
```

5.6 闭包

闭包指的是自包含的函数或 lambda，闭包可以从定义它的静态作用域中捕获变量，即使对闭包调用不在定义的作用域，仍可访问其捕获的变量。变量捕获发生在闭包定义时。

不是所有闭包内的变量访问都称为捕获，以下情形的变量访问不是捕获：

- 对定义在本函数或本 lambda 内的局部变量的访问；
- 对本函数或本 lambda 的形参的访问；

- 全局变量和静态成员变量在函数或 lambda 中的访问;
- 实例成员变量在实例成员函数中的访问。由于实例成员函数将 `this` 作为参数传入，在实例成员函数内通过 `this` 访问所有实例成员变量。

关于变量的捕获，可以分为以下两种情形:

- 捕获 `let` 声明的变量：在闭包中不能修改这些变量的值。如果捕获的变量是引用类型，可修改其可变实例成员变量的值。仅捕获了 `let` 声明的局部变量的函数或 lambda 可以作为一等公民使用，即可以赋值给变量，可以作为实参或返回值使用，可以作为表达式使用。
- 捕获 `var` 声明的变量：捕获了可变变量的函数或 lambda 只能被调用，不能作为一等公民使用，包括不能赋值给变量，不能作为实参或返回值使用，不能作为表达式使用。

  需要注意的是，捕获具有传递性，如果一个函数 `f` 调用了捕获 `var` 变量的函数 `g`，且存在 `g` 捕获的 `var` 变量不在函数 `f` 内定义，那么函数 `f` 同样捕获了 `var` 变量，此时，`f` 也不能作为一等公民使用。

以下示例中，`h` 仅捕获了 `let` 声明的变量 `y`，`h` 可以作为一等公民使用。

```
func f(){
    let y = 2
    func h() {
        print(y)  // OK, captured an immutable variable.
    }
    let d = h    // OK, h can be assigned to variable
    return h   // OK, h can be a return value
}
```

以下示例中，`g` 捕获了 `var` 声明的变量 `x`，`g` 不可以作为一等公民使用，仅能被调用。

```
func f() {
    var x = 1

    func g() {
        print(x)  // OK, captured a mutable variable.
    }
    let b = g  // Error, g cannot be assigned to a variable

    g  // Error, g cannot be used as an expression
    g()  // OK, g can be invoked

    // Lambda captured a mutable variable, cannot be assigned to a variable
    let e = { => print("${x}") }  // Error

   let i = { => x*x }()  // OK, lambda captured a mutable variable, can be invoked.
```

```
    return g  // Error, g cannot be used as a return value.
}
```

以下示例中，g 捕获了 var 声明的变量 x，f 调用了 g，且 g 捕获的 x 不在 f 内定义，f 同样不能作为一等公民使用：

```
func h(){
    var x = 1

    func g() { x }   // captured a mutable variable

    func f() {
        g()      // invoked g
    }
    return f // error
}
```

以下示例中，g 捕获了 var 声明的变量 x，f 调用了 g。但 g 捕获的 x 在 f 内定义，f 没有捕获其它 var 声明的变量。因此，f 仍作为一等公民使用：

```
func h(){
    func f() {
        var x = 1
        func g() { x }   // captured a mutable variable

        g()
    }
    return f // ok
}
```

访问了 var 修饰的全局变量、静态成员变量、实例成员变量的函数或 lambda 仍可作为一等公民使用。

```
class C {
    static var a: Int32 = 0
    static func foo() {
        a++       // OK
        return a
    }
}

var globalV1 = 0

func countGlobalV1() {
    globalV1++
```

```
    C.a = 99
    let g = C.foo  // OK
}

func g(){
    let f = countGlobalV1 // OK
    f()
}
```

捕获的变量必须满足以下规则：

- 被捕获的变量必须在闭包定义时可见，否则编译报错；
- 变量被捕获时必须已经完成初始化，否则编译报错；
- 如果函数外有变量，同时函数内有同名的局部变量，函数内的闭包因局部变量的作用域未开始而捕获了函数外的变量时，为避免用户误用，报 warning；

```
// 1. The captured variable must be defined before the closure.
let x = 4
func f() {
    print("${x}")    // Print 4.
    let x = 99
    func f1() {
        print("${x}")
    }
    let f2 = { =>
        print("${x}")
    }
    f1()          // Print 99.
    f2()          // Print 99.
}

// 2. The variable must be initialized before being captured.
let x = 4
func f() {
    print("${x}")    // Print 4.
    let x: Int64
    func f1() {
        print("${x}")    // Error: x is not initialized yet.
    }
    x = 99
    f1()
}
```

```
// 3. If there is a  local variable in a block, closures capture variables of the
 ↪  same name in the outer scope will report a warning.
let x = 4
func f() {
    print("${x}")    // Print 4.
    func f1() {
        print("${x}")   // warning
    }
    let f2 = { =>
        print("${x}")   // warning
    }
    let x = 99
    f1()   // print 4
    f2()   // print 4
}
```

5.7 函数重载

5.7.1 函数重载定义

在仓颉编程语言中，一个作用域中可见的同一个函数名对应的函数定义不构成重定义时便构成重载。函数存在重载时，在进行函数调用时需要根据函数调用表达式中的实参的类型和上下文信息明确是哪一个函数定义被使用。

被重载的多个函数必须遵循以下规则：

- 函数名必须相同，且满足以下情况之一：
  - 通过 func 关键字引入的函数名。
  - 在一个 class 或 struct 内定义的构造函数，包括主构造函数和 init 构造函数。
- 参数类型必须不同，即满足以下情况之一：
  - 函数参数数量不同。
  - 函数参数数量相同，但是对应位置的参数类型不同。
- 必须在同一个作用域中可见

注意，参数类型不受“是否是命名形参”、“是否有默认值”影响；且讨论参数类型时，一个类型与它的 alias 被视为相同的类型。例如下面例子中的四个函数的参数类型是完全相同的：

```
type Boolean = Bool
func f(a : Bool) {}
func f(a! : Bool) {}
func f(a! : Bool = false) {}
func f(a! : Boolean) {}
```

示例一：定义源文件顶层的 2 个函数的参数类型不同，f 构成了重载。

```
// f overloading
func f() {...}
func f(a: Int32) {...}
```

示例二：接口 I、类 C1、C2 中的函数 f1 构成了重载。

```
interface I {
    func f1() {...}
}

open class C1 {
  func f1(a: Int32) {...}
}

class C2 <: C1 & I {
    // f1 overloading
    func f1(a: Int32, b: String) {...}
}
```

示例三：类型内的构造函数之间构成重载。

```
class C {
    var name: String = "abc"

    // constructor overloading
    init(){
        print(name)
    }

    init(name: String){
        this.name = name
    }
}
```

示例四：如下示例，函数参数数量相同，相同位置的类型相同，仅参数类型中包含的类型变元的约束不同，不构成重载

```
interface I1{}
interface I2{}

func f<T>(a: T) where T <: I1 {}
func f<T>(a: T) where T <: I2 {} // Error, not overloading
```

5.8 mut 函数

mut 函数是一种特殊的实例成员函数。在 struct 类型的 mut 成员函数中，可以通过 this 修改成员变量，而在 struct 类型的非 mut 成员函数中，不可以通过 this 修改成员变量。

5.8.1 定义

mut 函数使用 mut 关键字修饰，只允许在 interface、struct 和 struct 扩展中定义，并且只能作用于实例成员函数（不支持静态成员函数）。

```
struct A {
    mut func f(): Unit {} // ok
    mut static func g(): Unit {} // error
    mut operator func +(rhs: A): A { // ok
        return A()
    }
}

extend A {
    mut func h(): Unit {} // ok
}

class B {
    mut func f(): Unit {} // error
}

interface I {
    mut func f(): Unit // ok
}
```

在 mut 函数中可以对 struct 实例的字段进行赋值，这些赋值会修改实例并立刻生效。与实例成员函数相同，this 不是必须的，可以由编译器推断。

```
struct Foo {
    var i = 0
    mut func f() {
        this.i += 1 // ok
        i += 1 // ok
    }
}

main() {
    var a = Foo()
    print(a.i) // 0
```

```
    a.f()
    print(a.i) // 2
    a.f()
    print(a.i) // 4
    return 0
}
```

mut 函数中的 this 具有额外的限制：

- 不能被捕获（意味着当前实例的字段也不能被捕获）；
- 不能作为表达式。

```
struct Foo {
    var i = 0
    mut func f(): Foo {
        let f1 = { => this } // error
        let f2 = { => this.i = 2 } // error
        let f3 = { => this.i } // error
        let f4 = { => i } // error
        return this //  error
    }
}
```

5.8.2 接口中的 mut 函数

struct 类型在实现 interface 的函数时必须保持一样的 mut 修饰。struct 以外的类型实现 interface 的函数时禁止使用 mut 修饰。

```
interface I {
    mut func f1(): Unit
    func f2(): Unit
}
struct A <: I {
    public mut func f1(): Unit {} // ok
    public func f2(): Unit {} // ok
}
struct B <: I {
    public func f1(): Unit {} // error
    public mut func f2(): Unit {} // error
}
class C <: I {
    public func f1(): Unit {} // ok
    public func f2(): Unit {} // ok
}
```

需要注意的是，当 struct 的实例赋值给 interface 类型时是拷贝语义，因此 interface 的 mut 函数并不能修改原本 struct 实例的值。

```
interface I {
    mut func f(): Unit
}
struct Foo <: I {
    var v = 0
    public mut func f(): Unit {
        v += 1
    }
}
main() {
    var a = Foo()
    var b: I = a
    b.f()
    print(a.v) // 0
    return 0
}
```

5.8.3 访问规则

如果一个变量使用 let 声明，并且类型可能是 struct（包含静态类型是 struct 类型，或者类型变元可能是 struct 类型），那么这个变量不能访问该类型使用 mut 修饰的函数。其它情况均允许访问。

```
interface I {
    mut func f(): Unit
}
struct Foo <: I {
    var i = 0
    public mut func f(): Unit {
        i += 1
    }
}
class Bar <: I {
    var i = 0
    public func f(): Unit {
        i += 1
    }
}
main() {
    let a = Foo()
    a.f() // error
```

```
    var b = Foo()
    b.f() // ok
    let c: I = Foo()
    c.f() // ok
    return 0
}
func g1<T>(v: T): Unit where T <: I {
    v.f() // error
}
func g2<T>(v: T): Unit where T <: Bar & I {
    v.f() // ok
}
```

如果一个变量的类型可能是 struct（包含静态类型是 struct 类型，或者类型变元可能是 struct 类型），那么这个变量不能将该类型使用 mut 修饰的函数被作为高阶函数使用，只能调用这些 mut 函数。

```
interface I {
    mut func f(): Unit
}
struct Foo <: I {
    var i = 0
    public mut func f(): Unit {
        i += 1
    }
}
class Bar <: I {
    var i = 0
    public func f(): Unit {
        i += 1
    }
}
main() {
    var a = Foo()
    var fn = a.f // error
    var b: I = Foo()
    fn = b.f // ok
    return 0
}
func g1<T>(v: T): Unit where T <: I {
    let fn = v.f // error
}
func g2<T>(v: T): Unit where T <: Bar & I {
    let fn = v.f // ok
```

```
}
```

非 mut 的实例成员函数（包括 lambda 表达式）不能访问 this 的 mut 函数，反之可以。

```
struct Foo {
    var i = 0
    mut func f(): Unit {
        i += 1
        g() // ok
    }
    func g(): Unit {
        f() // error
    }
}

interface I {
    mut func f(): Unit {
        g() // ok
    }
    func g(): Unit {
        f() // error
    }
}
```

第六章　类和接口

6.1　类

6.1.1　类的定义

在仓颉编程语言中，通过关键字 class 定义一个类，类定义包含以下几个部分：

- 可选的修饰符
- class 关键字
- 类名，必须是合法的 identifier
- 可选的类型参数
- 可选的指定父类或者父接口（写在 <: 后面，使用 & 分隔），如果存在父类，父类必须写在第一个，否则编译报错；
- 可选的泛型约束
- 类体

类只能定义在源文件顶层，类定义的语法如下：

```
classDefinition
    : classModifierList? 'class' identifier
    typeParameters?
    ('<:' superClassOrInterfaces)?
    genericConstraints?
    classBody
    ;
superClassOrInterfaces
    : classType ('&' superInterfaces)?
    | superInterfaces
    ;
```

类定义的例子如下：

```
interface I1<X> {}
interface I2 {}
open class A{}
```

```
// The following class B inherits class A and implements interface I2.
open class B <: A & I2 {}

/* The following class C declares 1 type parameters U. It inherits B and implements
 ↪ interfaces I1<U> and I2. Also, its type parameters have constraints U <: A */
class C<U> <: B & I1<U> & I2 where U <: A {}
```

6.1.1.1 类的修饰符

在此小节中，我们将介绍定义类可以使用的修饰符。

访问修饰符 类可以被所有访问修饰符修饰，默认的可访问性为 internal。详细内容请参考包和模块管理章节访问修饰符。

继承性修饰符

- open：类用 open 修饰，表示允许其他类从这个类继承。

```
/* The following class is declared with modifier `open`, indicating that it can
 ↪ be inherited. */
open class C1 {
    func foo(): Unit {
        return
    }
}

class C2 <: C1 {}
```

需要注意的是：

没有用 open 修饰的非抽象类不能被任何类继承。

- sealed：修饰符表示只能在 class 定义所在的包内继承该 class。
  - sealed 已经蕴含了 public 的语义，所以定义 sealed class 时的 public 修饰符是可选的。
  - sealed 的子类可以不是 sealed 类，仍可被 open/sealed 修饰，或不使用任何继承性修饰符。若 sealed 类的子类同时被 public 和 open 修饰，则其子类可在包外被继承；
  - sealed 的子类可以不被 public 修饰。

```
// package A
package A
public sealed class C1 {}   // OK
sealed class C2 {}          // OK, 'public' is optional when 'sealed' is used
```

```
class S1 <: C1 {}  // OK
public open class S2 <: C1 {}   // OK
public sealed class S3 <: C1 {}  // OK
open class S4 <: C1 {}   // OK

// package B
package B
import A.*

class SS1 <: S2 {}  // OK
class SS2 <: S3 {}  // Error, S3 is sealed class, cannot be inherited here.
```

需要注意的是：

没有用 open 或 sealed 修饰的非抽象类不能被任何类继承。

抽象类修饰符

- abstract：该类属于抽象类，与普通类不同的是，在抽象类中除了可以定义普通的函数，还允许声明抽象函数，只有用该修饰符修饰的类才为抽象类。如果一个函数没有函数体，我们称其为抽象函数。

  需要注意的是：

  - 抽象类 abstract 修饰符已经包含了可被继承的语义，因此抽象类定义时的 open 修饰符是可选的，也可以使用 sealed 修饰符修饰抽象类，表示该抽象类只能在本包被继承；
  - 抽象类中禁止定义 private 的抽象函数；

  - 不能为抽象类创建实例；
  - 允许抽象类的抽象子类不实现父类中的抽象函数；
  - 抽象类的非抽象子类必须实现父类中的所有抽象函数。

6.1.1.2 类继承

类只支持单继承。类通过 <: superClass 的方式来指定当前类的直接父类（父类是某个已经定义了的类）。

以下示例展示了类 C2 继承类 C1 的语法：

```
open class C1 {}
class C2 <: C1 {}
```

父类可以是一个泛型类，只需要在继承时提供合法的类型即可。例如：非泛型类 C2 与泛型类 C3 继承泛型类 C1：

```
open class C1<T> {}
class C2 <: C1<Int32> {}
class C3<U> <: C1<U> {}
```

所有类都有一个父类，对于没有定义父类的类，默认其父类为 Object。Object 例外，没有父类。

```
class Empty {}  //  Inherit from Object implicitly:  class Empty <: Object {}
```

类仅支持单继承。以下示例中的语法将引起编译错误。

```
open class C1 {}
open class C2 {}
class C3 <: C1 & C2 {}      // Error: Multiple inheritance is not supported.
```

当一个类继承另一个类时，将被继承的类称为父类，将产生继承行为的类称为子类。

```
open class C1 {}            // C1 is superclass
class C2 <: C1 {}           // C2 is subclass
```

子类将继承父类的所有成员，私有成员和构造函数除外。

子类可以直接访问父类的成员，但是在覆盖时，将不能直接通过名字来访问父类被覆盖的实例成员，这时可以通过 super 来指定（super 指向的是当前类对象的直接父类）或创建对象并通过对象来访问。

6.1.1.3 实现接口

类支持实现一个或多个接口，通过 <: I1 & I2 & ... & Ik 的方式声明当前类想要实现的接口，多个接口之间用 & 分隔。如果当前类也指定了父类，则接口需要出现在父类后面。例如：

```
interface I1 {}
interface I2 {}

// Class C1 implements interface I1
open class C1 <: I1 {}

// Class C2 inherits class C1 and implements interface I1, I2.
class C2 <: C1 & I1 & I2 {}
```

接口也可以是泛型的，此时在实现泛型接口时需要给出合法的类型参数。例如：

```
interface I1<T> {}
interface I2<U> {}
class C1 <: I1<Int32> & I2<Bool> {}
class C2<K, V> <: I1<V> & I2<K> {}
```

当类型实现接口时，对于非泛型接口不能直接实现多次，对于泛型接口不能用同样的类型参数直接实现多次。例如：

```
interface I1 {}
class C1 <: I1 & I1 {} // error

interface I2<T> {}
class C2 <: I2<Int32> & I2<Int32> {} // error
class C3<T> <: I2<T> & I2<Int32> {}  // ok
```

如果上述定义的泛型类 C3 在使用时被应用了类型参数 Int32，导致重复实现了两个相同类型的接口，那么编译器会在类型被使用到的位置报错：

```
interface I1<T> {}
open class C3<T> <: I1<T> & I1<Int32> {}  // ok
var a: C3<Int32> // error
var b = C3<Int32>() // error
class C4 <: C3<Int32> {}// error
```

关于接口的详细描述在章节接口。

6.1.1.4 类体

类体表示当前类包括的内容，类体由大括号包围，包含以下内容：

- 可选的静态初始化器
- 可选的主构造函数
- 可选的构造函数
- 可选的成员变量定义
- 可选的成员函数、成员操作符函数定义或声明
- 可选的成员属性定义或声明
- 可选的宏调用表达式
- 可选的类终结器

类体的语法定义如下：

```
classBody
    : '{'
          classMemberDeclaration*
          classPrimaryInit?
          classMemberDeclaration*
      '}'
    ;

classMemberDeclaration
    : classInit
    | staticInit
    | variableDeclaration
    | functionDefinition
    | operatorFunctionDefinition
    | macroExpression
    | propertyDefinition
    | classFinalizer
    ;
```

上述类体的语法定义中，包含以下内容：
`staticInit` 代表一个静态初始化器的定义，一个类最多只能定义一个静态初始化器；
`classPrimaryInit` 指的是主构造函数的定义，一个类最多只能定义一个；
`classInit` 表示 `init` 构造函数的定义；
`variableDeclaration` 表示成员变量的声明；
`operatorFunctionDefinition` 表示操作符重载成员函数的定义；
`macroExpression` 表示宏调用表达式，宏展开后依然要符合 `classMemberDeclaration` 的语法定义；
`propertyDefinition` 表示属性的定义；
`classFinalizer` 表示类的终结器的定义；
类体中引入的定义或声明均属于类的成员，将在类的成员章节中详细介绍。

6.1.2 类的成员

类的成员包括：

- 从父类（若存在）继承而来的成员。
- 如果类实现了接口，其成员还包括从接口中继承而来的成员。
- 在类体中声明或定义的成员，包括：静态初始化器、主构造函数、`init` 构造函数、静态成员变量、实例成员变量、静态成员函数、实例成员函数、静态成员属性、实例成员属性。

类的成员可以从不同的维度进行分类：
从是否被 `static` 修饰可以分为静态成员和实例成员。其中静态成员指不需要实例化类对象就能访问的成员，实例成员指必须先实例化类对象才能通过对象访问到的成员。
从成员的种类区分有静态初始化器、构造函数、成员函数、成员变量、成员属性。

需要注意的是：

- 所有的静态成员都不能通过对象名访问

6.1.2.1 构造函数

在仓颉编程语言中，有两种构造函数：主构造函数和 `init` 构造函数 (简称构造函数)。

主构造函数　主构造函数的语法定义如下：

```
classPrimaryInit
    : classNonStaticMemberModifier?  className '(' classPrimaryInitParamLists? ')'
      '{'
           superCallExression?
           ( expression
           | variableDeclaration
           | functionDefinition)*
      '}'
```

```
    ;

classNameIdentifier
    : identifier
    ;

classPrimaryInitParamLists
    : unnamedParameterList (',' namedParameterList)? (',' classNamedInitParamList)?
    | unnamedParameterList (',' classUnnamedInitParamList)?
        (',' classNamedInitParamList)?
    | classUnnamedInitParamList (',' classNamedInitParamList)?
    | namedParameterList (',' classNamedInitParamList)?
    | classNamedInitParamList
    ;

classUnnamedInitParamList
    : classUnnamedInitParam (',' classUnnamedInitParam)*
    ;

classNamedInitParamList
    : classNamedInitParam (',' classNamedInitParam)*
    ;

classUnnamedInitParam
    : classNonStaticMemberModifier? ('let'|'var') identifier ':' type
    ;

classNamedInitParam
    : classNonStaticMemberModifier? ('let'|'var') identifier'!' ':' type ('='
    ↪ expression)?
    ;

classNonStaticMemberModifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;
```

主构造函数的定义包括以下几个部分:

1、修饰符：可选。主构造函数可以使用 public、protected、private 其中之一修饰，都不使用是包内可见；详见访问修饰符

2、主构造函数名：与类型名一致。主构造函数名前不允许使用 `func` 关键字。

3、形参列表：主构造函数与 `init` 构造函数不同的是，前者有两种形参：普通形参和成员变量形参。普通形参的语法和语义与函数定义中的形参一致。

引入成员变量形参是为了减少代码冗余。成员变量形参的定义，同时包含形参和成员变量的定义，除此之外还表示了通过形参给成员变量赋值的语义。省略的定义和表达式会由编译器自动生成。

- 成员变量形参的语法和成员变量定义语法一致，此外，和普通形参一样支持使用`!`来标注是否为命名形参；
- 成员变量形参的修饰符有：`public`, `protected`, `private`；详见访问修饰符
- 成员变量形参只允许实例成员变量，即不允许使用 `static` 修饰；
- 成员变量形参不能与主构造函数外的成员变量同名；
- 成员变量形参可以没有初始值。这是因为主构造函数会由编译器生成一个对应的构造函数，将在构造函数体内完成将形参给成员变量的赋值；
- 成员变量形参也可以有初始值，初始值仅用于构造函数的参数默认值。成员变量形参的初值表达式中可以引用该成员变量定义之前已经定义的其他形参或成员变量（不包括定义在主构造函数外的实例成员变量），但不能修改这些形参和成员变量的值。需要注意的是，成员变量形参的初始值只在主构造函数中有效，不会在成员变量定义中包含初始值；
- 成员变量形参后不允许出现普通形参，并且要遵循函数定义时的参数顺序，命名形参后不允许出现非命名形参。

4、主构造函数体：如果显式调用父类构造函数，函数体内第一个表达式必须是调用父类构造函数的表达式；同时，主构造函数中不允许使用 `this` 调用本类中其它构造函数。父类构造函数调用之后，主构造函数体内允许写表达式、局部变量声明、局部函数定义，其中声明、定义和表达式需要满足 `init` 构造函数中对 `this` 和 `super` 使用的规则。具体规则详见 [init 构造函数]。

主构造函数定义的例子如下：

```
class Test{
    static let counter: Int64 = 3
    let name: String = "afdoaidfad"
    private Test(
        name: String,                  // regular parameter
        annotation!: String = "nnn",   // regular parameter
       var width!: Int64 = 1,       // member variable parameter with initial value
        private var length!: Int64,    // member variable parameter
        private var height!: Int64 = 3   // member variable parameter
    ) {
    }
}
```

主构造函数定义时，成员变量形参后不允许出现普通形参的例子如下：

```
class Test{
    static let counter: Int64 = 3
    let name: String = "afdoaidfad"
    private Test(
        name: String,                  // regular parameter
        annotation!: String = "nnn",   // regular parameter
       var width!: Int64 = 1,       // member variable parameter with initial value
        length!: Int64          // Error: regular parameters cannot be after member
         ↪ variable parameters
    ) {
    }
}
```

主构造函数是 `init` 构造函数的语法糖，编译器会自动生成与主构造函数对应的构造函数和成员变量的定义。自动生成的构造函数形式如下：

- 其修饰符与主构造函数修饰符一致；
- 其形参从左到右的顺序与主构造函数形参列表中声明的形参一致；
- 构造函数体内形式依次如下：
  - 依次是对成员变量的赋值，语法形式为 `this.x = x`，其中 `x` 为成员变量名；
  - 主构造函数体中的代码；

```
open class A<X> {
    A(protected var x: Int64, protected var y: X) {
        this.x = x
        this.y = y
    }
}

class B<X> <: A<X> {
     B(      // primary constructor, it's name is the same as the class
        x: Int64,    // regular parameter
        y: X,        // regular parameter
        v!: Int64 = 1,     // regular parameter
        private var z!: Int64 = v  // member variable parameter
        ) {
          super(x, y)
     }

     /* The corresponding init constructor with primary constructor auto-generated
        by compiler.

     private var z: Int64  // auto generated member variable definition
```

```
        init( x: Int64,
           y: X,
           v!: Int64 = 1,
           z!: Int64 = v) {         // auto generated named parameter definition
             super(x, y)
             this.z = z         // auto generated assign expression of member variable
        }
        */
    }
```

一个类最多可以定义一个主构造函数，除了主构造函数之外，可以照常定义其他构造函数，但要求其他构造函数必须和主构造函数所对应的构造函数构成重载。

init 构造函数 构造函数使用 init 关键字指定，不能带有 func 关键字，不能为构造函数显式定义返回类型，且必须有函数体。构造函数的返回类型为 Unit 类型。

构造函数的语法如下：

```
Init
    : nonSMemberModifier?  'init' '(' InitParamLists? ')'
      '{'
           (superCallExression | initCallExpression)?
           ( expression
           | variableDeclaration
           | functionDefinition)*
      '}'
    ;

InitParamLists
  : unnamedParameterList (',' namedParameterList)?
  | namedParameterList
  ;
```

可以在 init 前添加访问修饰符来限制该构造函数的可访问范围：详见访问修饰符

当构造一个类的对象时，实际上会调用此类的构造函数，如果没有参数类型匹配且可访问的构造函数，则会编译报错。

在一个类中，用户可以为这个类提供多个 init 构造函数，这些构造函数必须符合函数重载的要求。关于函数重载的详细描述，请参见函数重载。

```
class C {
    init() {}
    init(name: String, age: Int32) {}
}
```

创建类的实例时调用的构造函数，将根据以下顺序执行类中的表达式：

1. 先初始化主构造函数之外定义的有缺省值的变量；
2. 如果构造函数体内未显式调用父类构造函数或本类其它构造函数，则调用父类的无参构造函数 super()，如果父类没有无参构造函数，则报错；
3. 执行构造函数体内的代码；

如果一个类既没有定义主构造函数，也没有定义 init 构造函数，则会尝试生成一个（public 修饰的）无参构造函数。如果父类没有无参构造函数或者存在本类的实例成员变量没有初始值，则编译报错。

构造函数、this 和 super 的使用规则：

- 禁止使用实例成员变量 this.variableName 及其语法糖 variableName 和 super.variableName 作为构造函数参数的默认值；
- init 构造函数可以调用父类构造函数或本类其它构造函数，但两者之间只能调用一个。如果调用，必须在构造函数体内的第一个表达式处，在此之前不能有任何表达式或声明；
- 若构造函数没有显式调用其他构造函数，也没有显式调用父类构造函数，编译器会在该构造函数体的开始处插入直接父类的无参构造函数的调用。如果此时父类没有无参构造函数，则会编译报错；
- 构造函数体内调用完父类构造函数或本类其它构造函数之后允许使用 super.x 访问父类的实例成员变量 x；
- 若构造函数没有显式调用其他构造函数，则需要确保 return 之前本类声明的所有实例成员变量均完成初始化，否则编译报错；
- 可以被继承的类的构造函数中禁止调用实例成员函数或实例成员属性。
- 可以被继承的类的构造函数中禁止 this 逃逸。
- 构造函数在所有实例成员变量完成初始化之前，禁止使用隐式传参或捕获了 this 的函数或 lambda，禁止使用 super.f 访问父类的实例成员方法 f，禁止使用单独的 this 表达式，但允许使用 this.x 或其语法糖 x 来访问已经完成初始化的成员变量 x；
- 在构造函数体外，不允许通过 this 调用该类的构造函数；
- 禁止构造函数之间循环依赖，否则将编译报错。

```
var b: Int64 = 1
class A {
    var a: Int64 = 1
    var b: ()->Int64 =  { 3 }    // OK

    /* Cannot use lambda which captured `this` before all of the instance member
       variables are initialized. */
    var c: ()->Int64 =  { a }    // Error

    /* Cannot use function which captured `this` before all of the instance member
     ↪  variables are initialized. */
    var d: ()->Int64 = f       // Error

    var e: Int64 = a + 1    // OK
```

```
        func f(): Int64 {
            return a
        }
    }

    class B {
        var a: Int64 = 1
        var b: ()->Int64

        init() {
            b =  { 3 }
        }

        init(p: Int64) {
            this()
            b = { this.a } // OK
            b = f   // OK
        }

        func f(): Int64 {
            return a
        }
    }

    var globalVar: C = C()
    func f(c: C) {
        globalVar = c
    }
    open class C {
        init() {
            globalVar = this // Error, `this` cannot escape out of constructors of
             ↪  `open` class
            f(this) // Error, `this` cannot escape out of constructors of `open` class
            m() // Error: calling instance function is forbidden in constructors of
             ↪  `open` class
        }

        func m() {}
    }
```

6.1.2.2 静态初始化器

类或结构体中的静态变量也可以在静态初始化器中通过赋值表达式来初始化。不支持在枚举和接口中使用静态初始化器。

静态初始化器的语法如下：

```
staticInit
    : 'static' 'init' '(' ')'
    '{'
    expressionOrDeclarations?
    '}'
    ;
```

静态初始化器的规则如下：

- 静态初始化器会被自动调用，开发者不能显式调用；
- 静态初始化器会在它所属的包被加载时被调用，就像静态变量的初始化表达式；
- 一个类或结构中最多只能有一个静态初始化器；
- 对于一个非泛型的类或结构体，静态初始化器保证仅被调用一次；
- 对于一个泛型的类或结构体，静态初始化器在每个不同的类型实例化中，保证仅被调用一次；
  - 注意，如果没有该泛型类或结构体的类型实例化，则静态初始化器根本不会被调用；
- 静态初始化器在这个类或结构中所有静态成员变量的直接初始化后被调用，就像构造函数是在所有实例字段的直接初始化后被调用一样；
  - 这意味着可以在静态初始化器中引用进一步声明的静态成员变量；
  - 这也意味着静态初始化器可以位于类或结构中的任何位置，顺序并不重要；
- 在同一个文件中，跨越多个类，即使这些类之间存在继承关系，静态初始化器仍然以自上而下的顺序被调用。
  - 这意味着不能保证父类的所有静态成员变量必须在当前类的初始化之前被初始化。

```
class Foo <: Bar {
  static let y: Int64

  static init() { // called first
    y = x // error: not yet initialized variable
  }
}

open class Bar {
  static let x: Int64

  static init() { // called second
    x = 2
```

```
    }
}
```

- 静态成员变量必须只以一种方式初始化，要么直接通过右侧表达式，要么在静态初始化器中；
  - 尽管一个可变的静态变量可以同时被直接赋值和在静态初始化器中被赋值来进行初始化，但在这种情况下的变量初始化只是直接赋值，而静态初始化器中的赋值被认为是简单的重新赋值。这意味着在静态初始化器被调用之前，该变量会有一个直接赋值；
  - 如果一个不可变的静态变量同时被直接赋值和在静态初始化器中被赋值，编译器会报一个关于重新赋值的错误；
    - 如果一个不可变的静态变量在静态初始化器中被多次赋值，也会报告这个错误。
  - 如果一个静态变量既没有直接初始化也没有在静态初始化器中初始化，编译器会报一个关于未初始化变量的错误。
  - 上述情况可由一个特殊的初始化分析检测出来的，它取决于实现方式
- 静态初始化器不能有任何参数。
- 静态初始化器中不允许使用 `return` 表达式。
- 在静态初始化器中抛出异常会导致程序的终止，就像在静态变量的右侧表达式中抛出异常一样。
- 实例成员变量，或者未初始化完成的静态成员变量不能在静态初始化器中使用。
- 静态初始化器的代码是同步的，以防止部分初始化的类或结构的泄漏。
- 静态属性仍应以完整的方式声明，包括 `getter` 和 `setter`；
- 与静态函数不同，静态初始化器不能在扩展中（在 `extend` 内）使用。
- 由于静态初始化器是自动调用的，并且无法显式调用它，因此可见性修饰符（即 `public`、`private`）不能用于修饰静态初始化器。

下面用一个例子来演示初始化分析的规则：

```
class Foo {
    static let a: Int64
    static var c: Int64
    static var d: Int64 // error: uninitialized variable
    static var e: Int64 = 2
    static let f: Int64 // error: uninitialized variable
    static let g: Int64 = 1

    let x = c

    static init() {
        a = 1
        b = 2

        Foo.c // error: not yet initialized variable
        let anotherFoo = Foo()
        anotherFoo.x // error: not yet initialized variable
```

```
            c = 3
            e = 4
            g = 2 // error: reassignment
        }

        static let b: Int64
    }
```

6.1.2.3 成员变量

成员变量的声明　声明在类、接口中的变量称为成员变量。成员变量可以用关键字 let 声明为不可变的，也可以用关键字 var 声明为可变的。

主构造函数之外的实例成员变量声明时可以有初始值，也可以没有初始值。如果有初始值，初始值表达式可以使用此变量声明之前的成员变量。由于这些成员变量的初始化的执行顺序是在调用父类构造函数之前，初始值表达式中禁止使用带 super 的限定名访问父类的成员变量。

以下是变量的代码示例：

```
open class A {
    var m1: Int32 = 1
}
class C <: A {
    var a: Int32 = 10
    let b: Int32 = super.m1  // Error
}
```

变量的修饰符　类中的变量可以被访问修饰符修饰，详细内容请参考包和模块管理章节访问修饰符

另外，如果类中的一个变量用 static 修饰，则它属于类的静态变量。static 可以与其他访问修饰符同时使用。静态变量会被子类继承，子类和父类的静态变量是同一个。

```
class C {
    static var a = 1
}

var r1 = C.a  // ok
```

6.1.2.4 类成员函数

类成员函数的声明和定义　在类中允许定义函数，同时允许在抽象类中声明函数。定义和声明的区别在于该函数是否有函数体。

类成员的函数分为实例成员函数、静态成员函数。

类成员函数定义或声明的语法如下：

```
functionDefinition
    : modifiers 'func' identifier typeParameters? functionParameters (':'
    ↪ returnType)? genericConstraints? (('=' expression) | block)?
    ;
```

实例成员函数 实例成员函数的第一个隐式参数是 this，每当调用实例成员函数时，都意味着需要先传入一个完整的对象，因此在对象未创建完成时就调用实例成员函数的行为是被禁止的，但是该函数的类型将不包括该隐式参数。类对象创建完成的判断依据是已经调用了类的构造函数。

实例成员函数可以分为抽象成员函数和非抽象成员函数。

抽象成员函数

抽象成员函数只能在抽象类或接口中声明，没有函数体。

```
abstract class A {
    public func foo(): Unit     // abstract member function
}
```

非抽象成员函数

非抽象成员函数允许在任何类中定义，必须有函数体。

```
class Test {
    func foo(): Unit { // non-abstract member function
        return
    }
}
```

抽象实例成员函数默认具有 open 的语义。在抽象类中定义抽象实例成员函数时，open 修饰符是可选的，但必须显式指定它的可见性修饰符为 public 或 protected。

静态成员函数 静态成员函数用 static 关键字修饰，它不属于某个实例，而是属于它所在的类型。同时静态函数必须有函数体。

- 静态成员函数中不能使用实例成员变量，不能调用实例成员函数，不能调用 super 或 this 关键字。
- 静态成员函数中可以引用其他静态成员函数或静态成员变量。
- 静态成员函数可以用 private、protected、public、internal 修饰。详见访问修饰符
- 静态成员函数在被其它子类继承时，这个静态成员函数不会被拷贝到子类中。
- 抽象类和非抽象类中的静态成员函数都必须拥有实现。

例如：

```
class C<T> {
  static let a: Int32 = 0
  static func foo(b: T): Int32 {
    return a
```

```
    }
}

main(): Int64 {
    print("${C<Int32>.foo(3)}")
    print("${C<Bool>.foo(true)}")
    return 0
}
```

这段程序对于 C<Int32> 与 C<Bool> 分别有他们各自的静态成员变量 a 与静态函数 foo。

类中的静态函数可以声明新的类型变元，这些类型变元也可以存在约束。在调用时只需对类给出合法的类型，然后对静态函数给出合法的类型就可以了：

```
class C<T> {
    static func foo<U>(a: U, b: T): U { a }
}

var a: Bool = C<Int32>.foo<Bool>(true, 1)
var b: String = C<Bool>.foo<String>("hello", false)

func f<V>(a: V): V { C<Int32>.foo<V>(a, 0) }
```

类成员函数的修饰符　类成员函数可以被所有访问修饰符修饰，详见访问修饰符

其他可修饰的非访问修饰符如下：

- open：一个成员函数想要被覆盖，需要用 open 修饰符修饰。它与 static 修饰符有冲突。当带 open 修饰的实例成员被 class 继承时，该 open 的修饰符也会被继承。如果 class 中存在被 open 修饰的成员，而当前 class 没有被 open 修饰或不包含 open 语义，那么这些 open 修饰的成员仍然没有 open 效果，编译器对这种情况会报 warning 提示（对于继承下来的 open 成员或者 override 的成员不需要报 warning）。

  一个被 open 修饰的函数，必须被 public 或 protected 修饰。

```
// case 1
open class C1 {        // In this case, the 'open' modifier before 'class C1' is
↪  required.
    public open func f() {}
}
class C2 <: C1 {
    public override func f() {}
}
// case 2
open class A {
    public open func f() {}
```

```
}
open class B <: A {}
class C <: B {
    public override func f() {} // ok
}
// case 3
interface I {
   func f() {}
}
open class Base <: I {} // The function f Inherits the open modifier.
class Sub <: Base {
   public override func f() {} // ok
}
```

- override：当一个函数覆盖另一个可以被覆盖的函数时，允许可选地使用 override 进行修饰（override 不具备 open 的语义，如果用 override 修饰的函数还需要允许能被覆盖，需要重新用 open 修饰）。示例如上。函数覆盖的规则请参见覆盖章节。
- static：用 static 修饰的函数为静态成员函数，必须有函数体。静态成员函数不能用 open 修饰。

  静态成员函数内不可以访问所在类的实例成员；实例成员函数内能访问所在类的静态成员。

```
class C {
  static func f() {}   // Cannot be overwritten and must have a function body.
}
```

- redef：当一个静态函数重定义继承自父类型的静态函数时，允许可选地使用 redef 进行修饰。

```
open class C1 {
  static func f1() {}
  static func f2() {}
}
class C2 <: C1 {
  redef static func f1() {}
  redef static func f2() {}
}
```

6.1.2.5 类终结器

类终结器是类的一个实例成员函数，这个方法在类的实例被垃圾回收的时候被调用。

```
class C {
    // below is a finalizer
    ~init() {}
}
```

终结器的语法如下：

```
classFinalizer
    : '~' 'init' '(' ')' block
    ;
```

1. 终结器没有参数，没有返回类型，没有泛型类型参数，没有任何修饰符，也不可以被用户调用。
2. 带有终结器的类不可被 open 修饰，只有非 open 的类可以拥有终结器。
3. 一个类最多只能定义一个终结器。
4. 终结器不可以定义在扩展中。
5. 终结器被触发的时机是不确定的。
6. 终结器可能在任意一个线程上执行。
7. 多个终结器的执行顺序是不确定的。
8. 终结器向外抛出未捕获异常的行为由实现决定。
9. 终结器中创建线程或者使用线程同步功能的行为由实现决定。
10. 终结器执行结束之后，如果这个对象还可以被继续访问，后果由实现决定。
11. 不允许 this 逃逸出终结器。
12. 终结器中不允许调用实例成员方法。

举例如下：

```
class SomeType0 {
    ~init() {} // OK
}
class SomeType1 {
    ~init(x: Int64) {} // Error, finalizer cannot have parameters
}
class SomeType2 {
    private ~init() {} // Error, finalizer cannot have accessibility modifiers
}
class SomeType3 {
    open ~init() {} // Error, finalizer cannot have open modifier
}
open class SomeType4 {
    ~init() {} // Error, open class can't have a finalizer
}

var GlobalVar: SomeType5 = SomeType5()
class SomeType5 {
    ~init() {
        GlobalVar = this // do not escape `this` out of finalizer, otherwise,
        ↪ unexpected behavior may happen.
    }
}
```

6.1.2.6 类成员属性

类中也可以定义成员属性，定义成员属性的语法参见属性章节。

6.1.2.7 类的实例化

定义完非抽象的 class 类型之后，就可以创建对应的 class 实例。创建 class 实例的方式按照是否包含类型变元可分为两种：

1. 创建非泛型 class 的实例：ClassName(arguments)。其中 ClassName 为 class 类型的名字，arguments 为实参列表。ClassName(arguments) 会根据重载函数的调用规则（参见函数重载）调用对应的构造函数，然后生成 ClassName 的一个实例。举例如下：

```
class C {
    var a: Int32 = 1
    init(a: Int32) {
        this.a = a
    }
    init(a: Int32, b: Int32) {
        this.a = a + b
    }
}

main() : Int64 {
    var myC = C(2)      // invoke the first constructor
    var myC2 = C(3, 4)  // invoke the second constructor
    return 0
}
```

2. 创建泛型 class 的实例：ClassName<Type1, Type2, ... , TypeK>(arguments)。与创建非泛型 class 的实例的差别仅在于需要对泛型参数进行实例化，泛型实参可以显式指定，也可以省略（此时由编译器根据程序上下文推断出具体的类型）。举例如下：

```
class C<T, U> {
    var a: T
    var b: U
    init(a: T, b: U) {
        this.a = a
        this.b = b
    }
}

main() : Int64 {
    var myC = C<Int32, Int64>(3, 4)
```

```
    var myC2 = C(3,4)   // The type of myC2 is inferred to C<Int64, Int64>.
    return 0
}
```

6.1.3 Object 类

Object 类是所有 class 类型的父类（不包括 interface 类型），Object 类中不包含任何成员，即 Object 是一个“空”的类。Object 有 public 修饰的无参构造函数。

6.1.4 This 类型

在类内部，我们支持 This 类型占位符，它只能被作为实例成员函数的返回类型来使用，并且在编译时会被替换为该函数所在类的类型，从而进行类型检查。

1. 返回类型是 This 的函数，只能返回 This 类型表达式，其它表达式都不允许。
2. This 类型的表达式包含 this 和调用其它返回 This 的函数。
3. This 类型是当前类型的子类型，This 可以自动 cast 成当前类型，但反之不行。
4. 函数体内不能显式使用 This 类型。在返回值以外的地方使用 This 类型表达式都会被推断为当前类型。
5. 如果实例成员函数没有声明返回类型，并且只存在返回 This 类型表达式时，当前函数的返回类型会推断为 This。
6. 含 This 的 open 函数在 override 时，返回类型必须保持 This 类型。
7. 父类中的 open 函数返回类型如果是父类，子类在 override 时可以使用 This 作为返回类型。

```
open class C1 {
    func f(): This {  // its type is `() -> C1`
        return this
    }

    func f2() { // its type is `() -> C1`
        return this
    }

    public open func f3(): C1 {
        return this
    }
}

class C2 <: C1 {
    // member function f is inherited from C1, and its type is `() -> C2` now

    public override func f3(): This { // ok
        return this
    }
```

```
}

var obj1: C2 = C2()
var obj2: C1 = C2()

var x = obj1.f()    // During compilation, the type of x is C2
var y = obj2.f()    // During compilation, the type of y is C1
```

6.2 接口

接口用来定义一个抽象类型，它不包含数据，但可以定义类型的行为。一个类型如果声明实现某接口，并且实现了该接口中所有的成员，就被称为实现了该接口。

接口的成员可以包含实例成员函数、静态成员函数、操作符重载函数、实例成员属性和静态成员属性，这些成员都是抽象的，但函数和属性可以定义默认实现。

6.2.1 接口定义

6.2.1.1 接口定义的语法

接口的定义使用 interface 关键字，接口定义依次为：可缺省的修饰符、interface 关键字、接口名、可选的类型参数、是否指定父接口、可选的泛型约束、接口体的定义

以下示例是一些接口定义：

```
interface I1 {}
public interface I2<T> {}
public interface I3<U> {}
public interface I4<V> <: I2<V> & I3<Int32> {}
```

接口定义的语法如下：

```
interfaceDefinition
    : interfaceModifierList? 'interface' identifier
    typeParameters?
    ('<:' superInterfaces)?
    genericConstraints?
    interfaceBody
    ;
```

接口只可以定义在 top level。

接口体的 {} 不允许省略

```
interface I {}      // {} not allowed to be omitted
```

6.2.1.2 接口的修饰符

在此小节中，我们将介绍定义接口可以使用的修饰符。

访问修饰符

- 访问修饰符：默认，即不使用访问修饰符，表示只能在包内部访问。也可以使用 public 修饰符表示包外部可访问；

```
public interface I {}  // can be accessed outside the package
interface I2 {}        // can only be accessed inside the package
```

继承性修饰符

- 继承修饰符：默认，即不使用继承修饰符即可表示接口具有 open 修饰的语义，能在任何可以访问 interface 的位置继承、实现或扩展 interface，当然，也可以显式地使用 open 修饰符修饰；也可以使用 sealed 修饰符表示只能在 interface 定义所在的包内继承、实现或扩展该 interface。
- sealed 已经蕴含了 public 的语义，所以定义 sealed interface 时的 public 修饰符是可选的。
- 继承 sealed 接口的子接口或实现 sealed 接口的类仍可被 sealed 修饰或不使用 sealed 修饰。若 sealed 接口的子接口被 public 修饰，且不被 sealed 修饰，则其子接口可在包外被继承、实现或扩展；
- 继承、实现 sealed 接口的类型可以不被 public 修饰。

```
// package A
package A
public sealed interface I1 {}   // OK
sealed interface I2  {}          // OK, 'public' is optional when 'sealed' is used
open interface I3 {}             // OK
interface I4  {}                 // OK, 'open' is optional

class C1 <: I1 {}  // OK
public open class C2 <: I1 {}   // OK
public sealed class C3 <: I1 {}  // OK
extend Int64 <: I1 {}   // OK

// package B
package B
import A.*

class S1 <: C2 {}  // OK
class S2 <: C3 {}  // Error, C3 is sealed class, cannot be inherited here.
```

6.2.2 接口成员

接口的成员包括：

- 在接口体中声明（即在 {} 中声明）的成员：静态成员函数、实例成员函数、操作符重载函数、静态成员属性和实例成员属性。

- 从其他接口继承而来的成员。如以下示例中，I2 的成员将包括 I1 中允许被继承的成员。

```
interface I1 {
    func f(): Unit
}
interface I2 <: I1 {}
```

接口成员的 BNF 如下：

```
interfaceMemberDeclaration
        : (functionDefinition|macroExpression|propertyDefinition) end*
        ;
```

6.2.2.1 接口中的函数

接口中函数的声明和定义　接口中可以包含实例成员函数和静态成员函数，这些函数与普通实例成员函数和静态成员函数的编写方式一样，但可以没有函数实现，这些函数被称为抽象函数。

对于拥有实现的抽象函数，我们称该函数拥有默认实现。

以下是包含抽象函数的示例：

```
interface MyInterface {
    func f1(): Unit // Default implementation not included
    static func f2(): Unit // Default implementation not included

    func f3(): Unit { // Default implementation included
        return
    }
    static func f4(): Unit { // Default implementation included
        return
    }
}
```

抽象函数可以拥有命名参数，但不能拥有参数默认值。

```
interface MyInterface {
    func f1(a!: Int64): Unit // OK
    func f2(a!: Int64 = 1): Unit // Error, cannot have parameter default values
}
```

接口中函数和属性的修饰符　接口中定义的函数或属性已经蕴含 public 语义，使用 public 修饰会产生编译告警。不允许使用 protected、internal 和 private 修饰符修饰接口中定义的函数或属性。

以下是错误的示例：

```
interface MyInterface {
    public func f1(): Unit // Access modifiers cannot be used
```

```
    private static func f2(): Unit // Access modifiers cannot be used
}
```

使用 static 修饰的函数被称为静态成员函数，可以没有函数体。没有函数体的 static 函数不能直接使用接口类型调用，拥有函数体的 static 函数可以使用接口类型调用。当使用 interface 类型名直接调用其静态成员函数时，如果这个函数里面直接或间接调用了接口中（自己或其它接口）没有实现的其它静态函数，则编译报错。

```
interface I {
    static func f1(): Unit
    static func f2(): Unit {}
    static func f3(): Unit {
        f1()
    }
}

main() {
    I.f1() // Error, cannot directly call
    I.f2() // OK
    I.f3() // Error, f1 not implemented
}
```

接口中的实例成员函数默认具有 open 的语义。在接口中定义实例成员函数时，open 修饰符是可选的。

```
interface I {
    open func foo1() {} // ok
    func foo2() {}      // ok
}
```

使用 mut 修饰的函数是一种特殊的实例成员函数，可以用于抽象 struct 类型的可变行为。

```
interface I {
    mut func f(): Unit
}
```

6.2.2.2 接口中的成员属性

interface 中也可以定义成员属性，定义成员属性的语法参见 7 章。

6.2.2.3 接口的默认实现

interface 中的抽象函数和抽象属性都可以拥有默认实现。

interface 被其它 interface 或类型继承或实现的时候，如果该 interface 的默认实现没有被重新实现，则这些默认实现会被拷贝子类型中。

1. 实例成员函数的默认实现中可以使用 this，this 的类型是当前 interface。

2. 默认实现和非抽象的成员函数一样，可以访问当前作用域中所有可访问的元素。
3. 默认实现是一种语法糖，可以给实现类型提供默认行为，子接口可以沿用父接口的默认实现。
4. 默认实现不属于继承语义，因此不能使用 override/redef，也不能使用 super。
5. 继承接口的子类型为类时，默认实现保留 open 语义，可以被类的子类重写。

```
interface I {
    func f() { // f: () -> I
        let a = this // a: I
        return this
    }
}
```

6.2.3 接口继承

接口允许继承一个或多个接口。

```
interface I1<T> {}
interface I2 {}
interface I3<U> <: I1<U> {} // inherit a generic interface.
interface I4<V> <: I1<Int32> & I2 {} // inherit multiple interfaces.
```

子接口继承父接口时，会继承父接口的所有成员：

子接口继承父接口时，对于非泛型接口不能直接继承多次，对于泛型接口不能使用相同类型参数直接继承多次。例如：

```
interface I1 {}
interface I2 <: I1 & I1 {} // error

interface I3<T> {}
interface I4 <: I3<Int32> & I3<Int32> {} // error
interface I5<T> <: I3<T> & I3<Int32> {}  // ok
```

如果泛型接口 I3 在使用时被给了类型参数为 Int32，那么编译器会在类型使用的位置报错：

```
interface I1<T> {}
interface I2<T> <: I1<T> & I1<Int32> {}  // ok
interface I3 <: I2<Int32> {} // error

main() {
    var a: I2<Int32> // error
}
```

6.2.3.1 子接口中的默认实现

子接口如果继承了父接口中没有默认实现的函数或属性，则在子接口中允许仅写此函数或属性的声明（当然也允许定义默认实现），并且函数声明或定义前的 override 或 redef 修饰符是可选的。示例如下：

```
interface I1 {
    func f1(): Unit
    static func f2(): Unit
}
interface I2 <: I1 {
    func f1(): Unit        // ok
    static func f2(): Unit // ok
}
interface I3 <: I1 {
    override func f1(): Unit     // ok
    redef static func f2(): Unit // ok
}
interface I4 <: I1 {
    func f1(): Unit {}        // ok
    static func f2(): Unit {} // ok
}
interface I5 <: I1 {
    override func f1(): Unit {}     // ok
    redef static func f2(): Unit {} // ok
}
```

子接口如果继承了父接口中有默认实现的函数或属性，则在子接口中不允许仅写此函数或属性的声明而没有实现，如果子接口中给出新的默认实现，那么定义前的 override 或 redef 修饰符是可选的。示例如下：

```
interface I1 {
    func f1(): Unit {}
    static func f2(): Unit {}
}
interface I2 <: I1 {
    func f1(): Unit        // error, 'f1' must has a new implementation
    static func f2(): Unit // error, 'f2' must has a new implementation
}
interface I3 <: I1 {
    override func f1(): Unit {}     // ok
    redef static func f2(): Unit {} // ok
}
interface I4 <: I1 {
    func f1(): Unit {}        // ok
    static func f2(): Unit {} // ok
}
```

如果子接口继承的多个父接口中拥有相同签名成员的默认实现，子接口必须提供自己版本的新默认实现，否则会编译报错。

```
interface I1 {
    func f() {}
}
interface I2 {
    func f() {}
}
interface I3 <: I1 & I2 {} // error, I3 must implement f: () -> Unit
```

6.2.4 实现接口

6.2.4.1 实现接口时的覆盖与重载

一个类型实现一个或多个接口时规则如下：

1. 抽象类以外的类型实现接口时，必须实现所有的函数、属性。
2. 抽象类实现接口时，允许不实现接口中的函数和属性。
3. 实现函数的函数名、参数列表必须与接口中对应函数的相同。
4. 实现函数的返回类型应该与接口中对应函数的返回类型相同或者为其子类型。
5. 如果接口中的函数为泛型函数，则要求实现函数的类型变元约束比接口中对应函数更宽松或相同。
6. 实现属性的 `mut` 修饰符必须与接口中相应的属性相同。
7. 实现属性的类型，必须与接口中对应的属性相同。
8. 如果多个接口中只有同一函数或属性的一个默认实现，则实现类型可以不实现该函数或属性，使用默认实现。
9. 如果多个接口中包含同一函数或属性的多个默认实现，则实现类型必须要实现该函数或属性，无法使用默认实现。
10. 如果实现类型已经存在 (从父类继承或本类定义) 接口中同一函数或属性的实现，则不会再使用任何接口中的默认实现。
11. 类型在实现接口时，函数或属性定义前的 `override` 修饰符（或 `redef` 修饰符）是可选的，无论接口中的函数或属性是否存在默认实现。

抽象类以外的类型实现接口时，必须对接口中的抽象函数和抽象属性进行实现，如下示例中的 `f1`，`f3`；允许不使用接口中的函数默认实现，如下示例中的 `f2`；抽象类允许不实现接口中的实例成员函数，如下示例中抽象类 `C1` 并没有实现接口 `I` 中的 `f1`。

```
interface I {
    func f1(): Unit
    func f2(): Unit {
        return
    }
    static func f3(): Int64
}
class C <: I {
    public func f1(): Unit {}
    public func f2(): Unit {
```

```
        return
    }
    public static func f3(): Int64 {
        return 0
    }
}

abstract class C1 <: I {
    public static func f3(): Int64 {
        return 1
    }
}
```

示例：接口 I 中的函数 f 和 g 为泛型函数，class E 和 F 满足实现函数的类型变元约束比被实现函数的更宽松或相同的要求，编译成功；class D 不满足要求，则编译报错。

```
// C <: B <: A
interface I {
    static func f<T>(a: T): Unit where  T <: B
    static func g<T>(): Unit where T <: B
}

class D <: I  {
    public static func f<T>(a: T) where T <: C {} // Error, stricter constraint
    public static func g<T>() where T <: C {} // Error, stricter constraint
}

class E <: I  {
    public static func f<T>(a: T) where T <: A {} // OK, looser constraint
    public static func g<T>() where T <: A {} // OK, looser constraint
}

class F <: I  {
    public static func f<T>(a: T) where T <: B {} // OK, same constraint
    public static func g<T>() where T <: B {} // OK, same constraint
}
```

更多例子：

```
// case 1
interface I1 {
    func f(): Unit
}
```

```
interface I2 {
    func f(): Unit
}
class A <: I1 & I2 {
    public func f(): Unit {} // ok
}

// case 2
interface I1 {
    func f(): Unit
}
interface I2 {
    func f(): Unit {}
}
open class A {
    public open func f(): Unit {} // ok
}
class B <: A & I1 & I2 {
    public override func f(): Unit {} // ok
}

// case 3
interface I1 {
    func f(): Unit
}
interface I2 {
    func f(): Unit {}
}
class A <: I1 & I2 {} // ok, f from I2

// case 4
interface I1 {
    func f(): Unit {}
}
interface I2 {
    func f(): Unit {}
}
class A <: I1 & I2 {} // error
class B <: I1 & I2 { // ok,
    public func f(): Unit {}
}
```

```
// case 5
interface I1 {
    func f(a: Int): Unit {}
}
interface I2  {
    func f(a: Int): Unit {}
}

open class A  {
    public open func f(a: Int): Unit {}
}
open class B <: A & I1 & I2{ // ok, f from A
}
```

实现接口时函数重载的规则：　父作用域函数与子作用域函数的函数名必须相同，但参数列表必须不同。

以下几个例子中给出了类型实现接口时构成函数重载的一些情形，需要注意的是当类型现接口时需要为被重载的函数声明提供实现。

示例一：分别在 I1 和 I2 中声明了函数名相同、参数列表不同的函数 f。需要在实现类 C 中同时实现参数类型为 Unit 和参数类型为 Int32 的 f。

```
interface I1 {
    func f(): Unit
}
interface I2 {
    func f(a: Int32): Unit
}
class C <: I1 & I2 {
    public func f(): Unit {}        // The f in I1 needs to be implemented.
    public func f(a: Int32): Unit {}     // The f in I2 needs to be implemented.
}
```

示例二：分别在 I1 和 I2 中定义了函数相同、参数列表不同的默认函数 f。不需要在 C 中实现 f。

```
interface I1 {
    func f() {}
}
interface I2 {
    func f(a: Int32) {}
}
class C <: I1 & I2 {
}
```

示例三：在 I1 中声明了函数名为 f，参数类型为 Unit 的函数；在 I2 中定义了函数名为 f，参数类型为 Int32 的函数。类 C 中必须实现参数类型为 Unit 的函数 f。

```
interface I1 {
    func f(): Unit
}
interface I2 {
    func f(a: Int32):Unit {
        return
    }
}
class C <: I1 & I2 {
    public func f(): Unit {
        return
    }
}
```

6.2.5 Any 接口

Any 接口是一个语言内置的空接口，所有 interface 类型都默认继承它，所有非 interface 类型都默认实现它，因此所有类型都可以作为 Any 类型的子类型使用。

```
class A {}

struct B {}

enum C { D }

main() {
    var i: Any = A() // ok
    i = B() // ok
    i = C.D // ok
    i = (1, 2) // ok
    i = { => 123 } // ok
    return 0
}
```

在类型定义处可以显式声明实现 Any 接口，如果没有则会由编译器隐式实现，但不能使用扩展重新实现 Any 接口。

```
class A <: Any {} // ok

class B {} // Implicit implement Any

extend B <: Any {} // error
```

6.3 覆盖、重载、遮盖、重定义

6.3.1 覆盖

6.3.1.1 覆盖的定义

子类型中定义父类型中已经存在的同名非抽象、具有 open 语义的实例函数时，允许使用可选的 override 进行修饰以表示是对父类型中同名函数的覆盖。函数的覆盖需要遵循以下规则：

- 覆盖函数与被覆盖函数的函数名必须相同。
- 覆盖函数与被覆盖函数的参数列表必须相同。

  参数列表相同指函数的参数个数、参数类型相同

- 覆盖函数的返回类型与被覆盖函数的返回类型相同或为其子类型。
- 同一个函数覆盖多个父作用域中出现的函数，每个函数的覆盖规则由上面的其他规则确定。

示例：

- 类 C1、C2，接口 I 中的函数 f，参数参数列表相同，返回类型相同，满足覆盖的要求。
- 类 C1 与 C2 中的函数 f1，参数列表相同，在 C1 中的返回类型是 Father，在 C2 中的返回类型是 Child，由于 Child 是 Father 的子类型，因此满足覆盖的规则。

```
open class Father {}
class Child <: Father {}

open class C1 {
    public open func f() {}
    public open func f1(): Father { Father() }
}

interface I {
    func f() {}
}

class C2 <: C1 & I {
    public override func f() {}                  // OK.
    public override func f1(): Child { Child() } // OK.
}
```

需要注意的一些细节包括：

1. 类中用 private 修饰的函数不会被继承，无法在子类中访问。
2. 静态函数不能覆盖实例函数

6.3.1.2 覆盖函数的调用

如果子类型覆盖了父类型中的函数，并且在程序中调用了此函数，则编译器会根据运行时对象指向的类型来选择执行此函数的哪个版本。

以下示例中，在类 C1 定义了函数 f，其子类 C2 中覆盖了 f，同时定义了类型为 C1 的变量 a 与 b，并将 C1 的对象 myC1 赋给 a，将 C2 的对象 myC2 赋给 b。通过 a 和 b 来调用函数 f 时，会在运行时根据它们实际的类型来选择对应的函数。

```
open class C1 {
    public open func f(): Unit {
        return
    }
}
class C2 <: C1 {
    public override func f(): Unit {
        return
    }
}

var myC1 = C1()
var myC2 = C2()

// Assign the object of the superclass C1 to the variable of the C1 type.
var a: C1 = myC1
// Assign the object of the superclass C2 to the variable of the C1 type.
var b: C1 = myC2

// Invokes f of C1 based on the object type of at runtime
var c = a.f()
// Invokes f of C2 based on the object type of at runtime
var d = b.f()
```

6.3.2 重载

关于函数重载定义及调用的详细描述，请参见函数重载章节。

类和接口的静态成员函数和实例成员函数之间不允许重载。如果类或接口的静态成员函数和实例成员函数（包括本类/接口定义的和从父类或父接口继承过来的成员函数）同名，则报编译错误。

```
open class Base {}
class Sub <: Base {}
open class C{
    static func foo(a: Base) {
    }
```

```
}
open class B <: C {
    func foo(a: Sub) {   // Error
        C.foo(Sub())
    }
}

class A <: B {
    // Static and instance functions cannot be overloaded.
    static func foo(a: Sub) {}  // Error
}
```

重载决议时，父类型和子类型中定义的函数当成同一作用域优先级处理。

6.3.3 遮盖

子类型的成员不可遮盖父类型的成员。如果发生遮盖，将编译报错。
如下示例中，类 C1 和 C2 都有同名实例变量 x，编译报错。

```
open class C1 {
    let x = 1
}
class C2 <: C1 {
    let x = 2 // error
}
```

6.3.4 重定义

6.3.4.1 重定义函数的定义

子类型中定义父类型中已经存在的同名非抽象静态函数时，允许使用可选的 redef 进行修饰以表示是对父类型中同名函数的重定义。函数的重定义需要遵循以下规则：

- 函数与被重定义函数的函数名必须相同。
- 函数与被重定义函数的参数列表必须相同。

  参数列表相同指函数的参数个数、参数类型相同。

- 函数的返回类型与被重定义函数的返回类型相同或为其子类型。
- 如果被重定义函数为泛型函数，则要求重定义函数的类型变元约束比被实现函数更宽松或相同。
- 同一个函数重定义多个父作用域中出现的函数，每个函数的覆盖规则由上面的其他规则确定。

示例：

- 类 C1、C2，接口 I 中的函数 f，参数参数列表相同，返回类型相同，满足重定义的要求。
- 类 C1 与 C2 中的函数 f1，参数列表相同，在 C1 中的返回类型是 Father，在 C2 中的返回类型是 Child，由于 Child 是 Father 的子类型，因此满足重定义的规则。

```
open class Father {}
class Child <: Father {}

open class C1 {
    public static func f() {}
    public static func f1(): Father { Father() }
}

interface I {
    static func f() {}
}

class C2 <: C1 & I {
    public redef static func f() {}                  // OK.
    public redef static func f1(): Child { Child() } // OK.
}
```

示例：基类 Base 中的函数 f 和 g 为泛型函数，子类 E 和 F 满足重定义函数比被重定义函数的类型变元约束更宽松或相同的要求，编译成功；子类 D 不满足要求，则编译报错。

```
// C <: B <: A
open class Base {
    static func f<T>(a: T): T where T <: B {...}
    static func g<T>(): T where T <: B {...}
}

class D <: Base  {
    redef static func f<T>(a: T): T where T <: C {...} // Error, stricter constraint
    redef static func g<T>(): T where T <: C {...} // Error, stricter constraint
}

class E <: Base  {
    redef static func f<T>(a: T): T where T <: A {...} // OK, looser constraint
    redef static func g<T>(): T where T <: A {...} // OK, looser constraint
}

class F <: Base  {
    redef static func f<T>(a: T): T where T <: B {...} // OK, same constraint
    redef static func g<T>(): T where T <: B {...} // OK, same constraint
}
```

redef 修饰符不能用于修饰静态初始化器（因为静态初始化器不能被显式调用），否则编译器会报告错误。

6.3.4.2 重定义函数的调用

如果子类型重定义了父类型中的函数，并且在程序中调用了此函数，则编译器会根据类型来选择执行此函数的哪个版本。

以下示例中，在类 C1 定义了函数 f，其子类 C2 中重定义了 f。通过 C1 和 C2 来调用函数 f 时，会在编译时根据它们的类型来选择对应的函数。

```
open class C1 {
    static func f(): Unit {
        return
    }
}
class C2 <: C1 {
    redef static func f(): Unit {
        return
    }
}

// Invokes f of C1
var c = C1.f()
// Invokes f of C2
var d = C2.f()
```

6.3.5 访问控制等级限制

根据访问修饰符所允许的可访问范围大小，规定访问修饰符等级如下：

- 类型内部访问修饰符等级为 public > protected > default > private。

在此等级下，对跨访问等级的行为规定如下：

- 在子类继承父类时，子类的实例成员函数覆盖父类的实例成员函数，或者子类的静态成员函数重定义父类的静态成员函数，子类成员函数的可访问等级不得修改为小于父类成员函数的访问等级；
- 在类型实现接口时，类型的成员函数实现了接口中的抽象函数，类型的成员函数的可访问等级不得修改为小于抽象函数的访问等级。

以下是访问等级限制的代码示例：

```
open class A {
    protected open func f() {}
}
```

```
interface I {
    func m() {} // public by default
}
class C <: A & I {
    private override func f() {} // Error: the access control of override function
     ↪ is lower than the overriden function
    protected func m() {} // Error: the access control of function which
     ↪ implements abstract function is lower than the abstract function
}
```

6.4 非顶层成员的访问修饰符

在修饰非顶层成员时不同访问修饰符的语义如下：

- private 表示仅当前类型或扩展定义内可见。
- internal 表示仅当前包及子包（包括子包的子包）内可见。
- protected 表示当前 module 及当前类的子类可见。
- public 表示 module 内外均可见。

	Type/Extend	Package & Sub-Packages	Module & Sub-Classes	All Packages
private	Y	N	N	N
internal	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

类型成员的访问修饰符可以不同于类型本身。除接口外类型成员的默认修饰符（默认修饰符是指在省略情况下的修饰符语义，这些默认修饰符也允许显式写出）是 internal，接口中的成员函数和属性不可以写访问修饰符，它们的访问级别等同于 public。

6.5 泛型在类与接口中使用的限制

6.5.1 实例化类型导致函数签名重复

在定义泛型类时，由于函数是可以重载的，所以下面的 C1 类与成员函数的定义是合法的。

```
open class C1<T> {
    public func c1(a: Int32) {}  // ok
    public func c1(a: T) {}  // ok
}

interface I1<T> {
```

```
    func i1(a: Int32): Unit // ok
    func i1(a: T): Unit // ok
}

var a = C1<Int32>() // error
class C2 <: C1<Int32> {...} // error
var b: I1<Int32> // error
class C3 <: I1<Int32> {...} // error
```

但是当类 C1<T> 需要被实例化为 C1<Int32> 时，会出现两个函数签名完全相同的情形，此时，在使用到 C1<Int32> 类型位置报错。同理，当接口 I1<Int32> 需要被实例化时，也会造成其部声明的两个函数签名重复，此时也会报错。

6.5.2 类与接口的泛型成员函数

在仓颉语言中，非静态抽象函数与类中被 open 关键字修饰的函数不能声明泛型参数。

```
interface Bar {
    func bar<T>(a: T): Unit // error
}

abstract class Foo {
    func foo<T>(a: T): Unit // error
    public open func goo<T>(a: T) { } // error
}

class Boo {
    public open func Boo<T>(a: T) { } // error
}
```

更多可以见第 9 章泛型 7.2 小节。

第七章　属性

属性是一种特殊的语法，它不像字段一样会存储值，相反它们提供了一个 getter 和一个可选的 setter 来间接检索和设置值。

通过使用属性可以将数据操作封装成访问函数，使用的时候与普通字段无异，我们只需要对数据操作，对内部的实现无感知，可以更便利地实现访问控制、数据监控、跟踪调试、数据绑定等机制。

属性在使用时语法与字段一致，可以作为表达式或被赋值。

以下是一个简单的例子，b 是一个典型的属性，封装了外部对 a 的访问：

```
class Foo {
    private var a = 0
    mut prop b: Int64 {
        get() {
            print("get")
            a
        }
        set(value) {
            print("set")
            a = value
        }
    }
}

main() {
    var x = Foo()
    let y = x.b + 1 // get
    x.b = y // set
}
```

7.1　属性的语法

属性的语法规则为：

```
propertyDefinition
    : propertyModifier* 'prop' identifier ':' type propertyBody?
```

```
    ;

propertyBody
    : '{' propertyMemberDeclaration+ '}'
    ;

propertyMemberDeclaration
    : 'get' '(' ')' block end*
    | 'set' '(' identifier ')' block end*
    ;

propertyModifier
    : 'public'
    | 'private'
    | 'protected'
    | 'internal'
    | 'static'
    | 'open'
    | 'override'
    | 'redef'
    | 'mut'
    ;
```

没有 mut 修饰符声明的属性需要定义 getter 实现。用 mut 修饰符声明的属性需要单独的 getter 和 setter 实现。

特别是，对于数值类型、Bool、Unit、Nothing、Rune、String、Range、Function、Enum 和 Tuple 类型，在它们的扩展或定义体中不能定义 mut 修饰的属性，也不能实现有 mut 属性的接口。

如下面的示例所示，a 是没有 mut 声明的属性，b 是使用 mut 声明的属性。

```
class Foo {
    prop a: Int64 {
        get() {
            0
        }
    }
    mut prop b: Int64 {
        get() {
            0
        }
        set(v) {}
    }
}
```

没有 mut 声明的属性没有 setter，而且与用 let 声明的字段一样，它们不能被赋值。

```
class A {
    prop i: Int64 {
        get() {
            0
        }
    }
}
main() {
    var x = A()
    x.i = 1 // error
}
```

特别是，当使用 let 声明 struct 的实例时，不能为 struct 中的属性赋值，就像用 let 声明的字段一样。

```
struct A {
    var i1 = 0
    mut prop i2: Int64 {
        get() {
            i1
        }
        set(value) {
            i1 = value
        }
    }
}
main() {
    let x = A()
    x.i1 = 2 // error
    x.i2 = 2 // error
}
```

属性与字段不同，属性不可以赋初始值，必须要声明类型。

7.2 属性的定义

属性可以在 interface, class，struct，enum，extend 中定义。

```
class A {
    prop i: Int64 {
        get() {
            0
        }
```

```
        }
    }

    struct B {
        prop i: Int64 {
            get() {
                0
            }
        }
    }

    enum C {
        prop i: Int64 {
            get() {
                0
            }
        }
    }

    extend A {
        prop s: String {
            get() {
                ""
            }
        }
    }
```

可以在 interface 和 abstract class 中声明抽象属性，它的定义体可以省略。在实现类型中实现抽象属性时，它必须保持相同的名称、相同的类型和相同的 mut 修饰符。

```
    interface I {
        prop a: Int64
    }

    class A <: I {
        public prop a: Int64 {
            get() {
                0
            }
        }
    }
```

如同 interface 中的抽象函数可以拥有默认实现，interface 中的抽象属性也同样可以拥有默认实现。

拥有默认实现的抽象属性，实现类型可以不提供自己的实现（必须符合默认实现的使用规则）。

```
interface I {
    prop a: Int64 { // ok
        get() {
            0
        }
    }
}
class A <: I {} // ok
```

属性分为实例成员属性和静态成员属性。其中，实例成员属性只能由实例访问，在 getter 或 setter 的实现中可以访问 this、实例成员和其它静态成员。而静态成员属性只能访问静态成员。

```
class A {
    var x = 0
    mut prop X: Int64 {
        get() {
            x + y
        }
        set(v) {
            x = v + y
        }
    }
    static var y = 0
    static mut prop Y: Int64 {
        get() {
            y
        }
        set(v) {
            y = v
        }
    }
}
```

属性不支持重载，也不支持遮盖，不能和其它同级别成员重名。

```
open class A {
    var i = 0
    prop i: Int64 { // error
        get() {
            0
        }
    }
```

```
}
class B <: A {
    prop i: Int64 { // error
        get() {
            0
        }
    }
}
```

7.3 属性的实现

属性的 getter 和 setter 分别对应两个不同的函数。

1. getter 函数类型是 `()->T`，T 是该属性的类型，当使用该属性作为表达式时会执行 getter 函数。
2. setter 函数类型是 `(T)->Unit`，T 是该属性的类型，形参名需要显式指定，当对该属性赋值时会执行 setter 函数。

属性的实现同函数的实现规则一样，其中可以包含声明和表达式，可以省略 `return`，其返回值必须符合返回类型。

```
class Foo {
    mut prop a: Int64 {
        get() { // () -> Int64
            "123" // error
        }
        set(v) { // (Int64) -> Unit
            123
        }
    }
}
```

无论在属性内部还是外部，访问属性的行为都是一致的，因此属性递归访问时与函数一样可能会造成死循环。

```
class Foo {
    prop i: Int64 {
        get() {
            i // dead loop
        }
    }
}
```

需要注意的是，struct 的 setter 是 mut 函数，因此也可以在 setter 内部修改其它字段的值，并且 `this` 会受到 mut 函数的限制。

7.4 属性的修饰符

属性跟函数一样可以使用修饰符修饰，但只允许对整个属性修饰，不能对 getter 或 setter 独立修饰。

```
class Foo {
    public mut prop a: Int64 { // ok
        get() {
            0
        }
        set(v) {}
    }
    mut prop b: Int64 {
        public get() { // error
            0
        }
        public set(v) {} // error
    }
}
```

属性可以使用访问控制修饰符有 private，protected，public。

```
class Foo {
    private prop a: Int64 { // ok
        get() { 0 }
    }
    protected prop b: Int64 { // ok
        get() { 0 }
    }
    public static prop c: Int64 { // ok
        get() { 0 }
    }
}
```

实例属性像实例函数一样，可以使用 open 和 override 修饰。
使用 open 修饰的属性，子类型可以使用 override 覆盖父类型的实现（override 是可选的）。

```
open class A {
    public open mut prop i: Int64 {
        get() { 0 }
        set(v) {}
    }
}
class B <: A {
    override mut prop i: Int64 {
```

```
            get() { 1 }
            set(v) {}
        }
    }
```

静态属性像静态函数一样，可以使用 redef 修饰（redef 是可选的），子类型可以重新实现父类型的静态属性。

```
    open class A {
        static mut prop i: Int64 {
            get() { 0 }
            set(v) {}
        }
    }
    class B <: A {
        redef static mut prop i: Int64 {
            get() { 1 }
            set(v) {}
        }
    }
```

子类型 override/redef 使用 let 声明的实例属性必须要重新实现 getter。

子类型 override/redef 父类型中使用 mut 修饰符声明的的属性时，允许只重新实现 getter 或 setter，但不能均不重新实现。

```
    open class A {
        public open mut prop i1: Int64 {
            get() { 0 }
            set(v) {}
        }
        static mut prop i2: Int64 {
            get() { 0 }
            set(v) {}
        }
    }

    // case 1
    class B <: A {
        public override mut prop i1: Int64 {
            get() { 1 } // ok
        }
        redef static mut prop i2: Int64 {
            get() { 1 } // ok
```

```
        }
    }

    // case 2
    class B <: A {
        public override mut prop i1: Int64 {
            set(v) {} // ok
        }
        redef static mut prop i2: Int64 {
            set(v) {} // ok
        }
    }

    // case 3
    class B <: A {
        override mut prop i1: Int64 {} // error
        redef static mut prop i2: Int64 {} // error
    }
```

子类型的属性 override/redef 必须与父类保持相同的 mut 修饰符，并且还必须保持相同的类型。

```
    class P {}
    class S {}

    open class A {
        open prop i1: P {
            get() { P() }
        }
        static prop i2: P {
            get() { P() }
        }
    }

    // case 1
    class B <: A {
        override mut prop i1: P { // error
            set(v) {}
        }
        redef static mut prop i2: P { // error
            set(v) {}
        }
    }
    // case 2
```

```
class B <: A {
    override prop i1: S { // error
        get() { S() }
    }
    redef static prop i2: S { // error
        get() { S() }
    }
}
```

子类 override 父类的属性时，可以使用 super 调用父类的实例属性。

```
open class A {
    open prop v: Int64 {
        get() { 1 }
    }
}
class B <: A {
    override prop v: Int64 {
        get() { super.v + 1 }
    }
}
```

第八章　扩展

扩展可以为除函数、元组、接口外的在当前 package 可见的任何类型添加新功能。
可以添加的功能包括:

- 添加实例成员函数
- 添加静态成员函数
- 添加操作符重载
- 添加实例成员属性
- 添加静态成员属性
- 实现接口

扩展是在类型定义之后追加的功能，扩展不能破坏原有类型的封装性，因此以下功能是禁止的。

1. 扩展不能增加字段。
2. 扩展不能增加抽象成员。
3. 扩展不能增加 open 成员。
4. 扩展不能 override/redef 原有的成员。
5. 扩展不能访问原类型的私有成员。

8.1　扩展语法

一个简单的扩展示例如下:

```
extend String {
    func printSize() {
        print(this.size)
    }
}

"123".printSize() // 3
```

扩展定义的语法如下:

```
extendDefinition
    : 'extend' extendType
      ('<:' superInterfaces)? genericConstraints?
```

```
        extendBody
    ;

extendType
    : (typeParameters)? (identifier NL* DOT  NL*)* identifier (NL* typeArguments)?
    | INT8
    | INT16
    | INT32
    | INT64
    | INTNATIVE
    | UINT8
    | UINT16
    | UINT32
    | UINT64
    | UINTNATIVE
    | FLOAT16
    | FLOAT32
    | FLOAT64
    | CHAR
    | BOOLEAN
    | NOTHING
    | UNIT
    ;

extendBody
    : '{' extendMemberDeclaration* '}'
    ;

extendMemberDeclaration
    : (functionDefinition
    | operatorFunctionDefinition
    | propertyDefinition
    | macroExpression
    ) end*
    ;
```

扩展的定义使用 extend 关键字，扩展定义依次为 extend 关键字、可选的泛型形参、被扩展的类型、可选的实现接口、可选的泛型约束，以及扩展体的定义。扩展体的定义不允许省略 {}。

扩展只能定义在 top level。

扩展分为直接扩展和接口扩展两种用法，直接扩展不需要声明额外的接口。

8.1.1 直接扩展

直接扩展不需要声明额外的接口，可以用来直接为现有的类型添加新功能。

```
class Foo {}

extend Foo {
    func f() {}
}

main() {
    let a = Foo()
    a.f() // call extension function
}
```

8.1.2 接口扩展

接口扩展可以用来为现有的类型添加新功能并实现接口，增强抽象灵活性。使用接口扩展时必须要声明被实现的接口。

```
interface I {
    func f(): Unit
}

class Foo {}

extend Foo <: I {
    public func f() {}
}
```

对一个类型使用接口扩展功能实现接口 `I`，其等价于在类型定义时实现接口 `I`，但使用范围会受到扩展导入导出的限制，详细见扩展的导入导出。

```
func g(i: I) {
    i.f()
}

main() {
    let a = Foo()
    g(a)
}
```

我们可以在同一个扩展内同时实现多个接口，多个接口之间使用 `&` 分开，接口的顺序没有先后关系。

```
interface I1 {
    func f1(): Unit
```

```
}

interface I2 {
    func f2(): Unit
}

interface I3 {
    func f3(): Unit
}

class Foo {}

extend Foo <: I1 & I2 & I3 {
    public func f1() {}
    public func f2() {}
    public func f3() {}
}
```

如果被扩展类型已经实现过某个接口，则不能通过扩展重复实现该接口，包含使用扩展实现过的接口。

```
interface I {
    func f(): Unit
}

class Foo <: I {
    func f(): Unit {}
}
extend Foo <: I {} // error, can not repeat the implementation of the interface

class Bar {}
extend Bar <: I {
    func f(): Unit {}
}
extend Bar <: I {} // error, already implemented through the extension can not
 ↪  repeat the implementation
```

如果被扩展类型已经直接实现某个非泛型接口，则不能使用扩展重新实现该接口；如果被扩展类型已经直接实现某个泛型接口，则不能使用相同的类型参数扩展重新实现该接口。

```
interface I1 {}

class Foo <: I1 {}

extend Foo <: I1 {} // error
```

```
interface I2<T> {}

class Goo<T> <: I2<Int32> {}

extend Goo<T> <: I2<Int32> {} // error

extend Goo<T> <: I2<T> {} // ok
```

如果被扩展的类型已经包含接口要求的函数，则接口扩展不能再重新实现这些函数，也不会再使用接口中的默认实现。

```
class Foo {
    public func f() {}
}

interface I {
    func f(): Unit
}

extend Foo <: I {} // ok

extend Foo {
    public func g(): Unit {
        print("In extend!")
    }
}

interface I2 {
    func g(): Unit {
        print("In interface!")
    }
}

extend Foo <: I2 {} // ok, default implementation of g in I2 is no longer used
```

禁止定义孤儿扩展，孤儿扩展指的是接口扩展既不与接口（包含接口继承链上的所有接口）定义在同一个包中，也不与被扩展类型定义在同一个包中。

即接口扩展只允许以下两种情况：

1. 接口扩展与类型定义处在同一个包。
2. 接口扩展实现的接口，和接口的继承链上所有未被被扩展类型实现的接口，都必须在同一个包中。

其它情况的接口扩展都是不允许定义的。

```
// package pkg1
public class Foo {}
public class Goo {}

// package pkg2
public interface Bar {}
extend Goo <: Bar {}

// package pkg3
import pkg1.Foo
import pkg2.Bar

extend Foo <: Bar {} // error

interface Sub <: Bar {}

extend Foo <: Sub {} // error

extend Goo <: Sub {} // ok, 'Goo' has implemented the interface 'Bar' on the
 ↪  inheritance chain in pkg2.
```

8.2 扩展的成员

扩展的成员包括：静态成员函数、实例成员函数、静态成员属性、实例成员属性、操作符重载函数。

8.2.1 函数

扩展可以对被扩展类型添加函数，这些函数可以是泛型函数，支持泛型约束，支持重载，也支持默认参数和命名参数。这些函数都不能是抽象函数。

例如：

```
interface I {}
extend Int64 {
    func f<T>(a: T, b!: Int64 = 0) where T <: I {}
    func f(a: String, b: String) {}
}
```

8.2.1.1 修饰符

扩展内的函数定义支持使用 private，protected（仅限于被扩展类型是 class 类型）或 public 修饰。

使用 private 修饰的函数只能在本扩展内使用，外部不可见。

使用 protected 修饰的成员函数除了能在本包内被访问，对包外的当前 class 子类也可以访问。

没有使用 private，protected 或 public 修饰的函数只能在本包内使用。

```
// file1 in package p1
package p1

public open class Foo {}

extend Foo {
    private func f1() {}   // ok
    public func f2() {}    // ok
    protected func f3() {} // ok
    func f4() {}           // visible in the package
}

main() {
    let a = Foo()
    a.f1() // error, can not access private function
    a.f2() // ok
    a.f3() // ok
    a.f4() // ok
}

// file2 in package p2
package p2

import p1.*

class Bar <: Foo {
    func f() {
        f1() // error, can not access private function
        f2() // ok
        f3() // ok
        f4() // error, can not access default function
    }
}
```

扩展内的函数支持使用 `static` 修饰。

```
class Foo {}

extend Foo {
    static func f() {}
}
```

对 struct 类型的扩展可以定义 `mut` 函数。

```
struct Foo {
    var i = 0
}

extend Foo {
    mut func f() { // ok
        i += 1
    }
}
```

扩展内的函数定义不支持使用 open、override、redef 修饰。

```
class Foo {
    public open func f() {}
    static func h() {}
}

extend Foo {
    public override func f() {} // error
    public open func g() {} // error
    redef static func h() {} // error
}
```

8.2.2 属性

扩展可以对被扩展类型添加属性。这些属性都不能是抽象属性。
例如：

```
extend Int64 {
    mut prop i: Int64 {
        get() {
            0
        }
        set(value) {}
    }
}
```

8.2.2.1 修饰符

扩展内的属性定义支持使用 private，protected（仅限于被扩展类型是 class 类型）或 public 修饰。
使用 private 修饰的属性只能在本扩展内使用，外部不可见。
使用 protected 修饰的属性除了能在本包内被访问，对包外的当前 class 子类也可以访问。
没有使用 private，protected 或 public 修饰的属性只能在本包内使用。

```
// file1 in package p1
package p1

public open class Foo {}

extend Foo {
    private prop v1: Int64 {   // ok
        get() { 0 }
    }
    public prop v2: Int64 {    // ok
        get() { 0 }
    }
    protected prop v3: Int64 { // ok
        get() { 0 }
    }
    prop v4: Int64 {           // visible in the package
        get() { 0 }
    }
}

main() {
    let a = Foo()
    a.v1 // error, can not access private property
    a.v2 // ok
    a.v3 // ok
    a.v4 // ok
}

// file2 in package p2
package p2

import p1.*

class Bar <: Foo {
    func f() {
        v1 // error, can not access private function
        v2 // ok
        v3 // ok
        v4 // error, can not access default function
    }
}
```

扩展内的属性支持使用 static 修饰。

```
class Foo {}

extend Foo {
    static prop i: Int64 {
        get() { 0 }
    }
}
```

扩展内的属性定义不支持使用 open、override、redef 修饰。

```
class Foo {
    open prop v1: Int64 {
        get() { 0 }
    }
    static prop v2: Int64 {
        get() { 0 }
    }
}

extend Foo {
    override prop v1: Int64 { // error
        get() { 0 }
    }
    open prop v3: Int64 { // error
        get() { 0 }
    }
    redef static prop v2: Int64 { // error
        get() { 0 }
    }
}
```

8.3 泛型扩展

如果被扩展的类型是泛型类型，有两种扩展语法可以对泛型类型扩展功能。

一种是上面介绍的非泛型扩展。对于泛型类型，非泛型扩展可以针对特定的泛型实例化类型进行扩展。

在 extend 后允许是一个任意实例化完全的泛型类型。为这些类型增加的功能只有在类型完全匹配时才能使用。

```
extend Array<String> {} // ok
extend Array<Int> {} // ok
```

在扩展中，泛型类型的类型实参必须符合泛型类型定义处的约束要求，否则会编译报错。

```
class Foo<T> where T <: ToString {}

extend Foo<Int64> {} // ok

class Bar {}
extend Foo<Bar> {} // error
```

另一种是在 extend 后面引入泛型形参的泛型扩展。泛型扩展可以用来扩展未实例化或未完全实例化的泛型类型。

在 extend 后允许声明泛型形参，这些泛型形参必须被直接或间接使用在被扩展的泛型类型上。为这些类型增加的功能只有在类型和约束完全匹配时才能使用。

```
extend<T> Array<T> {} // ok
```

泛型扩展引入的泛型变元必须在被扩展类型中使用，否则报未使用错误。

```
extend<T> Array<T> {} // ok
extend<T> Array<Option<T>> {} // ok
extend<T> Array<Int64> {} // error
extend<T> Int64 <: Equatable<T> { // error
    ...
}
```

不管是泛型扩展还是非泛型扩展，对泛型类型扩展都不能重定义成员和重复实现接口。

当泛型扩展的泛型变元被直接使用在被扩展类型的类型实参时，对应的泛型变元会隐式引入被扩展类型定义时的泛型约束。

```
class Foo<T> where T <: ToString {}

extend<T> Foo<T> {} // T <: ToString
```

泛型扩展引入的泛型形参不能用于被扩展类型或被实现的接口，否则会编译报错。

```
extend<T> T {} // error
extend<T> Unit <: T {} // error
```

我们可以在泛型扩展中使用额外的泛型约束。通过这种方式添加的成员或接口，只有当该类型的实例在满足扩展的泛型约束时才可以使用，否则会报错。

```
class Foo<T> {
    var item: T
    init(it: T) {
        item = it
    }
}
```

```
interface Eq<T> {
    func equals(other: T): Bool
}

extend<T> Foo<T> <: Eq<Foo<T>> where T <: Eq<T> {
    public func equals(other: Foo<T>) {
        item.equals(other.item)
    }
}

class A {}
class B <: Eq<B> {
    public func equals(other: B) { true }
}

main() {
    let a = Foo(A())
    a.equals(a) // error, A has not implement Eq
    let b = Foo(B())
    b.equals(b) // ok, B has implement Eq
}
```

泛型类型不能重复实现同一接口。对于同一个泛型类型，无论它是否完全实例化，都不能重复实现同一接口。如果重复实现会在编译时报错。

```
interface I {}
extend Array<String> <: I {} // error, can not repeatedly implement the same
 ↪ interface
extend<T> Array<T> <: I {} // error, can not repeatedly implement the same
 ↪ interface

interface Bar<T> {}
extend Array<String> <: Bar<String> {} // error, can not repeatedly implement the
 ↪ same interface
extend<T> Array<T> <: Bar<T> {} // error, can not repeatedly implement the same
 ↪ interface
```

对于同一个泛型类型，无论它是否完全实例化，都不能定义相同类型或相同签名的成员。如果重复定义会在编译时报错。

```
// case 1
extend Array<String> {
    func f() {} // error, cannot be repeatedly defined
}
```

```
extend<T> Array<T> {
    func f() {} // error, cannot be repeatedly defined
}
// case 2
extend Array<String> {
    func g(a: String) {} // error, cannot be repeatedly defined
}
extend<T> Array<T> {
    func g(a: T) {} // error, cannot be repeatedly defined
}
// case 3
extend<T> Array<T> where T <: Int {
    func g(a: T) {} // error, cannot be repeatedly defined
}
extend<V> Array<V> where V <: String {
    func g(a: V) {} // error, cannot be repeatedly defined
}
// case 4
extend Array<Int>  {
    func g(a: Int) {} // ok
}
extend Array<String>  {
    func g(a: String) {} //ok
}
```

8.4 扩展的访问和遮盖

扩展的实例成员与类型定义处一样可以使用 this，this 的含义与类型定义中的保持一致。同样也可以省略 this 访问成员。

扩展的实例成员不能使用 super。

```
class A {
    var v = 0
}

extend A {
    func f() {
        print(this.v) // ok
        print(v) // ok
    }
}
```

扩展不能访问被扩展类型的 private 成员，其它修饰符修饰的成员遵循可见性原则。

```
class A {
    private var v1 = 0
    protected var v2 = 0
}

extend A {
    func f() {
        print(v1) // error
        print(v2) // ok
    }
}
```

扩展不允许遮盖被扩展类型的任何成员。

```
class A {
    func f() {}
}

extend A {
    func f() {} // error
}
```

扩展也不允许遮盖被扩展类型的其它扩展中已增加的任何成员。

```
class A {}

extend A {
    func f() {}
}

extend A {
    func f() {} // error
}
```

在同一个 package 内对同一类型可以扩展任意多次。
在扩展中可以直接使用（不加任何前缀修饰）其它对同一类型的扩展中的非 private 修饰的成员。

```
class Foo {}

extend Foo { // OK
    private func f() {}
    func g() {}
}
```

```
extend Foo { // OK
    func h() {
        g() // OK
        f() // Error
    }
}
```

扩展泛型类型时，可以使用额外的泛型约束。

泛型类型的扩展中能否直接使用其它对同一类型的扩展中的成员，除了满足上述可访问性规则之外，还需要满足以下约束规则：

- 如果两个扩展的约束相同，则两个扩展中可以直接使用对方的成员；
- 如果两个扩展的约束不同，且两个扩展的约束有包含关系，约束更严格的扩展中可以直接使用约束更宽松的扩展中的成员，反之，则不可直接使用；
- 当两个扩展的约束不同时，且两个约束不存在包含关系，则两个扩展中不可以直接使用对方的成员。

示例：假设对同一个类型 E<X> 的两个扩展分别为扩展 1 和扩展 2，X 的约束在扩展 1 中比扩展 2 中更严格，那扩展 1 中可直接使用扩展 2 中的成员，反之，扩展 2 中不可直接使用扩展 1 的成员。

```
// B <: A
class E<X> {}

interface I1 {
    func f1(): Unit
}
interface I2 {
    func f2(): Unit
}

extend<X> E<X> <: I1 where X <: B {  // extension 1
    public func f1(): Unit {
        f2() // OK
    }
}

extend<X> E<X> <: I2 where X <: A   { // extension 2
    public func f2(): Unit {
        f1() // Error
    }
}
```

8.5 扩展的继承

如果被扩展的类型是 class，那么扩展的成员会被子类继承。

```
open class A {}

extend A {
    func f() {}
}

class B <: A {
    func g() {
        f() // ok
    }
}

main() {
    let x = B()
    x.f() // ok
}
```

需要注意的是，如果在父类中扩展了成员，由于继承规则限制，子类中就无法再定义同名成员，同时也无法覆盖或重新实现（允许函数重载）。

```
open class A {}

extend A {
    func f() {}
    func g() {}
}

class B <: A {
    func f() {} // error
    override func g() {} // error
}
```

如果在同包内对同一个类型分别扩展实现父接口和子接口，那么编译器会先检查实现父接口的扩展，然后再检查实现子接口的扩展。

```
class Foo {}

interface I1 {
   func f() {}
}
interface I2 <: I1 {
   func g() {}
}
```

```
extend Foo <: I1 {} // first check
extend Foo <: I2 {} // second check
```

8.6 扩展的导入导出

extend 前不允许有修饰符，扩展只能与被扩展类型或接口一起导入导出。

8.6.1 直接扩展的导出

当直接扩展与被扩展类型的定义处在相同的 package 时，如果被扩展类型是导出的，扩展会与被扩展类型一起被导出，否则扩展不会被导出。

```
package pkg1

public class Foo {}

extend Foo {
    public func f() {}
}

///////

package pkg2
import pkg1.*

main() {
    let a = Foo()
    a.f() // ok
}
```

当直接扩展与被扩展类型的定义处在不同的 package 时，扩展永远不会被导出，只能在当前 package 使用。

```
package pkg1
public class Foo {}

///////

package pkg2
import pkg1.*

extend Foo {
```

```
    public func f() {}
}

func g() {
    let a = Foo()
    a.f() // ok
}

///////

package pkg3
import pkg1.*
import pkg2.*

main() {
    let a = Foo()
    a.f() // error
}
```

8.6.2 接口扩展的导出

当接口扩展与被扩展类型在相同的包时，对于外部可见类型只可以扩展同样外部可见的接口。

```
package pkg1
public class Foo {}

interface I1 {
    func f(): Unit
}

extend Foo <: I1 { // error
    public func f(): Unit {}
}

public interface I2 {
    func g(): Unit
}

extend Foo <: I2 { // ok
    public func g(): Unit {}
}
```

```
///////

package pkg2
import pkg1.*

main() {
    let a = Foo()
    a.g() // ok
}
```

当接口扩展与被扩展类型不在相同的包时，扩展的访问等级与相应接口相同。如果扩展多个接口，扩展的访问等级是 public 当且仅当所有接口都是 public 的。

```
package pkg1
public class Foo {}

public class Bar {}

///////

package pkg2

import pkg1.*

interface I1 {
    func f(): Unit
}

public interface I2 {
    func g(): Unit
}

extend Foo <: I1 & I2 { // not external
    public func f(): Unit {}
    public func g(): Unit {}
}

extend Bar <: I2 { // external
    public func g(): Unit {}
}
```

8.6.3 导入扩展

扩展会与被扩展的类型和扩展实现的接口一起被导入，不能指定导入扩展。

```
package pkg1
public class Foo {}

extend Foo {
    public func f() {}
}

///////

package pkg2
import pkg1.Foo // import Foo

main() {
    let a = Foo()
    a.f() // ok
}
```

特别的，由于扩展不支持遮盖任何被扩展类型的成员，当导入的扩展发生重定义时将会报错。

```
package pkg1
public class Foo {}

///////

package pkg2
import pkg1.Foo

public interface I1 {
    func f(): Unit {}
}

extend Foo <: I1 {} // ok, external

///////

package pkg3
import pkg1.Foo

public interface I2 {
    func f(): Unit {}
```

```
}

extend Foo <: I2 {} // ok, external

///////

package pkg4
import pkg1.Foo
import pkg2.I1 // error
import pkg3.I2 // error
```

第九章　泛型

如果有一个声明，在该声明中使用尖括号声明了类型形参，那么称这个声明是泛型的。在使用泛型声明时，类型形参可以被代换为其他的类型。通过在函数签名中声明类型形参，可以定义一个泛型函数；通过在 class、interface、struct、enum、typealias 定义中声明类型形参，可以定义泛型类型。

9.1　类型形参与类型变元

在仓颉编程语言中，用标识符表示类型形参，并用 <> 括起。通过在 <> 内用，分隔多个类型形参名称，可以提供多个类型形参，如 <T1，T2，T3>，T1，T2，T3 均为类型形参。

一旦声明了类型形参，这些形参就可以被当做类型来使用。

当使用标识符来引用声明的类型形参时，这些标识符被称为类型变元。

类型形参的语法如下：

```
typeParameters
    : '<' identifier (',' identifier)* '>'
    ;
```

9.2　泛型约束

在仓颉语言中可以用 <: 来表示一个类型是另一个类型的子类型。通过声明这种关系可以对泛型类型形参声明加以约束，使得它只能被替换为满足特定约束的类型。

泛型约束通过 where 之后的 <: 运算符来声明，由一个下界与一个上界来组成。其中 <: 左边称为约束的下界，下界只能为类型变元。<: 右边称为约束上界，约束上界可以为类型。当一个约束的上界是类型时，该约束为子类型约束。当上界为类型时，上界可以为任何类型。

一个类型变元可能同时受到多个上界约束，对于同一个类型形参的多个上界必须使用 & 连接，以此来简化一个类型形参有多个上界约束的情形，它本质上还是多个泛型约束。对不同类型变元的约束需要使用，分隔。

声明约束的语法如下：

```
upperBounds
    :  type ('&' type)*
    ;

genericConstraints
```

```
    : 'where' identifier  '<:' upperBounds (',' identifier  '<:' upperBounds)*
    ;
```

例如以下示例展示了泛型约束声明的写法:

```
interface Enumerable<U> where U <: Bounded {...}

interface Comparable<T> {...}

func collectionCompare<T>(a: T, b: T) where T <: Comparable<T> & Seqence {...}

func sort<T, V>(input: T)
    where T <:  Enumerable<V>, V <: Object {...}
```

类型变元 X 和 Y 的约束相同，指的是所有满足 X 约束的类型都满足 Y 的约束，且所有满足 Y 的约束的类型也都满足 X 的约束;

类型变元 X 比 Y 的约束更严格，指的是所有满足 X 约束的类型都满足 Y 的约束，反之，不一定满足;

如果 X 比 Y 的约束更严格，则 Y 比 X 的约束更宽松。

两个泛型类型的约束相同，指的是泛型类型的类型变元数量相同，且所有对应的类型变元约束均相同;

一个泛型类型 A 的约束比另一个泛型类型 B 的约束更严格，指的是 A 和 B 的类型变元数量相同，且 A 的所有类型变元的约束均比 B 中对应的类型变元更严格;

一个泛型类型 A 的约束比另一个泛型类型 B 的约束更严格，则 B 的约束比 A 更宽松。

比如下面的两个泛型 C 和 D，假设有 I1 <: I2，C 的约束比 D 更严格:

`class C<X, Y> where X <: I1, Y <: I1` 和 `class D<X, Y> where X <: I2, Y <: I2`

9.3 类型型变

在正式介绍泛型函数与泛型类型前，先简单介绍以下类型型变，以此来说明在仓颉编程语言中，泛型类型的子类型关系。

9.3.1 定义

如果 A 和 B 是类型，T 是类型构造器，设其有一个类型参数 X，那么:

- 如果 T(A) <: T(B) 当且仅当 A <: B，则 T 在 X 处是协变的。
- 如果 T(A) <: T(B) 当且仅当 B <: A，则 T 在 X 处是逆变的。
- 如果 T(A) <: T(B) 当且仅当 A = B，则 T 是不型变的。

9.3.2 泛型不型变

在仓颉编程语言中，所有的泛型都是不型变的。这意味着如果 A 是 B 的子类型，ClassName<A> 和 ClassName<B> 之间没有子类型关系。我们禁止这样的行为以保证运行时的安全。

9.3.3 函数类型的型变

函数的参数类型是逆变的，函数的返回类型是协变的。假设存在函数 f1 的类型是 S1 -> T1，函数 f2 的类型是 S2 -> T2。如果 S2 <: S1 并且 T1 <: T2，则 f1 的类型是 f2 的类型的子类型。

9.3.4 元组类型的协变

元组之间是存在子类型关系的，如果一个元组的每一个元素都是另一个元组的对应位元素的子类型，则该元组是另一个元组的子类型。假设有元组 Tuple1 和 Tuple2，它们的类型分别为 (A1, A2.., An)、(B1, B2.., Bn)，如果对于所有 i 都满足 Ai <: Bi，则 Tuple1 <: Tuple2。

9.3.5 型变的限制

现在以下两种情况的型变关系被禁止：

1. class 以外的类型实现接口，该类型和该接口之间的子类型关系不能作为协变和逆变的依据。
2. 实现类型通过扩展实现接口，该类型和该接口之间的子类型关系不能作为协变和逆变的依据。

这些限制除了影响型变关系以外，同时也会影响 override 对于子类型的判定。
不满足型变关系的类型，在发生 override 时不能作为子类型的依据。

```
interface I {
    func f(): Any
}

class Foo <: I {
    func f(): Int64 { // error
        ...
    }
}
```

9.4 泛型约束上界中导出的约束

对于一个约束 L <: T<T1..Tn>，其中的上界 T<T1..Tn> 的声明 T 的类型形参可能还需要满足一些约束，在实参 Ti 的代换后，这些约束需要被隐式地引入到当前声明的上下文中。例如：

```
interface Eq<T> {
    func eq(other: T): Bool
}

interface Ord<T> where T <: Eq<T> {
    func lt(other: T): Bool
}
```

```
func foo<T>(a: T) where T <: Ord<T> {
    a.eq(a)
    a.lt(a)
}
```

对于 foo 函数，虽然只声明了 T 受到 Ord 的约束，但是由于 Ord 的 T 类型受到了 Eq 的约束，所以在 foo 函数里是可以使用 Eq 中的 eq 函数的。这样，foo 函数的约束实际上是 T <: Eq & Ord。这样在声明一个泛型参数满足一个约束时，这一约束的上界中需要满足的约束也将被引入。

对于其他泛型声明，隐式地引入约束上界的约束这一规则也是有效的。例如：

```
interface A {}
class B<T> where T <: A {}
class C<U> where U <: B<U> {} // actual constraints are U <: A & B<U>
```

这里，对于类 C，它的泛型形参 U 所受到的约束实际上为 U <: A & B<U>。

注意：虽然当前声明中上界的约束会被隐式地引入，但当前声明仍然可以将这些约束显式地写出。

9.5 泛型函数与泛型类型的定义

9.5.1 泛型函数

如果一个函数声明了一个或多个类型形参，则将其称为泛型函数。语法上，类型形参紧跟在函数名后，并用 <> 括起，如果有多个类型形参，则用，分离。

```
func f<T>() {...}
func f1<T1, T2, T3, ...>() {...}
```

泛型函数的语法如下：

```
functionDefinition
    : functionModifierList? 'func' identifier
    typeParameters functionParameters
    (':' type)? genericConstraints?
    block?
    ;
```

需要注意的是：

- < 与 > 在使用时，会优先解析为泛型，如果成功，则直接就是泛型表达式，否则才为比较运算符，例如：

```
(c <d , e> (f))
```

这一表达式会被解析为函数调用表达式。

9.5.2 泛型类型

如果一个 class、interface、struct、enum、typealias 的定义中声明了一个或多个类型形参，则它们被称为泛型类型。语法上，类型形参紧跟在类型名（如类名、接口名等）后，并用 <> 括起，如果有多个类型形参，则用 “,” 分隔。

```
class C<T1, T2> {...}   // generic class
interface I<T1> {...}   // generic interface
type BinaryOperator<T> = (T, T) -> T    // generic typealias
```

对于泛型声明定义的语法可以参考相应的章节。

9.6 泛型类型检查

9.6.1 泛型声明的检查

9.6.1.1 泛型约束的健全性检查

对一个声明的所有类型形参，其每个形参的约束上界可以分为两种情况：

1. 上界也是类型变元，这个类型变元可能是它本身，也可能是其他的类型变元
2. 上界为具体类型时，可以分为两种情形：
    - 第一种情形是上界 class 与 interface 类型时，称为类相关类型。
    - 第二种情形是上界为除 class 与 interface 类型以外的类型，这些称为类无关类型

在仓颉语言中，对于一个类型变元的一个或多个具体类型上界需要满足如下规则：

1. 所有的约束上界只能属于同一种情形，即要么上界都是类相关类型，要么是类无关类型。例如：`T <: Object & Int32` 不合法。
2. 当上界是类相关类型时，如果存在多个类的上界，那么这些类需要在同一个继承链上，对于接口没有此限制。一个类型变元的多个泛型上界中不允许包含冲突的成员定义，具体来说，冲突指同名函数或相同操作符之间不构成重载，并且返回类型之间不具有子类关系。
3. 当上界是类无关类型的情形时，只能包含一种类无关的具体类型。不能同时为两个不同的具体类型。例如：`T <: Int32 & Bool` 不合法。
4. 类型变元上界为类无关类型的情形时不能存在递归约束。递归泛型约束是上界类型实参直接或间接依赖于下界类型变元自身的约束。例如：`T <: Option<T>`，由于 `Option` 是通过 `enum` 关键字声明的，所以此种递归泛型约束不合法。`T <: U, U <: (Int32) -> T` 也不合法，因为函数是值类型，`T` 类型间接地通过 `U` 依赖了自身。

9.6.1.2 类型兼容性检查

对于泛型声明的类型的检查主要是检查泛型类型与其所在的类型上下文中是否兼容，对于成员函数、变量的访问是否合法，例如：

```
open class C {
    func coo() {...}
}

class D <: C {...}

interface Tr {
    func bar(): Int64
}

func foo<T>(a: T) {
    var b: C = a // error, T is not a subclass of C
    a.coo() // error, T has no member function coo
    a.bar() // error, T did not implement Tr
}
```

在上述示例代码的 foo 的函数体中共有 3 处报错，原因分别如下：

1. 由于 foo 函数中声明的变量 b 的期望的类型是 C，所以这里需要检查 T 是否是 D 的子类型，即 T <: C，而这一约束不存在于 T 的上下文中，所以变量声明处编译报错。
2. 由于泛型类型 T 在当前上下文与 C 无关，所以也不能访问 C 的成员函数 coo。
3. 类似地，由于 T 类型的不存在 Tr 的约束，所以也不能访问 Tr 的成员函数 bar。

如果想要通过类型检查需要在声明体前加入泛型约束：

```
open class C {
    func coo() {...}
}

interface Tr {
    func bar(): Int64
}

func foo<T>(a: T) where T <: C & Tr {
    var b: C = a // OK, T is a sub type of C now
    a.coo() // OK, T is a sub type of C, so it has coo member function
    a.bar() // OK, T is constrained by Tr
}
```

特别地，如果一个类型变元 T 的泛型上界包含一个函数类型或重载了函数调用操作符 () 的类型，则类型为 T 的值可以被作为函数调用。当上界为函数类型时，该调用的返回类型为 T 的上界的返回类型，当上界为重载了函数调用操作符 () 的类型时，该调用的返回类型为上界类型中匹配的函数调用操作符的返回类型。

9.6.2 泛型声明使用的检查

对于泛型声明的使用检查主要是将实参代入到泛型声明的形参，然后检查约束是否成立。

如果我们直接用 C 类型调用上一小节中定义的 foo 函数：

```
main(): Int64 {
    foo<C>(C()) // error C does not implement Tr
    return 0
}
```

那么会得到类型 C 没有实现 Tr 的错误，这是因为在 foo 函数的约束中有 T <: C & Tr，其中的形参 T 会被代替为 C，首先 C <: C 成立，但是 C <: Tr 不成立。

如果为 C 类型加入了实现 Tr 的声明，那么就可以满足 T <: Tr 这一约束：

```
extend C <: Tr {
    func bar(): Int64 {...}
}
```

特别的是，当 interface 作为泛型约束时，调用时泛型变元实例化的类型必须完全实现所有上界约束中的 interface static 函数。

意味着如果作为泛型约束的 interface 中存在 static 函数，就无法将未实现对应 static 函数的 interface 或抽象类作为泛型变元实例化的类型。

```
interface I {
    static func f(): Unit
}

func g<T>(): Unit where T <: I {}

main() {
    g<I>() // error
    return 0
}
```

9.6.3 泛型实例化的深度

为保证泛型实例化不会出现死循环或耗尽内存，在编译过程中会对实例化的层数做出限制。例如：

```
class A<T>{
    func test(a: A<(A<T>, A<T>)>): Bool {true}
}

main(): Int64 {
    var a : A<Int32> = A<Int32>()
    return 0
}
```

这段程序会报 `infinite instantiation` 的错误。

9.7 泛型实例化

一个泛型声明在所有类型形参的取值都确定之后形成一个对应的非泛型语言结构的过程称之为泛型声明的实例化。

9.7.1 泛型函数的实例化

```
func show<T>(a: T) where T <: ToString {
    a.toString()
}
```

在给定 `T = Int32` 的取值之后会形成这样的实例（这里假定 `show$Int32` 是编译器实例化后的内部表示，后面也会使用类似的表示）：

```
func show$Int32(a: Int32) {
    a.toString()
}
```

9.7.1.1 实例化泛型函数的限制

在仓颉语言中，以下情形不能声明泛型函数：

1. 接口与抽象类中的非静态抽象函数
2. 类与抽象类中被 open 关键字修饰的实例成员函数
3. 操作符重载函数

例如，以下函数的声明与定义都是不合法的：

```
abstract class AbsClass {
    public func foo<T>(a: T): Unit // error: abstract generic function in abstract
    ↪ class
    public open func bar<T>(a: T) { // error: open generic function in abstract
    ↪ class
        ...
    }
}

interface IF {
    func foo<T>(a: T): Unit // error: abstract generic function in interface
}

open class Foo {
```

```
    public open func foo<T>(a: T): Unit { // error: open generic function in class
        ...
    }
}
```

而以下的泛型函数是合法的:

```
class Foo {
    static func foo<T>(a: T) {...} // generic static function in class
    func bar<T>(a: T) {...} // generic non-open function in class
}

abstract class Bar {
    func bar<T>(a: T) {...} // generic non-open function in abstract class
}

struct R {
    func foo<T>(a: T) {...} // generic function in struct
}

enum E {
    A | B | C
    func e<T>(a: T) {...} // generic function in enum
}
```

9.7.2 类与接口的实例化

```
class Foo<T> <: IBar<T>{
    var a: T
    init(a: T) {
        this.a = a
    }
    static func foo(a: T) {...}
    public func bar(a: T, b: Int32): Unit {...}
}

interface IBar<T> {
    func bar(a: T, b: Int32): Unit
}
```

在给定 T=Int32 时会生成以下实例的声明:

```
class Foo$Int32 <: IBar$Int32 {
    var a: Int32
```

```
    static func foo(a: Int32) {...}
    func bar(a: Int32) {...}
}

interface IBar$Int32 {
    func bar(a: Int32, b: Int32)
}
```

9.7.3 struct 的实例化

结构体的实例化与类的实例化十分类似。

```
struct Foo<T> {
    func foo(a: T) {...}
}
```

当给定 T=Int32 时会生成以下实例的声明：

```
struct Foo$Int32 {
    func foo(a: Int32) {...}
}
```

9.7.4 Enum 的实例化

```
enum Either<T,R> {
      Left(T)
    | Right(R)
}
```

当 Either 被给定参数 Int32 与 Bool 时，类型在被实例化后得到：

```
enum Either$Int32$Bool {
      Left(Int32)
    | Right(Bool)
}
```

在使用一个泛型声明时，例如调用泛型函数、构造泛型类型的值等，在编译时实际发生作用的都是确定类型形参后的实例，也就是说只有当所有泛型参数都为具体类型后才会发生实例化。

9.8 泛型函数重载

在仓颉编程语言中，支持泛型函数之间的重载，也支持泛型函数与非泛型函数之间的重载，重载的定义详见函数重载。

函数调用时，重载的处理过程如下：

第一步：构建函数调用的候选集，最终进入候选集的函数均为通过类型检查可以被调用的函数，详见重载函数候选集，在构建候选集时，对于泛型函数有额外的规则，下面会详细介绍；

第二步：根据作用域优先级规则（详见作用域优先级）和最匹配规则（详见最匹配规则）选择最匹配的函数，如果无法确定唯一的最匹配函数，则报无法决议的错误；

第三步：如果实参类型有多个，根据最匹配函数确定实参类型，如果不能确定唯一的实参类型，则报错。

构建函数调用的候选集时，对于泛型函数需要注意以下几点：

1、在函数调用时，对于泛型函数 `f`，进入候选集的可能是部分实例化后的泛型函数或是完全实例化的函数。具体是哪种形式进入候选集，由调用表达式的形式决定：

- 调用表达式的形式为：`C<TA>.f(A)`，即 `f` 为某个泛型类型的静态成员函数，先对类型进行实例化；再对实例化后类型的静态成员函数进行函数调用的类型检查，如果能通过则进入候选集。假设 `C` 的类型形参为 `X`，则进入候选集的函数是将 `f` 中 `X` 代换成 `TA` 之后的 `f'`：

$$\sigma = [X \mapsto TA]$$

$$f' = \sigma f$$

```
// The context contains the following types: Base, Sub, and Sub <: Base.
class F<X> {
   static func foo<Y>(a: X, b: Y) {}  // foo1
   static func foo<Z>(a: Sub, b: Z) {}  // foo2
}

/* The functions that enter the candidate set are foo1 and partial instantiated
 ↪  foo2: foo<Y>(a:Base, b: Y) and foo<Z>(a: Sub, b: Z) */
var f = F<Base>.foo(Sub(), Base())  // foo2
```

- 调用表达式的形式为：`obj.f(A)`，且 `obj` 为泛型类型实例化类型的实例，即 `f` 为某个泛型类型的非静态成员函数。

  在该表达式中 `obj` 的类型需要先确定，再根据 `obj` 的类型来获取候选集函数。`obj` 的类型是实例化后的类型，也是将实例化后的类型的非静态成员函数进行函数调用的类型检查，通过类型检查的进入候选集。

```
// The context contains the following types: Base, Sub, and Sub <: Base.
class C<T, U>{
    init (a: T, b: U) {}
    func foo(a: T, b: U) {}     // foo1
    func foo(a: Base, b: U) {}   // foo2
}

/* It is inferred that the type of obj is C<Sub, Rune>.
   The functions that enter the candidate set are instantiated foo1, foo2:
   foo(a:Sub, b:Rune) and foo(a: Base, b: Rune)
```

```
 */
main() {
    C(Sub(), 'a').foo(Sub(), 'a')   // choose foo1
    return 0
}
```

2、如果函数调用时未提供类型实参，也就是函数调用的形式为：`f(a)`，则要求进入候选集的泛型函数 `f` 满足以下要求：

- 如果 `f` 是泛型函数，其形式为：`f<X_1,...,X_m>(p1: T_1, ..., pn: T_n): R`，调用表达式中未提供类型实参，形式为 `f(a_1, ..., a_n)`，`f` 可以根据实参的类型 `(A1,...,An)` 能推断出一组类型实参 `TA_1,...,TA_m`，满足 `f` 的所有泛型约束，且将 `f` 中的 `X_1,...,X_m` 分别代换成 `TA_1,...,TA_m`，能通过函数调用的类型检查，检查规则如下：
  - 将推断出的类型实参 `TA_1,...,TA_m` 代换到 `f` 的函数形参 `(T1,...,Tn)` 后，满足在调用表达式所在的上下文中 `(A1,...,An)` 是代换后形参类型的子类型：

$$\sigma = [X_1 \mapsto TA_1, ..., X_m \mapsto TA_m]$$
$$\Delta \vdash (A1, ..., An) <: \sigma(T_1, ..., T_n)$$

  - 如果调用表达式中提供了返回类型 `RA`，则需要根据返回类型进行类型检查，将 `f` 的返回类型 `R` 中的 `X_1,...,X_m` 分别代换成 `TA_1,...,TA_m` 后，满足在调用表达式所在的上下文中代换后的返回类型是 `RA` 的子类型。

$$\sigma = [X_1 \mapsto TA_1, ..., X_m \mapsto TA_m]$$
$$\Delta \vdash \sigma R <: RA$$

3、如果函数调用时提供了类型实参，也就是函数调用的形式为：`f<TA>(a)`，则要求进入候选集的 `f` 满足以下要求：

- `f` 的类型形参与 `TA` 的数量相同，且类型实参 `TA` 满足 `f` 的泛型约束，且类型实参代入之后能通过函数调用的类型检查规则

第十章　重载

10.1　函数重载

10.1.1　函数重载的定义

在仓颉编程语言中，如果一个作用域中，同一个函数名对应多个参数类型不完全相同的函数定义，这种现象称为函数重载。

函数重载定义详见函数重载定义。

需要注意的是：

1. class、interface、struct 类型中的静态成员函数和实例成员函数之间不能重载
2. 同一个类型的扩展中的静态成员函数和实例成员函数之间不能重载，同一个类型的不同扩展中两个函数都是 private 的除外
3. enum 类型的 constructor、静态成员函数和实例成员函数之间不能重载

下例中，class A 中的实例成员函数 f 和静态成员函数 f 重载，将编译报错。

```
class A {
    func f() {}
    //Static member function can not be overloaded with instance member function.
    static func f(a: Int64) {}  // Error
}
```

下例中，class A 的扩展中实例成员函数 g 和静态成员函数 g 重载，将编译报错。

```
class A {}

extend A {
    func g() {}
    static func g(a: Int64) {}  // Error
}
```

下例中，实例成员函数 h 和静态成员函数 h 在 class A 的不同扩展中，且都是 private，编译不报错。

```
extend A {
    private func h() {}
}
```

```
extend A {
    private static func h(a: Int64) {}   // OK
}
```

下例中，enum E 的 constructor f , 实例成员函数 f 和静态成员函数 f 重载，将编译报错。

```
enum E {
    f(Int64)  // constructor

    // Instance member function can not be overloaded with constructor.
    func f(a: Float64) {}  // Error

    // Static member function can not be overloaded with instance member function
     ↪  or constructor.
    static func f(a: Bool) {}  // Error
}
```

在进行函数调用时，需要根据函数调用表达式中的实参的类型和上下文信息明确是哪一个函数定义被使用，分为以下几个步骤：

第 1 步、构建函数候选集；

第 2 步、函数重载决议；

第 3 步、确定实参类型。

10.1.2 重载函数候选集

调用函数 f 时，首先需要确定哪些函数是可以被调用的函数，可以被调用的函数集合称为重载函数候选集（以下简称候选集），用于函数重载决议。

构建函数候选集主要是两个步骤：

1. 查找可见函数集，即根据调用表达式的形式和上下文确定所有可见的函数；
2. 对可见函数集中的函数进行函数调用的类型检查，通过类型检查的函数（即可以被调用的函数）进入候选集；

10.1.2.1 可见函数集

可见函数集需要满足以下规则：

1. 作用域可见；
2. 根据 f 是普通函数还是构造函数确定可见函数集：2.1. 如果 f 是构造函数，则根据以下规则确定可见函数集：

- 如果 f 是有构造函数的类型名字，则可见函数集仅包括 f 中定义的构造函数；
- 否则，如果 f 是 enum 的 constructor 名字，则可见函数集仅包括指定的 enum 中名为 f 的 constructor；

- 否则，如果 f 是 super，则表示该调用表达式出现在 class/interface 中，f 仅包含调用表达式所在 class/interface 的直接父类构造函数；
- 否则，如果 f 是无构造函数的类型名，则可见函数集为空；

  2.2. 如果 f 是普通函数，则根据是否有限定前缀来确定可见函数集：

- 不带限定前缀直接通过名字调用 f(...)，可见函数集包含以下几种方式引入的名为 f 的函数：

  1) 作用域中可见的局部函数；
  2) 如果函数调用在 class/interface/struct/enum/extend 的静态上下文中，则包含类型的静态成员函数；
  3) 如果函数调用在 class/interface/struct/enum/extend 的非静态上下文中，则包含类型的静态和非静态成员函数；
  4) 函数调用所在 package 内定义的全局函数；
  5) 函数调用所在 package 中定义的 extend 中声明的函数；
  6) 函数调用所在 package 通过 import 方式导入的函数；

- 对于带限定前缀的函数调用 c.f(...)，可见函数集根据限定前缀来确定：

  1) 如果 c 是包名，可见函数集仅包含 package c 中定义的全局函数 f，包括 c 中通过 public import 重导出的函数，不包括 c 中仅通过 import 导入的函数；
  2) 如果 c 是 class 或 interface 或 struct 或 enum 定义的类型名，则可见函数集仅包括 c 的静态成员方法 f；
  3) 如果 c 是 this，即 this.f(...)，表示该调用表达式出现在 class 或 interface 或 struct 或 enum 的定义或扩展中。如果出现在类型定义中，可见函数集仅包含当前类型中的非静态成员方法 f(包括继承得到的，但不包括扩展中的)；如果出现在类型扩展中，可见函数集包含类型中的非静态成员方法 f，也包含扩展中的非静态成员方法 f；
  4) 如果 c 是 super，即 super.f(...)，表示该调用表达式出现在 class 或 interface 中，则可见函数集仅包含当前类型的父类或父接口的非静态成员方法 f；
  5) 如果函数调用的形式是 对象名.f(...) 的形式：
     如果对象的类型中有成员方法 f，则可见函数集仅包含所有名字为 f 的成员方法；

  2.3. 如果 f 是类型实例，则根据其类型或类型扩展中定义的 () 操作符重载函数来确定可见函数集。

  假设 `f` 的类型是 T，则可见函数集包括：

  1) T 内定义的 `()` 操作符重载函数；
  2) T 的扩展（当前作用域可见）中定义的 `()` 操作符重载函数；
  3) T 如果有父类，还包括从父类中继承的 `()` 操作符重载函数；
  4) T 如果实现了接口，还包括从接口中获得的带缺省实现的 `()` 操作符重载函数

1. 对于泛型函数 f，进入可见函数集的可能是部分实例化后的泛型函数或是完全实例化的函数。具体是哪种形式进入可见函数集，由调用表达式的形式决定（详见，泛型函数重载）.

10.1.2.2 类型检查

对于可见函数集中的函数进行类型检查，只有通过函数调用类型检查的函数才能进入候选集。

```
open class Base {}
class Sub <: Base {}

func f<X, Y>(a: X, b: Y) {}  // f1, number of type parameters is not matched
func f<X>(a: Base, b: X) where X <: Sub {} // f2

func test() {
    f<Sub>(Base(), Sub())  // candidate set: { f2 }
}
```

需要注意的是：

1. 如果实参有多个类型，在类型检查阶段，如果实参的多个类型中有一个能通过类型检查，则认为该实参能通过候选函数的类型检查；

```
open class A {}
open class B <: A {}

func g(a: A): B {  //g1
    B()
}
func g(a: B): B {  //g2
    B()
}
func g(a: Int64): Unit {}  //g3

// (A)->B <: (B)->B <: (B)->A
func f(a: (A)->B) {}   //f1, g1 can pass the type check
func f(a: (B)->B) {}   //f2, g1, g2 can pass the type check
func f(a: (B)->A) {}   //f3, g1, g2 can pass the type check
func f(a: Bool) {}     //f4, no g can pass the type check

func test() {
    f(g)  // candidate set: { f1, f2, f3 }
}
```

10.1.3 函数重载决议

如果候选集为空，即没有匹配项，编译报错；
如果候选集中只有一个函数，即只有一个匹配项，选择匹配的函数进行调用；

如果候选集中有多个函数，即有多个匹配项。先按照作用域优先级规则，选择候选集中作用域级别最高的。如果作用域级别最高的只有一个，则选择该函数；否则，根据最匹配规则，选择最匹配项。若无法确定唯一最匹配项则编译报错。

10.1.3.1 作用域优先级

作用域级别越高，函数重载决议时优先级越高，即：候选集中两个函数，选作用域级别高的，如果作用域级别相同，则根据最匹配规则章节中的规则进行选择。

作用域优先级需要注意的是：

1. 候选集中父类和子类中定义的函数，在函数重载时当成同一作用域优先级处理。
2. 候选集中类型中定义的函数和扩展中定义的函数，在函数重载时当成同一作用域优先级处理。
3. 候选集中定义在同一类型的不同扩展中的函数，在函数重载时当成同一作用域优先级处理。
4. 候选集中定义在扩展中或类型中的操作符重载函数和内置操作符，在函数重载时同一作用域优先级处理。
5. 除以上提到的类型作用域级别之外，其它的情况根据作用域级别章节中的规则，判断作用域级别；

```
/* According to the scope-level precedence principle, two functions in the
 ↪  candidate set, with different scope-levels, are preferred to the one with
 ↪  higher scope-level. */
func outer() {
    func g(a: B) {
        print("1")
    }
    func g(a: Int32) {
        print("3")
    }

    func inner() {
        func g(a: A) { print("2") }
        func innermost() {
            g(B())  // Output: 2
            g(1)   // Output: 3
        }
        g(B())   // Output: 2
        innermost()
    }

    inner()
    g(B())    // Output: 1
}
```

上例中，函数 innermost 中调用函数 g，且传入实参 B()，根据作用域级优先原则，优先选作用域级别高的，因此，选第 7 行定义的函数 g。

```
/* The inherited names are at the same scope level as the names defined or
 ↪ declared in the class. */
open class Father {
    func f(x: Child) { print("in Father") }
}

class Child <: Father {
    func f(x: Father) { print("in Child") }
}

func display() {
    var obj: Child = Child()
    obj.f(Child())  // in Father
}
```

上例中，函数 display 调用函数 f，且传入实参 Child()，父类和子类中构成重载的函数均属于同一作用域级别，将根据最匹配规则选择 Father 类中定义的 f(x: Child) {...}。

下例中，类型 C 中定义的函数 f 和 C 的扩展中定义的函数 f，在函数重载时当成同一作用域优先级处理。

```
open class Base {}
class Sub <: Base {}

class C {
    func f(a: Sub): Unit {}  // f1
}

extend C {
    func f(a: Base): Unit {}  // f2
    func g() {
        f(Sub()) // f1
    }
}

var obj = C()
var x = obj.f(Sub()) // f1
```

10.1.3.2 最匹配规则

候选集中最高优先级有多个函数：{f_1, ⋯, f_n}，这些函数被称为匹配项，若其中存在唯一一个匹配项 f_m，比其它所有匹配项都更匹配，则 f_m 为最匹配函数。

对于两个匹配项谁更匹配的比较是对两个匹配项形参的比较，分两步进行：

第一步，需要明确匹配项中用于比较的形参数量和顺序；

- 如果函数参数有缺省值，未指定实参的可选形参不参与比较。

```
open class Base {}
class Sub <: Base {}

func f(a!: Sub = Sub(), b!: Int32 = 2) {}    // This is f1,
func f(a!: Base = Base()) {}    // This is f2.

var x1 = f(a: Sub()) // parameters involved in comparison in f1 is only a
```

- 如果函数有命名参数，实参顺序可能会和形参声明的顺序不一致，那么用于比较的形参顺序需要和实参顺序保持一致，确保两个候选函数用于比较的形参是对应的。

第二步，比较两个匹配项的形参确定哪个匹配项更匹配；

对于两个匹配项：`f_i` 和 `f_j`，如果满足以下条件，则称 `f_i` 比 `f_j` 更匹配：

对于一个函数调用表达式：`e<T1, ..., T_p>(e_1, ...,e_m, a_m+1:e_m+1, a_k:e_k)`，实参个数为 `k` 个，则参与用于比较的形参个数为 `k` 个。

`f_i` 和 `f_j` 参与比较的形参分别为 `(a_i1, ..., a_ik)` 和 `(a_j1, ..., a_jk)`，对应的形参类型分别为 `(A_i1, ..., A_ik)`，`(A_j1, ..., A_jk)`。

若将 `f_i` 的形参 `(a_i1, ..., a_ik)` 作为实参传递给 `f_j` 能通过 `f_j` 的函数调用类型检查，且将 `f_j` 的形参 `(a_j1, ..., a_jk)` 作为参数传递给 `f_i` 不能通过函数调用类型检查，则称 `f_i` 比 `f_j` 更匹配。如下所示：

```
func f_i<X1,..., Xp>(a_i1: A_i1,..., a_ik: A_ik) {// f_i may not have type
 ↪ parameters
        f_j(a_i1, ..., a_ik)  // If this call expression can pass the type checking
}

func f_j<X1,..., Xq>(a_j1: A_j1,..., a_jk: A_jk) {// f_j may not have type
 ↪ parameters
         f_i(a_j1, ..., a_jk) // And this expression cannot pass the type checking
}
```

以下列举了一些函数重载决议的例子：

```
interface I3 {}
interface I1 <: I2 & I3 {}
interface I2 <: I4 {}
interface I4 <: I3 {}

func f(x: I4) {}  // f1
func f(x: I3) {}  // f2
```

```
class C1 <: I1 {}
var obj = C1()
var result = f(obj)   // choose f1, because I4 <: I3

open class C1 {}
open class C2 <: C1 {}
class C3 <: C2 {}

func f(a: C1, b: C2, c: C1) {}  // f1
func f(a: C3, b: C3, c: C2) {}   // f2
func f(a: C3, b: C2, c: C1) {}   // f3

// function call
var x = f(C3(), C3(), C3())   // f2

open class A {}
class B <: A {}
func foo<X>(a: X, b: X): Int32 {}       // foo1
func foo(a: A, b: B): Int32 {}          // foo2
func foo<X>(a: A, b: X): Int32 {}       // foo3

foo(A(), A())   // Error: cannot resolve.
foo(A(), 1)     // foo3. foo3 is the only function in candidate set.
```

10.1.3.3 确定实参类型

如果实参有多个类型:

确定最匹配函数后，如果实参的多个类型中有且只有一个类型 T 能通过该函数的类型检查，则确定实参的类型为 T；否则，编译报错。

```
// Sub <: Base
func f(a: (Base)->Int64, b : Sub) {} //f1
func f(a: (Sub)->Int64, b: Base) {}  //f2
func g(a: Base): Int64 { 0 }   // g1
func g(a: Sub): Int64 { 0 }    // g2

func test() {
    f(g, Base())  // Error, both of g can pass f2's type check.

    f(g, Sub())  // OK, only g1 passes f1's type check.
}
```

10.2 操作符重载

仓颉编程语言中定义了一系列使用特殊符号表示的操作符（详见第 4 章表达式），例如，算术操作符（+, -, *, /等），逻辑操作符（!, &&, ||），函数调用操作符 () 等。默认情况下，这些操作符的操作数只能是特定的类型，例如，算术操作符的操作数只能是数值类型，逻辑操作符的操作数只能是 Bool 类型。如果希望扩展某些操作符支持的操作数类型，或者允许自定义类型也能使用这些操作符，可以使用操作符重载（operator overloading ）来实现。

如果需要在某个类型 Type 上重载某个操作符 opSymbol，可以通过为 Type 定义一个函数名为 opSymbol 的操作符函数（operator function）的方式实现，这样，在 Type 的实例使用 opSymbol 时，就会自动调用名为 opSymbol 的操作符函数。

操作符函数定义的语法如下：

```
operatorFunctionDefinition
        : functionModifierList? 'operator' 'func'
          overloadedOperators typeParameters?
          functionParameters (':' type)?
          (genericConstraints)? ('=' expression | block)?
        ;
```

操作符函数定义与普通函数定义相似，区别如下：

- 定义操作符函数时需要在 func 关键字前面添加 operator 修饰符；
- overloadedOperators 是操作符函数名，它必须是一个操作符，且只有固定的操作符可以被重载。哪些操作符可重载将在可以被重载的操作符章节中详细介绍；
- 操作符函数的参数个数和操作符要求的操作数个数必须相同；
- 操作符函数只能定义在 class、interface、struct、enum 和 extend 中；
- 操作符函数具有实例成员函数的语义，所以禁止使用 static 修饰符；
- 操作符函数不能为泛型函数。

另外，需要注意：

- 被重载后的操作符不改变它们固有的优先级和结合性（各操作符的优先级和结合律详见第 4 章表达式）。
- 一元操作符是作为前缀操作符使用还是后缀操作符使用，与此操作符默认的用法保持一致。由于仓颉编程语言中不存在既可以作为前缀使用又可以作为后缀使用的操作符，所以不会产生歧义。
- 操作符函数的调用方式遵循操作符固有的使用方式（根据操作数的数量和类型决定调用哪个操作符函数)。

如果要在一个类型 Type 之上重载某个操作符 opSymbol，需要在类型上定义名为 opSymbol 的操作符函数。在类型上定义操作符函数有两种方式：

1. 使用 extend 的方式为其添加操作符函数，从而实现操作符在这些类型上的重载。对于无法直接包含函数定义的类型（是指除 struct、class、enum 和 interface 之外其他的类型），只能采用这种方式；

2. 对于可以直接包含函数定义的类型 (包括 class、interface、enum 和 struct)，可以直接在其内部定义操作符函数的方式实现操作符的重载。

10.2.1 定义操作符函数

因为操作符函数实现的是特定类型之上的操作符，所以定义操作符函数与定义普通实例成员函数的差别在于对参数类型的约定：

1. 对于一元操作符，操作符函数没有参数，对返回值的类型没有要求。

   例如，假设 opSymbol1 是一元操作符，那么操作符函数 opSymbol1 定义为：

```
operator func opSymbol1(): returnType1 {
    functionBody1
}
```

1. 对于二元操作符，操作符函数只有一个参数 right，对返回值的类型没有要求。

   例如，假设 opSymbol2 是二元操作符，那么操作符函数 opSymbol2 可以定义为：

```
operator func opSymbol2(right: anyType): returnType2 {
    functionBody2
}
```

同样地，对于定义了操作符函数的类型 TypeName，就可以像使用普通的一元或二元操作符一样在 TypeName 类型的实例上使用这些操作符（保持各操作符固有的使用方式）。另外，因为操作符函数是实例函数，所以在 TypeName 类型的实例 A 上使用重载操作符 opSymbol，其实是函数调用 A.opSymbol(arguments) 的语法糖（根据操作符的类型，参数的个数和类型，调用不同的操作符函数）。

下面举例说明如何在 class 类型中定义操作符函数，进而实现在 class 类型上重载特定的操作符。

假设我们希望在一个名为 Point（包含两个 Int32 类型的成员变量 x 和 y）的 class 类型上实现一元负号（-）和二元加法（+）两个操作。其中，-实现对一个 Point 实例中两个成员变量 x 和 y 取负值，然后返回一个新的 Point 对象，+ 实现对两个 Point 实例中两个成员变量 x 和 y 分别求和，然后返回一个新的 Point 对象。

首先，定义名为 Point 的 class，并在其中分别定义函数名为-和 + 的操作符函数，如下所示：

```
class Point {
    var x: Int32 = 0
    var y: Int32 = 0
    init (a: Int32, b: Int32) {
        x = a
        y = b
    }
    operator func -(): Point {
        return Point(-x, -y)
    }
    operator func +(right: Point): Point {
        return Point(x + right.x, y + right.y)
    }
}
```

接下来，就可以在 Point 的实例上直接使用一元-操作符和二元 + 操作符：

```
main(): Int64 {
    let p1 = Point(8, 24)
    let p2 = -p1     // p2 = Point(-8, -24)
    let p3 = p1 + p2 // p3 = Point(0, 0)
    return 0
}
```

10.2.2 操作符函数的作用域以及调用时的搜索策略

本节介绍操作符函数的作用域（名字和作用域请参考第 3 章）以及调用操作符函数时的搜索策略。

10.2.2.1 操作符函数的作用域

操作符函数和相同位置定义或声明的普通函数具备相同的作用域级别。

10.2.2.2 调用操作符函数时的搜索策略

这里介绍调用操作符函数（即使用操作符）时的搜索策略，因为一元操作符函数的搜索是二元操作符函数搜索的一个子情况，所以这里只介绍二元操作符（记为 op）的搜索策略（一元操作符遵循一样的策略）：

第一步，确定左操作数 lOperand 和右操作数 rOperand 的类型（假设分别为 lType 和 rType）；

第二步，在调用表达式 lOperand op rOperand 的当前作用域内，搜索和 lType 关联的所有名字为 op，右操作数类型为 rType 的操作符函数。如果有且仅有一个这样的操作符函数，则将表达式调用转换成此操作符函数的调用；如果没有找到这样的函数，则继续执行第 3 步；

第三步，在更低优先级的作用域内重复第 2 步。如果在最低优先级的作用域内仍然没有找到匹配的操作符函数，则终止搜索，并产生一个编译错误（“函数未定义”错误）。

10.2.3 可以被重载的操作符

下表列出了所有可以被重载的操作符（优先级从高到低）：

Operator	Description
()	Function call
[]	Indexing
!	NOT: unary
-	Negative: unary
**	Power: binary

Operator	Description
*	Multiply: binary
/	Divide: binary
%	Remainder: binary
+	Add: binary
-	Subtract: binary
<<	Bitwise left shift: binary
>>	Bitwise right shift: binary
<	Less than: binary
<=	Less than or equal: binary
>	Greater than: binary
>=	Greater than or equal: binary
==	Equal: binary
!=	Not equal: binary
&	Bitwise AND: binary
^	Bitwise XOR: binary
|	Bitwise OR: binary

需要注意的是：

1. 除了上表中列出的操作符，其他操作符（完整的操作符列表见 1.4 节）不支持被重载；
2. 一旦在某个类型上重载了除关系操作符（<、<=、>、>=、== 和!=）之外的其他二元操作符，并且操作符函数的返回类型与左操作数的类型一致或是其子类型，那么自然也就可以在此类型上使用对应的复合赋值符号。当操作符函数的返回类型与左操作数的类型不一致且不是其子类型时，在使用对应的复合赋值符号时将报类型不匹配错误；
3. 仓颉编程语言不支持自定义操作符，即不允许定义除上表中所列 `operator` 之外的其他操作符函数。

4. 函数调用操作符（()）重载函数对输入参数和返回值类型没有要求。

5. 对于类型 T, 如果 T 已经默认支持了上述若干可重载操作符，那么通过扩展的方式再次为其实现同签名的操作符函数时将报重定义错误。例如，为数值类型重载其已支持的同签名算术操作符、位操作符或关系操作符等操作符时，为 Rune 重载同签名的关系操作符时，为 Bool 类型重载同签名的逻辑操作符、判等或不等操作符时，等等这些情况，均会报重定义错。

存在以下特殊场景：

- 不能使用 `this` 或 `super` 调用 `()` 操作符重载函数。
- 对于枚举类型，当构造器形式和 `()` 操作符重载函数形式都满足时，优先匹配构造器形式。

```
// Scenario for `this` or `super`.
open class A {
    init(x: Int64) {
        this() // error, missing argument for call with parameter list: (Int64)
    }
    operator func ()(): Unit {}
}

class B <: A {
    init() {
        super() // error, missing argument for call with parameter list: (Int64)
    }
}

// Scenario for enum constructor.
enum E {
    Y | X | X(Int64)
    operator func ()(a: Int64){a}
    operator func ()(a: Float64){a}
}

main() {
    let e = X(1) // ok, X(1) is to call the constructor X(Int64).
    X(1.0)   // ok, X(1.0) is to call the operator () overloading function.
    let e1 = X
    e1(1) // ok, e1(1) is to call the operator () overloading function.
    Y(1) // oK, Y(1) is to call the operator () overloading function.
}
```

10.2.4 索引操作符重载

索引操作符（[]）分为取值 let a = arr[i] 和赋值 arr[i] = a 两种形式，它们通过是否存在特殊的命名参数 value 来区分不同的重载。索引操作符重载不要求同时重载两种形式，可以只重载赋值不重载取值，反之亦可。

索引操作符取值形式 [] 内的参数序列对应操作符重载的非命名参数，可以是 1 个或多个，可以是任意类型。不可以有其它命名参数。返回类型可以是任意类型。

```
class A {
    operator func [](arg1: Int64, arg2: String): Int64 {
        return 0
    }
}

func f() {
    let a = A()
    let b: Int64 = a[1, "2"]
    // b == 0
}
```

索引操作符赋值形式 [] 内的参数序列对应操作符重载的非命名参数，可以是 1 个或多个，可以是任意类型。= 右侧的表达式对应操作符重载的命名参数，有且只能有一个命名参数，该命名参数的名称必须是 value，不能有默认值，value 可以是任意类型。返回类型必须是 Unit 类型。

需要注意的是，value 只是一种特殊的标记，在索引操作符赋值时并不需要使用命名参数的形式调用。

```
class A {
    operator func [](arg1: Int64, arg2: String, value!: Int64): Unit {
        return
    }
}

func f() {
    let a = A()
    a[1, "2"] = 0
}
```

特别的，数值类型、Bool、Unit、Nothing、Rune、String、Range、Function、Tuple 类型不支持重载索引操作符赋值形式。

第十一章　包和模块管理

在仓颉编程语言中，程序以包的形式进行组织，包是最小的编译单元。包可以定义子包，从而构成树形结构。没有父包的包称为 root 包，root 包及其子包（包括子包的子包）构成的整棵树称为 module。module 的名称与 root 包相同。module 是仓颉的最小发布单元。

包由一个或多个源码文件组成，同一个包的源码文件必须在同一个目录，并且同一个目录里的源码文件只能属于同一个包。子包的目录是其父包目录的子目录。

11.1　包

11.1.1　包的声明

包声明以关键字 package 开头，后接 root 包至当前包路径上所有包的包名，以 . 分隔（路径本身不是包名）。

包声明的语法如下:

```
packageHeader
    : packageModifier? (MACRO NL*)? PACKAGE NL* fullPackageName end+
    ;

fullPackageName
    : (packageNameIdentifier NL* DOT NL*)* packageNameIdentifier
    ;

packageNameIdentifier
    : Ident
    | contextIdent
    ;

packageModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    ;
```

每个包都有包名，包名是这个包可唯一识别的标识符。

除 root 包外，包名必须和其所在的目录名一致。

包声明必须在文件的首行（注释和空白字符不计），每个文件只能有一个 package 声明。特别地，root 包的文件可以不声明 package，对于不包含 package 声明的文件，它会用 default 作为包名。

```
// ./src/main.cj
// Leaving package declaration out is equivalent to 'package default'

main() {
    println("Hello World")
}
```

package 可以被 internal、protected 或者 public 修饰。package 的默认修饰符（默认修饰符是指在省略情况下的修饰符语义，这些默认修饰符也允许显式写出）为 public。同一个包在不同文件中的 package 声明必须使用相同的访问修饰符修饰。特别地，root 包不能被 internal 或者 protected 修饰。

- internal 表示仅当前包及子包（包括子包的子包）内可见。当前包的子包（包括子包的子包）内可以导入这个包或者这个包的成员。
- protected 表示仅当前 module 内可见。同一个 module 内的其它包可以导入这个包或者这个包的成员，不同 module 的包无法访问。
- public 表示 module 内外均可见。其它包可以导入这个包或者这个包的成员。

11.1.2 包的成员

包的成员是在顶层声明的类、接口、struct、enum、类型别名、全局变量、扩展、函数。顶层声明包括的内容，可以参考其对应的章节。

当前包的父包和子包并不是当前包的成员，访问父包或者子包需要包的导入机制，未被导入的包名不在 top-level 作用域中。

如下所示的例子中，package a.b 是 package a 的子包。

```
// src/a.cj
package a
let a = 0 // ok
```

```
// src/b/b.cj
package a.b
let a = 0 // ok
let b = 0 // ok
```

如下所示的例子中，package a.b 是 package a 的子包。

```
// src/a.cj
package a

let u = 0     // ok
let _ = b.x   // error: undeclared identifier 'b'
```

```
let _ = a.u // error: undeclared identifier 'a'
let _ = a.b.x // error: undeclared identifier 'a'

// src/b/b.cj
package a.b

let x = 1     // ok
let _ = a.u   // error: undeclared identifier 'a'
let _ = a.b.x // error: undeclared identifier 'a'
let _ = b.x   // error: undeclared identifier 'b'
```

特别地，子包不能和当前包的成员同名，这是为了保证访问路径中的名称是唯一的。

11.1.3 访问修饰符

仓颉中，可以使用访问修饰符来保护对类型、变量、函数等元素的访问。仓颉有 4 种不同的访问修饰符。

- private
- internal
- protected
- public

11.1.3.1 修饰顶层元素

在修饰顶层元素时不同访问修饰符的语义如下：

- private 表示仅当前文件内可见。不同的文件无法访问这类成员。
- internal 表示仅当前包及子包（包括子包的子包）内可见。同一个包内可以不导入就访问这类成员，当前包的子包（包括子包的子包）内可以通过导入来访问这类成员。
- protected 表示仅当前 module 内可见。同一个包的文件可以不导入就访问这类成员，不同包但是在同一个 module 内的其它包可以通过导入访问这些成员，不同 module 的包无法访问。
- public 表示 module 内外均可见。同一个包的文件可以不导入就访问这类成员，其它包可以通过导入访问这些成员。

	File	Package & Sub-Packages	Module	All Packages
private	Y	N	N	N
internal	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

不同顶层声明支持的访问修饰符和默认修饰符规定如下：

- pacakge 支持使用 internal、protected、public，默认修饰符为 public。
- import 支持使用全部访问修饰符，默认修饰符为 private。
- 其他顶层声明支持使用全部访问修饰符，默认修饰符为 internal。

11.1.3.2 修饰非顶层成员

在修饰非顶层成员时不同访问修饰符的语义如下:

- private 表示仅当前类型或扩展定义内可见。
- internal 表示仅当前包及子包（包括子包的子包）内可见。
- protected 表示当前 module 及当前类的子类可见。
- public 表示 module 内外均可见。

	Type/Extend	Package & Sub-Packages	Module & Sub-Classes	All Packages
private	Y	N	N	N
internal	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

类型成员的访问修饰符可以不同于类型本身。除接口外类型成员的默认修饰符（默认修饰符是指在省略情况下的修饰符语义，这些默认修饰符也允许显式写出）是 internal，接口中的成员函数和属性不可以写访问修饰符，它们的访问级别等同于 public。

11.1.3.3 访问修饰符的合法性检查

仓颉的访问级别排序为 public > protected > internal > private。
类型的访问级别:

- 非泛型类型的访问级别由类型声明的访问修饰符决定
- 泛型实例化类型的访问级别等同于该泛型类型与该泛型类型实参的访问级别中最低的一个

一个声明的访问修饰符不得高于该声明中用到的类型的访问修饰符的级别。具体地:

- 变量、属性声明的访问级别不得高于其类型的访问级别
- 函数声明的访问级别不得高于参数类型、返回值类型，以及 where 约束中的类型上界的访问级别
- 类型别名的访问级别不得高于原类型的访问级别
- 类型声明的访问级别不得高于 where 约束中的类型上界的访问级别
- 子包的访问级别不得高于其父包的访问级别
- import 的访问修饰符不得高于其导入声明的访问级别

```
private open class A {}

protected let a = A() // error: 'protected' declaration 'a' uses 'private' type 'A'
let (a, b) = (A(), 1) // error: 'internal' declaration 'a' uses 'private' type 'A'

func f(_: A) {} // error: 'internal' declaration 'f' uses 'private' type 'A'
func f() { A() } // error: 'internal' declaration 'f' uses 'private' type 'A'
```

```
func f<T>() where T <: A {} // error: 'internal' declaration 'f' uses 'private'
 ↪  type 'A'

public type X = A // error: 'public' declaration 'X' uses 'private' type 'A'
public type ArrayA = Array<A> // error: 'public' declaration 'ArrayA' uses
 ↪  'private' type 'A'

protected struct S<T> where T <: A {} // error: 'protected' declaration 'S' uses
 ↪  'private' type 'A'
```

```
// src/a.cj
public package a

// src/a/b/b.cj
protected package a.b // ok

// src/a/b/c/c.cj
public package a.b.c // error
```

特别地，类继承时子类访问级别与父类访问级别、类型实现/继承接口时子类型访问级别与父接口访问级别不受上述规则限制。

```
private open class A {}
public enum E { U | V }
interface I {}

public class C <: A {} // ok

public interface J <: I {} // ok

extend E <: I {} // ok
```

11.1.4 包的导入

导入是一种用来将其他包或其他包中的成员引入到当前仓颉程序中的机制。

当源码中没有 import 声明的时候，当前文件只能访问当前包中的成员和编译器默认导入的成员。通过 import 声明，可以让编译器在编译这个仓颉文件时找到所需要的外部名称。

import 语句在文件中的位置必须在包声明之后，其他声明或定义之前。

import 相关的语法如下：

```
importList
    : importModifier? NL* IMPORT NL* importContent end+
    ;
```

```
importSingle
    : (packageNameIdentifier NL* DOT NL*)* (identifier | packageNameIdentifier)
    ;

importSpecified
    : (identifier '.')+ identifier
    ;

importAlias
   : importSingle NL* AS NL* identifier
   ;

importAll
   : (packageNameIdentifier NL* DOT NL*)+ MUL
   ;

importMulti
   : (packageNameIdentifier NL* DOT NL*)* LCURL NL*
   (importSingle | importAlias | importAll) NL*
   (COMMA NL* (importSingle | importAlias | importAll))* NL*
   COMMA? NL* RCURL
   ;
```

import 语法有如下几种形式:

- 单导入
- 别名导入
- 全导入
- 批量导入

通过 import，可以导入一个或多个其他包或者其他包中的成员，也可以通过 as 语法为导入的名称定义别名。如果导入的名称是包，则可以用它继续访问包中的成员（子包不是包的成员），但包名本身不能作为表达式。

```
package a
public let x = 0
```

```
package demo

import a

main() {
    println(a.x) // ok, prints 0
}
```

任意包和包之间不能产生循环依赖，即使是同一个 module 下的包之间也不可以。对于任意两个包 p1 和 p2，如果 p1 导入了 p2 或者 p2 的成员，那么我们称 p1 和 p2 具有依赖关系，p1 依赖 p2。依赖关系具有传递性，如果 p1 依赖 p2，p2 依赖 p3，那么 p1 依赖 p3。包的循环依赖是指存在包相互依赖的情况。

```
package p.a
import p.b // error

pacakge p.b
import p.a // error
```

禁止使用 import 导入当前包或当前包中的成员。

```
package a

import a // error
import a.x // error

public let x = 0
```

导入的成员的作用域级别低于当前包声明的成员。导入的非函数成员会被当前包的同名成员遮盖；导入的函数成员若可以和当前包的同名函数构成重载，调用时会根据泛型函数重载 和函数重载 的规则进行函数决议；导入的函数成员若和当前包的同名函数不构成重载，则按照遮盖处理。

```
package a
public let x = 0
public func f() {}

import a.x // warning: imported 'x' is shadowed
import a.f

let x = 1
func f(x: Int64) { x }

let _ = f()  // ok, found 'a.f'
let _ = f(1) // ok, found 'f' defined in this package
```

11.1.4.1 单导入

单导入语法用来导入单个成员，目标成员必须是对当前包可见的。导入的成员名称会作为当前作用域内可以访问的名称。

import 语法中的路径最后一个名称表示指定的成员，这个名称可以是顶层变量、函数、类型，也可以是包。

下面是导入顶层变量、函数、类型的例子。有两个包分别是 a 和 b，在 b 包中导入 a 包的成员。

```
package a
```

```
public let x = 0
public func f() { 0 }
public class A {}
```

```
import a.x
import a.f
import a.A

private func g(_: A) { x + f() } // ok
```

如下所示的例子中，c 是 a 的子包。

```
package a.c

public let y = 1
```

```
import a

private func g(_: a.A) { a.x + a.f() } // ok
privage func h(_: A) { // error: undeclared identifier 'A'
    x + f() // error: undeclared identifier 'x' and 'f'
}
let _ = a.c.y // error: 'c' is not a member of 'a'
let _ = a // error: undeclared identifier 'a'
```

```
import a.c

let _ = c.y // ok
```

单导入的成员被当前包成员遮盖时，编译器会给出告警提示无用导入。

```
import a.x // warning: imported 'x' is shadowed
import a.f // warning: imported 'f' is shadowed

func f() { 1 }

let x = 1
let _ = f() // ok, call 'f' defined in this package, value is 1
```

11.1.4.2 别名导入

别名导入可以使用 as 语法为导入成员重命名。以别名导入的内容在当前包中只会以别名的形式引入作用域，而不会引入原来的名称（但不禁止分别导入原名和别名）。导入的内容可以是包或者包的成员。

```
package a
```

```
public let x = 0
public let y = 1
public func f() { 0 }

import a as pkgA
import a.x as x1
import a.x as x2 // ok

let _ = x        // error: undeclared identifier 'x'
let _ = a.x      // error: undeclared identifier 'a'
let _ = x1       // ok
let _ = x2       // ok
let _ = pkgA.x   // ok
let _ = pkgA.x1  // error: 'x1' is not a member of 'pkgA'
```

11.1.4.3 全导入

全导入通过 * 语法导入其他包中所有对当前包可见的顶层成员（不包括子包）。
示例如下：

```
package a

public let x = 0
public func f() { 0 }
public class A {}

import a.*

private func g(_: A) { x + f() } // ok
```

与单导入不同，当全导入的成员被当前包成员遮盖时，编译器不会给出告警。

```
import a.*

let x = 1
func f() { x } // ok, 'x' defined in this package

let _ = f() // ok, call 'f' defined in this package, value is 1
```

如果导入的成员不被当前包的成员遮盖，但多个导入成员重名时，编译器不会给出告警，但如果这些重名的导入不构成重载，这个名字在本包中不可用，在使用该名称时编译器会因无法找到唯一的名称而报错。

```
package b
```

```
public let x = 1
public func f(x: Int64) { x }

import a.*
import b.*

let _ = x // error: ambiguous 'x'
```

如果导入的重名成员可以构成函数重载，调用时会根据泛型函数重载 和函数重载 的规则进行函数决议。

```
import a.*
import b.*

func f(b: Bool) { b }

let _ = f()     // ok, call 'a.f'
let _ = f(1)    // ok, call 'b.f'
let _ = f(true) // ok, call 'f' defined in this package
```

带访问修饰符的全导入不会导入比其访问级别低的声明。

```
package a
protected import a.b.*
let _ = x // ok
let _ = y // ok
let _ = z // error: undeclared identifier 'z'

package a.b
public let x = 0
protected let y = 1
internal let z = 2
```

11.1.4.4 批量导入

批量导入使用 {} 语法，在一个 import 声明里同时导入多个成员。通常用来省略重复的包路径前缀。批量导入的 {} 中支持单导入、别名导入和全导入，但不允许嵌套批量导入。

```
import std.{
    time,
    fs as fileSystem,
    io.*,
    collection.{HashMap, HashSet} // syntax error
}
```

{} 的前缀可以为空。

```
import {
    std.time,
    std.fs as fileSystem,
    std.io.*,
}
```

使用批量导入语法与使用多个独立 import 的语法是等价的。

```
import std.{
    os.process,
    time,
    io.*,
    fs as fileSystem
}
```

等价于:

```
import std.os.process
import std.time
import std.io.*
import std.fs as fileSystem
```

11.1.4.5 导入名称冲突检查

如果多个单导入的名称产生重名（包括重复导入）且不构成函数重载，并且该名字在本包中没有被遮盖，编译器会给出名称冲突告警，这个名字在本包中不可用，在使用该名称时编译器会因无法找到唯一的名称而报错。若该名称被当前包成员遮盖时，编译器会给出告警提示无用导入。

```
package b

public let x = 1
public func f(x: Int64) { x }
```

```
package c
public let f = 0

import a.x // warning: imported 'x' is shadowed
import a.x // warning: imported 'x' is shadowed
import b.x // warning: imported 'x' is shadowed

let x = 0
let y = x // y = 0

import a.x
import a.x // warning: 'x' has been imported
```

```
import b.x // warning: 'x' has been imported

let _ = x // error: ambiguous 'x'
```

如果导入的重名成员之间或者导入的成员与当前包中的同名函数之间可以构成函数重载，调用时会根据泛型函数重载 和函数重载 的规则进行函数决议。

```
import a.f
import b.f

func f(b: Bool) { b }

let _ = f()     // ok, call 'a.f'
let _ = f(1)    // ok, call 'b.f'
let _ = f(true) // ok, call 'f' defined in this package
```

多个别名导入同名，或者别名导入和本包定义同名时的处理规则与单导入相同。

```
import a.x as x1 // warning: imported 'x1' is shadowed

let x1 = 10
let _ = x1 // ok, 'x1' defined in this package
```

```
package b

public let x = 1
public func f(x: Int64) { x }
```

```
import a.x as x1
import a.x as x1 // warning: 'x1' has been imported
import b.x as x1 // warning: 'x1' has been imported

let _ = x1 // error: ambiguous 'x1'
```

```
import a.f as g
import b.f as g

func g(b: Bool) { b }

let _ = g()     // ok, call 'a.f'
let _ = g(1)    // ok, call 'b.f'
let _ = g(true) // ok, call 'g' defined in this package
```

如果导入名称冲突的其中一方来自全导入，这种情况下编译器也不会给出报警，但冲突的声明都不可用。批量导入依据其等价的单导入、别名导入、多导入做名称冲突检查。

11.1.4.6 import 的访问修饰符

import 可以被 private、internal、protected、public 访问修饰符修饰。其中，被 public、protected 或者 internal 修饰的 import 可以把导入的成员重导出（如果这些导入的成员没有因为名称冲突或者被遮盖导致在本包中不可用）。其他包可以根据访问修饰符 的访问规则通过 import 导入这些被重导出的对象。具体地：

- private import 表示导入的内容仅当前文件内可访问，private 是 import 的默认修饰符，不写访问修饰符的 import 等价于 private import。
- internal import 表示导入的内容在当前包及其子包（包括子包的子包）均可访问。非当前包访问需要显式 import。
- protected import 表示导入的内容在当前 module 内都可访问。非当前包访问需要显式 import。
- public import 表示导入的内容外部都可访问。非当前包访问需要显式 import。

在下面的例子中，b 是 a 的子包，在 a 中通过 public import 重导出了 b 中定义的函数 f。

```
package a

public let x = 0
public import a.b.f

internal package a.b

public func f() { 0 }

import a.f  // ok
let _ = f() // ok

//// case 1
package demo

public import std.time.Duration // warning: imported 'Duration' is shadowed

struct Duration {}

//// case 2
// ./a.cj
package demo

public import std.time.Duration

// ./b.cj
package demo

func f() {
```

```
    let a: Duration = Duration.second // ok, access the re-exported 'Duration'
}

//// case 3
// ./a/a.cj
package demo.a

public let x = 0

// ./b/b.cj
package demo.b

public import demo.a.* // warning: imported 'x' is shadowed, will not re-export
 ↪  'demo.a.x'

var x = 0

//// case 4
// ./a/a.cj
package demo.a

public let x = 0

// ./b/b.cj
package demo.b

public let x = 0

// ./c/c.cj
package demo.c

public import demo.a.* // warning, because there are duplicate names, will not
 ↪  re-export 'demo.a.x'
public import demo.b.* // warning, because there are duplicate names, will not
 ↪  re-export 'demo.b.x'
```

特别地，包不可以被重导出：如果被 `import` 导入的是包，那么该 `import` 不允许被 `public`、`protected` 或者 `internal` 修饰。

```
public import a.b // error: cannot re-export package
```

第十二章　异常

在编写软件系统时，检测和处理程序中的错误行为往往是十分困难的，为了保证系统的正确性和健壮性，很多软件系统中都包含大量的代码用于错误检测和错误处理。异常是一类特殊的可以被程序员捕获并处理的错误，是程序执行时出现的一系列不正常行为的统称，例如，索引越界、除零错误、计算溢出、非法输入等。

异常不属于程序的正常功能，一旦发生异常，要求程序必须立即处理，即将程序的控制权从正常功能的执行处转移至处理异常的部分。仓颉编程语言提供异常处理机制用于处理程序运行时可能出现的各种异常情况。异常处理主要涉及：

- try 表达式（try expression），包括普通 try 表达式和 try-with-resources 表达式。
- throw 表达式（throw expression）由关键字 `throw` 以及尾随的表达式组成。尾随表达式的类型必须继承于 `Exception` 或 `Error` 类。

下面将分别介绍 try 表达式和 throw 表达式

12.1　Try 表达式

根据是否涉及资源的自动管理，将 try 表达式分为两类：不涉及资源自动管理的普通 try 表达式，以及会进行资源自动管理的 try-with-resources 表达式。try 表达式的语法定义为：

```
tryExpression
    : 'try' block 'finally' block
    | 'try' block ('catch' '(' catchPattern ')' block)+ ('finally' block)?
    | 'try' '(' ResourceSpecifications ')' block ('catch' '(' catchPattern ')'
    ↪  block)* ('finally' block)?
    ;
catchPattern
    : wildcardPattern
    | exceptionTypePattern
    ;
exceptionTypePattern
    : ('_' | identifier) ':' type ('|' type)*
    ;
ResourceSpecifications
    : ResourceSpecification (',' ResourceSpecification)*
```

```
    ;

ResourceSpecification
    : identifier (':' classType)? '=' expression
    ;
```

接下来分别对普通 try 表达式和 try-with-resources 表达式进行介绍。

12.1.1 普通 Try 表达式

普通 try 表达式（本小节提到的 try 表达式特指普通的 try 表达式）包括三个部分：try 块，catch 块和 finally 块。

1. **Try** 块，以关键字 `try` 开始，后面紧跟一个由表达式与声明组成的块（用一对花括号括起来，定义了新的局部作用域，可以包含任意表达式和声明，后简称“块”），try 后面的块内可以抛出异常，并被紧随的 catch 块所捕获并处理（如果不存在 catch 块或未被捕获，则在执行完 finally 块后，继续抛出至调用它的函数）；

2. **Catch** 块，一个 try 表达式可以包含零个或多个 catch 块（当没有 catch 块时必须有 finally 块）。每个 catch 块以关键字 `catch` 开头，后跟一条 `(catchPattern)` 和一个*表达式与声明组成的块*，`catchPattern` 通过模式匹配的方式匹配待捕获的异常，一旦匹配成功，则交由*表达式与声明组成的块*进行处理，并且忽略它后面的所有 catch 块。当某个 catch 块可捕获的异常类型均可被定义在它前面的某个 catch 块所捕获时，会在此 catch 块处报“catch 块不可达”的 warning。

3. **Finally** 块，以关键字 `finally` 开始，后面紧跟一个用花括号括起来的表达式与声明组成的块。原则上，finally 块中主要实现一些“善后”的工作，如释放资源等，且要尽量避免在 finally 块中再抛异常。并且无论异常是否发生（即无论 try 块中是否抛出异常），finally 块内的内容都会被执行（若异常未被处理，执行完 finally 块后，继续向外抛出异常）。另外，一个 try 表达式中可以包含一个 finally 块，也可以不包含 finally 块（但此时必须至少包含一个 catch 块）。

其中 `catchPattern` 有两种模式：

- ** 通配符模式（“_”）** ：可以捕获同级 try 块内抛出的任意类型的异常，等价于类型模式中的 `e: Exception`，即捕获 Exception 及其子类所定义的异常。示例：

```
// Catch with wildcardPattern.
let arrayTest: Array<Int64> = Array<Int64>([0, 1, 2])
try {
    let lastElement = arrayTest[3]
} catch (_) {
    print("catch an exception!")
}
```

- 类型模式：可以捕获指定类型（或其子类型）的异常，语法上主要有两种格式：

- `identifier : ExceptionClass`。此格式可以捕获类型为 ExceptionClass 及其子类的异常，并将捕获到的异常实例转换成 ExceptionClass，然后与 `identifier` 定义的变量进行绑定，接着就可以在 catch 块中通过 `identifier` 定义的变量访问捕获到的异常实例。
- `identifier : ExceptionClass_1 | ExceptionClass_2 | ... | ExceptionClass_n`。此格式可以通过连接符 `|` 将多个异常类进行拼接，连接符 `|` 表示“或”的关系：可以捕获类型为 `ExceptionClass_1` 及其子类的异常，或者捕获类型为 `ExceptionClass_2` 及其子类的异常，依次类推，或捕获类型为 `ExceptionClass_n` 及其子类的异常（假设 `n` 大于 `1`）。当待捕获异常的类型属于上述“或”关系中的任一类型或其子类型时，此异常将被捕获。但是由于无法静态地确定被捕获异常的类型，所以会将被捕获异常的类型转换成由 `|` 连接的所有类型的最小公共父类，并将异常实例与 `identifier` 定义的变量进行绑定。因此在此类模式下，catch 块内只能通过 `identifier` 定义的变量访问 `ExceptionClass_i`（`1 <= i <= n`）的最小公共父类中的成员变量和成员函数。当然，也可以使用通配符代替类型模式中的 `identifier`，差别仅在于不会进行绑定操作。

关于类型模式用法的示例如下：

```
// The first situation.
main() {
    try {
        throw ArithmeticException()
    } catch (e: Exception) { // Caught.
        print("Exception and its subtypes can be caught here")
    }
}

// The second situation.
// User defined exceptions.
open class Father <: Exception {
    var father: Int64 = 0
    func whatFather() {
        print("I am Father")
    }
}
class ChildOne <: Father {
    var childOne: Int64 = 1
    func whatChildOne() {
        print("I am ChildOne")
    }
    func whatChild() {
        print("I am method in ChildOne")
    }
}
class ChildTwo <: Father {
```

```
    var childTwo: Int64 = 2
    func whatChildTwo() {
        print("I am ChildTwo")
    }
    func whatChild() {
        print("I am method in ChildTwo")
    }
}

// Function main.
main() {
    var a = 1
    func throwE() {
        if (a == 1) {
            ChildOne()
        } else {
            ChildTwo()
        }
    }
    try {
        throwE()
    } catch (e: ChildOne | ChildTwo) {
        e.whatFather()        // ok: e is an object of Father
        //e.whatChildOne()    // error: e is an object of Father
        //e.whatChild()       // error: e is an object of Father
        print(e.father)       // ok: e is an object of Father
        //print(e.childOne)   // error: e is an object of Father
        //print(e.childOTwo)  // error: e is an object of Father
    }
    return 0
}
```

使用 finally 块的例子如下：

```
// Catch with exceptionTypePattern.
try {
    throw IndexOutOfBoundsException()
} catch (e: ArithmeticException | IndexOutOfBoundsException) {
    print("exception info: " + e.toString())
} catch (e: Exception) {
    print("neither ArithmeticException nor IndexOutOfBoundsException, exception
     ↪  info: " + e.toString())
} finally {
```

```
        print("the finally block is executed")
    }
```

12.1.1.1 Try 表达式的类型

类似于 if 表达式，

1. 如果 try 表达式的值没有被读取或者返回，那么整个 try 表达式的类型为 Unit，try 块和 catch 块不要求存在公共父类型；否则，按如下规则检查；
2. 在上下文没有明确的类型要求时，要求 try 块和所有 catch 块（如果存在）拥有最小公共父类型。整个 try 表达式的类型就是该最小公共父类型；
3. 在上下文有明确的类型要求时，try 块和任一 catch 块（如果存在）的类型都必须是上下文所要求的类型的子类型，但此时不要求它们拥有最小公共父类型。

需要注意的是，虽然 finally 块会在 try 块和 catch 之后执行，但是它不会对整个 try 表达式的类型产生影响，并且 finally 块的类型始终是 Unit（即使 finally 块内表达式的类型不是 Unit）。

12.1.1.2 Try 表达式的求值顺序

关于 try {e1} catch (catchPattern) {e2} finally {e3} 表达式执行规则的额外规定：

- 若进入 finally 块前执行到 return e 表达式，则会将 e 求值至 v，然后立刻执行 finally block；若进入 finally 块前执行到 break 或 continue 表达式，则会立刻执行 finally block。
  - 若 finally 块中无 return 表达式，则处理完 finally 块后会将缓存的结果 v 返回（或抛出异常）。也就是说，即使 finally 块中有对上述 e 中引用变量的赋值，也不会影响已经求值的结果 v，举例如下：

    ```
    func f(): Int64 {
        var x = 1;
        try {
            return x + x;  // Return 2.
        } catch (e: Exception) { // Caught.
            print("Exception and its subtypes can be caught here")
        } finally {
            x = 2;
      }  // The return value is 2 but not 4.
    ```

  - 若在 finally 块中执行到 return e2 或 throw e2，则会对 e2 求值至结果 v2，并立即返回或抛出 v2；若在 finally 块中执行到 break 或 continue，则会终止 finally 的执行并立即跳出循环。举例如下：

    ```
    func f(): Int64 {
        var x = 1;
        try {
            return x + x; // Return 2.
    ```

```
        } catch (e: Exception) { // Caught.
            print("Exception and its subtypes can be caught here")
        } finally {
            x = 2;
            return x + x; // Return 4
        }  // The return value is 4 but not 2.
    }
```

总之，finally 块一定会被执行。如果 finally 块中有任何控制转移表达式，都会覆盖进入 finally 之前的控制转移表达式。

Try 表达式中 throw 的处理更为复杂，具体请参考下一小节。

12.1.1.3 Try 表达式处理异常的逻辑

关于 `try {e1} catch (catchPattern) {e2} finally {e3}` 表达式执行时抛出异常的规定：

1. 若执行 e1 的过程中没有抛出异常（这种情况下不会执行 e2），
    - 当执行 e3 的过程中亦无异常抛出，那么整个 try 表达式不会抛出异常，
    - 当执行 e3 的过程中抛出异常 E3，那么整个 try 表达式抛出异常 E3；
2. 若执行 e1 的过程中抛出异常 E1，且执行 e3 的过程中抛出异常 E3，那么整个 try 表达式抛出异常 E3（无论 E1 是否被 catch 块所捕获）；
3. 若执行 e1 的过程中抛出异常 E1，且执行 e3 的过程中无异常抛出，那么：
    1. 当 E1 可以被 catch 捕获且执行 e2 的过程中无异常抛出时，整个 try 表达式无异常抛出；
    2. 当 E1 可以被 catch 捕获且执行 e2 的过程中抛出异常 E2 时，整个 try 表达式抛出异常 E2；
    3. 当 E1 未能被 catch 捕获时，整个 try 表达式抛出异常 E1。

12.1.2 Try-With-Resources 表达式

Try-with-resources 表达式主要是为了自动释放非内存资源。不同于普通 try 表达式，try-with-resources 表达式中的 catch 块和 finally 块均是可选的，并且 try 关键字其后的块之间可以插入一个或者多个 ResourceSpecification 用来申请一系列的资源（ResourceSpecification 并不会影响整个 try 表达式的类型）。这里所讲的资源对应到语言层面即指对象，因此 ResourceSpecification 其实就是实例化一系列的对象（多个实例化之间使用 “,” 分隔）。使用 try-with-resources 表达式的例子如下所示：

```
class MyResource <: Resource {
    var flag = false
    public func isClosed() { flag }
    public func close() { flag = true }
    public func hasNextLine() { false }
    public func readLine() { "line" }
    public func writeLine(_: String) {}
```

```
}

main() {
    try (input = MyResource(),
        output = MyResource()) {
        while (input.hasNextLine()) {
            let lineString = input.readLine()
            output.writeLine(lineString)
        }
    } catch (e: Exception) {
        print("Exception happened when executing the try-with-resources
        ↪ expression")
    } finally {
        print("end of the try-with-resources expression")
    }
}
```

在 try-with-resources 表达式中的 ResourceSpecification 的类型必须实现 Resource 接口：

```
interface Resource {
    func isClosed(): Bool
    func close(): Unit
}
```

Try-with-resources 表达式会首先（依声明顺序）执行实例化对应的一系列资源申请（上例中，先实例化 input 对象，再实例化 output 对象），在此过程中若某个资源申请失败（例如，output 实例化失败），则在它之前申请成功的资源（如 input 对象）会被全部释放（释放过程中若抛出异常，会被忽略），并抛出申请此资源（output 对象）失败的异常。

如果所有资源均申请成功，则继续执行 try 之后紧跟的块。在执行块的过程中，无论是否发生异常，之后均会按照资源申请时的逆序依次自动释放资源（上例中，先释放 output 对象，再释放 input 对象）。在释放资源的过程中，若某个资源在被释放时发生异常，并不会影响其它资源的继续释放，并且整个 try 表达式抛出的异常遵循如下原则：（1）如果 try 之后紧跟的块中有异常抛出，则释放资源的过程中抛出的异常会被忽略；（2）如果 try 之后紧跟的块中没有抛出异常，释放资源的过程中抛出的首个异常将会被抛出（后续释放资源过程中抛出的异常均会被忽略）。

需要说明的是，try-with-resources 表达式中一般没有必要再包含 catch 块和 finally 块，也不建议用户再手动释放资源。因为 try 块执行的过程中无论是否发生异常，所有申请的资源都会被自动释放，并且执行过程中产生的异常均会被向外抛出。但是，如果需要显式地捕获 try 块或资源申请和释放过程中可能抛出的异常并处理，仍可在 try-with-resources 表达式中包含 catch 块和 finally 块：

```
try (input = MyResource(),
    output = MyResource()) {
    while (input.hasNextLine()) {
        let lineString = input.readLine()
```

```
            output.writeLine(lineString)
        }
    } catch (e: Exception) {
        print("Exception happened when executing the try-with-resources expression")
    } finally {
        print("end of the try-with-resources expression")
    }
```

事实上，上述 try-with-resources 表达式等价于下述普通 try 表达式：

```
try {
    var freshExc = None<Exception> // A fresh variable that could store any
    ↪ exceptions
    let input = MyResource()
    try {
        var freshExc = None<Exception>
        let output = MyResource()
        try {
            while (input.hasNextLine()) {
                let lineString = input.readLine()
                output.writeLine(lineString)
            }
        } catch (e: Exception) {
            freshExc = e
        } finally {
            try {
                if (!output.isClosed()) {
                    output.close()
                }
            } catch (e: Exception) {
                match (freshExc) {
                    case Some(v) => throw v // Exception raised from the user code
                    ↪ will be thrown
                    case None => throw e
                }
            }
            match (freshExc) {
                case Some(v) => throw v
                case None => ()
            }
        }
    } catch (e: Exception) {
        freshExc = e
```

```
        } finally {
            try {
                if (!input.isClosed()) {
                    input.close()
                }
            } catch (e: Exception) {
                match (freshExc) {
                    case Some(v) => throw v
                    case None => throw e
                }
            }
            match (freshExc) {
                case Some(v) => throw v
                case None => ()
            }
        }
    } catch (e: Exception) {
        print("Exception happened when executing the try-with-resources expression")
    } finally {
        print("end of the try-with-resources expression")
    }
```

可以看到，try 块（即用户代码）中若抛出的异常会被记录在 freshExc 变量中，并最终被层层向外抛出，其优先级高于释放资源的过程中可能出现的异常。try-with-resources 表达式的类型是 Unit。

12.2 Throw 表达式

throw 表达式的语法定义为：

```
throwExpression
    : 'throw' expression
    ;
```

Throw 表达式由关键字 throw 和一条表达式组成，用于抛出异常。Throw 表达式的类型是 Nothing。需要注意的是，关键字 throw 之后的表达式只能是一个继承于 Exception 或 Error 的类型的对象。Throw 表达式会改变程序的执行逻辑：throw 表达式在执行时会抛出一个异常，捕获此异常的代码块将被执行，而非 throw 后剩余的表达式。

使用 throw 表达式的例子如下：

```
// Catch with exceptionTypePattern.
let listTest = [0, 1, 2]
try {
    throw ArithmeticException()
```

```
    let temp = listTest[0] + 1 // Will never be executed.
} catch (e: ArithmeticException) {
    print("an arithmeticException happened: " + e.toString())
} finally {
    print("the finally block is executed")
}
```

当 throw 表达式抛出一个异常后，必须要能够将其捕获并处理，搜寻异常捕获代码的顺序是函数调用链的逆序：当一个异常被抛出时，首先在抛出异常的函数内搜索匹配的 catch 块，如果未找到，则终止此函数的执行，并在调用这个函数的函数内继续寻找相匹配的 catch 块，若仍未找到，该函数同样需要终止，并继续搜索调用它的函数，以此类推，直到找到相匹配的 catch 块。但是，如果在调用链内的所有函数中均未找到合适的 catch 块，则程序跳转到 Exception 中 terminate 函数内执行，使得程序非正常退出。

下面的例子展示了异常在不同位置被捕获的场景：

```
// Caught by catchE().
func catchE() {
    let listTest = [0, 1, 2]
    try {
        throwE() // caught by catchE()
    } catch (e: IndexOutOfBoundsException) {
        print("an IndexOutOfBoundsException happened: " + e.toString())
    }
}

// Terminate function is executed.
func notCatchE() {
    let listTest = [0, 1, 2]
    throwE()
}

// func throwE()
func throwE() {
    throw IndexOutOfBoundsException()
}
```

第十三章　语言互操作

13.1　C 语言互操作

13.1.1　unsafe 上下文

因为 C 语言太容易造成不安全，所以仓颉规定所有和 C 语言互操作的功能都只能发生在 unsafe 上下文中。unsafe 上下文是用 unsafe 关键字引入的。

unsafe 关键字可以有以下几种用法:

1. 修饰一个代码块。unsafe 表达式的类型就是这个代码块的类型。
2. 修饰一个函数。

所有的 unsafe 函数和 CFunc 类型函数必须在 unsafe 上下文中调用。

所有的 unsafe 函数，均不能作为一等公民使用，包括不能赋值给变量，不能作为实参或返回值使用，不能作为表达式使用，只能在 unsafe 上下文中被调用。

语法定义为:

```
unsafeExpression
    : 'unsafe' '{' expressionOrDeclarations '}'
    ;

unsafeFunction
    : 'unsafe' functionDefinition
    ;
```

13.1.2　调用 C 函数

13.1.2.1　foreign 关键字和 @C

仓颉编程语言要调用 C 函数，需要先在仓颉代码中声明这个函数且用 foreign 关键字修饰。foreign 函数只能存在于 top-level 作用域中，且仅包内可见，故不能使用其他任何函数修饰符。

@C 只支持修饰 foreign 函数、top-level 作用域中的非泛型函数和 struct 类型。在修饰 foreign 函数时，@C 可省略。如未特别说明，C 语言互操作章节中的 foreign 函数均视为 @C 修饰的 foreign 函数。

在 @C 修饰下，foreign 关键字只允许修饰 top-level 作用域中的非泛型函数。foreign 函数只是声明，没有函数体，其参数和返回类型均不可省略。用 foreign 修饰的函数使用原生 C ABI，且不会做名字修饰。

```
foreign func foo(): Unit
foreign var a: Int32 = 0 // compiler error
foreign func bar(): Unit { // compiler error
    return
}
```

多个 foreign 函数声明，可以使用 foreign 块来声明。foreign 块是指在 foreign 关键字后使用一对花括号括起来的声明序列。foreign 块内仅可包含函数。在 foreign 块上添加注解等同于为 foreign 块中的每一个成员加上注解。

```
foreign {
    func foo(): Unit
    func bar(): Unit
}
```

foreign 函数需要能链接到同名的 C 函数，且参数和返回类型需要保持一致。只有满足于 CType 约束的类型可以被用于 foreign 函数的参数和返回类型。关于 CType 的定义，请参考下文的 CType 接口部分。

foreign 函数不支持命名参数和参数默认值。foreign 函数允许变长参数，使用 ... 表达，只能用于参数列表的最后。变长参数均需要满足 CType 约束，但不必是同一类型。

13.1.2.2 CFunc

仓颉中的 CFunc 类型函数是指可以被 C 语言代码调用的函数，共有以下三种形式：

1. @C 修饰的 foreign 函数
2. @C 修饰的仓颉函数
3. 类型为 CFunc 的 lambda 表达式
    - 与普通的 lambda 表达式不同，CFunc lambda 不能捕获变量。

```
// Case 1
foreign func free(ptr: CPointer<Int8>): Unit

// Case 2
@C
func callableInC(ptr: CPointer<Int8>) {
    print("This function is defined in Cangjie.")
}

// Case 3
let f1: CFunc<(CPointer<Int8>) -> Unit> = { ptr =>
    print("This function is defined with CFunc lambda.")
}
```

以上三种形式声明/定义的函数的类型均为 CFunc<(CPointer<Int8>) -> Unit>。CFunc 对应 C 语言的函数指针类型。这个类型为泛型类型，其泛型参数表示该 CFunc 入参和返回值类型，使用方式如下：

```
foreign func atexit(cb: CFunc<()->Unit>)
```

与 foreign 函数一样，其他形式的 CFunc 的参数和返回类型必须满足 CType 约束，且不支持命名参数和参数默认值。

CFunc 在仓颉代码中被调用时，需要处在 unsafe 上下文中。

仓颉支持将一个 CPointer<T> 类型的变量类型转换为一个具体的 CFunc，其中 CPointer 的泛型参数 T 可以是满足 CType 约束的任意类型，使用方式如下：

```
main() {
    var ptr = CPointer<Int8>()
    var f = CFunc<() -> Unit>(ptr)
    unsafe { f() } // core dumped when running, because the pointer is nullptr.
}
```

CFunc 的参数和返回类型不允许依赖外部的泛型参数。

```
func call<T>(f: CFunc<(T) -> Unit>, x: T) where T <: CType { // error
    unsafe { f(x) }
}

func f<T>(x: T) where T <: CType {
    let g: CFunc<(T) -> T> = { x: T => x } // error
}

class A<T> where T <: CType {
    let x: CFunc<(T) -> Unit> // error
}
```

13.1.2.3 inout 参数

在仓颉中调用 CFunc 时，其实参可以使用 inout 关键字修饰组成引用传值表达式，此时，该参数按引用传递。引用传值表达式的类型为 CPointer<T>，其中 T 为 inout 表达式修饰的表达式的类型。

引用传值表达式具有以下约束：

- 仅可用于对 CFunc 的调用处；
- 其修饰对象的类型必须满足 CType 约束，但不可以是 CString；
- 其修饰对象必须是用 var 定义的变量；
- 通过仓颉侧引用传值表达式传递到 C 侧的指针，仅保证在函数调用期间有效，即此种场景下 C 侧不应该保存指针以留作后用。

inout 修饰的变量，可以是定义在 top-level 作用域中的变量、局部变量、struct 中的成员变量，但不能直接或间接来源于 class 的实例。

```
foreign func foo1(ptr: CPointer<Int32>): Unit
```

```
@C
func foo2(ptr: CPointer<Int32>): Unit {
    let n = unsafe { ptr.read() }
    println("*ptr = ${n}")
}

let foo3: CFunc<(CPointer<Int32>) -> Unit> = { ptr =>
    let n = unsafe { ptr.read() }
    println("*ptr = ${n}")
}

struct Data {
    var n: Int32 = 0
}

class A {
    var data = Data()
}

main() {
    var n: Int32 = 0
    unsafe {
        foo1(inout n)  // OK
        foo2(inout n)  // OK
        foo3(inout n)  // OK
    }
    var data = Data()
    var a = A()
    unsafe {
        foo1(inout data.n)   // OK
        foo1(inout a.data.n) // Error, n is derived indirectly from instance member
         ↪ variables of class A
    }
}
```

13.1.2.4 调用约定

函数调用约定描述调用者和被调用者双方如何进行函数调用（如参数如何传递、栈由谁清理等），函数调用和被调用双方必须使用相同的调用约定才能正常运行。仓颉编程语言通过 `@CallingConv` 来表示各种调用约定，支持的调用约定如下：

- **CDECL**, CDECL 表示 clang 的 C 编译器在不同平台上默认使用的调用约定。

- **STDCALL**, STDCALL 表示 Win32 API 使用的调用约定。

通过 C 语言互操作机制调用的 C 函数，未指定调用约定时将采用默认的 CDECL 调用约定。如下调用 C 标准库函数 clock 示例：

```
@CallingConv[CDECL]   // Can be omitted in default.
foreign func clock(): Int32

main() {
    println(clock())
}
```

@CallingConv 只能用于修饰 foreign 块、单个 foreign 函数和 top-level 作用域中的 CFunc 函数。当 @CallingConv 修饰 foreign 块时，会为 foreign 块中的每个函数分别加上相同的 @CallingConv 修饰。

13.1.3 类型映射

在仓颉编程语言中声明 foreign 函数时，参数以及返回值的类型必须和要调用的 C 函数的参数和返回类型相一致。仓颉编程语言类型与 C 语言类型不同，有些简单值类型，如 C 语言 int32_t 类型可以直接使用仓颉编程语言 Int32 与之对应，但是一些相对复杂类型如结构体则需要在仓颉侧声明对应的同样内存布局的类型。仓颉编程语言可以用于和 C 交互的类型都满足 CType 约束，它们可以分成基础类型和复杂类型。基础类型包括整型、浮点型、Bool 类型、CPointer 类型、Unit 类型。复杂类型包括用 @C 修饰的 struct、CFunc 类型等。

13.1.3.1 基础类型

仓颉编程语言和 C 之间传递参数时，基础的值类型会进行复制，如 int、short 等。仓颉编程语言与 C 语言相匹配的基础类型映射关系表，如下：

Cangjie Type	C type	Size
Unit	void	0
Bool	bool	1
Int8	int8_t	1
UInt8	uint8_t	1
Int16	int16_t	2
UInt16	uint16_t	2
Int32	int32_t	4
UInt32	uint32_t	4
Int64	int64_t	8
UInt64	uint64_t	8
IntNative	ssize_t	platform dependent
UIntNative	size_t	platform dependent
Float32	float	4

Cangjie Type	C type	Size
Float64	double	8

注意： 1. int 类型、long 类型等由于其在不同平台上的不确定性，需要程序员自行指定对应仓颉编程语言类型。2. IntNative/UIntNative 在 C 互操作场景中，其与 C 语言中的 ssize_t/size_t 是一致的。3. 在 C 互操作场景中，与 C 语言类似，Unit 类型仅可作为 CFunc 中的返回类型和 CPointer 的泛型参数。

仓颉编程语言中 foreign 函数的参数、返回值的类型需要与 C 函数参数、返回值的类型相对应。对于类型映射关系明确且平台无关的类型（参考基础类型映射关系表），可以直接使用标准对应的仓颉编程语言基础类型。比如在 C 语言中有一个 add 函数声明如下：

```
int64_t add(int64_t X,  int64_t Y) { return X+Y; }
```

在仓颉编程语言中调用 add 函数，代码示例如下：

```
foreign func add(x: Int64, y: Int64): Int64

main() {
    let x1: Int64 = 42
    let y1: Int64 = 42
    var ret1 = unsafe { add(x1, y1) }
    ...
}
```

13.1.3.2 指针

仓颉编程语言提供 CPointer<T> 类型对应 C 语言的指针 T* 类型，其中 T 必须满足 CType 约束。CPointer 类型必须满足：

- 大小和对齐与平台相关
- 对它做加减法算术运算、读写内存，是需要在 unsafe 上下文操作的
- CPointer<T1> 可以在 unsafe 上下文中使用类型强制转换，变成 CPointer<T2> 类型

CPointer 有一些成员方法，如下所示：

```
func isNull() : bool

// operator overloading
unsafe operator func + (offset: int64) : CPointer<T>
unsafe operator func - (offset: int64) : CPointer<T>

// read and write access
unsafe func read() : T
unsafe func write(value: T) : Unit
```

```
// read and write with offset
unsafe func read(idx: int64) : T
unsafe func write(idx: int64, value: T) : Unit
```

CPointer 可以使用类型名构造一个实例，它的值对应 C 语言的 NULL。

```
func test() {
    let p = CPointer<Int64>()
    let r1 = p.isNull() // r1 == true
}
```

仓颉支持将一个 CFunc 变量的类型转换为一个 CPointer 类型，其中 CPointer 的泛型参数 T 可以是满足 CType 约束的任意类型，使用方式如下：

```
foreign func rand(): Int32
main() {
    var ptr = CPointer<Int8>(rand)
    0
}
```

13.1.3.3 字符串

C 语言字符串实际是以'\0'终止的一维字符数组。仓颉编程语言提供 CString 与 C 语言字符串相匹配。通过 CString 的构造函数或 LibC 的 mallocCString 创建 C 语言字符串，如需在仓颉端释放，则调用 LibC 的 free 方法。

声明 foreign 函数时，需要根据要调用的 C 语言函数的声明来确定函数名称、参数类型、返回值类型。C 语言标准库 printf 函数的声明如下：

```
int printf(const char *format, ...)
```

参数 const char * 类型对应仓颉的类型为 CString。返回类型 int 对应的仓颉的 Int32 类型。创建字符串并调用 printf 函数示例如下：

```
package demo

foreign func printf(fmt: CString, ...): Int32

main() {
    unsafe {
        let str: CString = LibC.mallocCString("hello world!\n")
        printf(str)
        LibC.free(str)
    }
}
```

13.1.3.4 Array 类型

仓颉的 Array 类型是不满足 CType 约束的，因此它不能用于 foreign 函数的参数和返回值。但是当 Array 内部的元素类型满足 CType 约束的时候，仓颉允许使用下面这两个函数获取和释放指向数组内部元素的指针。

```
unsafe func acquireArrayRawData<T>(arr: Array<T>): CPointerHandle<T> where T <:
 ↪ CType
unsafe func releaseArrayRawData<T>(h: CPointerHandle<T>): Unit where T <: CType

struct CPointerHandle<T> {
    let pointer: CPointer<T>
    let array: Array<T>
}
```

参考如下示例，假设我们要把一个 Array<UInt8> 写入到文件中，可以这样做：

```
foreign func fwrite(buf: CPointer<UInt8>, size: UIntNative, count: UIntNative,
 ↪ stream: CPointer<Unit>): UIntNative

func writeFile(buffer: Array<UInt8>, file: CPointer<Unit>) {
    unsafe {
        let h = acquireArrayRawData(buffer)
        fwrite(h.pointer, 1, buffer.size, file)
        releaseArrayRawData(h)
    }
}
```

13.1.3.5 VArray 类型

仓颉使用 VArray 类型与 C 数组类型映射。当 VArray<T, $N> 中的元素类型 T 满足 CType 约束时，VArray<T, $N> 类型也满足 CType 约束。

```
struct A {} // A is not a CType.
let arr1: VArray<A, $2> // arr1 is not a CType.
let arr2: VArray<Int64, $2> // arr2 is a CType.
```

VArray 允许作为 CFunc 签名的参数类型，但不允许为返回类型。

如果 CFunc 签名中参数类型被声明为 VArray<T, $N>，对应的实参也只能是被 inout 修饰的 VArray<T, $N> 类型的表达式，但参数传递时，仍然以 CPointer<T> 传递。

如果 CFunc 签名中参数类型被声明为 CPointer<T>，对应的实参可以是被 inout 修饰的 VArray<T, $N> 类型的表达式，参数传递时，仍然为 CPointer<T> 类型。

CPointer<VArray<T, $N>> 等同于 CPointer<T>。

```
foreign func foo1(a: VArray<Int32, $3>): Unit
foreign func foo2(a: CPointer<Int32>): Unit
```

```
var a: VArray<Int32, $3> = [1, 2, 3]
unsafe {
    foo1(inout a) // Ok.
    foo2(inout a) // Ok.
}
```

13.1.3.6 结构体

foreign 函数的签名中包含结构体类型时，需要在仓颉侧定义同样内存布局的 struct，使用 @C 修饰。

参考如下示例，一个 C 图形库（libskialike.so）中有一个计算两点之间距离的函数 distance，其在 C 语言头文件中相关结构体和函数声明如下：

```
struct Point2D {
    float x;
    float y;
};
float distance(struct Point2D start, struct Point2D end);
```

声明 foreign 函数时，需要根据要调用的 C 语言函数的声明来确定函数名称、参数类型、返回值类型。当创建 C 侧结构体时，需要确定结构体各个成员名称和类型。代码示例如下：

```
package demo

@C
struct Point2D {
    var x: Float32
    var y: Float32
}

foreign func distance(start: Point2D, end: Point2D): Float32
```

用 @C 修饰的 struct 必须满足以下限制：

- 成员变量的类型必须满足 CType 约束
- 不能实现或者扩展 interfaces
- 不能作为 enum 的关联值类型
- 不允许被闭包捕获
- 不能具有泛型参数

用 @C 修饰的 struct 自动满足 CType 约束。

@C struct 中的 VArray 类型成员变量保证与 C 中数组的内存布局一致。

例如，对于以下 C 的结构体类型：

```
struct S {
    int a[2];
    int b[0];
}
```

在仓颉中，可以声明为如下结构体与 C 代码对应：

```
@C
struct S {
    var a: VArray<Int32, $2> = VArray<Int32, $2>(item: 0)
    var b: VArray<Int32, $0> = VArray<Int32, $0>(item: 0)
}
```

注意：C 语言中允许结构体的最后一个字段为未指明长度的数组类型，该数组被称为柔性数组（flexible array），仓颉不支持包含柔性数组的结构体的映射。

13.1.3.7 函数

仓颉中的函数类型是不满足 CType 约束的，故提供了 CFunc 作为 C 语言中函数指针的映射。如下 C 语言代码中定义的函数指针类型，在仓颉中可以映射为 CFunc<() -> Unit>。

```
// Function pointer in C.
typedef void (*FuncPtr) ();
```

CFunc 的具体细节参见前面 CFunc 一节。

13.1.4 CType 接口

CType 接口是一个语言内置的空接口，它是 CType 约束的具体实现，所有 C 互操作支持的类型都隐式地实现了该接口，因此所有 C 互操作支持的类型都可以作为 CType 类型的子类型使用。

```
@C
struct Data {}

@C
func foo() {}

main() {
    var c: CType = Data() // ok
    c = 0 // ok
    c = true // ok
    c = CString(CPointer<UInt8>()) // ok
    c = CPointer<Int8>() // ok
    c = foo // ok
}
```

CType 接口是仓颉中的一个 interface 类型，它本身不满足 CType 约束。同时，CType 接口不允许被继承、显式实现、扩展。

CType 接口不会突破其子类型的使用限制。

```
@C
struct A {} // implicit implement CType

class B <: CType {} // error

class C {} // error

extend C <: CType {} // error

class D<T> where T <: CType {}

main() {
    var d0 = D<Int8>() // ok
    var d1 = D<A>() // ok
}
```

第十四章　元编程

元编程允许把代码表示成可操作的数据对象，可以对其增、删、改、查。为此，仓颉编程语言提供了编译标记用于元编程。编译标记分为两类：宏和注解。宏机制支持基于 Tokens 类型的编译期代码变换，支持将代码转为数据、拼接代码等基础操作。

14.1　quote 表达式和 Tokens 类型

仓颉通过 quote 表达式引用具体代码，表示成可操作的数据对象。quote 表达式的语法定义为：

```
quoteExpression
    : 'quote' quoteExpr
    ;

quoteExpr
    : '(' quoteParameters ')'
    ;

quoteParameters
    : (quoteToken | quoteInterpolate | macroExpression)+
    ;

quoteToken
    : '.' | ',' | '(' | ')' | '[' | ']' | '{' | '}' | '**' | '*' | '%' | '/' | '+'
    ↪ | '-'
    | '|>' | '~>'
    | '++' | '--' | '&&' | '||' | '!' | '&' | '|' | '^' | '<<' | '>>' | ':' | ';'
    | '=' | '+=' | '-=' | '*=' | '**=' | '/=' | '%='
    | '&&=' | '||=' | '&=' | '|=' | '^=' | '<<=' | '>>='
    | '->' | '=>' | '...' | '..=' | '..' | '#' | '@' | '?' | '<:' | '<' | '>' |
    ↪ '<=' | '>='
    | '!=' | '==' | '_' | '\\' | '`' | '$'
    | 'Int8' | 'Int16' | 'Int32' | 'Int64' | 'UInt8' | 'UInt16' | 'UInt32' |
    ↪ 'UInt64' | 'Float16'
```

```
    | 'Float32' | 'Float64' | 'Rune' | 'Bool' | 'Unit' | 'Nothing' | 'struct' |
    ↪ 'enum' | 'This'
    | 'package' | 'import' | 'class' | 'interface' |'func' | 'let' | 'var' | 'type'
    | 'init' | 'this' | 'super' | 'if' | 'else' | 'case' | 'try' | 'catch' |
    ↪ 'finally'
    | 'for' | 'do' | 'while' | 'throw' | 'return' | 'continue' | 'break' | 'as' |
    ↪ 'in' | '!in'
    | 'match' | 'from' | 'where' | 'extend' | 'spawn' | 'synchronized' | 'macro' |
    ↪ 'quote' | 'true' | 'false'
    | 'sealed' | 'static' | 'public' | 'private' | 'protected'
    | 'override' | 'abstract' | 'open' | 'operator' | 'foreign'
    | Identifier | DollarIdentifier
    | literalConstant
    ;

quoteInterpolate
    : '$' '(' expression ')'
    ;
```

quote 表达式的语法规则总结如下:

- 通过关键字 quote 定义 quote 表达式。
- quote 表达式由 () 括起，内部引用的可以是代码（Token）、代码插值、宏调用表达式。
- quote 表达式为 Tokens 类型。Tokens 是仓颉标准库提供的类型，是由词法单元（token）组成的序列。

上述语法定义中的 quoteToken 是指仓颉编译器支持的任意合法 token，注释、终结符等不被 **Lexer** 解析的 **token** 除外。但当换行符作为两条表达式分隔符时，不会被忽略，它会被解析成一个 quoteToken。如下的例子，只有两条赋值表达式之间的换行符会被解析成 quoteToken，quote 表达式里的其他换行符都会被忽略。

```
// The newline character between assignment expression is preserved, others are
↪ ignored.
quote(

var a = 3
var b = 4

)
```

不允许未经过代码插值的连续 quote，关于代码插值，详见下一节。当 quote 表达式引用如下 token 时，需要进行转义:

- quote 表达式中不允许不匹配的小括号，但是通过\转义的小括号，不计入匹配规则。

- 当 @ 表示普通 token，而非宏调用表达式时，需要通过\进行转义。
- 当 $ 表示普通 token，而非代码插值时，需要通过\进行转义。

注：本章所提到的 token（字母均小写），指仓颉 Lexer 解析出来的词法单元。

下面是一些以 Tokens 或代码插值为参数的 quote 表达式用例：

```
let a0: Tokens = quote(==)  // ok
let a1: Tokens = quote(2+3) // ok
let a2: Tokens = quote(2) + quote(+) + quote(3)  // ok
let a3: Tokens = quote(main(): Int64 {
    0
}) // ok
let a4: Tokens = quote(select * from Users where id=100086) // ok

let b0: Tokens = quote(()  // error: unmatched `(`
let b1: Tokens = quote(quote(x))  // ok -- `b1.size == 4`
```

quote 表达式还可以引用宏调用表达式，例如：

```
quote(@SayHi("say hi")) // a quoted, un-expanded macro -- macro expansion happens
 ↪  later
```

仓颉编程语言提供 Tokens 类型表示代码从原始文本信息解析后的 token 序列。Tokens 支持通过 operator + 将两个对象的 token 结果拼接到新对象。下面例子里 a1 和 a2 基本等价。

```
let a1: Tokens = quote(2+3) // ok
let a2: Tokens = quote(2) + quote(+) + quote(3) // ok
```

14.1.1 代码插值

在 quote 表达式里使用 $ 作为代码插值运算符，运算符后面接表达式，表示将该表达式的值转成 tokens。该运算符为一元前缀运算符，只能用在 quote 表达式中，优先级比其他运算符高。

代码插值可作用于所有仓颉合法表达式。代码插值本身不是表达式，不具备类型。

```
var rightOp: Tokens = quote(3)
quote(2 + $rightOp)        // quote(2 + 3)
```

能被代码插值的表达式类型必须实现 ToTokens 接口。ToTokens 接口形式如下：

```
interface ToTokens  {
    func toTokens(): Tokens
}
```

- 仓颉中大部分的值类型：数值类型、Rune 类型、Bool 类型、String 类型已有默认实现。
- Tokens 类型默认实现 ToTokens 接口，toTokens 返回它自己。
- 用户自定义数据结构须主动实现 ToTokens 接口。仓颉标准库提供了丰富的生成 Tokens 的接口。例如，对于 String 类型的变量 str，用户可以直接使用 str.toTokens() 获取 str 对应的 Tokens。

14.1.2 quote 表达式求值规则

quote 表达式对括号内的引用规则总结如下：

- 代码插值：取被插值的表达式.toTokens() 结果。
- 普通 tokens：取对应的 Tokens 结果。

quote 表达式根据以上情况再按照表达式出现的顺序拼接成更大的 Tokens。
下面是一些 quote 表达式求值示例，注释里表示程序执行返回的 Tokens 结果。

```
var x: Int64 = 2 + 3

// tokens in the quote are obtained from the corresponding Tokens.
quote(x + 2)        // quote(x + 2)

// The Tokens type can only add with Tokens type.
quote(x) + 1        // error! quote(x) is Tokens type, can't add with integer

// The value of code interpolation in quote equals to the tokens result
 ↪  corresponding to the value of the interpolation expression.
quote($x)           // quote(5)
quote($x + 2)       // quote(5 + 2)
quote($x + (2 + 3)) // quote(5 + (2 + 3))
quote(1 + ($x + 1) * 2)    // quote(1 + (5 + 1) * 2)
quote(1 + $(x + 1) * 2)    // quote(1 + 6 * 2)

var t: Tokens = quote(x)   // quote(x)

// without interpolation, the `t` is the token `t`
quote(t)                   // quote(t)

// with interpolation, `t` is evaluated and expected to implement ToTokens
quote($t)                  // quote(x)
quote($t+1)                // quote(x+1)

// quote expressions can be used inside of interpolations, and cancel out
quote($(quote(t)))         // quote(t)
quote($(quote($t)))        // quote(x)

quote($(t+1))              // error! t is Tokens type, can't add with integer

public macro PlusOne(input: Tokens): Tokens {
  quote(($input + 1))
}
```

```
// Macro invocations are preserved, unexpanded, until they are re-inserted into
 ↪  the code by
// the macro expander. However, interpolation still happens, including among the
 ↪  arguments
// to the macro
quote(@PlusOne(x))           // quote(@PlusOne(x))
quote(@PlusOne($x))          // quote(@PlusOne(5))
quote(@PlusOne(2+3))         // quote(@PlusOne(2+3))
quote(1 + @PlusOne(x) * 2)   // quote(1 + @PlusOne(x) * 2)

// When the macro invocation is outside the quote, the macro expansion happens
 ↪  early
var y: Int64 = @PlusOne(x)   // 5 + 1
quote(1 + $y * 2 )           // quote(1 + 6 * 2)
```

14.2 宏

宏是实现元编程的重要技术。宏和函数的相同点在于都可以被调用，具有输入和输出。不同点在于宏代码在编译期进行展开，仓颉将宏调用表达式替换为展开后的代码。用户可通过宏编写代码模板（即生成代码的代码）。另外，宏提供自定义文法的能力。用户可基于宏灵活定义自己的 DSL 文法。最后，仓颉提供丰富的代码操作接口供用户方便地完成代码变换。

宏分为无属性宏和有属性宏两类。相对于无属性宏，有属性宏可根据属性不同对宏的输入进行相应的分析变换得到不同的输出结果。

14.2.1 宏定义

仓颉编程语言中，宏定义遵循以下语法规则：

```
macroDefinition
    : 'public' 'macro' identifier
    (macroWithoutAttrParam | macroWithAttrParam)
    (':' identifier)?
    ('=' expression | block)
    ;

macroWithoutAttrParam
    : '(' macroInputDecl ')'
    ;

macroWithAttrParam
    : '(' macroAttrDecl ',' macroInputDecl ')'
```

```
    ;

macroInputDecl
    : identifier ':' identifier
    ;

macroAttrDecl
    : identifier ':' identifier
    ;
```

总结如下：

- 通过关键字 macro 定义一个宏。
- macro 前需要有 public 修饰，表示对包外部可见。
- macro 后需要带有宏的名字。
- 宏的参数由 () 括起。无属性宏固定 1 个 Tokens 类型参数对应宏的输入，有属性宏固定 2 个 Tokens 类型依次对应宏的属性和输入。
- 可缺省的函数返回类型固定为 Tokens。

下面是无属性宏的示例。它包含了宏定义具备的所有要素：public 表示对包外部可见；macro 关键字；foo 是宏的标识符；形参 x 和它的类型 Tokens；返回值和输入同值、同类型。

```
public macro foo(x: Tokens): Tokens { x }

public macro bar(x: Tokens): Tokens {
  return quote($x)      // or just `return x`
}
```

下面是有属性宏的示例。和无属性宏相比，多一个 Tokens 类型的输入，宏定义体内可以进行更灵活的操作。

```
public macro foo(attr: Tokens, x: Tokens): Tokens { attr + x }

public macro bar(attr: Tokens, x: Tokens): Tokens {
    return quote($attr + $x)
}
```

宏一旦被定义，宏名字不能再被赋值。另外，宏对参数类型、参数个数有严格要求。

14.2.2 宏调用

宏调用表达式遵循以下语法规则：

```
macroExpression
    : '@' identifier macroAttrExpr?
    (macroInputExprWithoutParens | macroInputExprWithParens)
    ;

macroAttrExpr
    : '[' quoteToken* ']'
    ;

macroInputExprWithoutParens
    : functionDefinition
    | operatorFunctionDefinition
    | staticInit
    | structDefinition
    | structPrimaryInit
    | structInit
    | enumDefinition
    | caseBody
    | classDefinition
    | classPrimaryInit
    | classInit
    | interfaceDefinition
    | variableDeclaration
    | propertyDefinition
    | extendDefinition
    | macroExpression
    ;

macroInputExprWithParens
    : '(' macroTokens ')'
    ;

macroTokens
    : (quoteToken | macroExpression)*
    ;
```

宏调用表达式规则总结如下:

- 通过关键字 `@` 定义宏调用表达式。
- 宏调用使用 `()` 括起来。括号里面可以是任意合法 `tokens`，不可以是空。
- 调用有属性宏时，宏属性使用 `[]` 括起来。里面可以是任意合法 `tokens`，不可以是空。

当宏调用表达式引用如下 `token` 时，需要转义:

- 宏调用表达式参数中不允许不匹配的小括号，例如 `@ABC(we have two (( and one ))`。但通过\转义的小括号，不计入匹配规则。
- 有属性宏的中括号内不允许不匹配的中括号，例如 `@ABC[we have two [[ and one ]]()`。但通过\进行转义的中括号，不计入匹配规则。
- 宏调用表达式参数中引用 `@` 时，需要通过\转义。

宏调用用于某些声明或表达式前面的时候可省略括号，其语义和不带括号的宏调用有所不同。带括号的宏调用，其参数可以是任意合法 tokens；不带括号的宏调用，其参数必须是以下声明或表达式中的一种。

- 函数声明
- `struct` 声明
- `enum` 声明
- `enum constructor`
- 类声明
- 静态初始化器
- 类和 `struct` 构造函数和主构造函数
- 接口声明
- 变量声明
- 属性声明
- 扩展声明
- 宏调用表达式

另外，对于不带括号的宏调用，其参数还必须满足：

- 如果参数是声明，那么该宏调用只能出现在该声明允许出现的位置。

下面是无属性宏、有属性宏、不带括号的宏调用的示例。

```
func foo()
{
    print("In foo\n")
}

// Non-attribute macros
public macro Twice(input: Tokens): Tokens
{
    print("Compiling the macro `Twice` ...\n")
    quote($input; $input)
}

@Twice(foo())  // After Macro expand: foo(); foo()
@Twice()       // error, parameters in macro invocation can not be empty.

// Attributed macros
```

```
public macro Joint(attr: Tokens, input: Tokens): Tokens
{
    print("Compiling the macro `Joint` ...\n")
    quote($attr; $input)
}

@Joint[foo()](foo()) // After Macro expand: foo(); foo()
@Joint[foo()]()      // error, parameters in macro invocation can not be empty.
@Joint[](foo())      // error, attribute in macro invocation can not be empty.

// Non-attribute macros
public macro MacroWithoutParens(input: Tokens): Tokens
{
    print("Compiling the macro `MacroWithoutParens` ...\n")
    quote(func foo() { $input })
}

@MacroWithoutParens
var a: Int64 = 0  // After Macro expand: func foo() { var a: Int64 = 0 }

public macro echo(input: Tokens) {
    return input
}

@echo class A {}  // ok, class can only be defined in top-level, so is current
 ↪  macro invocation

func goo() {
  @echo func tmp() {} // ok, function can be defined in another function body,
                      // so is current macro invocation
  @echo class B {}    // error, class can only be defined in top-level, so is
   ↪  current macro invocation
}
```

宏调用作为一种表达式，它可以出现在任意表达式允许出现的位置上。另外宏调用还可以出现在如下声明可以出现的位置：函数声明、struct 声明、enum 声明、类声明、接口声明、变量声明（函数参数除外）、属性声明、扩展声明。

宏只在编译期可见。宏调用表达式在编译期会被宏展开后的代码替换。宏展开是指在代码编译时，宏定义会被执行，执行后的结果会被解析成仓颉语法树，替换宏调用表达式。替换后的节点经过语义分析之后拥有相应的类型信息，该类型可以认为是宏调用表达式的类型。例如上述例子中，当用户执行 `@Twice`，`@Joint`，`@MacroWithoutParens` 这些宏调用的时候，宏定义中的代码会被编译并执行，打印出 `Compiling the macro ...` 等字样，执行后的结果作为新的语法树节点，替换掉原来的宏调用表达式。

当宏调用出现在不同的上下文时，需要满足以下规则：

1. 如果宏调用出现在期望表达式或者声明的上下文里，那么宏调用会在编译期被展开成程序文本。

2. 如果宏调用出现在期望 token 序列的上下文里（作为宏调用表达式或 quote 表达式的参数），那么宏调用会在该 token 序列被求值的时刻调用，而且宏调用的返回值（token 序列）会被直接使用而不展开成程序文本。

14.2.3 宏作用域和包的导入

宏必须在源文件顶层定义，作用域是整个 package。

宏定义所在的 package，需使用 macro package 来声明，被 macro package 限定的包，仅允许宏定义声明对外可见，其他声明仅包内可见，其他声明被修饰为对外可见时将报错。

```
//define.cj
macro package define         // modify package with macro
import std.ast.*
// public func A(){}       // error: func A can not be modified by `public`
public macro M(input:Tokens): Tokens{ // only macros can be modified by `public`
    return input
}

// call.cj
package call
import define.*
main(){
    @M()
    return 0
}
```

宏的导入遵守仓颉通用的包导入规则。

需要特殊说明的是，macro package 仅允许被 macro package 重导出。

```
// A.cj
macro package A
public macro M1(input:Tokens):Tokens{
    return input
}

// B.cj
package B
//public import A.* // error: macro packages can only be `public import` in macro
 ↪ packages
public func F1(input:Int64):Int64{
    return input
```

```
}

// C.cj
macro package
public import A.*  // only macro packages can be public import in macro packages
// public import B.*  // error: normal packages can not be public import in macro
 ↪ packages
import B.*
public macro M2(input:Tokens):Tokens{
    return @M1(input) + Token(TokenKind.ADD) + quote($(F1(1)))
}
```

在同一个 package 中，宏对于同名的约定和函数一致，属性宏和非属性宏的规则如下表所示:

Same name, Same package	attribute macros	non-attribute macros
attribute macros	NO	YES
non-attribute macros	YES	NO

14.2.4 嵌套宏和递归宏

宏允许嵌套调用其他宏。

宏调用表达式包含宏调用时，如 `@Outer @Inner(2+3)`，优先执行内层的宏，再执行外层的宏。内层宏返回的 Tokens 结果和其他 Tokens 拼接一起交给外层宏调用。内层宏和外层宏可以是不同的宏，也可以是同一个宏。

内层宏可以调用库函数 `AssertParentContext` 来保证内层宏调用一定嵌套在特定的外层宏调用中。如果内层宏调用这个函数时没有嵌套在给定的外层宏调用中，该函数将抛出一个错误。第二个函数 `InsideParentContext` 只在内层宏调用嵌套在给定的外层宏调用中时返回 true。

```
public macro Inner(input: Tokens): Tokens {
    AssertParentContext("Outer")
    // ...or...
    if (InsideParentContext("Outer")) {
        // ...
    }
}
```

内层宏也可以通过发送键/值对的方式与外层宏通信。当内层宏执行时，通过调用标准库函数 `SetItem` 发送信息；随后，当外层宏执行时，调用标准库函数 `GetChildMessages` 接收每一个内层宏发送的信息（一组键/值对映射）。

```
public macro Inner(input: Tokens): Tokens {
    AssertParentContext("Outer")
    SetItem("key1", "value1")
```

```
        SetItem("key2", "value2")
        // ...
    }

    public macro Outer(input: Tokens): Tokens {
        let messages = GetChildMessages("Inner")
        for (m in messages) {
            let value1 = m.getString("key1")
            let value2 = m.getString("key2")
            // ...
        }
    }
```

宏定义体包含宏调用时，如果宏调用出现在期望 token 序列的上下文里，则两个宏可以是不同的宏或同一个宏（即支持递归）。否则，被嵌套调用的宏不能和调用的宏相同。具体的使用可以参考下面两个例子。

下面例子里，嵌套调用的宏出现在 quote 表达式，则支持递归调用：

```
    public macro A(input: Tokens): Tokens {
        print("Compiling the macro A ...\n")
        let tmp = A_part_0(input)
        if cond {
            return quote($tmp)
        }
        let bb: Tokens = quote(@A(quote($tmp)))  // ok
        A_part_1()
    }

    main():Int64 {
        var res: Int64 = @A(2+3)  // ok, @A will be treated as Int64 after macro expand
        return res
    }
```

在这个例子中，当宏 A 未被外部调用时，宏 A 不会被执行（即使内部调用了自己），即不会打印 Compiling the macro A ...。if cond 是递归的终止条件。注意：宏递归和函数递归有类似的约束，需要有终止条件，否则会陷入死循环，导致编译无法停止。

下面例子里，嵌套调用的宏出现的地方不在 quote 表达式，则不支持递归调用：

```
    public macro A(input: Tokens): Tokens {
        let tmp = A_part_0(input)
        if cond {
            return quote($tmp)
        }
        let bb: Tokens = @A(quote($tmp))  // error, recursive macro expression not in
        ↪ quote
```

```
    A_part_1()
}

main():Int64 {
    var res: Int64 = @A(2+3)  // error, type mismatch
    return res
}
```

总结下宏嵌套调用和递归调用规则：

- 宏调用表达式允许嵌套调用。
- 宏定义允许嵌套调用其他宏，但只允许在 quote 表达式中递归调用自己。

14.2.5 限制

- 宏有条件地支持递归调用自己。具体情况请参考前面小节说明。
- 除了宏递归调用外，宏定义和宏调用必须位于不同的 package。宏调用的地方必须 import 宏定义所在的 package，保证宏定义比宏调用点先编译。不支持宏的循环依赖导入。例如下面的用法是非法的：pkgA 导入 pkgB，pkgB 又导入 pkgA，存在循环依赖导入。

```
// ======= file A.cj
macro package pkgA
import pkgB.*
public macro A(..) {
  @B(..)  // error
}

// ======= file B.cj
macro package pkgB
import pkgA.*
public macro B(..) {
  @A(..)  // error
}
```

- 宏允许嵌套调用其他宏。被调用的宏也要求定义点和调用点必须位于不同的 package。

14.2.6 内置编译标记

14.2.6.1 源码位置

仓颉提供了几个内置编译标记，用于在编译时获取源代码的位置。

- @sourcePackage() 展开后是一个 String 类型的字面量，内容为当前宏所在的源码的包名
- @sourceFile() 展开后是一个 String 类型的字面量，内容为当前宏所在的源码的文件名

- `@sourceLine()` 展开后是一个 `Int64` 类型的字面量，内容为当前宏所在的源码的代码行

- `@sourcePackage()` is expanded to a `String` literal, which is the name of the current package - `@sourceFile()` is expanded to a `String` literal, which is the name of the current source file - `@sourceLine()` is expanded to an `Int64` literal, which is the source file line number

这几个编译标记可以在任意表达式内部使用，只要能符合类型检查规则即可。示例如下：

```
func test1() {
    let s: String = @sourceFile()  // The value of `s` is the current source file
    ↪ name
}

func test2(n!: Int64 = @sourceLine()) { /* at line 5 */
    // The default value of `n` is the source file line number of the definition of
    ↪ `test2`
    println(n) // print 5
}
```

14.2.6.2 @Intrinsic

@Intrinsic 标记可用于修饰全局函数。@Intrinsic 修饰的函数是由编译器提供的特殊函数。

1. @Intrinsic 修饰的函数不允许写函数体。
2. @Intrinsic 修饰的函数，是编译器决定的一份名单，跟随标准库发布。名单之外的任何其它函数用 @Intrinsic 修饰都会报错。
3. @Intrinsic 标记只在 std module 内可见，std module 外用户可以定义自己的叫做 Intrinsic 的宏或者注解，不会发生名字冲突。
4. @Intrinsic 修饰的函数，不允许作为一等公民使用。

@Intrinsic directive can modify global functions. These functions are special functions provided by the compiler.

1. Functions modified by @Intrinsic are not allowed to have function body.
2. The list of @Intrinsic functions are determined by compiler, and released with std library. Other functions outside of this list can not be modified by @Intrinsic.
3. @Intrinsic directive is only visible inside std module. User defined macro or annotation with the same name "Intrinsic" does not conflict with the one in std module.
4. @Intrinsic modified function is not allowed to be used as a first-class citizen.

示例代码如下：

```
@Intrinsic
func invokeGC(heavy: Bool):Unit

public func GC(heavy!: Bool = false): Unit {
    unsafe { return invokeGC (heavy) }  // CJ_MCC_InvokeGC(heavy)
}
```

14.2.6.3 @FastNative

为了提升与 C 语言互操作的性能，仓颉提供 @FastNative 用于优化对 C 函数的调用。值得注意的是 @FastNative 只能用于 foreign 声明的函数。

开发者在使用 @FastNative 修饰 foreign 函数时，应确保对应的 C 函数满足以下两点要求：1. 函数的整体执行时间不宜太长。例如：不允许函数内部存在很大的循环；不允许函数内部产生阻塞行为，如，调用 sleep、wait 等函数。2. 函数内部不能调用仓颉方法。

14.2.6.4 条件编译

条件编译是一种在程序代码中根据特定条件选择性地编译不同代码段的技术。条件编译的作用主要体现在以下几个方面:

- 平台适应：支持根据当前的编译环境选择性地编译代码，用于实现跨平台的兼容性。
- 功能选择：支持根据不同的需求选择性地启用或禁用某些功能，用于实现功能的灵活配置。例如，选择性地编译包含或排除某些功能的代码。
- 调试支持：支持调试模式下编译相关代码，用于提高程序的性能和安全性。例如，在调试模式下编译调试信息或记录日志相关的代码，而在发布版本中将其排除。
- 性能优化：支持根据预定义的条件选择性地编译代码，用于提高程序的性能。

条件编译遵循以下语法规则:

{{conditionalCompilationDefinition|pretty}}

条件编译语法规则总结如下:

- 内置标记 @When 只能用于声明节点和导入节点。
- 编译条件使用 [] 括起来，[] 内支持输入一组或多组编译条件（请参见多条件编译）。

@When[...] 作为一种内置编译标记，在导入前处理，由宏展开生成的代码中含有 @When[...] 会编译报错。

As a builtin compiler directive, @When[...] is processed before import. If the code generated by macro expansion contains @When[...], a compilation error is reported.

编译条件 编译条件主要由关系表达式或逻辑表达式组成，编译器根据编译条件评估获取一个布尔值，从而决定编译哪段代码。条件变量是编译器计算编译条件的基准，根据条件变量是否由编译器提供可将条件变量分为内置条件变量和用户自定义条件变量。同时，条件变量的作用域仅限于 @When 的 [] 内, 其他作用域内使用未定义的条件变量标识符会触发未定义错误。

编译器内置条件变量 编译器为条件编译提供了五个内置条件变量: os、backend、cjc_version、debug 和 test 用于获取当前构建环境中对应的值，内置条件变量支持比较运算和逻辑运算。其中，os、backend 和 cjc_version 三个变量支持比较运算，在比较运算中条件变量只能作为二元操作符的左操作数，二元操作符的右操作数必须是一个 String 类型的字面量值；逻辑运算仅适用于条件变量 debug 和 test。

os 表示当前编译环境所在的操作系统。它用于实时获取编译器所在的具体操作系统类型，进而与目标操作系统的字面量值构成一个编译条件。如果想为 Windows 操作系统生成代码，可以将变量 os 与字面量值“Windows”进行比较判断。类似地，如果想为 Linux 生成代码，可以将 os 与字面量值“Linux”判等。

```
@When[os == "Linux"]
func foo() {
    print("Linux")
}
@When[os == "Windows"]
func foo() {
    print("Windows")
}
main() {
    foo() // Compiling and running the code will print "Linux" or "Windows" on
     ↪  Linux or Windows
    return 0
}
```

backend 表示当前编译器所使用的后端。它用于实时获取编译器当前所使用的后端类型，进而与目标后端的字面量值构成一个编译条件。如果想为 cjnative 后端编译代码，可以将 backend 与字面量值“cjnative”进行比较判断。

支持的后端有：cjnative、cjnative-x86、cjnative-x86_64、cjnative-arm、cjnative-aarch64、cjvm、cjvm-x86、cjvm-x86_64、cjvm-arm、cjvm-aarch64。

```
@When[backend == "cjnative"]
func foo() {
    print("cjnative backend")
}
@When[backend == "cjvm"]
func foo() {
    print("cjvm backend")
}
main() {
    foo() // Compile and execute with the cjnative and cjvm backend version, and
     ↪  print "llvm backend" and "cjvm backend" respectively.
}
```

cjc_version 表示当前编译器的版本号。版本号是一个 String 类型的字面量值，采用 X.Y.Z 格式，其中 X、Y 和 Z 为非负的整数且不允许包含前导零，例如“0.18.8”。它用于实时获取编译器当前的版本号，进而与目标版本号比较，确保编译器能够基于特定的版本编译代码。共支持六种类型 ==、!=、>、<、>=、<= 的比较操作。条件的结果由从左到右依次比较这些字段时的第一个差异决定。例如，0.18.8 < 0.18.11，0.18.8 == 0.18.8。

debug 是一个条件编译标识符，表示当前编译的代码是否是一个调试版本。用于在编译代码时进行调试和发布版本之间的切换。使用 @When[debug] 修饰的代码只会在调试版本中编译；使用 @When[!debug] 修饰

的代码只会在发布版本中编译。

test 是一个条件编译标识符，用于标记测试代码。测试代码通常与普通源码位于相同的文件中，使用 @When[test] 修饰的代码，只会在使用 --test 选项时才会被编译，正常的构建时该部分代码将会被排除。

用户自定义条件变量　用户自定义条件变量是用户根据自己的需要定义条件变量，进而在代码中使用这些条件变量来控制代码的编译。用户自定义条件变量与内置变量本质上相似，唯一不同点是用户自定义条件变量的值由用户自身配置设定，而内置变量的值由编译器根据当前编译环境决定。用户自定义条件变量遵循如下规则：

- 自定义条件变量必须是一个合法的标识符
- 自定义条件变量仅支持等于和不等于两种关系运算符的比较运算。
- 用户自定义的编译条件变量必须是 String 类型。

一个用户自定义条件变量的示例如下：

```
//source.cj
@When[feature == "lion"]    // "feature" is user custom conditional variable.
func foo() {
    print("feature lion, ")
}
@When[platform == "dsp"]    // "platform" is user custom conditional variable.
func fee() {
    println("platform dsp")
}
main() {
    foo()
    fee()
}
```

配置自定义条件变量　配置自定义条件变量的方式有两种：编译选项和配置文件。编译选项 --cfg 允许接收字符串用于配置自定义条件变量的值，具体规则如下：

- 允许多次使用 --cfg。
- 允许使用多个条件赋值语句，用逗号 (,) 分隔。
- 不允许多次指定条件变量的值。
- = 操作符的左边必须是一个合法的标识符。
- = 操作符的右边是一个字面量。

使用 --cfg 对用户自定义条件变量 feature 和 platform 进行配置。

```
cjc --cfg "feature = lion, platform = dsp" source.cj // ok

cjc --cfg "feature = lion" --cfg "platform = dsp" source.cj // ok

cjc --cfg "feature = lion, feature = dsp" source.cj // error
```

采用 .toml 文件格式作为用户自定条件变量的配置文件，配置文件命名为 cfg.toml。

- 采用键值对的方式对自定义条件变量配置，键值对由 = 分隔，每个键值对单独占一行。
- 健名是一个合法的标识符。
- 键值是一个双引号括起来的字面量值。

默认以当前目录作为条件编译配置文件的搜索路径，支持使用 --cfg 设置配置文件搜索路径。

```
cjc --cfg "/home/cangjie/conditon/compile" source.cj // ok
```

14.2.6.5 多条件编译

允许使用逻辑与（&&）、逻辑或（||）组合多个编译条件，其优先级和结合性与逻辑表达式一致（请参见逻辑表达式）。允许使用圆括号将编译条件括起来，圆括号括起来的编译条件被视作一个单独的计算单元被优先计算。

```
@When[(backend == "cjnative" || os == "Linux") && cjc_version >= "0.40.1"]
func foo() {
    print("This is Multi-conditional compilation")
}

main() {
    foo() // Conditions for compiling source code: cjc_version greater than
     ↪  0.40.1; compiler backend is cjnative or os is Linux.
    return 0
}
```

第十五章　并发

15.1　仓颉线程

仓颉编程语言提供抢占式的并发模型，其中仓颉线程是基本的执行单元。仓颉线程由仓颉运行时自行管理并非是底层 native 线程（如操作系统线程）。在无歧义的情况下，我们直接使用“线程”一词作为“仓颉线程”的简称。每个线程都具有如下性质：- 每个线程在执行的任意时刻都可能被另一个线程抢占；- 多个线程之间并发执行；- 当线程发生阻塞时，该线程将被挂起；- 不同线程之间可以共享内存（需要显式同步）。

仓颉程序执行从初始化全局变量开始，然后调用程序入口 main。当 main 退出时，整个程序执行便退出而不会等待其余线程执行完毕。

15.1.1　创建线程

通过 spawn 关键字、一个可缺省的 ThreadContext 类型入参（关于 ThreadContext 的介绍见下方）和一个不包含形参的 lambda 可以创建并启动一个线程，同时该表达式返回一个 Future<T> 的实例（关于 Future<T> 的介绍见下方 15.1.2）。spawn 表达式的 BNF 如下：

```
spawnExpression
    : 'spawn' ( '(' expression ')' )? trailingLambdaExpression
    ;
```

其中 spawn 的可选参数的类型为 ThreadContext。以下是 spawn 表达式缺省 ThreadContext 类型入参的一个示例。

```
func add(a: Int32, b: Int32): Int32 {
    println("This is a new thread")
    return a + b
}

main(): Int64 {
    println("This is main")

    // Create a thread.
    let fut: Future<Int32> = spawn {
        add(1, 2)
    }
```

```
    println("main waiting...")
    // Waiting for the results of thread execution.
    let res: Int32 = fut.get()
    // Print the result.
    println("1 + 2 = ${res}")
}

// OUTPUT maybe:
// This is main
// main waiting...
// This is a new thread
// 1 + 2 = 3
```

执行 spawn 表达式的过程中，闭包会被封装成一个运行时任务并提交给调度器，然后由调度器的策略决定任务的执行时机。一旦 spawn 表达式执行完成，任务对应的线程便处于可执行状态。需要注意的是，spawn 表达式的闭包中不可捕获 var 声明的局部变量；关于闭包，详见闭包。

15.1.2 Future<T> 泛型类

spawn 表达式的返回值是一个 Future<T> 对象可用于获取线程的计算结果。其中，T 的类型取决于 spawn 表达式中的闭包的返回值类型。

```
func foo(): Int64 {...}
func bar(): String {...}

// The return type of foo() is Int64.
let f1: Future<Int64> = spawn {
    foo()
}

// The return type of bar() is String.
let f2: Future<String> = spawn {
    bar()
}

// Waiting for the threads' execution results.
let r1: Int64 = f1.get()
let r2: String = f2.get()
```

一个 Future<T> 对象代表一个未完成的计算或任务。Future<T> 是一个泛型类，其定义如下：

```
class Future<T> {
    ... ...
```

```
    /**
     * Blocking the current thread,
     * waiting for the result of the thread corresponding to this Future<T> object.
     * @throws Exception or Error if an exception occurs in the corresponding
      ↪ thread.
     */
    func get(): T

    /**
     * Blocking the current thread,
     * waiting for the result of the thread corresponding to this Future<T> object.
     * If the corresponding thread has not completed execution within `ns`
      ↪ nanoseconds,
     * the method will return `Option<T>.None`.
     * If `ns <= 0`, same as `get()`.
     * @throws Exception or Error if an exception occurs in the corresponding
      ↪ thread.
     */
    func get(ns: Int64): Option<T>

    /**
     * Non-blocking method that immediately returns None if thread has not
      ↪ finished execution.
     * Returns the computed result otherwise.
     * @throws Exception or Error if an exception occurs in the corresponding
      ↪ thread.
     */
    func tryGet(): Option<T>

    // Get the associated thread object.
    prop thread: Thread
}
```

如果线程由于发生异常而终止，那么它会将异常传递到 Future<T> 对象，并在终止前默认打印出异常信息。当线程对应的 Future<T> 对象调用 get() 时，异常将被再次抛出。例如：

```
main(args:Array<String>) {
    let fut: Future<Int64> = spawn {
        if (args.size != -1) {
            throw IllegalArgumentException()
        }
        return 100
```

```
        }
        try {
            let res: Int64 = fut.get()
            print("val = ${res}\n")
        } catch (e: IllegalArgumentException) {
            print("Exception occured\n")
        }
    }
    // OUTPUT:
    // An exception has occurred:
    // IllegalArgumentException
    //     ...
    // Exception occurred
```

15.1.3 Thread 类

每个 Future<T> 对象都有一个关联的线程对象，其类型为 Thread，可通过 Future<T> 的 thread 属性获取。线程类 Thread 主要用于获取线程的相关信息，例如线程标识等，其部分定义如下。

```
class Thread {
    ... ...
    // Get the currently running thread
    static prop currentThread: Thread

    // Get the unique identifier (represented as an integer) of the thread object
    prop id: Int64

    // Get or set the name of the thread
    mut prop name: String
}
```

静态属性 currentThread 可获取当前正在执行的线程对象。下面的示例代码能够打印出当前执行线程的标识。注意：线程在每次执行时可能被赋予不同的线程标识。

```
main() {
    let fut: Future<Unit> = spawn {
        let tid = Thread.currentThread.id
        println("New thread id: ${tid}")
    }

    fut.get()
}
// OUTPUT:
// New thread id: 2
```

15.1.4 线程睡眠

在仓颉标准库的 sync 包中提供了 sleep 函数能够让线程睡眠指定时长。如果睡眠时长为零，即参数时长为 duration.Zero, 那么当前线程只会让出执行资源并不会进入睡眠。

```
func sleep(duration: Duration)
```

15.1.5 线程终止

在正常执行过程中，若线程对应的闭包执行完成，则该线程执行结束。此外，还可以通过 Future<T> 对象的 cancel() 方法向对应的线程发送终止请求，这一方法仅发送请求而不会影响线程的执行。线程类的 hasPendingCancellation 属性用于检查当前执行的线程是否存在终止请求，程序员可以通过该检查来自行决定是否提前终止线程以及如何终止线程。

```
class Future<T> {
    ... ...
    // Send a termination request to its executing thread.
    public func cancel(): Unit
}

class Thread {
    ... ...
    // Check whether the thread has any cancellation request
    prop hasPendingCancellation: Bool
}
```

以下示例展示如何终止线程。

```
main(): Unit {
    let future = spawn { // Create a new thread
        while (true) {
            //...
            if (Thread.currentThread.hasPendingCancellation) {
                return   // Terminate when having a request
            }
            //...
        }
    }
    //...
    future.cancel()    // Send a termination request
    future.get() // Wait for thread termination
}
```

15.2 线程上下文

线程总是被运行在特定的上下文里，线程上下文会影响线程运行时的行为。用户创建线程时，除了缺省 spawn 表达式入参，也可以通过传入不同 ThreadContext 类型的实例，选择不同的线程上下文，然后一定程度上控制并发行为。

15.2.1 接口 ThreadContext

ThreadContext 接口类型定义如下：

- 结束方法 end 用于向当前 context 发送结束请求（关于线程上下文结束的介绍见下方 thread context 结束）。
- 检查方法 hasEnded 用于检查当前 context 是否已结束。

```
interface ThreadContext {
    func end(): Unit
    func hasEnded(): Bool
}
```

目前不允许用户自行实现 ThreadContext 接口，仓颉语言根据使用场景，提供了如下线程上下文类型：

- MainThreadContext: 用户可以在终端应用开发中使用此线程上下文；

具体定义可在终端框架库中查阅。

15.2.2 线程上下文关闭

当关闭 ThreadContext 对象时，可以通过 ThreadContext 对象提供的 end 方法向对应的 context 发送关闭请求。此方法向运行在此 context 上的所有线程发送终止请求（参见线程终止），并马上返回，运行时待所有绑定至此 context 上的仓颉 thread 运行结束后，关闭 context。

同样的，关闭请求不会影响在此 context 上所有线程的执行，用户可以通过 hasPendingCancellation 检查来自行决定是否提前终止线程以及如何终止线程。

15.2.3 线程局部变量

类型 ThreadLocal<T> 可用于定义*线程局部变量*。和普通变量相比，线程局部变量有不同的访问语义。当多个线程共享使用同一线程局部变量时，每个线程都有各自的一份值拷贝。线程对变量的访问会读写线程本地的值，而不会影响其他线程中变量的值。因而，ThreadLocal<T> 类是线程安全的。类 ThreadLocal<T> 如下定义所示，其中：

- 构造方法 init 用于构造线程局部变量。在构造完但未设置变量值时，线程对变量的读取将得到空值 None；
- 读写方法 get/set 用于访问线程局部变量的值。当线程将变量的值设为 None 后，当前线程中的变量值将被清除。

```
class ThreadLocal<T> {
    // Construct a thread-local variable contains None.
    public init()

    // Get the value of the thread-local variable of the current executing thread.
    public func get(): ?T

    // Set a value to the thread-local variable.
    public func set(value: ?T): Unit
}
```

以下示例展示如何使用线程局部变量。

```
let tlv = ThreadLocal<Int64>() // Define a thread-local variable

main(): Unit {
    for (i in 0..3) { // Spawn three threads
        spawn {
            tlv.set(i) // Each thread sets a different value
            // ...
            println("${tlv.get()}") // Each thread prints its own value
        }
    }
    // ...
    println("${tlv.get()}") // Print `None`
                            // since the current thread does not set any value.
}
```

15.3 同步机制

15.3.1 原子操作

原子操作确保指令被原子地执行，即执行过程中不会被中断。对原子变量的一个写操作，对于后续对同一个原子变量的读操作来说，总是可见的。此外，原子操作是非阻塞的动作，不会导致线程阻塞。仓颉语言提供了针对整数类型（包括 Int8、Int16、Int32、Int64、UInt8、UInt16、UInt32、UInt64），布尔类型（Bool）和引用类型的原子操作。

- 对于整数类型，我们提供基本的读写、交换以及算术运算的操作：
  - load：读取
  - store：写入
  - swap：交换
  - compareAndSwap：比较再交换
  - fetchAdd：加法

  - fetchSub：减法
  - fetchAnd：与
  - fetchOr：或
  - fetchXor：异或

```
// Signed Integers.
class AtomicInt8 {
    ... ...
    init(val: Int8)

    public func load(): Int8
    public func store(val: Int8): Unit
    public func swap(val: Int8): Int8
    public func compareAndSwap(old: Int8, new: Int8): Bool

    public func fetchAdd(val: Int8): Int8
    public func fetchSub(val: Int8): Int8
    public func fetchAnd(val: Int8): Int8
    public func fetchOr(val: Int8): Int8
    public func fetchXor(val: Int8): Int8

    ... ... // Operator overloading, etc.
}

class AtomicInt16 {...}
class AtomicInt32 {...}
class AtomicInt64 {...}

// Unsigned Integers.
class AtomicUInt8  {...}
class AtomicUInt16 {...}
class AtomicUInt32 {...}
class AtomicUInt64 {...}
```

- 对于布尔类型，仅提供基本的读写、交换操作，不提供算术运算的操作：

  - load：读取
  - store：写入
  - swap：交换
  - compareAndSwap：比较再交换

```
// Boolean.
class AtomicBool {
    ... ...
```

```
    init(val: Bool)

    public func load(): Bool
    public func store(val: Bool): Unit
    public func swap(val: Bool): Bool
    public func compareAndSwap(old: Bool, new: Bool): Bool

    ... ... // Operator overloading, etc.
}
```

- 对于引用类型，仅提供基本的读写、交换操作，不提供算术运算的操作：
  - load：读取
  - store：写入
  - swap：交换
  - compareAndSwap：比较再交换

```
class AtomicReference<T> where T <: Object {
    ... ...
    init(val: T)

    public func load(): T
    public func store(val: T): Unit
    public func swap(val: T): T
    public func compareAndSwap(old: T, new: T): Bool
}
```

- 此外，对于可空引用类型，可以通过 AtomicOptionReference 保存“空引用”（以 None 表示）。

```
class AtomicOptionReference<T> where T <: Object {
    public init(val: Option(T))
    public func load(): Option<T>
    public func store(val: Option<T>): Unit
    public func swap(val: Option<T>): Option<T>
    public func compareAndSwap(old: Option<T>, new: Option<T>): Bool
}
```

以上方法均包含一个隐藏参数 memory order。当前只支持 Sequential Consistency 内存模型，后续可以考虑增加更宽松的内存模型，例如 acquire/release 内存顺序等。

15.3.2 IllegalSynchronizationStateException

IllegalSynchronizationStateException 是一个运行时异常，当出现违规行为，例如当线程尝试去释放当前线程未获取的互斥量的时候，内置的并发原语会抛出此异常。

15.3.3 IReentrantMutex

IReentrantMutex 是可重入式互斥并发原语的公共接口，提供如下三个方法，需要开发者提供具体实现：

- lock(): Unit：获取锁，如果获取不到，阻塞当前线程
- tryLock(): Bool：尝试获取锁
- unlock(): Unit：释放锁

```
interface IReentrantMutex {
    // Locks the mutex, blocks the current thread if the mutex is not available.
    public func lock(): Unit

    // Tries to lock the mutex, returns false if the mutex is not available,
    ↪ otherwise locks the mutex and returns true.
    public func tryLock(): Bool

    // Unlocks the mutex. If the mutex was locked repeatedly N times, this method
    ↪ should be invoked N times to
    // fully unlock the mutex. When the mutex is fully unlocked, unblocks one of
    ↪ the threads waiting on its `lock`
    // (no particular admission policy implied).
    // Throws ISSE("Mutex is not locked by the current thread") if the current
    ↪ thread does not hold this mutex.
    public func unlock(): Unit
}
```

注意：

1. 开发者在实现该接口时需要保证底层互斥锁确实支持嵌套锁。
2. 开发者在实现该接口时需要保证在出现锁状态异常时抛出 IllegalSynchronizationStateException。

15.3.4 ReentrantMutex

ReentrantMutex 提供如下方法：

- lock(): Unit：获取锁，如果获取不到，阻塞当前线程
- tryLock(): Bool：尝试获取锁
- unlock(): Unit：释放锁

ReentrantMutex 是内置的锁，在同一时刻最多只能被一个线程持有。如果给定的 ReentrantMutex 已经被另外一个线程持有，lock 方法会阻塞当前线程直到该互斥锁被释放，而 tryLock 方法会立即返回 false。ReentrantMutex 是可重入锁，即如果一个线程尝试去获取一个它已经持有的 ReentrantMutex 锁，他会立即获取该 ReentrantMutex。为了成功释放该锁，unlock 的调用次数必须与 lock 的调用次数相匹配。

注意：**ReentrantMutex** 是内置的互斥锁，不允许开发者继承。

```
// Base class for built-in reentrant mutual exclusion concurrency primitives.
sealed class ReentrantMutex <: IReentrantMutex {
    // Constructor.
    init()

    // Locks the mutex, blocks current thread if the mutex is not available.
    public func lock(): Unit

    // Tries to lock the mutex, returns false if the mutex is not available,
     ↪  otherwise locks the mutex and returns true.
    public func tryLock(): Bool

    // Unlocks the mutex. If the mutex was locked repeatedly N times, this method
     ↪  should be invoked N times to
    // fully unlock the mutex. Once the mutex is fully unlocked, unblocks one of
     ↪  the threads waiting in its `lock` method, if any
    // (no particular admission policy implied).
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪  thread does not hold this mutex.
    public func unlock(): Unit
}
```

15.3.5 synchronized

通过 synchronized 关键字和一个 ReentrantMutex 对象，对其后面修饰的代码块进行保护，使得同一时间只允许一个线程执行里面的代码。ReentrantMutex 或其派生类的实例可以作为参数传递给 synchronized 关键字，这将导致如下转换：

```
//=====================================
// `synchronized` expression
//=====================================
let m: ReentrantMutex = ...
synchronized(m) {
    foo()
}
//=====================================
// is equivalent to the following program.
//=====================================
let m: ReentrantMutex = ...
m.lock()
try {
    foo()
```

```
} finally {
    m.unlock()
}
```

注意：synchronized 关键字与 IReentrantMutex 接口不兼容。

- 一个线程在进入 synchronized 修饰的代码块之前，必须先获取该 ReentrantMutex 对象的锁，如果无法获取该锁，则当前线程被阻塞；
- 一个线程在退出 synchronized 修饰的代码块之后，会自动释放该 ReentrantMutex 对象的锁；只会执行一个 unlock 操作，因此允许嵌套同一个互斥锁的 synchronized 代码块。
- 在到达 synchronized 修饰的代码块中的 return e 表达式后，会先将 e 求值到 v，再调用 m.unlock()，最后返回 v。
- 若 synchronized 修饰的代码块内有 break 或 continue 表达式，并且该表达式的执行会导致程序跳出该代码块，那么会自动调用 m.unlock() 方法。
- 当在 synchronized 修饰的代码块中发生异常并跳出该代码块时，会自动调用 m.unlock() 方法。

synchronized 表达式的 BNF 如下：

```
synchronizedExpression
    : 'synchronized' '(' expression ')' block
    ;
```

其中 expression 是一个 ReentrantMutex 的对象。

15.3.6 Monitor

Monitor 是一个内置的数据结构，它绑定了互斥锁和单个与之相关的条件变量（也就是等待队列）。Monitor 可以使线程阻塞并等待来自另一个线程的信号以恢复执行。这是一种利用共享变量进行线程同步的机制，主要提供如下方法：

- wait(timeout!: Duration = Duration.Max): Bool：等待信号，阻塞当前线程
- notify(): Unit：唤醒一个在 Monitor 上等待的线程（如果有）
- notifyAll(): Unit：唤醒所有在 Monitor 上等待的线程（如果有）

调用 Monitor 对象的 wait、notify 或 notifyAll 方法前，需要确保当前线程已经持有对应的 Monitor 锁。wait 方法包含如下动作：

1. 添加当前线程到该 Monitor 对应的等待队列中
2. 阻塞当前线程，同时完全释放该 Monitor 锁，并记录获取次数
3. 等待某个其它线程使用同一个 Monitor 实例的 notify 或 notifyAll 方法向该线程发出信号
4. 唤醒当前线程并同时获取 Monitor 锁，恢复第二步记录的获取次数

注意：wait 方法接受一个可选参数 timeout。需要注意的是，业界很多常用的常规操作系统不保证调度的实时性，因此无法保证一个线程会被阻塞“精确时长”——即可能会观察到由系统导致的不精确情况。此外，当前语言规范明确允许实现中发生虚假唤醒——在这种情况下，wait 返回值是由实现决定的——可能为 true 或 false。因此鼓励开发者始终将 wait 包在一个循环中：

```
synchronized (obj) {
    while (<condition is not true>) {
        obj.wait();
    }
}
```

Monitor 定义如下:

```
class Monitor <: ReentrantMutex {
    // Constructor.
    init()

    // Blocks until either a paired `notify` is invoked or `timeout` pass.
    // Returns `true` if the monitor was signalled by another thread or `false` on
     ↪  timeout. Spurious wakeups are allowed.
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪  thread does not hold this mutex.
    func wait(timeout!: Duration = Duration.Max): Bool

    // Wakes up a single thread waiting on this monitor, if any (no particular
     ↪  admission policy implied).
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪  thread does not hold this mutex.
    func notify(): Unit

    // Wakes up all threads waiting on this monitor, if any (no particular
     ↪  admission policy implied).
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪  thread does not hold this mutex.
    func notifyAll(): Unit
}
```

15.3.7 MultiConditionMonitor

MultiConditionMonitor 是一个内置的数据结构，它绑定了互斥锁和一组与之相关的动态创建的条件变量。该类应仅当在 Monitor 类不足以实现高级并发算法时被使用。

提供如下方法:

- newCondition(): ConditionID：创建一个新的等待队列并与当前对象关联，返回一个特定的 ConditionID 标识符
- wait(id: ConditionID, timeout!: Duration = Duration.Max): Bool：等待信号，阻塞当前线程
- notify(id: ConditionID): Unit：唤醒一个在 Monitor 上等待的线程（如果有）

- `notifyAll(id: ConditionID): Unit`：唤醒所有在 `Monitor` 上等待的线程（如果有）

初始化时，`MultiConditionMonitor` 没有与之相关的 `ConditionID` 实例。每次调用 `newCondition` 都会将创建一个新的等待队列并与当前对象关联，并返回如下类型作为唯一标识符：

```
external struct ConditionID {
   private init() { ... } // constructor is intentionally private to prevent
                          // creation of such structs outside of
                           ↪ MultiConditionMonitor
}
```

请注意使用者不可以将一个 `MultiConditionMonitor` 实例返回的 `ConditionID` 传给其它实例，或者手动创建 `ConditionID`（例如使用 `unsafe`）。由于 `ConditionID` 所包含的数据（例如内部数组的索引，内部队列的直接地址，或任何其他类型数据等）和创建它的 `MultiConditionMonitor` 相关，所以将“外部”`conditonID` 传入 `MultiConditionMonitor` 中会导致 `IllegalSynchronizationStateException`。

```
class MultiConditionMonitor <: ReentrantMutex {
    // Constructor.
    init()

    // Returns a new ConditionID associated with this monitor. May be used to
     ↪ implement
    // "single mutex -- multiple wait queues" concurrent primitives.
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪ thread does not hold this mutex.
    func newCondition(): ConditionID

    // Blocks until either a paired `notify` is invoked or `timeout` pass.
    // Returns `true` if the specified condition was signalled by another thread
     ↪ or `false` on timeout.
    // Spurious wakeups are allowed.
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪ thread does not hold this mutex.
    // Throws ISSE("Invalid condition") if `id` was not returned by `newCondition`
     ↪ of this MultiConditionMonitor instance.
    func wait(id: ConditionID, timeout!: Duration = Duration.Max): Bool

    // Wakes up a single thread waiting on the specified condition, if any (no
     ↪ particular admission policy implied).
    // Throws ISSE("Mutex is not locked by the current thread") if the current
     ↪ thread does not hold this mutex.
    // Throws ISSE("Invalid condition") if `id` was not returned by `newCondition`
     ↪ of this MultiConditionMonitor instance.
    func notify(id: ConditionID): Unit
```

```
        // Wakes up all threads waiting on the specified condition, if any (no
        ↪ particular admission policy implied).
        // Throws ISSE("Mutex is not locked by the current thread") if the current
        ↪ thread does not hold this mutex.
        // Throws ISSE("Invalid condition") if `id` was not returned by `newCondition`
        ↪ of this MultiConditionMonitor instance.
        func notifyAll(id: ConditionID): Unit
    }
```

示例：使用 MultiConditionMonitor 去实现一个“长度固定的有界 FIFO 队列”，当队列为空，get() 会被阻塞；当队列满了时，put() 会被阻塞。

```
    class BoundedQueue {
        // Create a MultiConditionMonitor, two Conditions.
        let m: MultiConditionMonitor = MultiConditionMonitor()
        var notFull: ConditionID
        var notEmpty: ConditionID

        var count: Int64 // Object count in buffer.
        var head: Int64  // Write index.
        var tail: Int64  // Read index.

        // Queue's length is 100.
        let items: Array<Object> = Array<Object>(100, {i => Object()})

        init() {
            count = 0
            head = 0
            tail = 0

            synchronized(m) {
              notFull  = m.newCondition()
              notEmpty = m.newCondition()
            }
        }

        // Insert an object, if the queue is full, block the current thread.
        public func put(x: Object) {
            // Acquire the mutex.
            synchronized(m) {
              while (count == 100) {
                // If the queue is full, wait for the "queue notFull" event.
```

```
            m.wait(notFull)
          }
          items[head] = x
          head++
          if (head == 100) {
            head = 0
          }
          count++

          // An object has been inserted and the current queue is no longer
          // empty, so wake up the thread previously blocked on get()
          // because the queue was empty.
          m.notify(notEmpty)
        } // Release the mutex.
      }

      // Pop an object, if the queue is empty, block the current thread.
      public func get(): Object {
        // Acquire the mutex.
        synchronized(m) {
          while (count == 0) {
            // If the queue is empty, wait for the "queue notEmpty" event.
            m.wait(notEmpty)
          }
          let x: Object = items[tail]
          tail++
          if (tail == 100) {
            tail = 0
          }
          count--

          // An object has been popped and the current queue is no longer
          // full, so wake up the thread previously blocked on put()
          // because the queue was full.
          m.notify(notFull)

          return x
        } // Release the mutex.
      }
    }
```

15.4 内存模型

内存模型主要解决并发编程中内存可见性的问题，它规定了一个线程对一个变量的写操作，何时一定可以被其他线程对同一变量的读操作观测到。

- 如果存在数据竞争，那么行为是未定义的。
- 如果没有数据竞争，一个读操作所读到的值是：在 happens-before 顺序上离它最近的一个写操作所写入的值。

主要解决并发编程中内存可见性的问题，即一个线程的写操作何时会对另一个线程可见。

15.4.1 数据竞争 Data Race

如果两个线程对 同一个数据访问, 其中至少有一个是写操作, 而且这两个操作之间没有 happens-before 关系（在 15.4.2 节定义），那么形成一个数据竞争。

对“同一个数据访问”的定义：

1. 对同一个 primitive type、enum、array 类型的变量或者 struct/class 类型的同一个 field 的访问，都算作对同一个数据的访问。
2. 对 struct/class 类型的不同 field 的访问，算作对不同数据的访问。

15.4.2 Happens-Before

- 程序顺序规则：同一个线程中的每个操作 happens-before 于该线程中的任意后续操作。

```
var a: String

main(): Int64 {
    a = "hello, world"
    println(a)
    return 0
}

// OUTPUT:
// hello, world
```

- 线程启动规则：如果线程 A 通过 spawn 创建线程 B，那么线程 A 的 spawn 操作 happens-before 于线程 B 中的任意操作。

```
var a: String = "123"

func foo(): Unit {
    println(a)
}
```

```
main(): Int64 {
    a = "hello, world"
    let fut: Future<Unit>= spawn {
        foo()
    }
    fut.get()
    return 0
}

// OUTPUT:
// hello, world
```

- 线程终止规则：如果线程 A 调用 `futureB.get()` 并成功返回，那么线程 B 中的任意操作 happens-before 于线程 A 中的 `futureB.get()` 调用。如果线程 A 调用 `futureB.cancel()` 并且线程 B 在此之后访问 `hasPendingCancellation`，那么这两个调用构成 happens-before 关系。

```
var a: String = "123"

func foo(): Unit {
    a = "hello, world"
}

main(): Int64 {
    let fut: Future<Unit> = spawn {
        foo()
    }

    fut.get()
    println(a)
    return 0
}

// OUTPUT:
// hello, world
```

- 线程同步规则：在同一个线程同步对象（例如互斥锁、信号量等）上的操作存在一个全序。一个线程对一个同步对象的操作（例如对互斥锁的解锁操作）happens-before 于这个全序上后续对这个同步对象的操作（例如对互斥锁的上锁操作）。
- 原子变量规则：对于所有原子变量的操作存在一个 全序。一个线程对一个原子变量的操作 happens-before 于这个全序上后续所有的原子变量的操作。
- 传递性规则：如果 A happens-before B 且 B happens-before C，那么 A happens-before C。

第十六章　常量求值

16.1　const 变量

const 变量是一种特殊的变量，它可以定义在编译时完成求值，并且在运行时不可改变的变量。

const 变量与 let/var 声明的变量的区别是必须在定义时就初始化，且必须用 const 表达式初始化。因此 const 变量的类型只能是 const 表达式支持的类型。

与 let/var 一样，const 变量也支持省略类型。

const 表达式见下文定义。

```
const a: Int64 = 0 // ok
const b: Int64 // error, b is uninitialized
const c = f() // error, f() is not const expression
const d = 0 // ok, the type of d is Int64

func f(): Unit {}
```

const 变量定义后可以像 let/var 声明的变量一样使用。与 let/var 声明的变量不同，const 由于在编译时就可以得到结果，可以大幅减少程序运行时需要的计算。

```
const a = 0

main(): Int64 {
    let b: Int64 = a // ok
    print(a) // ok
    let c: VArray<Int64, $0> = [] // ok
    return 0
}
```

const 变量可以是全局变量，局部变量，静态成员变量。const 变量不能在扩展中定义。

```
const a = 0 // ok

class C {
    const b = 0 // error, const member field must be modified by static
    static const c = 0 //ok
```

```
        var v: Int64

        init() {
            const d = 0 // ok
            v = b + c + d
        }
    }

    extend C {
        const e = 0 // error, const cannot be defined in extend
    }
```

const 变量可以访问对应类型的所有实例成员，也可以调用对应类型的所有非 mut 实例成员函数。

```
    struct Foo {
        let a = 0
        var b = 0

        const init() {}

        func f1() {}
        const func f2() {}
        mut func f3() {
            b = 123
        }
    }

    main(): Int64 {
        const v = Foo()
        print(v.a) // ok
        print(v.b) // ok
        v.f1() // ok
        v.f2() // ok
        v.f3() // error, f3 is mut function
        return 0
    }
```

const 变量初始化后该类型实例的所有成员都是 const 的（深度 const，包含成员的成员），因此不能被用于左值。

```
    struct Foo {
        let a = 0
        var b = 0
```

```
        const init() {}
    }

    func f() {
        const v = Foo()
        v.a = 1 // error
        v.b = 1 // error
    }
```

16.2 const 表达式

某些特定形式的表达式，被称为 const 表达式，这些表达式具备了可以在编译时求值的能力。在 const 上下文中，这些是唯一允许的表达式，并且始终会在编译时进行求值。而在其它非 const 上下文，const 表达式不保证在编译时求值。

以下表达式都是 const 表达式。

1. 数值类型、Bool、Unit、Rune、String 类型的字面量（不包含插值字符串）。
2. 所有元素都是 const 表达式的 array 字面量（不能是 `Array` 类型），tuple 字面量。
3. const 变量，const 函数形参，const 函数中的局部变量。
4. const 函数，包含使用 const 声明的函数名、符合 const 函数要求的 lambda、以及这些函数返回的函数表达式。
5. const 函数调用（包含 const 构造函数），该函数的表达式必须是 const 表达式，所有实参必须都是 const 表达式。
6. 所有参数都是 const 表达式的 enum 构造器调用，和无参数的 enum 构造器。
7. 数值类型、Bool、Unit、Rune、String 类型的算数表达式、关系表达式、位运算表达式，所有操作数都必须是 const 表达式。
8. if、match、try、控制转移表达式（包含 return、break、continue、throw）、is、as。这些表达式内的表达式必须都是 const 表达式。
9. const 表达式的成员访问（不包含属性的访问），tuple 的索引访问。
10. const init 和 const 函数中的 this 和 super 表达式。
11. const 表达式的 const 实例成员函数调用，且所有实参必须都是 const 表达式。

16.3 const 上下文

const 上下文是一类特定上下文，在这些上下文内的表达式都必须是 const 表达式，并且这些表达式始终在编译时求值。

const 上下文是指 const 变量初始化表达式。

16.4 const 函数

const 函数是一类特殊的函数，这些函数具备了可以在编译时求值的能力。在 const 上下文中调用这种函数时，这些函数会在编译时执行计算。而在其它非 const 上下文，const 函数会和普通函数一样在运行时执行。

const 函数与普通函数的区别是限制了部分影响编译时求值的功能，const 函数中只能出现声明、const 表达式、受限的部分赋值表达式。

1. const 函数声明必须使用 const 修饰。
2. 全局 const 函数和 static const 函数中只能访问 const 声明的外部变量，包含 const 全局变量、const 静态成员变量，其它外部变量都不可访问。const init 函数和 const 实例成员函数除了能访问 const 声明的外部变量，还可以访问当前类型的实例成员变量。
3. const 函数中的表达式都必须是 const 表达式，const init 函数除外。
4. const 函数中可以使用 let、const 声明新的局部变量。但不支持 var。
5. const 函数中的参数类型和返回类型没有特殊规定。如果该函数调用的实参不符合 const 表达式要求，那这个函数调用不能作为 const 表达式使用，但仍然可以作为普通表达式使用。
6. const 函数不一定都会在编译时执行，例如可以在非 const 函数中运行时调用。
7. const 函数与非 const 函数重载规则一致。
8. 数值类型、Bool、Unit、Rune、String 类型和 enum 支持定义 const 实例成员函数。
9. 对于 struct 和 class，只有定义了 const init 才能定义 const 实例成员函数。class 中的 const 实例成员函数不能是 open 的。struct 中的 const 实例成员函数不能是 mut 的。

```
const func f(a: Int64): Int64 { // ok
    let b = 6
    if (a > 5 && (b + a) < 15 ) {
        return a * a
    }
    return a
}

class A {
    var a = 0
}

const func f1(a: A): Unit { // ok
    return
}

const func f2(): A {
    return A() // error, A did not contains const init
}
```

16.4.1 接口中的 const 函数

接口中也可以定义 const 函数，但会受到以下规则限制。

1. 接口中的 const 函数，实现类型必须也用 const 函数才算实现接口。
2. 接口中的非 const 函数，实现类型使用 const 或非 const 函数都算实现接口。
3. 接口中的 const 函数与接口的 static 函数一样，只有在该接口作为泛型约束的时候，受约束的泛型变元或变量才能使用这些 const 函数。

```
interface I {
    const func f(): Int64
    const static func f2(): Int64
}

const func g<T>(i: T) where T <: I {
    return i.f() + T.f2()
}
```

16.5 const init

有无 const init 决定了哪些自定义 struct/class 可以用在 const 表达式上。一个类型能否定义 const init 取决于以下条件。

1. 如果当前类型是 class，则不能具有 var 声明的实例成员变量，否则不允许定义 const init。如果当前类型具有父类，当前的 const init 必须调用父类的 const init（可以显式调用或者隐式调用无参 const init），如果父类没有 const init 则报错。
2. Object 类型的无参 init 也是 const init，因此 Object 的子类可以使用该 const init。
3. 当前类型的实例成员变量如果有初始值，初始值必须要是 const 表达式，否则不允许定义 const init。
4. const init 内可以使用赋值表达式对实例成员变量赋值，除此以外不能有其它赋值表达式。

const init 与 const 函数的区别是 const init 内允许对实例成员变量进行赋值（需要使用赋值表达式）

```
struct R1 {
    var a: Int64
    let b: Int64

    const init() { // ok
        a = 0
        b = 0
    }
}

struct R2 {
    var a = 0
```

```
    let b = 0

    const init() {} // ok
}

func zero(): Int64 {
    return 0
}

struct R3 {
    let a = zero()

    const init() {} // error, Initialization of a is not const expression
}

class C1 {
    var a = 0

    const init() {} // error, a can not be var binding
}

struct R4 {
    var a = C1()

    const init() {} // error, Initialization of a is not const expression
}

open class C2 {
    let a = 0
    let b = R2()

    const init() {} // ok
}

class C3 <: C2 {
    let c = 0

    const init() {} // ok
}
```

第十七章　注解

注解（Annotation）是一种只用在声明上的特殊语法，它用来给编译器或者运行时提供额外信息来实现特定的功能。

注解最基本的语法示例如下:

```
@Annotation1[arg1, arg2]
@Annotation2
class Foo {}
```

注解可以用在不同的声明上。

其语法规格定义为:

```
annotationList: annotation+;

annotation
    : '@' (identifier '.')* identifier ('[' annotationArgumentList ']')?
    ;

annotationArgumentList
    : annotationArgument (',' annotationArgument)*  ','?
    ;

annotationArgument
    : identifier ':' expression
    | expression
    ;
```

注解和宏调用使用了一样的语法。由于宏处理的阶段比注解更早，编译器在解析时会先尝试将该语法当作宏处理。如果作为宏不成功则继续尝试将该语法作为注解处理。如果两种处理方式都不成功则应该编译报错。

17.1　自定义注解

自定义注解机制用来让反射获取标注内容，目的是在类型元数据之外提供更多的有用信息，以支持更复杂的逻辑。

自定义注解可以用在类型声明、成员变量声明、成员属性声明、成员函数声明、构造函数声明、构造器声明、（前面提到的函数里的）函数参数声明上。

开发者可以通过自定义类型标注 @Annotation 方式创建自己的自定义注解。

@Annotation 只能修饰 class，并且不能是 abstract 或 open 或 sealed 修饰的 class。

当一个 class 声明它标注了 @Annotation，那么它必须要提供至少一个 const init 函数，否则编译器会报错。

下面是一个自定义注解的例子：

```
@Annotation
public class CustomAnnotation {
    let name: String
    let version: Int64

    public const init(name: String, version!: Int64 = 0) {
        this.name = name
        this.version = version
    }
}

// use annotation
@CustomAnnotation["Sample", version: 1]
class Foo {}
```

注解信息需要在编译时生成信息绑定到类型上，自定义注解在使用时必须使用 const init 构建出合法的实例。

我们规定：

1. 注解声明语法与声明宏语法一致，后面的 [] 括号中需要按顺序或命名参数规则传入参数，且参数必须是 const 表达式。
2. 对于拥有无参构造函数的注解类型，声明时允许省略括号。

```
@Annotation
public class MyAnnotation { ... }

@MyAnnotation // ok
class Foo {}
```

对于同一个注解目标，同一个注解类不允许声明多次，即不可重复。

编译器不承诺生成的多个 Annotation 元数据的顺序跟代码中 Annotation 的标注顺序保持一致。

```
@MyAnnotation
@MyAnnotation // error
class Foo {}
```

```
@MyAnnotation
@MyAnnotation2
class Bar {}
```

Annotation 不会被继承，因此一个类型的注解元数据只会来自它定义时声明的注解。如果需要父类型的注解元数据信息，需要开发者自己用反射接口查询。

```
@BaseAnno
open class Base {}

@InterfaceAnno
interface I {}

@SubAnno
class Sub <: Base & I {} // Sub has only SubAnno annotation information.
```

自定义注解可以限制自己可以使用的位置，这样可以减少开发者的误用，这类注解需要在声明 @Annotation 时标注 target 参数。

target 参数需要以变长参数的形式传入该自定义注解希望支持的位置，target 接收的参数是一个 Array<AnnotationKind>。

可以限制的位置包括:

- 类型声明（class、struct、enum、interface）
- 成员函数/构造函数中的参数
- 构造函数声明
- 成员函数声明
- 成员变量声明
- 成员属性声明

```
public enum AnnotaitionKind {
    | Type
    | Parameter
    | Init
    | MemberProperty
    | MemberFunction
    | MemberVariable
}

@Annotation[target: [Type, MemberFunction]]
class CustomAnnotation{}
@Annotation
class SubCustomAnnotation{}

@Annotation[target: []]
class MyAnno{}
```

当没有限定 target 的时候，该自定义注解可以用在以上全部位置。当限定 target 时，只能用在声明的列表中。

附录 A　仓颉语法

A.1　词法

A.1.1　注释

```
DelimitedComment
    : '/*' ( DelimitedComment | . )*? '*/'
    ;

LineComment
    : '//' ~[\u000A\u000D]*
    ;
```

A.1.2　空白和换行

```
WS
    : [\u0020\u0009\u000C]
    ;

NL: '\u000A' | '\u000D' '\u000A' ;
```

A.1.3　符号

```
DOT: '.' ;
COMMA: ',' ;
LPAREN: '(' ;
RPAREN: ')' ;
LSQUARE: '[' ;
RSQUARE: ']' ;
LCURL: '{' ;
RCURL: '}' ;
EXP: '**' ;
MUL: '*' ;
MOD: '%' ;
DIV: '/' ;
```

```
ADD: '+' ;
SUB: '-' ;
PIPELINE: '|>' ;
COMPOSITION: '~>' ;
INC: '++' ;
DEC: '--' ;
AND: '&&' ;
OR: '||' ;
NOT: '!' ;
BITAND: '&' ;
BITOR: '|' ;
BITXOR: '^' ;
LSHIFT: '<<' ;
RSHIFT: '>>' ;
COLON: ':' ;
SEMI: ';' ;
ASSIGN: '=' ;
ADD_ASSIGN: '+=' ;
SUB_ASSIGN: '-=' ;
MUL_ASSIGN: '*=' ;
EXP_ASSIGN: '**=' ;
DIV_ASSIGN: '/=' ;
MOD_ASSIGN: '%=' ;
AND_ASSIGN: '&&=' ;
OR_ASSIGN: '||=' ;
BITAND_ASSIGN: '&=' ;
BITOR_ASSIGN: '|=' ;
BITXOR_ASSIGN: '^=' ;
LSHIFT_ASSIGN: '<<=' ;
RSHIFT_ASSIGN: '>>=' ;
ARROW: '->' ;
BACKARROW: '<-' ;
DOUBLE_ARROW: '=>' ;
ELLIPSIS: '...' ;
CLOSEDRANGEOP: '..=' ;
RANGEOP: '..' ;
HASH: '#' ;
AT: '@' ;
QUEST: '?' ;
UPPERBOUND: '<:';
LT: '<' ;
```

```
GT: '>' ;
LE: '<=' ;
GE: '>=' ;
NOTEQUAL: '!=' ;
EQUAL: '==' ;
WILDCARD: '_' ;
BACKSLASH: '\\' ;
QUOTESYMBOL: '`';
DOLLAR: '$';
QUOTE_OPEN: '"' ;
TRIPLE_QUOTE_OPEN: '"""' NL;
QUOTE_CLOSE: '"' ;
TRIPLE_QUOTE_CLOSE: '"""' ;
LineStrExprStart: '${' ;
MultiLineStrExprStart: '${' ;
```

A.1.4 关键字

```
INT8: 'Int8' ;
INT16: 'Int16' ;
INT32: 'Int32' ;
INT64: 'Int64' ;
INTNATIVE: 'IntNative' ;
UINT8: 'UInt8' ;
UINT16: 'UInt16' ;
UINT32: 'UInt32' ;
UINT64: 'UInt64' ;
UINTNATIVE: 'UIntNative' ;
FLOAT16: 'Float16' ;
FLOAT32: 'Float32' ;
FLOAT64: 'Float64' ;
RUNE: 'Rune' ;
BOOLEAN: 'Bool' ;
UNIT: 'Unit' ;
Nothing: 'Nothing' ;
STRUCT: 'struct' ;
ENUM: 'enum' ;
THISTYPE: 'This';
PACKAGE: 'package' ;
IMPORT: 'import' ;
CLASS: 'class' ;
INTERFACE: 'interface' ;
```

```
FUNC: 'func';
MAIN: 'main';
LET: 'let' ;
VAR: 'var' ;
CONST: 'const' ;
TYPE_ALIAS: 'type' ;
INIT: 'init'  ;
THIS: 'this' ;
SUPER: 'super' ;
IF: 'if' ;
ELSE: 'else' ;
CASE: 'case' ;
TRY: 'try' ;
CATCH: 'catch' ;
FINALLY: 'finally' ;
FOR: 'for' ;
DO: 'do' ;
WHILE: 'while' ;
THROW: 'throw' ;
RETURN: 'return' ;
CONTINUE: 'continue' ;
BREAK: 'break' ;
IS: 'is' ;
AS: 'as' ;
IN: 'in' ;
MATCH: 'match' ;
FROM: 'from' ;
WHERE: 'where';
EXTEND: 'extend';
SPAWN: 'spawn';
SYNCHRONIZED: 'synchronized';
MACRO: 'macro';
QUOTE: 'quote';
TRUE: 'true';
FALSE: 'false';
STATIC: 'static';
PUBLIC: 'public' ;
PRIVATE: 'private' ;
PROTECTED: 'protected' ;
OVERRIDE: 'override' ;
REDEF: 'redef' ;
```

```
ABSTRACT: 'abstract' ;
OPEN: 'open' ;
OPERATOR: 'operator' ;
FOREIGN: 'foreign';
INOUT: 'inout';
PROP: 'prop';
MUT: 'mut';
UNSAFE: 'unsafe';
GET: 'get';
SET: 'set';
```

A.1.5 字面量

```
IntegerLiteralSuffix
   : 'i8' |'i16' |'i32' |'i64' |'u8' |'u16' |'u32' | 'u64'
   ;

IntegerLiteral
   : BinaryLiteral IntegerLiteralSuffix?
   | OctalLiteral IntegerLiteralSuffix?
   | DecimalLiteral '_'* IntegerLiteralSuffix?
   | HexadecimalLiteral IntegerLiteralSuffix?
   ;
BinaryLiteral
    : '0' [bB] BinDigit (BinDigit | '_')*
    ;
BinDigit
    : [01]
    ;
OctalLiteral
    : '0' [oO] OctalDigit (OctalDigit | '_')*
    ;
OctalDigit
    : [0-7]
    ;
DecimalLiteral
    : (DecimalDigitWithOutZero (DecimalDigit | '_')*) | DecimalDigit
    ;
fragment DecimalFragment
    : DecimalDigit (DecimalDigit | '_')*
    ;
fragment DecimalDigit
```

```
    : [0-9]
    ;
fragment DecimalDigitWithOutZero
    : [1-9]
    ;
HexadecimalLiteral
    : '0' [xX] HexadecimalDigits
    ;

HexadecimalDigits
    : HexadecimalDigit (HexadecimalDigit | '_')*
    ;

HexadecimalDigit
    : [0-9a-fA-F]
    ;

FloatLiteralSuffix
    : 'f16' | 'f32' | 'f64'
    ;

FloatLiteral
    : (DecimalLiteral DecimalExponent | DecimalFraction DecimalExponent? |
     ↪  (DecimalLiteral DecimalFraction) DecimalExponent?)  FloatLiteralSuffix?
    | ( Hexadecimalprefix (HexadecimalDigits | HexadecimalFraction |
     ↪  (HexadecimalDigits HexadecimalFraction)) HexadecimalExponent)
    ;

fragment DecimalFraction : '.' DecimalFragment ;
fragment DecimalExponent : FloatE Sign? DecimalFragment  ;
fragment Sign : [-] ;
fragment Hexadecimalprefix : '0' [xX] ;


DecimalFraction : '.' DecimalLiteral ;
DecimalExponent : FloatE Sign? DecimalLiteral  ;
HexadecimalFraction : '.' HexadecimalDigits ;
HexadecimalExponent : FloatP Sign? DecimalFragment ;
FloatE : [eE] ;
FloatP : [pP] ;
Sign : [-] ;
```

```
    Hexadecimalprefix : '0' [xX] ;

    RuneLiteral
        : '\'' (SingleChar | EscapeSeq ) '\''
        ;

    SingleChar
        :   ~['\\\r\n]
        ;

    EscapeSeq
        : UniCharacterLiteral
        | EscapedIdentifier
        ;

    UniCharacterLiteral
        : '\\' 'u' '{' HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit '}'
        | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
         ↪  HexadecimalDigit '}'
        ;

    EscapedIdentifier
        : '\\' ('t' | 'b' | 'r' | 'n' | '\'' | '"' | '\\' | 'f' | 'v' | '0' | '$')
        ;

    ByteLiteral
        : 'b' '\'' (SingleCharByte | ByteEscapeSeq) '\''
        ;

    ByteEscapeSeq
```

```
    : HexCharByte
    | ByteEscapedIdentifier
    ;

SingleCharByte
    // ASCII 0x00~0x7F without \n \r \' \" \\
    // +-------+-----+-----+
    // | Rune  | Hex | Dec |
    // +-------+-----+-----+
    // | \n    |  0A |  10 |
    // | \r    |  0D |  13 |
    // | \"    |  22 |  34 |
    // | \'    |  27 |  39 |
    // | \\    |  5C |  92 |
    // +-------+-----+-----+
    :
     ↪  [\u0000-\u0009\u000B\u000C\u000E-\u0021\u0023-\u0026\u0028-\u005B\u005D-\u007F]
    ;

fragment ByteEscapedIdentifier
    : '\\' ('t' | 'b' | 'r' | 'n' | '\'' | '"' | '\\' | 'f' | 'v' | '0')
    ;

fragment HexCharByte
    : '\\' 'u' '{' HexadecimalDigit '}'
    | '\\' 'u' '{' HexadecimalDigit HexadecimalDigit '}'
    ;

ByteStringArrayLiteral
    : 'b' '"' (SingleCharByte | ByteEscapeSeq)* '"'
    ;

JStringLiteral
    : 'J' '"' (SingleChar | EscapeSeq)* '"'
    ;

LineStrText
    : ~["\\\r\n]
    | EscapeSeq
    ;
```

```
TRIPLE_QUOTE_CLOSE
    : MultiLineStringQuote? '"""' ;

MultiLineStringQuote
    : '"'+
    ;

MultiLineStrText
    : ~('\\')
    | EscapeSeq
    ;

MultiLineRawStringLiteral
    : MultiLineRawStringContent
    ;

fragment MultiLineRawStringContent
    : HASH MultiLineRawStringContent HASH
    | HASH '"' .*? '"' HASH
    ;
```

A.1.6 标识符

```
identifier
    : Identifier
    | PUBLIC
    | PRIVATE
    | PROTECTED
    | OVERRIDE
    | ABSTRACT
    | OPEN
    | REDEF
    | GET
    | SET
    ;

Identifier
    : '_'* Letter (Letter | '_' | DecimalDigit)*
    | '`' '_'* Letter (Letter | '_' | DecimalDigit)* '`'
    ;

Letter
```

```
    : [a-zA-Z]
    ;

DollarIdentifier
    : '$' Identifier
    ;
```

A.2 语法

A.2.1 编译单元

```
translationUnit
    : NL* preamble end* topLevelObject* (end+ mainDefinition)? NL* (topLevelObject
    ↪  (end+ topLevelObject?)*)? EOF
    ;

end
    : NL | SEMI
    ;
```

A.2.2 包定义和包导入

```
preamble
    : packageHeader? importList*
    ;

packageHeader
    : PACKAGE NL* packageNameIdentifier end+
    ;

packageNameIdentifier
    : identifier (NL* DOT NL* identifier)*
    ;

importList
    : (FROM NL* identifier)? NL* IMPORT NL* importAllOrSpecified
    (NL* COMMA NL* importAllOrSpecified)* end+
    ;

importAllOrSpecified
    : importAll
    | importSpecified (NL* importAlias)?
```

```
    ;

importSpecified
    : (identifier NL* DOT NL*)+ identifier
    ;

importAll
    : (identifier NL* DOT NL*)+ MUL
    ;

importAlias
    : AS NL* identifier
    ;
```

A.2.3 top-level 定义

```
topLevelObject
    : classDefinition
    | interfaceDefinition
    | functionDefinition
    | variableDeclaration
    | enumDefinition
    | structDefinition
    | typeAlias
    | extendDefinition
    | foreignDeclaration
    | macroDefinition
    | macroExpression
    ;
```

A.2.3.1 class 定义

```
classDefinition
    : (classModifierList NL*)? CLASS NL* identifier
    (NL* typeParameters NL*)?
    (NL* UPPERBOUND NL* superClassOrInterfaces)?
    (NL* genericConstraints)?
    NL* classBody
    ;

superClassOrInterfaces
    : superClass (NL* BITAND NL* superInterfaces)?
```

```
    | superInterfaces
    ;

classModifierList
    : classModifier+
    ;

classModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
    | ABSTRACT
    | OPEN
    ;

typeParameters
    : LT NL* identifier (NL* COMMA NL* identifier)* NL* GT
    ;

superClass
    : classType
    ;

classType
    : (identifier NL* DOT  NL*)*  identifier (NL* typeParameters)?
    ;

typeArguments
    : LT NL* type (NL* COMMA NL* type)* NL* GT
    ;

superInterfaces
    : interfaceType (NL* COMMA NL* interfaceType )*
    ;

interfaceType
    : classType
    ;

genericConstraints
```

```
    : WHERE NL* (identifier | THISTYPE) NL* UPPERBOUND NL* upperBounds (NL* COMMA
     ↪  NL* (identifier | THISTYPE) NL* UPPERBOUND NL* upperBounds)*
    ;

upperBounds
    : type (NL* BITAND NL* type)*
    ;

classBody
    : LCURL end*
         classMemberDeclaration* NL*
         classPrimaryInit? NL*
         classMemberDeclaration* end* RCURL
    ;

classMemberDeclaration
    : (classInit
    | staticInit
    | variableDeclaration
    | functionDefinition
    | operatorFunctionDefinition
    | macroExpression
    | propertyDefinition
    ) end*
    ;

classInit
    : (classNonStaticMemberModifier | CONST NL*)? INIT NL* functionParameters NL*
     ↪  block
    ;

staticInit
    : STATIC INIT LPAREN RPAREN
    LCURL
    expressionOrDeclarations?
    RCURL
    ;

classPrimaryInit
    : (classNonStaticMemberModifier | CONST NL*)?  className NL* LPAREN NL*
          classPrimaryInitParamLists
```

```
        NL* RPAREN NL*
      LCURL NL*
           (SUPER callSuffix)? end
           expressionOrDeclarations?
      NL* RCURL
    ;

className
    : identifier
    ;

classPrimaryInitParamLists
    : unnamedParameterList (NL* COMMA NL* namedParameterList)? (NL* COMMA NL*
    ↪  classNamedInitParamList)?
    | unnamedParameterList (NL* COMMA NL* classUnnamedInitParamList)? (NL* COMMA
    ↪  NL* classNamedInitParamList)?
    | classUnnamedInitParamList (NL* COMMA NL* classNamedInitParamList)?
    | namedParameterList (NL* COMMA NL* classNamedInitParamList)?
    | classNamedInitParamList
    ;

classUnnamedInitParamList
    : classUnnamedInitParam (NL* COMMA NL* classUnnamedInitParam)*
    ;

classNamedInitParamList
    : classNamedInitParam (NL* COMMA NL* classNamedInitParam)*
    ;

classUnnamedInitParam
    : (classNonSMemberModifier NL*)? (LET | VAR) NL* identifier NL* COLON NL* type
    ;

classNamedInitParam
    : (classNonSMemberModifier NL*)? (LET | VAR) NL* identifier NL* NOT NL* COLON
    ↪  NL* type (NL* ASSIGN NL* expression)?
    ;

classNonStaticMemberModifier
    : PUBLIC
    | PRIVATE
```

```
    | PROTECTED
    | INTERNAL
    ;
```

A.2.3.2 interface 定义

```
interfaceDefinition
    : (interfaceModifierList NL*)? INTERFACE NL* identifier
    (NL* typeParameters NL*)?
    (NL* UPPERBOUND NL* superInterfaces)?
    (NL* genericConstraints)?
    (NL* interfaceBody)
    ;

interfaceBody
    : LCURL end* interfaceMemberDeclaration* end* RCURL
    ;

interfaceMemberDeclaration
    : (functionDefinition
    | operatorFunctionDefinition
    | macroExpression
    | propertyDefinition) end*
    ;

interfaceModifierList
    : (interfaceModifier NL*)+
    ;

interfaceModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
    | OPEN
    ;
```

A.2.3.3 function 定义

```
functionDefinition
    :(functionModifierList NL*)? FUNC
```

```
        NL* identifier
        (NL* typeParameters NL*)?
        NL* functionParameters
        (NL* COLON NL* type)?
        (NL* genericConstraints)?
        (NL* block)?
        ;

    operatorFunctionDefinition
        : (functionModifierList NL*)? OPERATOR NL* FUNC
        NL* overloadedOperators
        (NL* typeParameters NL*)?
        NL* functionParameters
        (NL* COLON NL* type)?
        (NL* genericConstraints)?
        (NL* block)?
        ;

    functionParameters
        : (LPAREN (NL* unnamedParameterList ( NL* COMMA NL* namedParameterList)? )?
            NL* RPAREN NL*)
        | (LPAREN NL* (namedParameterList NL*)? RPAREN NL*)
        ;

    nondefaultParameterList
        : unnamedParameter (NL* COMMA NL* unnamedParameter)*
            (NL* COMMA NL*  namedParameter)*
        | namedParameter (NL* COMMA NL* namedParameter)*
        ;

    unnamedParameterList
        : unnamedParameter (NL* COMMA NL*  unnamedParameter)*
        ;

    unnamedParameter
        : (identifier | WILDCARD) NL* COLON NL* type
        ;

    namedParameterList
        : (namedParameter | defaultParameter)
          (NL* COMMA NL* (namedParameter | defaultParameter))*
```

```
    ;

namedParameter
    : identifier NL* NOT NL* COLON NL* type
    ;

defaultParameter
    : identifier NL* NOT NL* COLON NL* type NL* ASSIGN NL* expression
    ;

functionModifierList
    : (functionModifier NL*)+
    ;

functionModifier
    : PUBLIC
    | PRIVATE
    | PROTECTED
    | INTERNAL
    | STATIC
    | OPEN
    | OVERRIDE
    | OPERATOR
    | REDEF
    | MUT
    | UNSAFE
    | CONST
    ;
```

A.2.3.4 变量定义

```
variableDeclaration
    : variableModifier* NL* (LET | VAR | CONST) NL* patternsMaybeIrrefutable (
    ↪  (NL* COLON NL* type)? (NL* ASSIGN NL* expression)
                                                                | (NL* COLON NL*
                                                                ↪  type)
                                                                )
    ;

variableModifier
    : PUBLIC
    | PRIVATE
```

```
    | PROTECTED
    | INTERNAL
    | STATIC
    ;
```

A.2.3.5 enum 定义

```
enumDefinition
    : (enumModifier NL*)? ENUM NL* identifier (NL* typeParameters NL*)?
    (NL* UPPERBOUND NL* superInterfaces)?
    (NL* genericConstraints)? NL* LCURL end* enumBody end* RCURL
    ;

enumBody
    : (BITOR NL*)? caseBody (NL* BITOR NL* caseBody)*
    (NL*
    ( functionDefinition
    | operatorFunctionDefinition
    | propertyDefinition
    | macroExpression
    ))*
    ;

caseBody
    : identifier ( NL* LPAREN NL* type (NL* COMMA NL* type)* NL* RPAREN)?
    ;

enumModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
    ;
```

A.2.3.6 struct 定义

```
structDefinition
    : (structModifier NL*)? STRUCT NL* identifier (NL* typeParameters NL*)?
    (NL* UPPERBOUND NL* superInterfaces)?
    (NL* genericConstraints)? NL* structBody
    ;

structBody
```

```
        : LCURL end*
            structMemberDeclaration* NL*
            structPrimaryInit? NL*
            structMemberDeclaration*
          end* RCURL
        ;

    structMemberDeclaration
        : (structInit
        | staticInit
        | variableDeclaration
        | functionDefinition
        | operatorFunctionDefinition
        | macroExpression
        | propertyDefinition
        ) end*
        ;

    structInit
        : (structNonStaticMemberModifier | CONST NL*)? INIT NL* functionParameters NL*
         ↪ block
        ;

    staticInit
        : STATIC INIT LPAREN RPAREN
        LCURL
        expressionOrDeclarations?
        RCURL
        ;

    structPrimaryInit
        : (structNonStaticMemberModifier | CONST NL*)? structName NL* LPAREN NL*
         ↪ structPrimaryInitParamLists? NL* RPAREN NL*
          LCURL NL*
               expressionOrDeclarations?
          NL* RCURL
        ;

    structName
        : identifier
        ;
```

```
structPrimaryInitParamLists
    : unnamedParameterList (NL* COMMA NL* namedParameterList)? (NL* COMMA NL*
    ↪  structNamedInitParamList)?
    | unnamedParameterList (NL* COMMA NL* structUnnamedInitParamList)? (NL* COMMA
    ↪  NL* structNamedInitParamList)?
    | structUnnamedInitParamList (NL* COMMA NL* structNamedInitParamList)?
    | namedParameterList (NL* COMMA NL* structNamedInitParamList)?
    | structNamedInitParamList
    ;

structUnnamedInitParamList
    : structUnnamedInitParam (NL* COMMA NL* structUnnamedInitParam)*
    ;

structNamedInitParamList
    : structNamedInitParam (NL* COMMA NL*  structNamedInitParam)*
    ;

structUnnamedInitParam
    : (structNonStaticMemberModifier NL*)? (LET | VAR) NL* identifier NL* COLON
    ↪  NL* type
    ;

structNamedInitParam
    : (structNonStaticMemberModifier NL*)? (LET | VAR) NL* identifier NL* NOT NL*
    ↪  COLON NL* type (NL* ASSIGN NL* expression)?
    ;

structModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
    ;

structNonStaticMemberModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
```

```
    ;
```

A.2.3.7 类型别名定义

```
typeAlias
    : (typeModifier NL*)? TYPE_ALIAS NL* identifier (NL* typeParameters)? NL*
    ↪  ASSIGN NL* type end*
    ;

typeModifier
    : PUBLIC
    | PROTECTED
    | INTERNAL
    | PRIVATE
    ;
```

A.2.3.8 扩展定义

```
extendDefinition
    : EXTEND NL* extendType
    (NL* UPPERBOUND NL* superInterfaces)? (NL* genericConstraints)?
    NL* extendBody
    ;

extendType
    : (typeParameters)? (identifier NL* DOT  NL*)* identifier (NL* typeArguments)?
    | INT8
    | INT16
    | INT32
    | INT64
    | INTNATIVE
    | UINT8
    | UINT16
    | UINT32
    | UINT64
    | UINTNATIVE
    | FLOAT16
    | FLOAT32
    | FLOAT64
    | RUNE
    | BOOLEAN
    | NOTHING
    | UNIT
```

```
    ;

extendBody
    : LCURL end* extendMemberDeclaration* end* RCURL
    ;

extendMemberDeclaration
    : (functionDefinition
    | operatorFunctionDefinition
    | macroExpression
    | propertyDefinition
    ) end*
    ;
```

A.2.3.9 foreign 声明

```
foreignDeclaration
    : FOREIGN NL* (foreignBody | foreignMemberDeclaration)
    ;

foreignBody
    : LCURL end* foreignMemberDeclaration* end* RCURL
    ;

foreignMemberDeclaration
    : (classDefinition
    | interfaceDefinition
    | functionDefinition
    | macroExpression
    | variableDeclaration) end*
    ;
```

A.2.3.10 Annotation

```
annotationList: annotation+;

annotation
    : AT (identifier NL* DOT)* identifier (LSQUARE NL* annotationArgumentList NL*
    ↪ RSQUARE)?
    ;

annotationArgumentList
    : annotationArgument (NL* COMMA NL* annotationArgument)* NL* COMMA?
```

```
    ;

annotationArgument
    : identifier NL* COLON NL* expression
    | expression
    ;
```

A.2.3.11 宏声明

```
macroDefinition
    : PUBLIC NL* MACRO NL* identifier NL*
    (macroWithoutAttrParam | macroWithAttrParam) NL*
    (COLON NL* identifier NL*)?
    (ASSIGN NL* expression | block)
    ;

macroWithoutAttrParam
    : LPAREN NL* macroInputDecl NL* RPAREN
    ;

macroWithAttrParam
    : LPAREN NL* macroAttrDecl NL* COMMA NL* macroInputDecl NL* RPAREN
    ;

macroInputDecl
    : identifier NL* COLON NL* identifier
    ;

macroAttrDecl
    : identifier NL* COLON NL* identifier
    ;
```

A.2.3.12 属性定义

```
propertyDefinition
    : propertyModifier* NL* PROP NL* identifier NL* COLON NL* type NL*
    ↪ propertyBody?
    ;

propertyBody
    : LCURL end* propertyMemberDeclaration+ end* RCURL
    ;
```

```
propertyMemberDeclaration
    : GET NL* LPAREN RPAREN NL* block end*
    | SET NL* LPAREN identifier RPAREN NL* block end*
    ;

propertyModifier
    : PUBLIC
    | PRIVATE
    | PROTECTED
    | INTERNAL
    | STATIC
    | OPEN
    | OVERRIDE
    | REDEF
    | MUT
    ;
```

A.2.3.13 程序入口定义

```
mainDefinition
    : MAIN
      NL* functionParameters
      (NL* COLON NL* type)?
      NL* block
    ;
```

A.2.4 类型

```
type
    : arrowType
    | tupleType
    | prefixType
    | atomicType
    ;

arrowType
    : arrowParameters NL* ARROW NL* type
    ;

arrowParameters
    : LPAREN NL* (type (NL* COMMA NL* type)* NL*)? RPAREN
    ;
```

```
tupleType
    : LPAREN NL* type (NL* COMMA NL* type)+ NL* RPAREN
    ;

prefixType
    : prefixTypeOperator type
    ;

prefixTypeOperator
    : QUEST
    ;

atomicType
    : charLangTypes
    | userType
    | parenthesizedType
    ;

charLangTypes
    : numericTypes
    | RUNE
    | BOOLEAN
    | Nothing
    | UNIT
    | THISTYPE
    ;

numericTypes
    : INT8
    | INT16
    | INT32
    | INT64
    | INTNATIVE
    | UINT8
    | UINT16
    | UINT32
    | UINT64
    | UINTNATIVE
    | FLOAT16
    | FLOAT32
```

```
    | FLOAT64
    ;

userType
    : (identifier NL* DOT NL*)* identifier ( NL* typeArguments)?
    ;

parenthesizedType
    : LPAREN NL* type NL* RPAREN
    ;
```

A.2.5 表达式语法

```
expression
    : assignmentExpression
    ;

assignmentExpression
    : leftValueExpressionWithoutWildCard NL* assignmentOperator NL* flowExpression
    | leftValueExpression NL* ASSIGN NL* flowExpression
    | tupleLeftValueExpression NL* ASSIGN NL* flowExpression
    | flowExpression
    ;

tupleLeftValueExpression
    : LPAREN NL* (leftValueExpression | tupleLeftValueExpression) (NL* COMMA NL*
    ↪  (leftValueExpression | tupleLeftValueExpression))+ NL* COMMA? NL* RPAREN
    ;

leftValueExpression
    : leftValueExpressionWithoutWildCard
    | WILDCARD
    ;

leftValueExpressionWithoutWildCard
    : identifier
    | leftAuxExpression QUEST? NL* assignableSuffix
    ;

leftAuxExpression
    : identifier (NL* typeArguments)?
    | type
```

```
        | thisSuperExpression
        | leftAuxExpression QUEST? NL* DOT NL* identifier (NL* typeArguments)?
        | leftAuxExpression QUEST? callSuffix
        | leftAuxExpression QUEST? indexAccess
        ;

    assignableSuffix
        : fieldAccess
        | indexAccess
        ;

    fieldAccess
        : NL* DOT NL* identifier
        ;

    flowExpression
        : coalescingExpression (NL* flowOperator NL* coalescingExpression)*
        ;

    coalescingExpression
        : logicDisjunctionExpression (NL* QUEST QUEST NL* logicDisjunctionExpression)*
        ;

    logicDisjunctionExpression
        : logicConjunctionExpression (NL* OR NL* logicConjunctionExpression)*
        ;

    logicConjunctionExpression
        : rangeExpression (NL* AND NL* rangeExpression)*
        ;

    rangeExpression
        : bitwiseDisjunctionExpression NL* (CLOSEDRANGEOP | RANGEOP) NL*
        ↪  bitwiseDisjunctionExpression (NL* COLON NL* bitwiseDisjunctionExpression)?
        | bitwiseDisjunctionExpression
        ;

    bitwiseDisjunctionExpression
        : bitwiseXorExpression (NL* BITOR NL* bitwiseXorExpression)*
        ;
```

```
bitwiseXorExpression
    : bitwiseConjunctionExpression (NL* BITXOR NL* bitwiseConjunctionExpression)*
    ;

bitwiseConjunctionExpression
    : equalityComparisonExpression (NL* BITAND NL* equalityComparisonExpression)*
    ;

equalityComparisonExpression
    : comparisonOrTypeExpression (NL* equalityOperator NL*
    ↪  comparisonOrTypeExpression)?
    ;

comparisonOrTypeExpression
    : shiftingExpression (NL* comparisonOperator NL* shiftingExpression)?
    | shiftingExpression (NL* IS NL* type)?
    | shiftingExpression (NL* AS NL* type)?
    ;

shiftingExpression
    : additiveExpression (NL* shiftingOperator NL* additiveExpression)*
    ;

additiveExpression
    : multiplicativeExpression (NL* additiveOperator NL* multiplicativeExpression)*
    ;

multiplicativeExpression
    : exponentExpression (NL* multiplicativeOperator NL* exponentExpression)*
    ;

exponentExpression
    : prefixUnaryExpression (NL* exponentOperator NL* prefixUnaryExpression)*
    ;

prefixUnaryExpression
    : prefixUnaryOperator* incAndDecExpression
    ;

incAndDecExpression
    : postfixExpression (INC | DEC )?
```

```
    ;

postfixExpression
     : atomicExpression
     | type NL* DOT NL* identifier
     | postfixExpression NL* DOT NL* identifier (NL* typeArguments)?
     | postfixExpression callSuffix
     | postfixExpression indexAccess
     | postfixExpression NL* DOT NL* identifier callSuffix?
      ↪  trailingLambdaExpression
     | identifier callSuffix? trailingLambdaExpression
     | postfixExpression (QUEST questSeperatedItems)+
     ;

questSeperatedItems
     : questSeperatedItem+
     ;

questSeperatedItem
     : itemAfterQuest (callSuffix | callSuffix? trailingLambdaExpression |
      ↪  indexAccess)?
     ;

itemAfterQuest
     : DOT identifier (NL* typeArguments)?
     | callSuffix
     | indexAccess
     | trailingLambdaExpression
     ;

callSuffix
    : LPAREN NL* (valueArgument (NL* COMMA NL* valueArgument)* NL*)? RPAREN
    ;

valueArgument
    : identifier NL* COLON NL* expression
    | expression
    | refTransferExpression
    ;

refTransferExpression
```

```
    : INOUT (expression DOT)? identifier
    ;

indexAccess
    : LSQUARE NL* (expression | rangeElement) NL* RSQUARE
    ;

rangeElement
    :  RANGEOP
    | ( CLOSEDRANGEOP | RANGEOP ) NL* expression
    | expression NL* RANGEOP
    ;

atomicExpression
    : literalConstant
    | collectionLiteral
    | tupleLiteral
    | identifier (NL* typeArguments)?
    | unitLiteral
    | ifExpression
    | matchExpression
    | loopExpression
    | tryExpression
    | jumpExpression
    | numericTypeConvExpr
    | thisSuperExpression
    | spawnExpression
    | synchronizedExpression
    | parenthesizedExpression
    | lambdaExpression
    | quoteExpression
    | macroExpression
    | unsafeExpression
    ;

literalConstant
    : IntegerLiteral
    | FloatLiteral
    | RuneLiteral
    | ByteLiteral
    | booleanLiteral
```

```
        | stringLiteral
        | ByteStringArrayLiteral
        | unitLiteral
        ;

    booleanLiteral
        : TRUE
        | FALSE
        ;

    stringLiteral
        : lineStringLiteral
        | multiLineStringLiteral
        | MultiLineRawStringLiteral
        ;

    lineStringContent
        :  LineStrText
        ;

    lineStringLiteral
        : QUOTE_OPEN (lineStringExpression | lineStringContent)* QUOTE_CLOSE
        ;

    lineStringExpression
        : LineStrExprStart SEMI* (expressionOrDeclaration (SEMI+
        ↪  expressionOrDeclaration?)*) SEMI* RCURL
        ;

    multiLineStringContent
        : MultiLineStrText
        ;

    multiLineStringLiteral
        : TRIPLE_QUOTE_OPEN (multiLineStringExpression | multiLineStringContent)*
        ↪  TRIPLE_QUOTE_CLOSE
        ;

    multiLineStringExpression
        : MultiLineStrExprStart end* (expressionOrDeclaration (end+
        ↪  expressionOrDeclaration?)*) end* RCURL
```

```
    ;

collectionLiteral
    : arrayLiteral
    ;

arrayLiteral
    : LSQUARE (NL* elements)? NL* RSQUARE
    ;

elements
    : element ( NL* COMMA NL* element )*
    ;

element
    : expressionElement
    | spreadElement
    ;

expressionElement
    : expression
    ;

spreadElement
    : MUL expression
    ;

tupleLiteral
    : LPAREN NL* expression (NL* COMMA NL*expression)+ NL* RPAREN
    ;

unitLiteral
    : LPAREN NL* RPAREN
    ;

ifExpression
    : IF NL* LPAREN NL* (LET NL* deconstructPattern NL* BACKARROW NL*)? expression
    ↪  NL* RPAREN NL* block
    (NL* ELSE (NL* ifExpression | NL* block))?
    ;
```

```
deconstructPattern
    : constantPattern
    | wildcardPattern
    | varBindingPattern
    | tuplePattern
    | enumPattern
    ;

matchExpression
    : MATCH NL* LPAREN NL* expression NL* RPAREN NL* LCURL NL* matchCase+ NL* RCURL
    | MATCH NL* LCURL NL* (CASE NL* (expression | WILDCARD) NL* DOUBLE_ARROW NL*
     ↪ expressionOrDeclaration (end+ expressionOrDeclaration?)*)+ NL* RCURL
    ;

matchCase
    : CASE NL* pattern NL* patternGuard? NL* DOUBLE_ARROW NL*
     ↪ expressionOrDeclaration (end+ expressionOrDeclaration?)*
    ;

patternGuard
    : WHERE NL* expression
    ;

pattern
   : constantPattern
   | wildcardPattern
   | varBindingPattern
   | tuplePattern
   | typePattern
   | enumPattern
   ;

constantPattern
   : literalConstant NL* ( NL* BITOR NL* literalConstant)*
   ;

wildcardPattern
   : WILDCARD
   ;

varBindingPattern
```

```
    : identifier
    ;

tuplePattern
    : LPAREN NL* pattern (NL* COMMA NL* pattern)+ NL* RPAREN
    ;

typePattern
    : (WILDCARD | identifier) NL* COLON NL* type
    ;

enumPattern
    : NL* ((userType NL* DOT NL*)? identifier enumPatternParameters?) (NL* BITOR
    ↪  NL* ((userType NL* DOT NL*)? identifier enumPatternParameters?))*
    ;

enumPatternParameters
    : LPAREN NL* pattern (NL* COMMA NL* pattern)* NL* RPAREN
    ;

loopExpression
    : forInExpression
    | whileExpression
    | doWhileExpression
    ;

forInExpression
    : FOR NL* LPAREN NL* patternsMaybeIrrefutable NL* IN NL* expression NL*
    ↪  patternGuard? NL* RPAREN NL* block
    ;

patternsMaybeIrrefutable
    : wildcardPattern
    | varBindingPattern
    | tuplePattern
    | enumPattern
    ;

whileExpression
    : WHILE NL* LPAREN NL* (LET NL* deconstructPattern NL* BACKARROW NL*)?
    ↪  expression NL* RPAREN NL* block
```

```
    ;

doWhileExpression
    : DO NL* block NL* WHILE NL* LPAREN NL* expression NL* RPAREN
    ;

tryExpression
    : TRY NL* block NL* FINALLY NL* block
    | TRY NL* block (NL* CATCH NL* LPAREN NL* catchPattern NL* RPAREN NL* block)+
    ↪  (NL* FINALLY NL* block)?
    | TRY NL* LPAREN NL* resourceSpecifications NL* RPAREN NL* block
    (NL* CATCH NL* LPAREN NL* catchPattern NL* RPAREN NL* block)* (NL* FINALLY NL*
    ↪  block)?
    ;

catchPattern
    : wildcardPattern
    | exceptionTypePattern
    ;

exceptionTypePattern
    : (WILDCARD | identifier) NL* COLON NL* type (NL* BITOR NL* type)*
    ;

resourceSpecifications
    : resourceSpecification (NL* COMMA NL* resourceSpecification)*
    ;

resourceSpecification
    : identifier (NL* COLON NL* classType)? NL* ASSIGN NL* expression
    ;

jumpExpression
    : THROW NL* expression
    | RETURN (NL* expression)?
    | CONTINUE
    | BREAK
    ;

numericTypeConvExpr
    : numericTypes LPAREN NL* expression NL* RPAREN
```

```
    ;

thisSuperExpression
    : THIS
    | SUPER
    ;

lambdaExpression
    : LCURL NL* lambdaParameters? NL* DOUBLE_ARROW NL* expressionOrDeclarations?
    ↪ RCURL
    ;

trailingLambdaExpression
    : LCURL NL* (lambdaParameters? NL* DOUBLE_ARROW NL*)?
    ↪ expressionOrDeclarations? RCURL
    ;

lambdaParameters
    : lambdaParameter (NL* COMMA NL* lambdaParameter)*
    ;

lambdaParameter
    : (identifier | WILDCARD) (NL* COLON NL* type)?
    ;

spawnExpression
    : SPAWN (LPAREN NL* expression NL* RPAREN)? NL* trailingLambdaExpression
    ;

synchronizedExpression
    : SYNCHRONIZED LPAREN NL* expression NL* RPAREN NL* block
    ;

parenthesizedExpression
    : LPAREN NL* expression NL* RPAREN
    ;

block
    : LCURL expressionOrDeclarations RCURL
    ;
```

```
    unsafeExpression
        : UNSAFE NL* block
        ;

    expressionOrDeclarations
        : end* (expressionOrDeclaration (end+ expressionOrDeclaration?)*)?
        ;

    expressionOrDeclaration
        : expression
        | varOrfuncDeclaration
        ;

    varOrfuncDeclaration
        : functionDefinition
        | variableDeclaration
        ;

    quoteExpression
        : QUOTE quoteExpr
        ;

    quoteExpr
        : LPAREN NL* quoteParameters NL* RPAREN
        ;

    quoteParameters
        : (NL* quoteToken | NL* quoteInterpolate | NL* macroExpression)+
        ;

    quoteToken
        : DOT | COMMA | LPAREN | RPAREN | LSQUARE | RSQUARE | LCURL | RCURL | EXP | MUL
         ↪  | MOD | DIV | ADD | SUB
        | PIPELINE | COMPOSITION
        | INC | DEC | AND | OR | NOT | BITAND | BITOR | BITXOR | LSHIFT | RSHIFT |
         ↪  COLON | SEMI
        | ASSIGN | ADD_ASSIGN | SUB_ASSIGN | MUL_ASSIGN | EXP_ASSIGN | DIV_ASSIGN |
         ↪  MOD_ASSIGN
        | AND_ASSIGN | OR_ASSIGN | BITAND_ASSIGN | BITOR_ASSIGN | BITXOR_ASSIGN |
         ↪  LSHIFT_ASSIGN | RSHIFT_ASSIGN
        | ARROW | BACKARROW | DOUBLE_ARROW | ELLIPSIS | CLOSEDRANGEOP | RANGEOP | HASH
         ↪  | AT | QUEST | UPPERBOUND | LT | GT | LE | GE
```

```
    | NOTEQUAL | EQUAL | WILDCARD | BACKSLASH | QUOTESYMBOL | DOLLAR
    | INT8 | INT16 | INT32 | INT64 | INTNATIVE | UINT8 | UINT16 | UINT32 | UINT64 |
     ↪ UINTNATIVE | FLOAT16
    | FLOAT32 | FLOAT64 | RUNE | BOOL | UNIT | NOTHING | STRUCT | ENUM | THIS
    | PACKAGE | IMPORT | CLASS | INTERFACE | FUNC | LET | VAR | CONST | TYPE
    | INIT | THIS | SUPER | IF | ELSE | CASE | TRY | CATCH | FINALLY
    | FOR | DO | WHILE | THROW | RETURN | CONTINUE | BREAK | AS | IN
    | MATCH | FROM | WHERE | EXTEND | SPAWN | SYNCHRONIZED | MACRO | QUOTE | TRUE |
     ↪ FALSE
    | STATIC | PUBLIC | PRIVATE | PROTECTED
    | OVERRIDE | ABSTRACT | OPEN | OPERATOR | FOREIGN
    | Identifier | DollarIdentifier
    | literalConstant
    ;

quoteInterpolate
    : DOLLAR LPAREN NL* expression NL* RPAREN
    ;

macroExpression
    : AT Identifier macroAttrExpr? NL* (macroInputExprWithoutParens |
     ↪ macroInputExprWithParens)
    ;

macroAttrExpr
    : LSQUARE NL* quoteToken* NL* RSQUARE
    ;

macroInputExprWithoutParens
    : functionDefinition
    | operatorFunctionDefinition
    | staticInit
    | structDefinition
    | structPrimaryInit
    | structInit
    | enumDefinition
    | caseBody
    | classDefinition
    | classPrimaryInit
    | classInit
    | interfaceDefinition
```

```
        | variableDeclaration
        | propertyDefinition
        | extendDefinition
        | macroExpression
        ;

    macroInputExprWithParens
        : LPAREN NL* macroTokens NL* RPAREN
        ;

    macroTokens
        : (quoteToken | macroExpression)*
        ;

    assignmentOperator
        : ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | EXP_ASSIGN
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | AND_ASSIGN
        | OR_ASSIGN
        | BITAND_ASSIGN
        | BITOR_ASSIGN
        | BITXOR_ASSIGN
        | LSHIFT_ASSIGN
        | RSHIFT_ASSIGN
        ;

    equalityOperator
        : NOTEQUAL
        | EQUAL
        ;

    comparisonOperator
        : LT
        | GT
        | LE
        | GE
```

```
    ;

shiftingOperator
    : LSHIFT | RSHIFT
    ;

flowOperator
    : PIPELINE | COMPOSITION
    ;

additiveOperator
    : ADD | SUB
    ;

exponentOperator
    : EXP
    ;

multiplicativeOperator
    : MUL
    | DIV
    | MOD
    ;

prefixUnaryOperator
    : SUB
    | NOT
    ;

overloadedOperators
    : LSQUARE RSQUARE
    | NOT
    | ADD
    | SUB
    | EXP
    | MUL
    | DIV
    | MOD
    | LSHIFT
    | RSHIFT
    | LT
```

```
    | GT
    | LE
    | GE
    | EQUAL
    | NOTEQUAL
    | BITAND
    | BITXOR
    | BITOR
    ;
```