程序设计实践

The Practice of Programming

设计与实现

算法和数据结构

查找

排序

可增长数组

链表

树

散列表

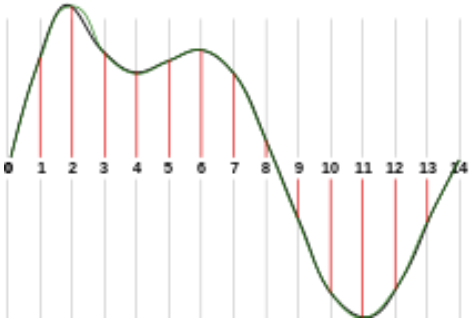
数组下标访问

最简单，最快，最常用的“算法”

应用条件：

1. 要查询的数据的索引都是不太大的整数。
2. 这些整数稀疏程度不高。

语音编解码背景知识



每秒8000次采样

每个采样值用16位整数记录

一系列16位整数（PCM格式）

压缩

每个16位整数压缩为8位表示（G711格式）

传输

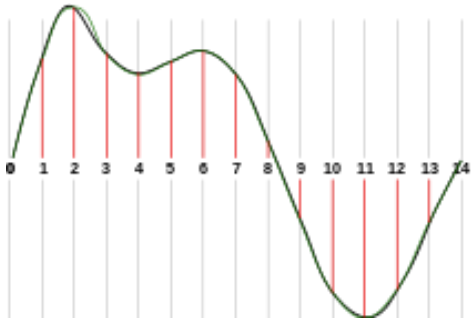
每个采样8位的G711格式数据

解压缩

上一页的程序示例实现的是这个解压缩过程

每个8位整数解压为16位

还原



因为G711编码后的取值，只有256种可能性，所以只要事先构造出一个映射表，就可以将复杂的压缩算法：

$$F^{-1}(y)=\mathrm{sgn}(y)\begin{cases}\frac{|y|(1+\ln(A))}{A}, & |y|<\frac{1}{1+\ln(A)}\\ \frac{\exp(|y|(1+\ln(A))-1)}{A}, & \frac{1}{1+\ln(A)}\le|y|<1.\end{cases}$$

变为一个简单的数组下标访问

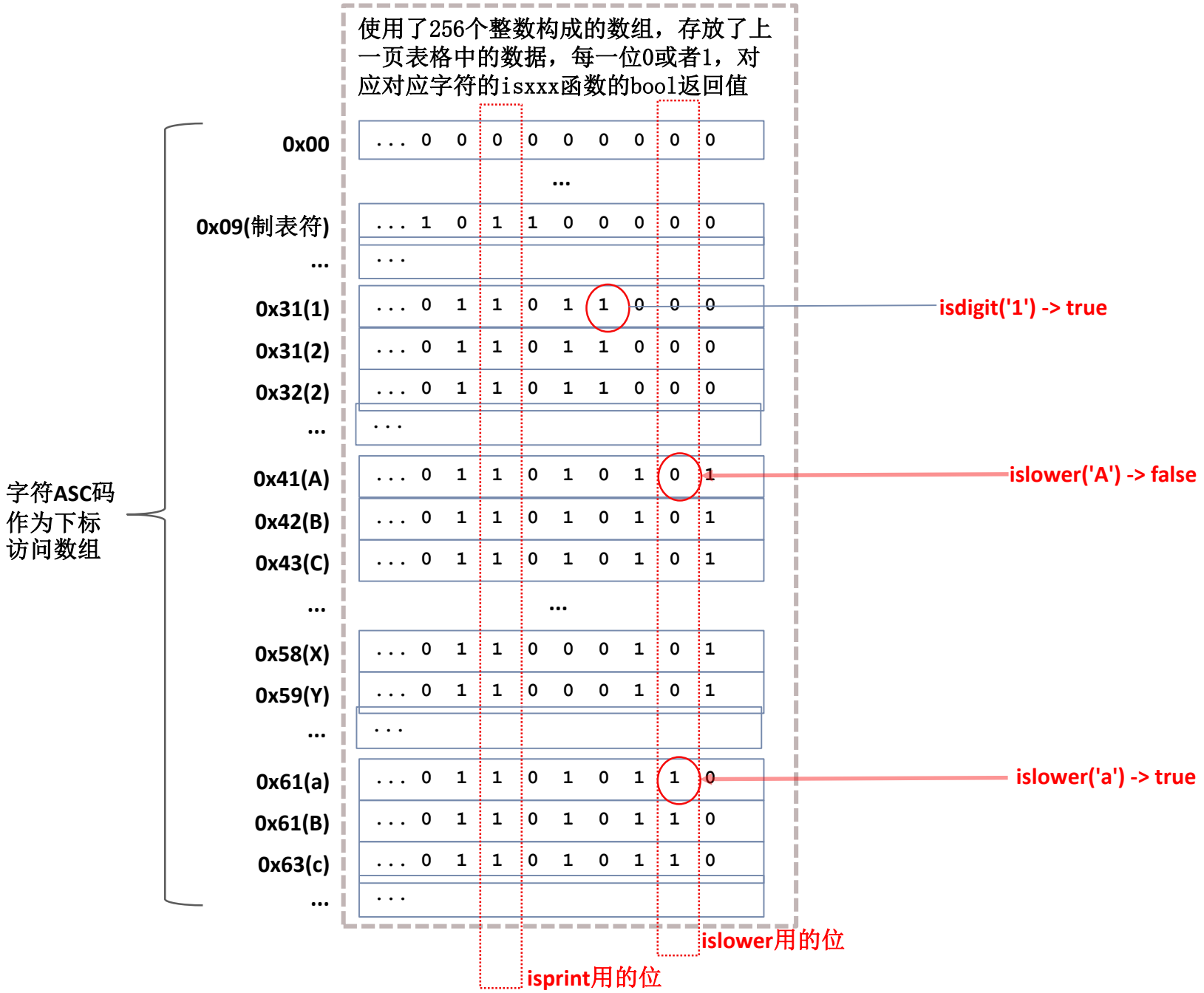
数组下标访问（续）

```
// G711语音解码程序：
static short _alawDecompressTable[256] =
{
    0xEA80, 0xEB80, 0xE880, 0xE980, 0xEE80, 0xEF80, 0xEC80, 0xED80,
    0xE280, 0xE380, 0xE080, 0xE180, 0xE680, 0xE780, 0xE480, 0xE580,
    0xF540, 0xF5C0, 0xF440, 0xF4C0, 0xF740, 0xF7C0, 0xF640, 0xF6C0,
    0xF140, 0xF1C0, 0xF040, 0xF0C0, 0xF340, 0xF3C0, 0xF240, 0xF2C0,
    0x03B0, 0x0390, 0x03F0, 0x03D0, 0x0330, 0x0310, 0x0370, 0x0350,
    ...
};

short alawToPcm(unsigned char sample)
{
    return _alawDecompressTable[sample];
}
```

C标准库中， isdigit, isalpha等函数的实现，使用了数组下标直接访问

	isblank	isgraph	isprint	isspace	isxdigit	isdigit	isalpha	islower	isupper
0x00	0	0	0	0	0	0	0	0	0
...									
0x09(制表符)	1	0	1	1	0	0	0	0	0
...									
0x30(0)	0	1	1	0	1	1	0	0	0
0x31(1)	0	1	1	0	1	1	0	0	0
0x32(2)	0	1	1	0	1	1	0	0	0
...									
0x41(A)	0	1	1	0	1	0	1	0	1
0x42(B)	0	1	1	0	1	0	1	0	1
0x43(C)	0	1	1	0	1	0	1	0	1
...									
0x58(X)	0	1	1	0	0	0	1	0	1
0x59(Y)	0	1	1	0	0	0	1	0	1
...									
0x61(a)	0	1	1	0	1	0	1	1	0
0x62(b)	0	1	1	0	1	0	1	1	0
0x63(c)	0	1	1	0	1	0	1	1	0
...									


使用了256个整数构成的数组，存放了上一页表格中的数据，每一位0或者1，对应对应字符的isxxx函数的bool返回值
0x00
0x09(制表符)
0x31(1)
0x31(2)
0x32(2)
0x41(A)
0x42(B)
0x43(C)
0x58(X)
0x59(Y)
0x61(a)
0x61(B)
0x63(c)
字符ASC码作为下标访问数组
isdigit('1') -> true
islower('A') -> false
islower('a') -> true
islower用的位
isprint用的位

顺序查找(Sequential Search)

简单，低效的查找算法

可能的应用场景：

1. 要查询的数据特别少，例如少于10条。
2. 被查找数据的排列顺序不能自由设定，例如字符串中查找。
3. 查询条件太复杂，以至于匹配条件的开销大于查找。
4. 更新频繁，却很少查找。

有关字符串顺序查找的标准库：

1. C： strchr, strstr
2. C++： std::find
3. Java： String.indexOf

二分查找(Binary Search)

高效，但是对应用条件要求比较苛刻

应用条件：

1. 数据已经有序
2. 数据基本有序，查找前可以很快调整成有序
3. 数据无序，但是排序后更新次数远小于查找次数。

二分查找标准库：

1. C++： std::binary_search
2. Java： Arrays.binarySearch

排序(Sorting)

排序算法：

1. 选择、冒泡、插入排序
2. 快速排序
3. 其它排序算法…

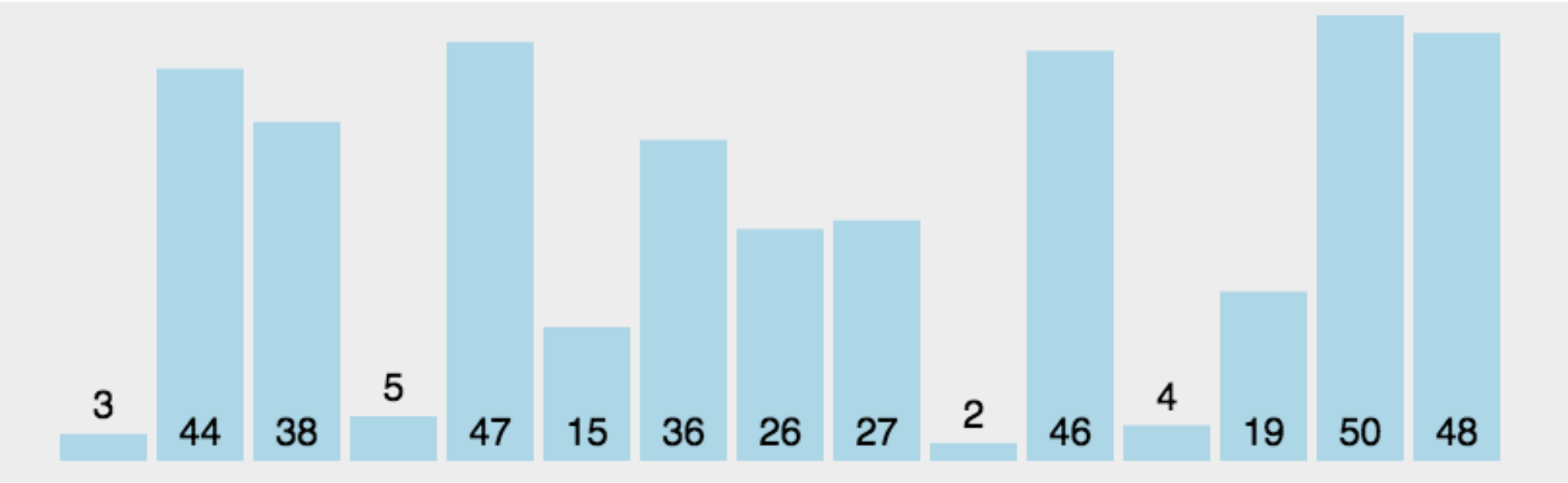
库：

1. C：qsort（快速排序）
2. C++：std::sort
3. Java：**Array.sort**

常见排序算法及动画演示
https://www.cnblogs.com/onepixel/articles/7674659.html

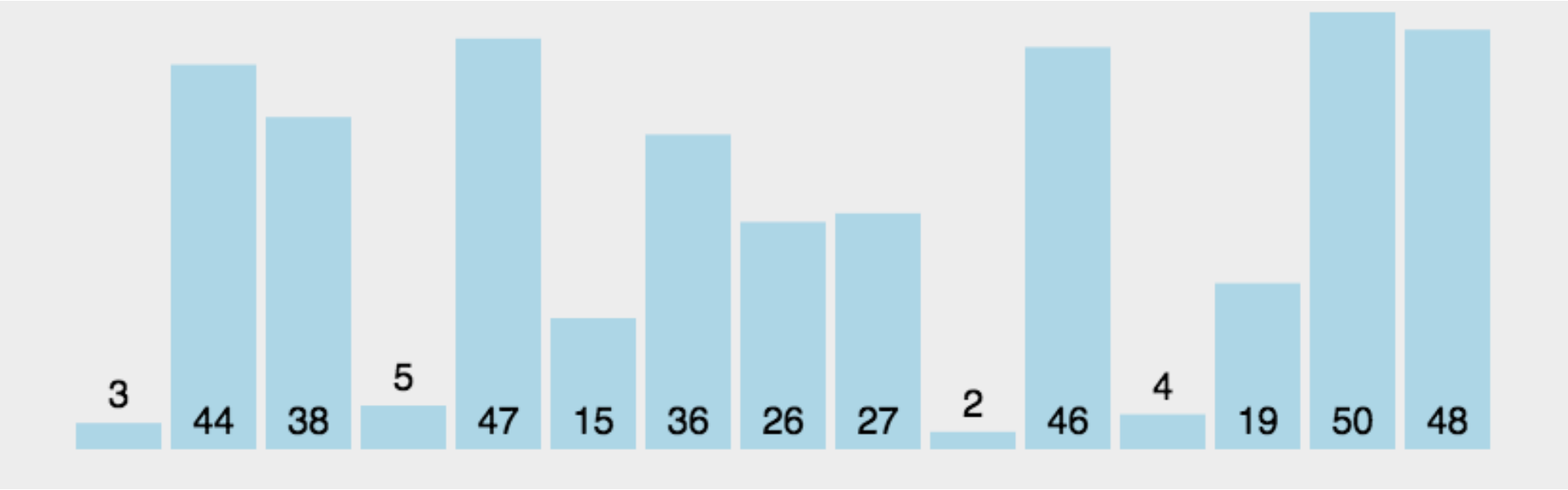
排序(Sorting)

选择排序



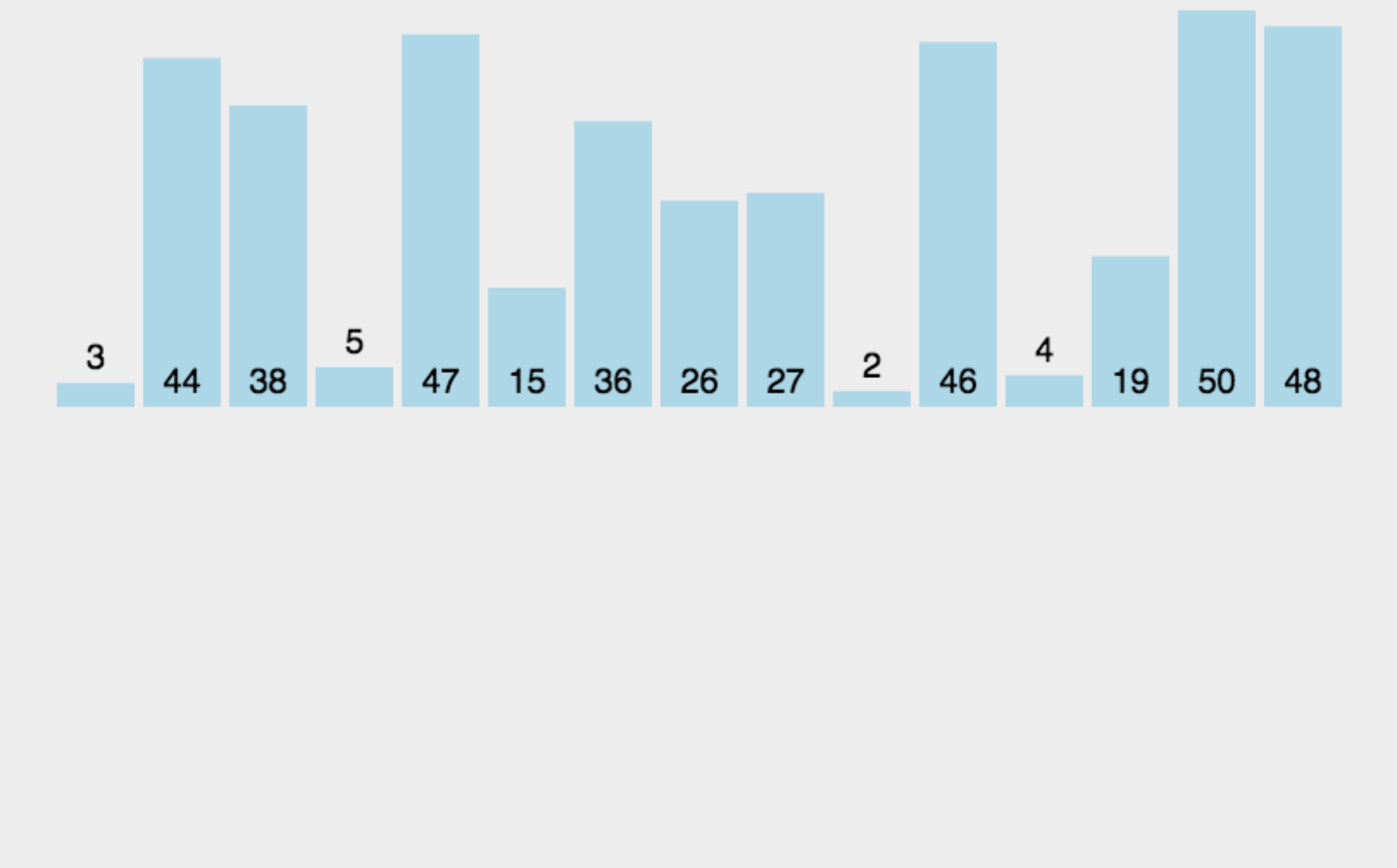
排序(Sorting)

冒泡排序



排序(Sorting)

插入排序



快速排序(Quicksort)

诞生于1960年代的最好的排序算法

```
void quick_sort(int v[], int n){
    if(n <= 1)
        return;
    swap(v, 0, rand() % n);
    int last = 0;

    for(int i = 1; i < n; i++) {
        if(v[i] < v[0])
            swap(v, ++last, i);
    }
    swap(v, 0, last);

    quick_sort(v, last);
    quick_sort(v + last + 1, n - last - 1);
}
```

如果元素个数小于等于1，则不用排了

随机取数组中间一个元素，作为支点元素，暂且和第0个元素交换

last以及之前的元素，都确定比支点元素要小，last暂且初值为0

从1到最后扫描一遍，找出所有比支点元素小的，排到++last那里去

把此前暂且交换到开头的支点元素与last交换，至此支点元素之前的元素，都比它小，之后的都比它大(或者等于)。以支点元素为界，数组被拆分成两部分

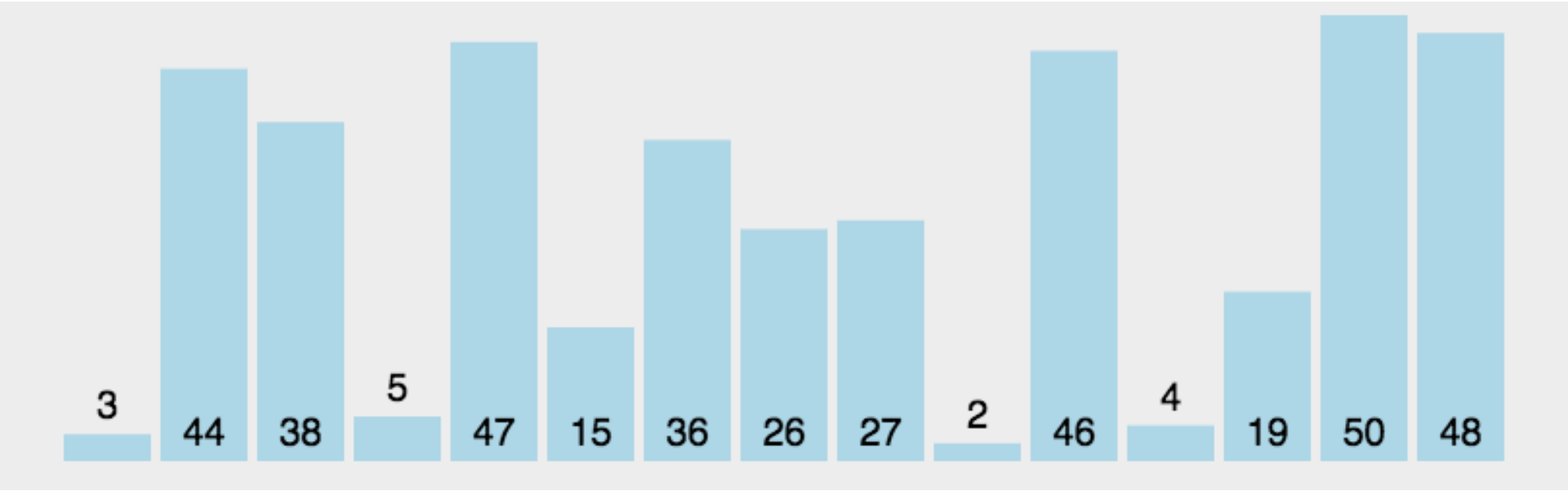
递归调用，对前半部分排序

递归调用，对后半部分排序

```
void swap(int v[], int i, int j){
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

排序(Sorting)

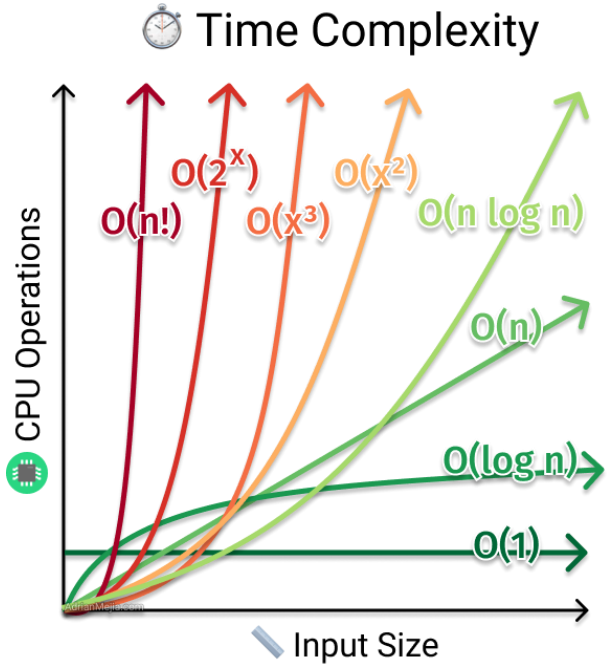
快速排序



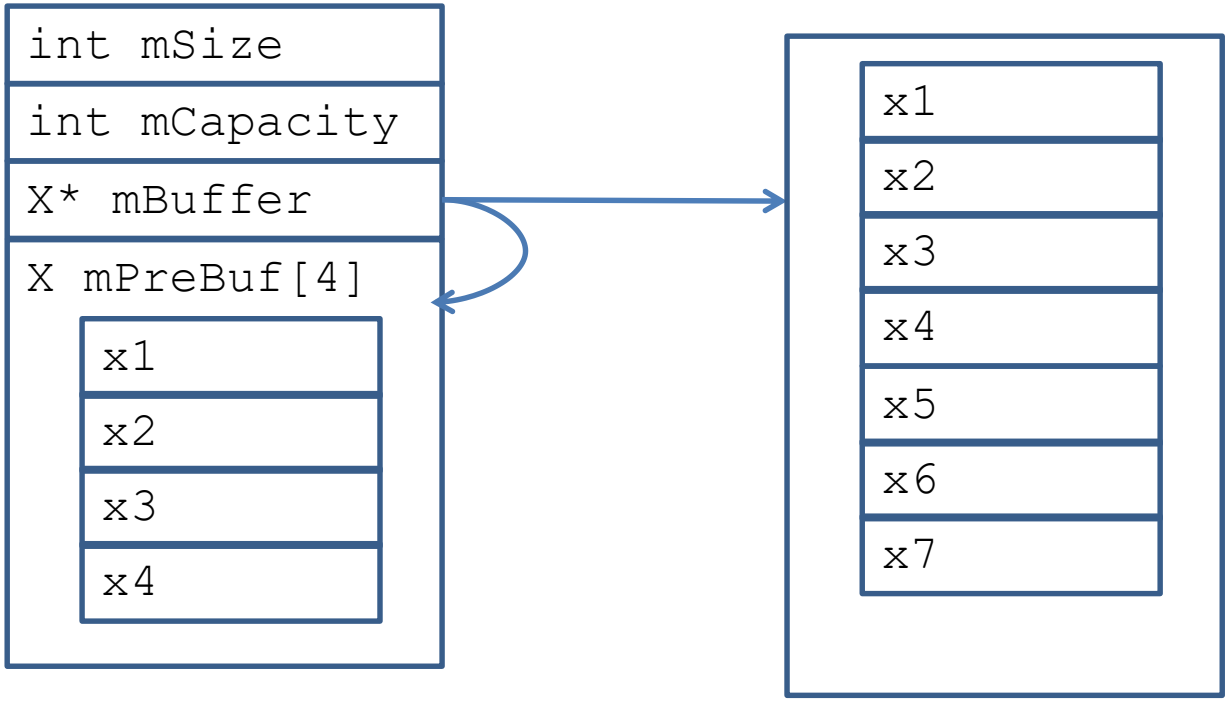
棕色：已确定排序位置的元素
黄色：当前正在处理的子序列的支点元素
红色：正在处理的元素
蓝色：子序列中尚未与支点元素比较的元素
绿色：子序列中已与支点比过，并且值小于支点的元素
紫色：子序列中已与支点比过，并且值大于支点的元素

算法复杂度大O记法

记法	名称	举例
O(1)	常数(constant)	数组下标访问
O(logn)	对数(logarithmic)	二分查找
O(n)	线性(linear)	顺序查找
O(nlogn)	nlogn	快速排序
O(n²)	平方(quadratic)	简单排序
O(n³)	立方(cubic)	矩阵乘法
O(2^n)	指数(exponential)	集合划分



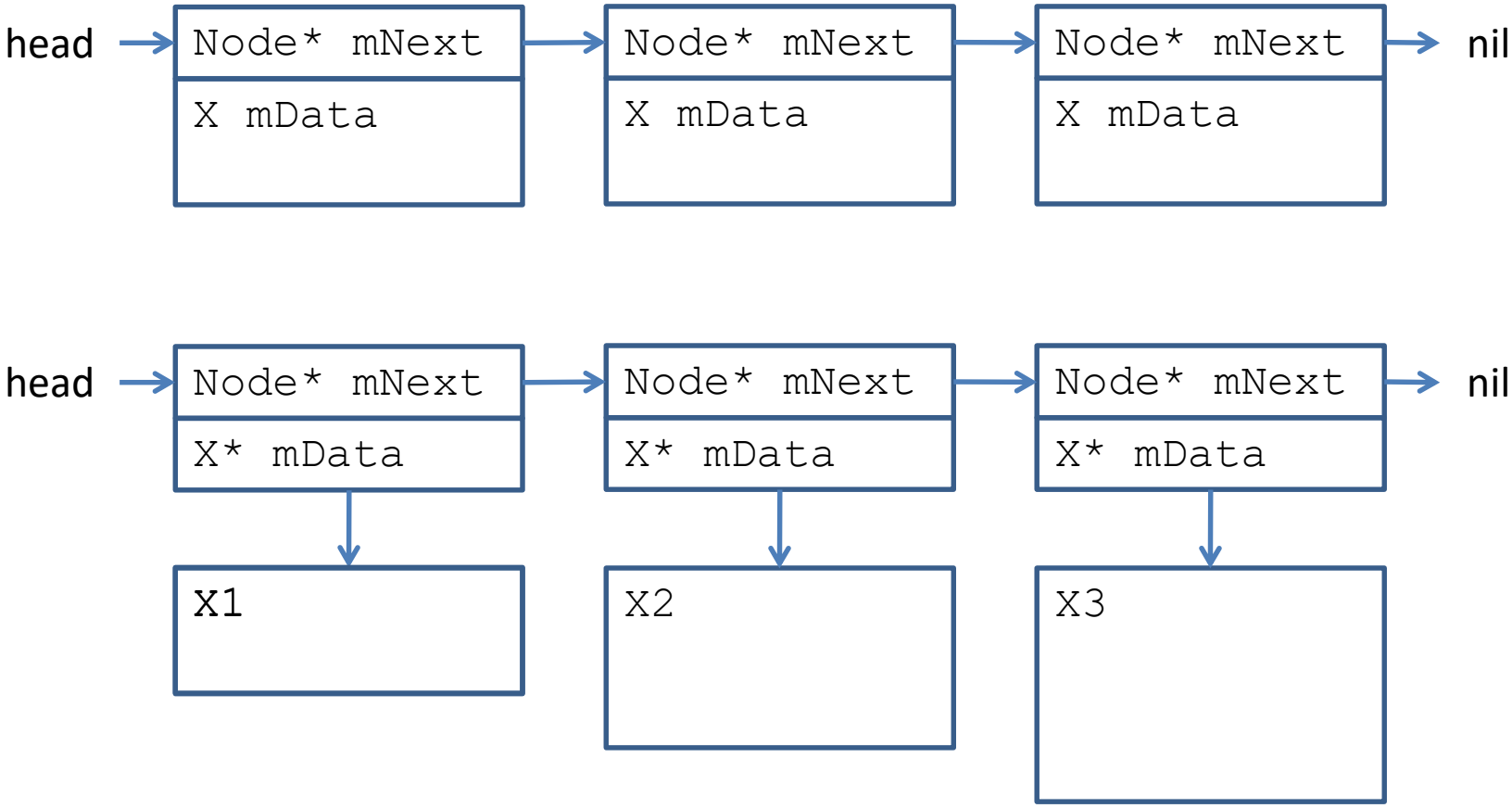
可增长数组(Growing Array)



库：

1. C++：std::string
2. Java：String

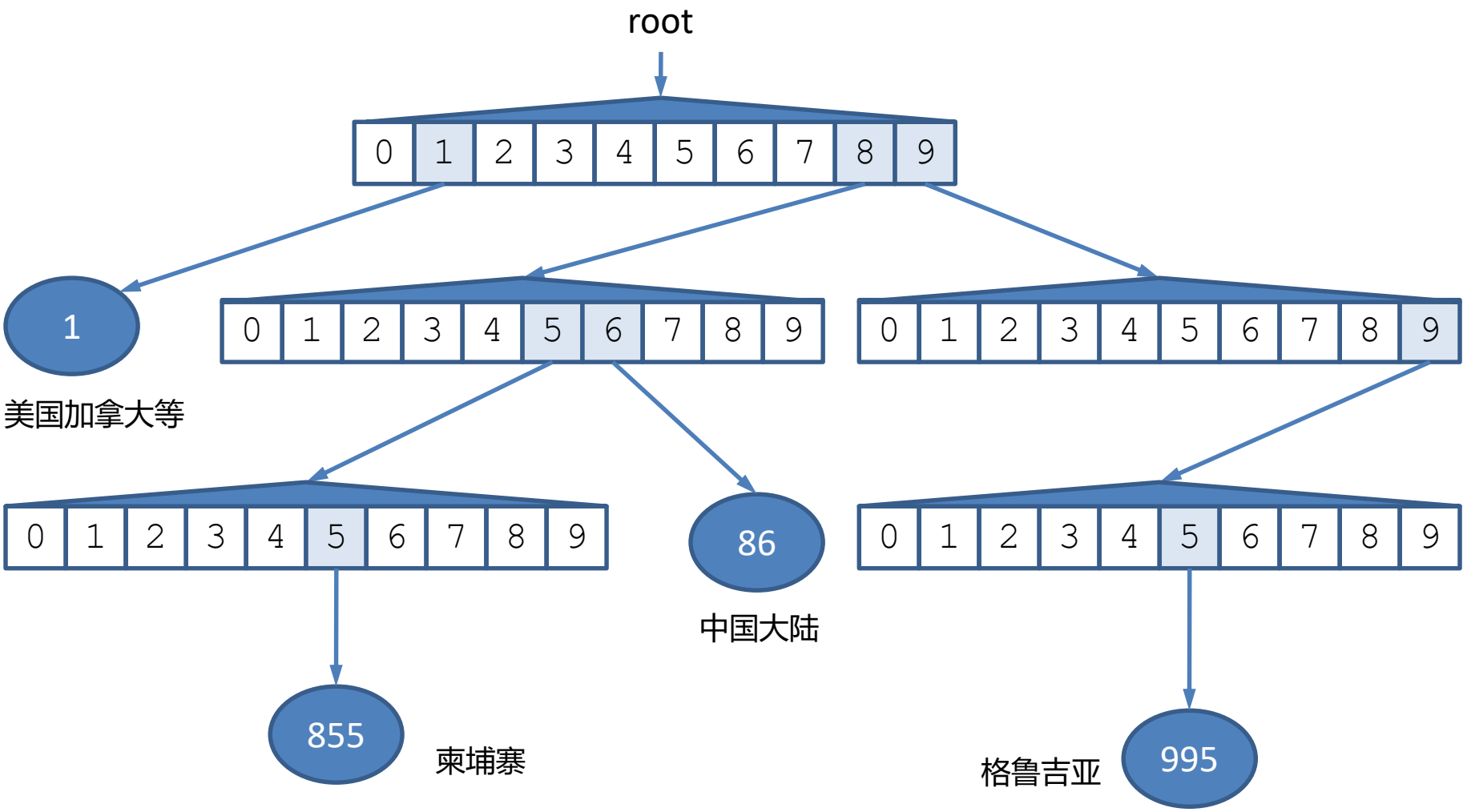
链表(List)



C++标准库：std::list

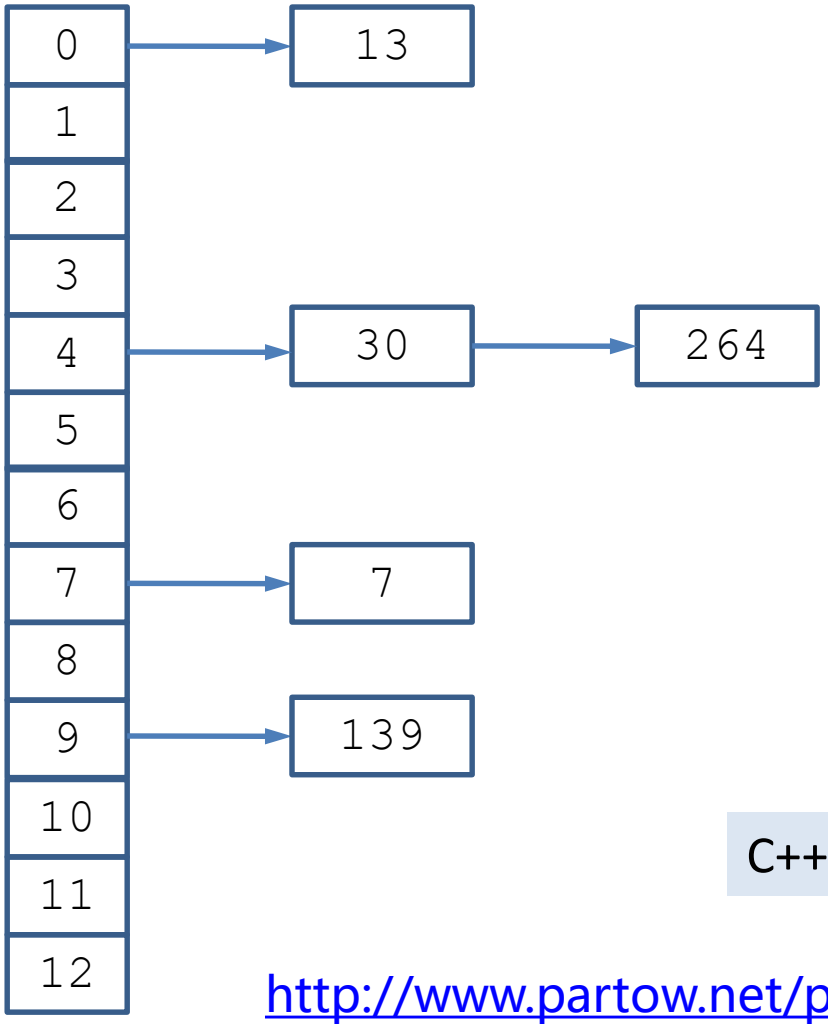
树

问题：如何从一个带地区号的电话号码中求出地区号？
（例如12024952266，861062282297，9952259000）



散列表(Hash Table)

计算机科学领域伟大的发明之一
应用最广泛的一个数据结构



索引：整数
散列数组大小：13
散列算法：(n % 13)

索引：字符串
散列数组大小：？
散列算法：？

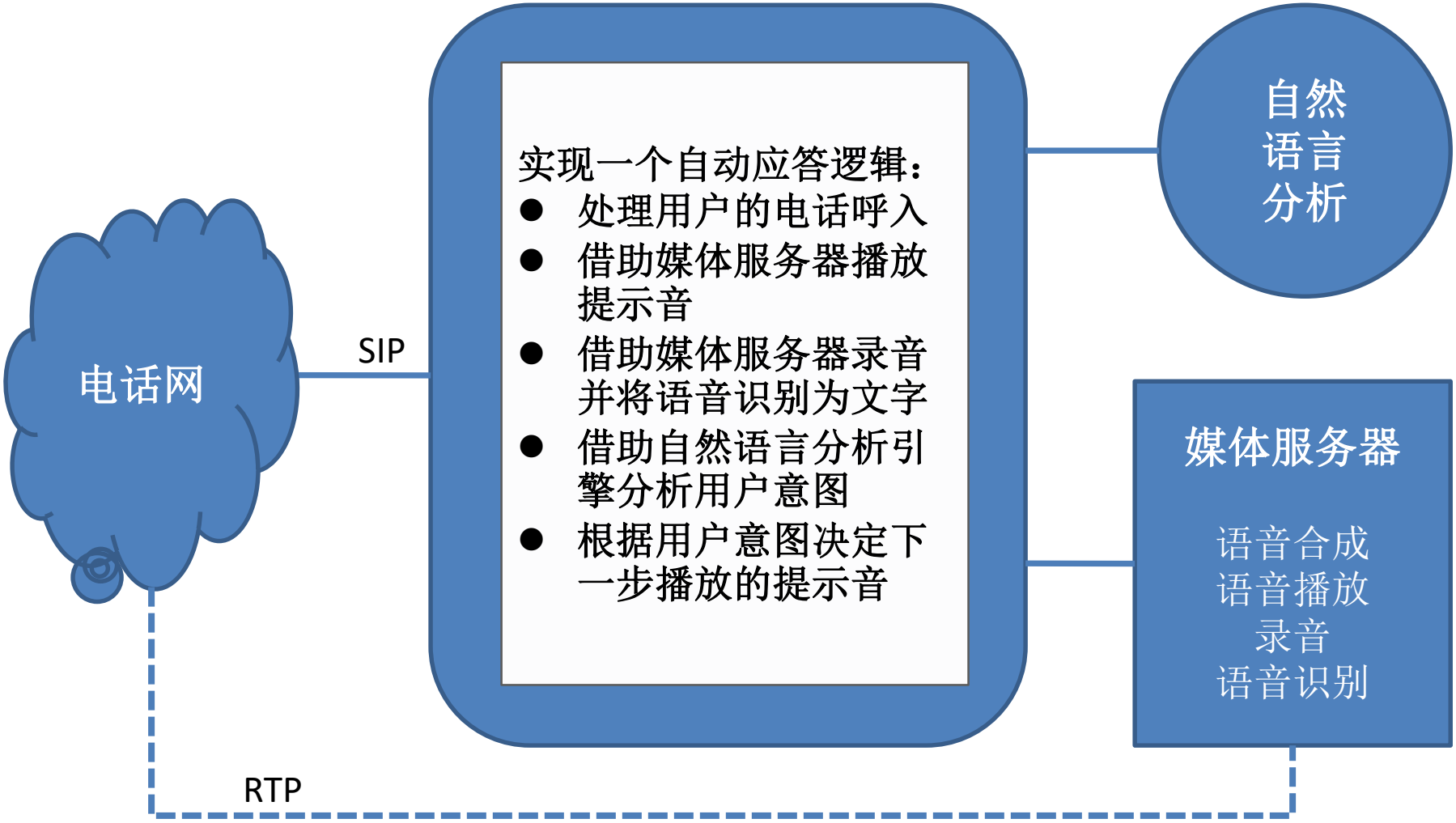
C++标准库：std::hash_map

http://www.partow.net/programming/hashfunctions/
这个网站有很多散列算法的例子

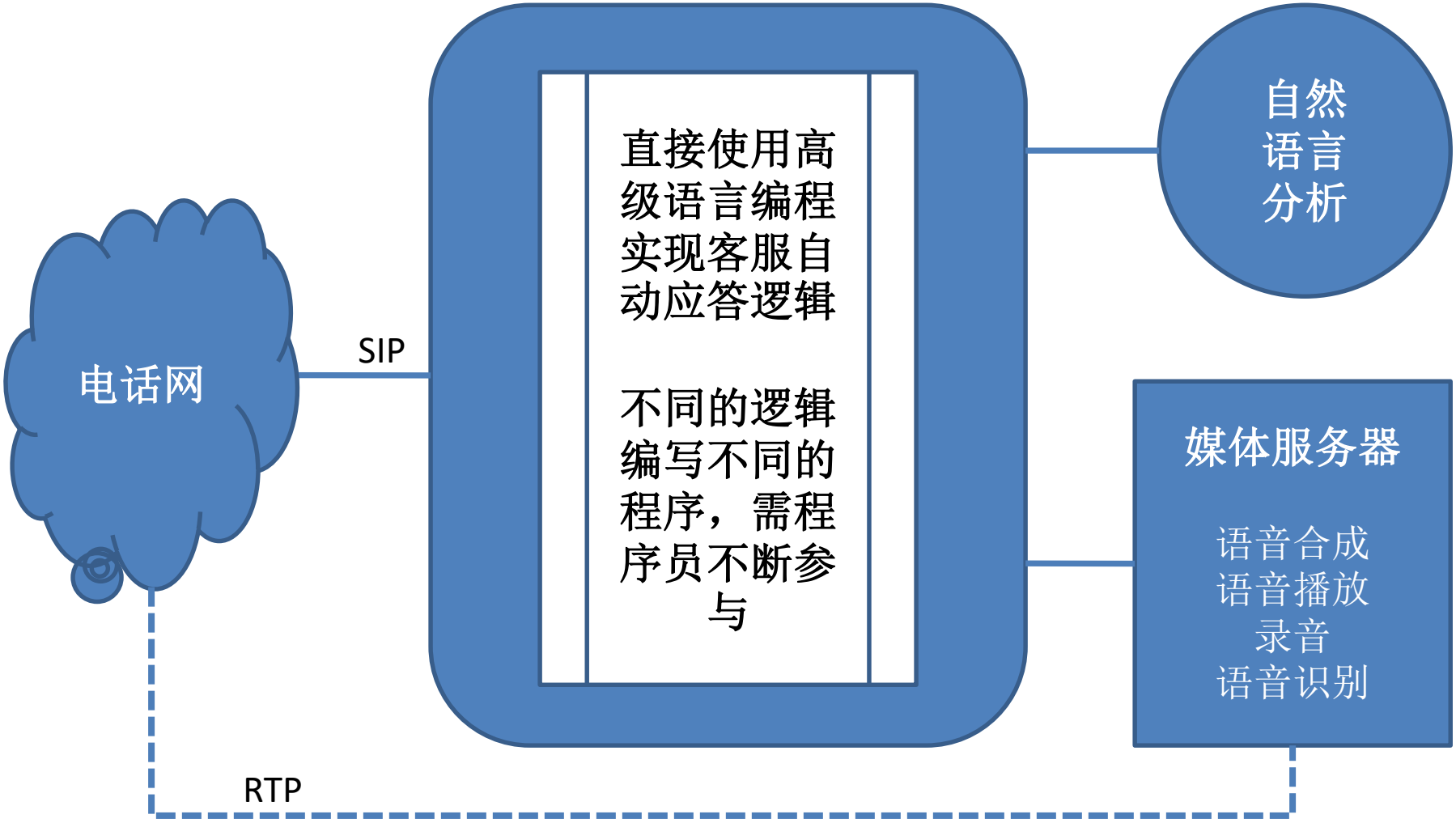
设计实现一个脚本解释器

需求

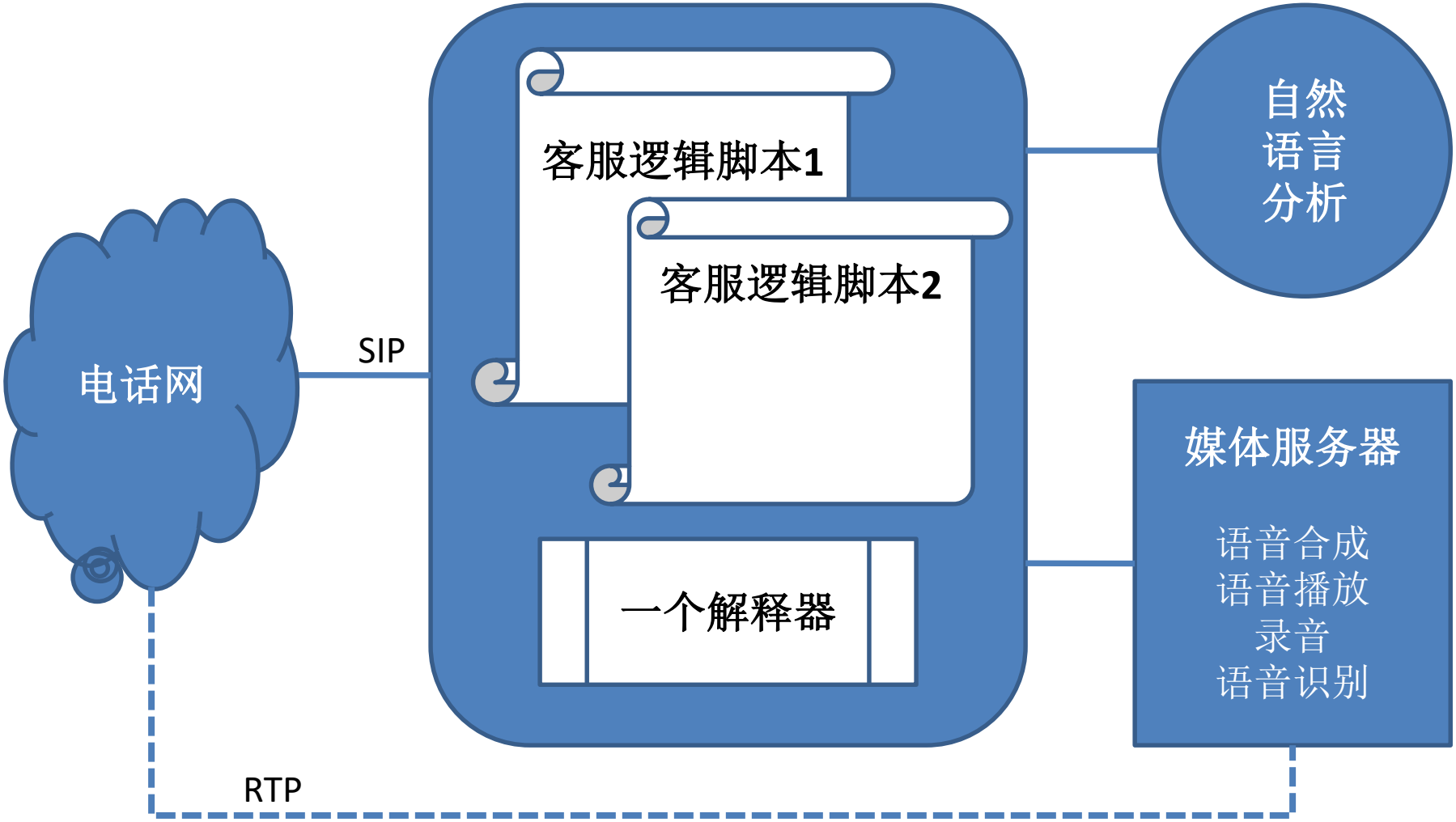
实现一个语音客服机器人



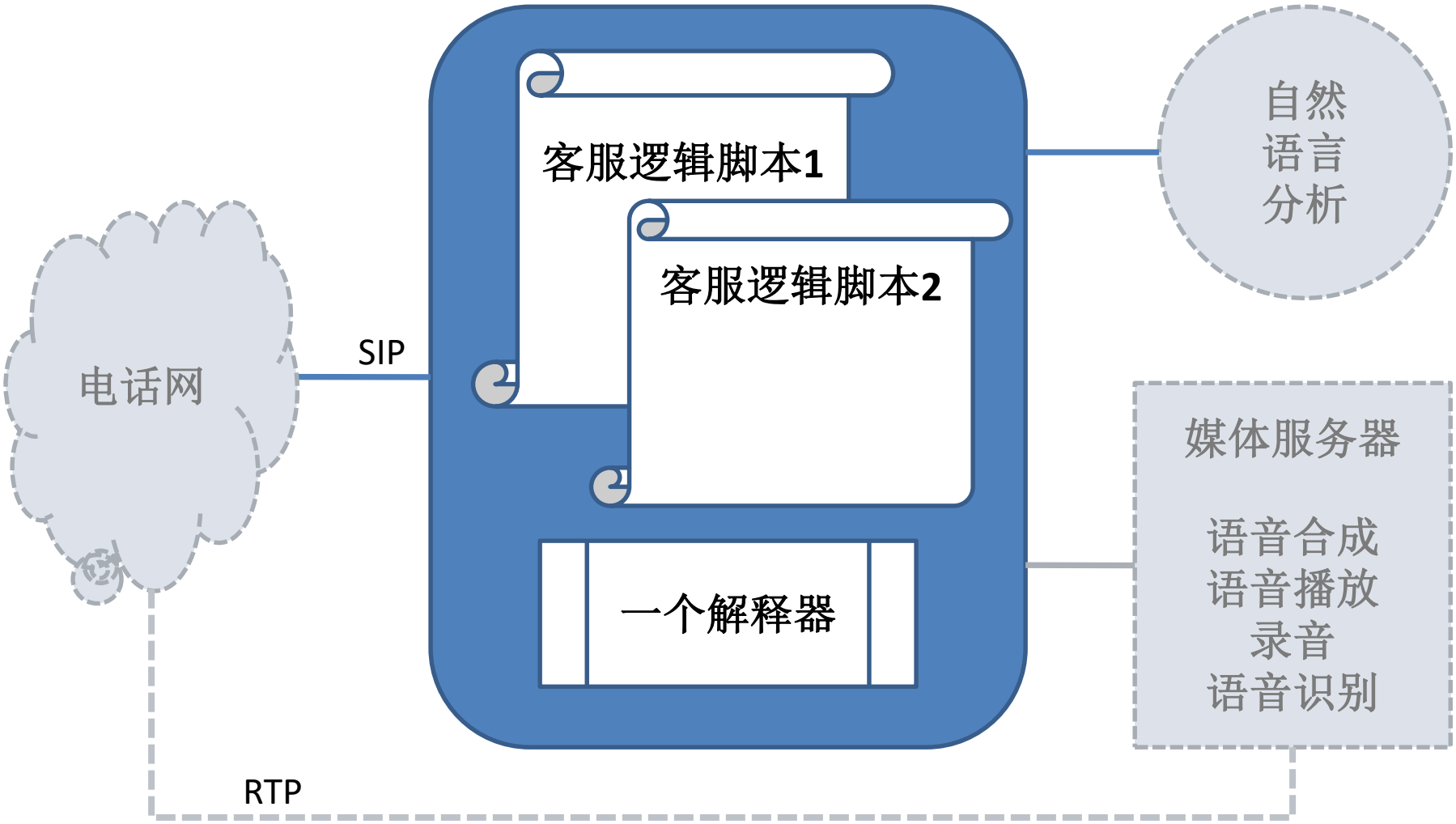
方案1


直接使用高级语言编程实现客服自动应答逻辑
不同的逻辑编写不同的程序，需程序员不断参与
自然语言分析
媒体服务器
语音合成
语音播放
录音
语音识别
电话网
SIP
RTP

方案2


客服逻辑脚本1
客服逻辑脚本2
一个解释器
自然
语言
分析
电话网
SIP
媒体服务器
语音合成
语音播放
录音
语音识别
RTP

我们要做的部分


客服逻辑脚本1
客服逻辑脚本2
一个解释器
自然
语言
分析
电话网
SIP
媒体服务器
语音合成
语音播放
录音
语音识别
RTP

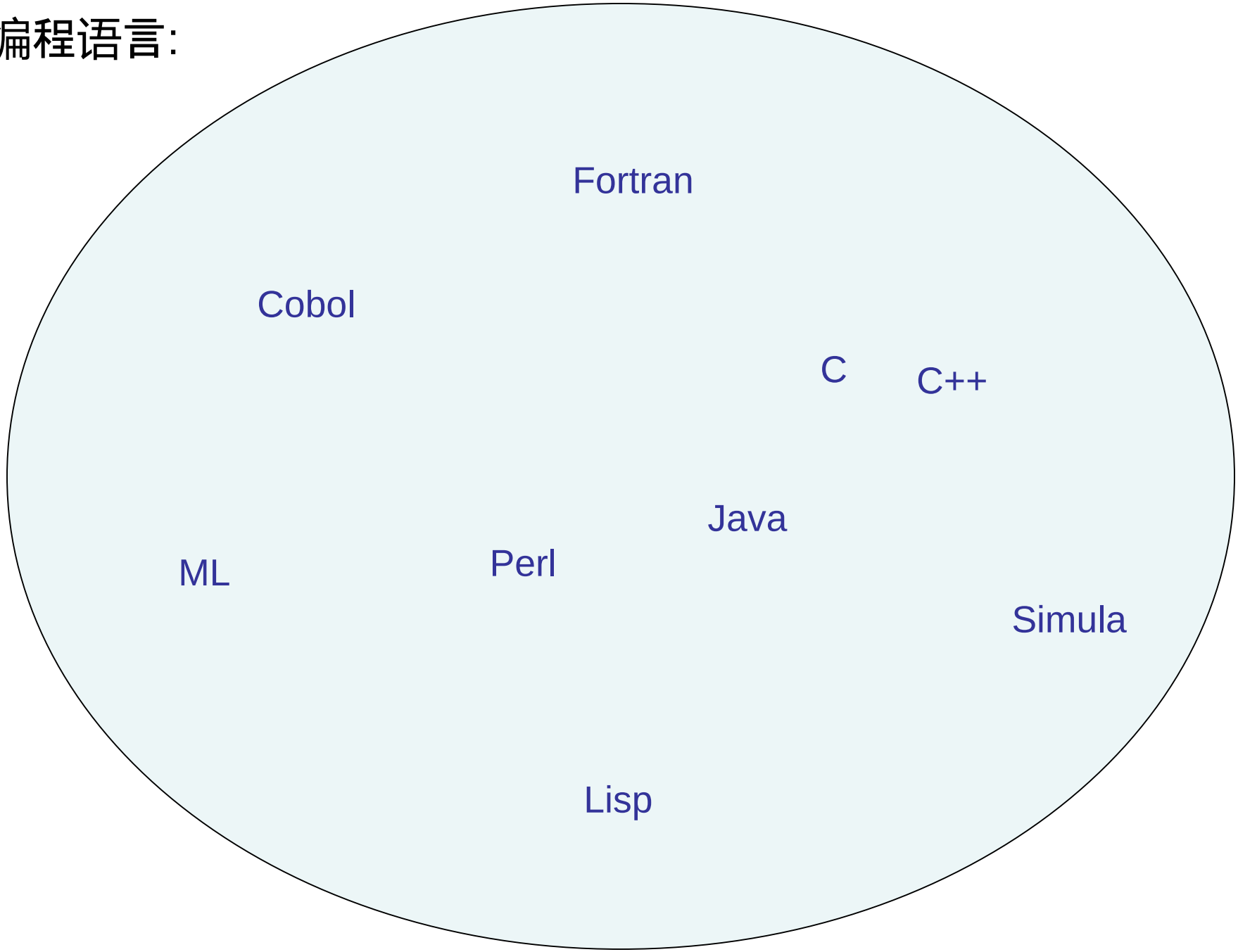
领域专用语言

Domain-Specific Languages

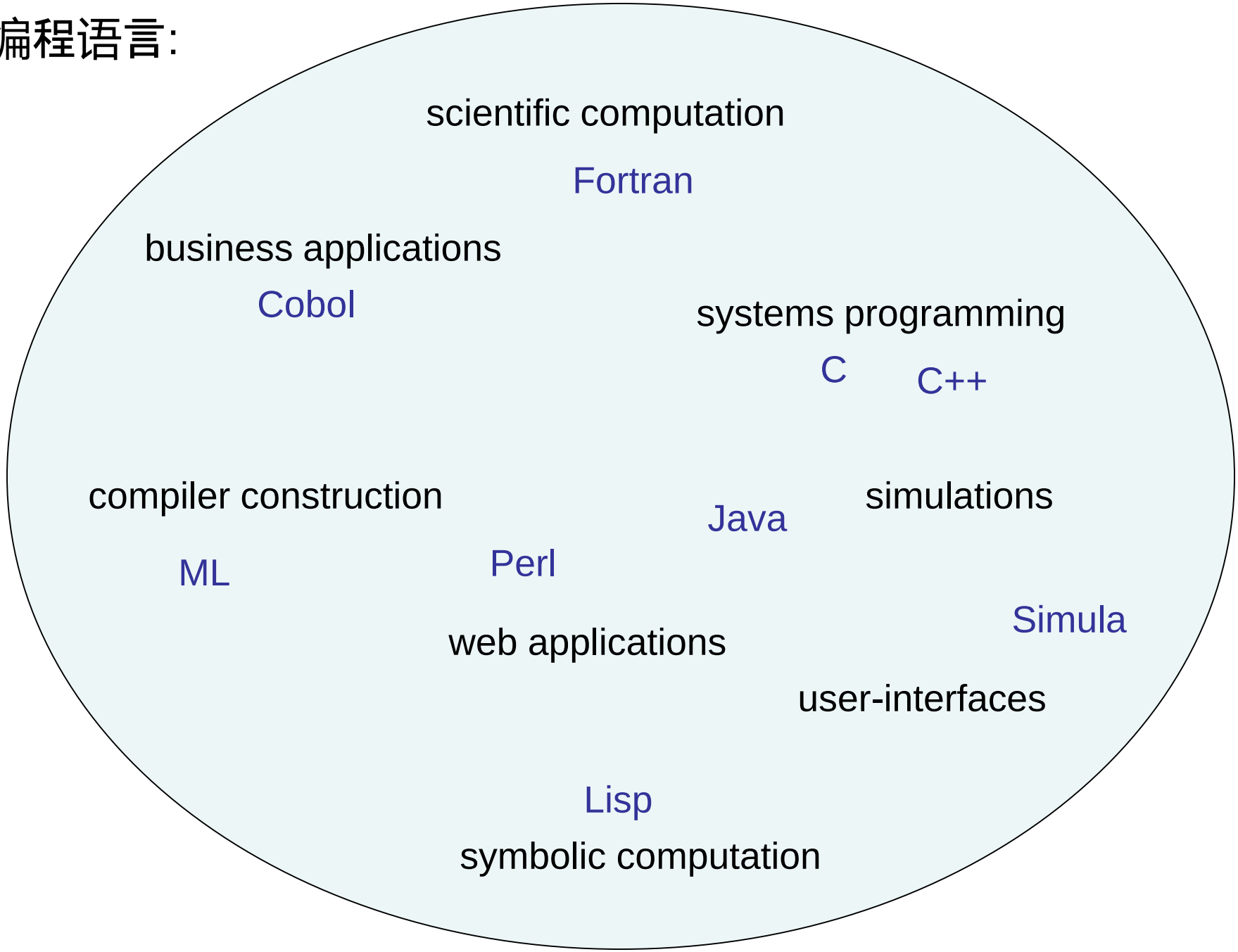
DSL

简介

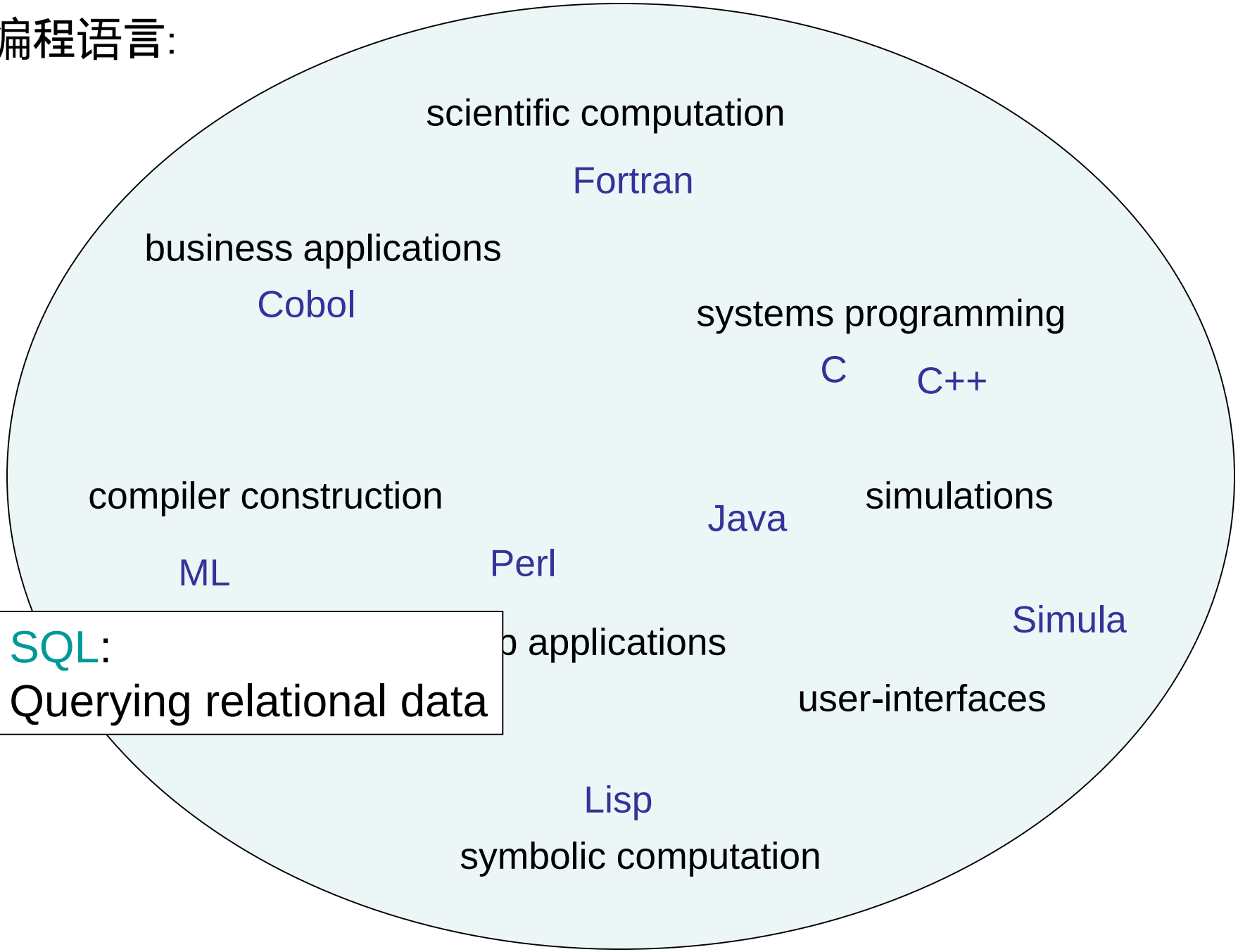
编程语言:



编程语言:

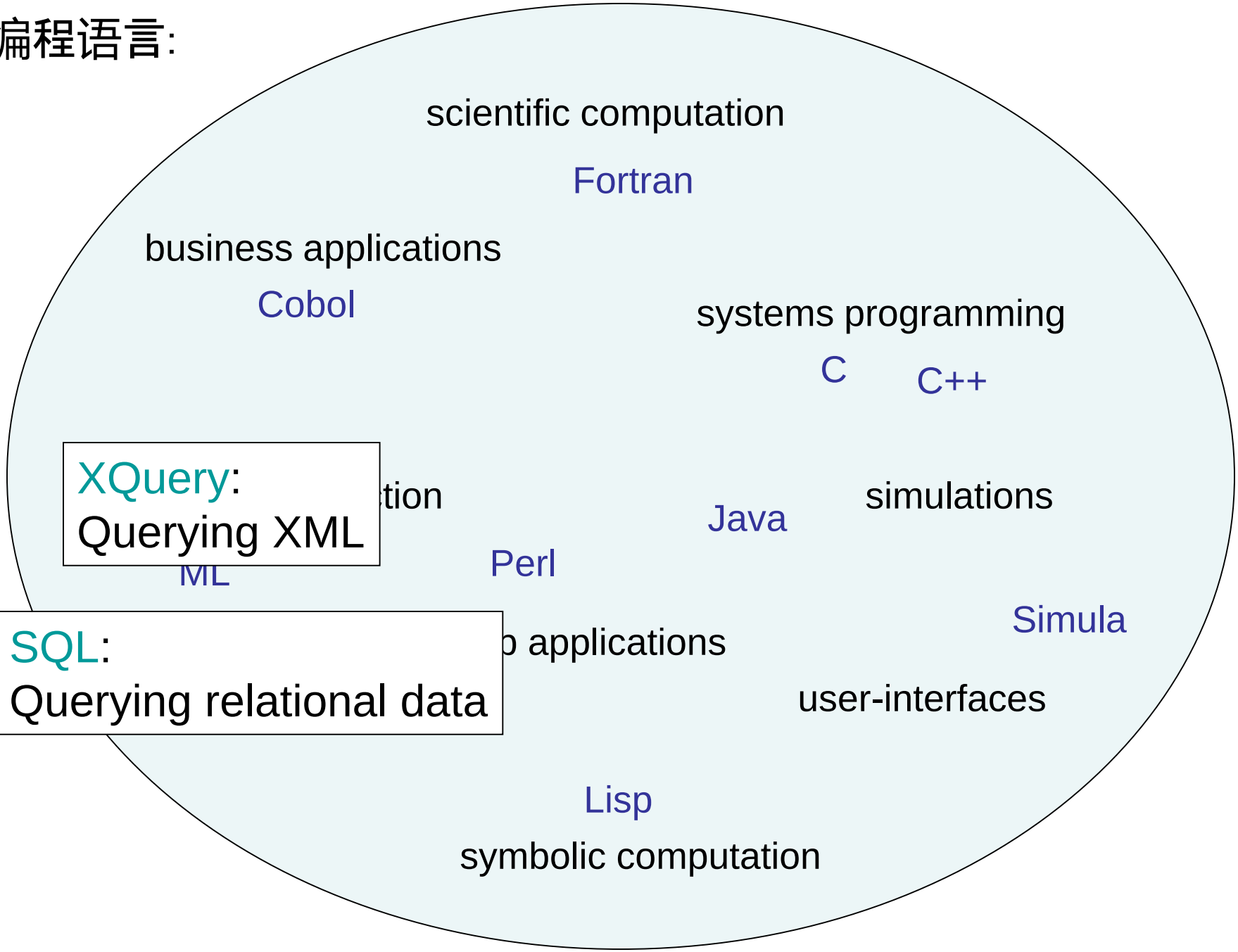

scientific computation
Fortran
business applications
Cobol
systems programming
C
C++
compiler construction
Java
simulations
ML
Perl
Simula
web applications
user-interfaces
Lisp
symbolic computation

编程语言:

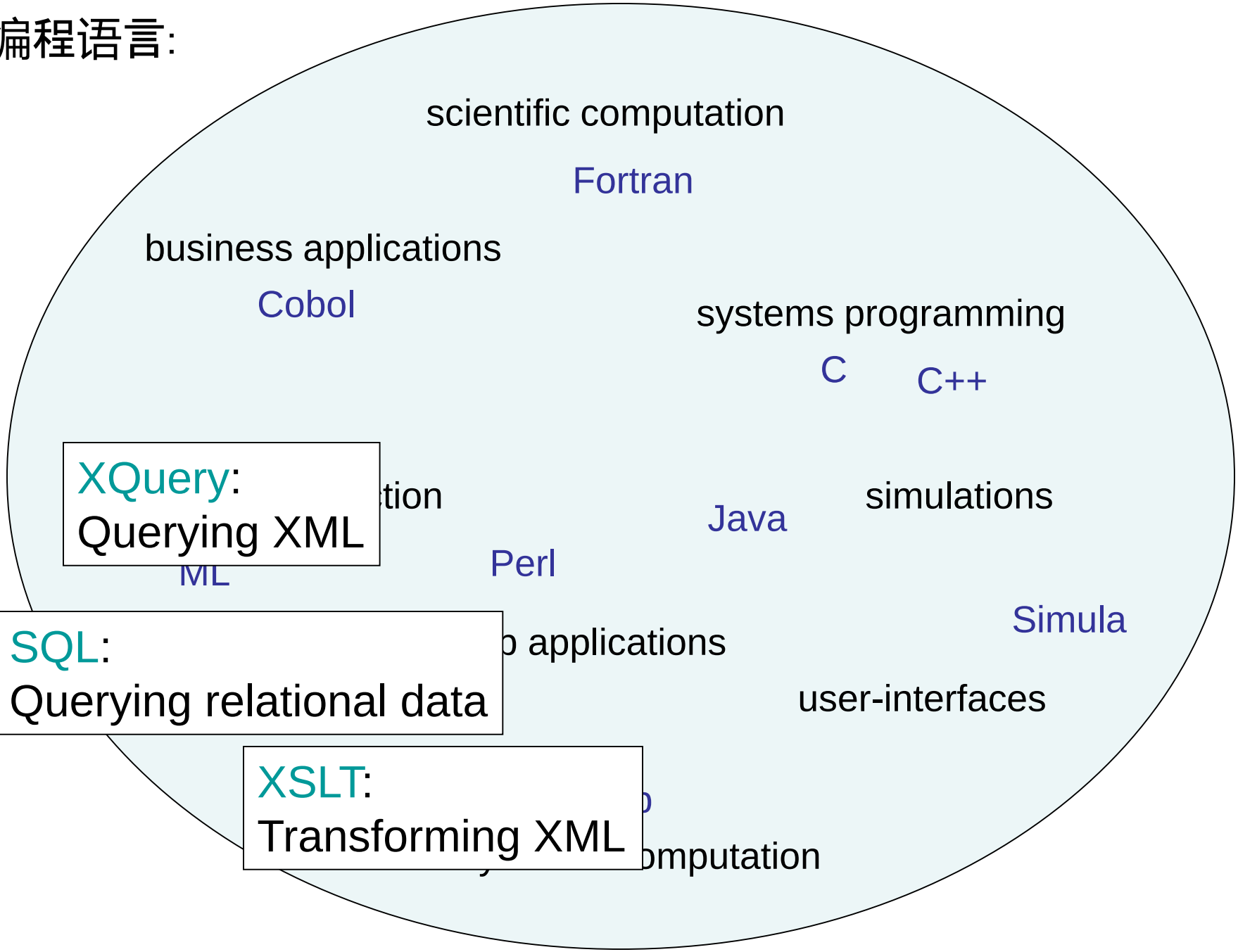


SQL:
Querying relational data

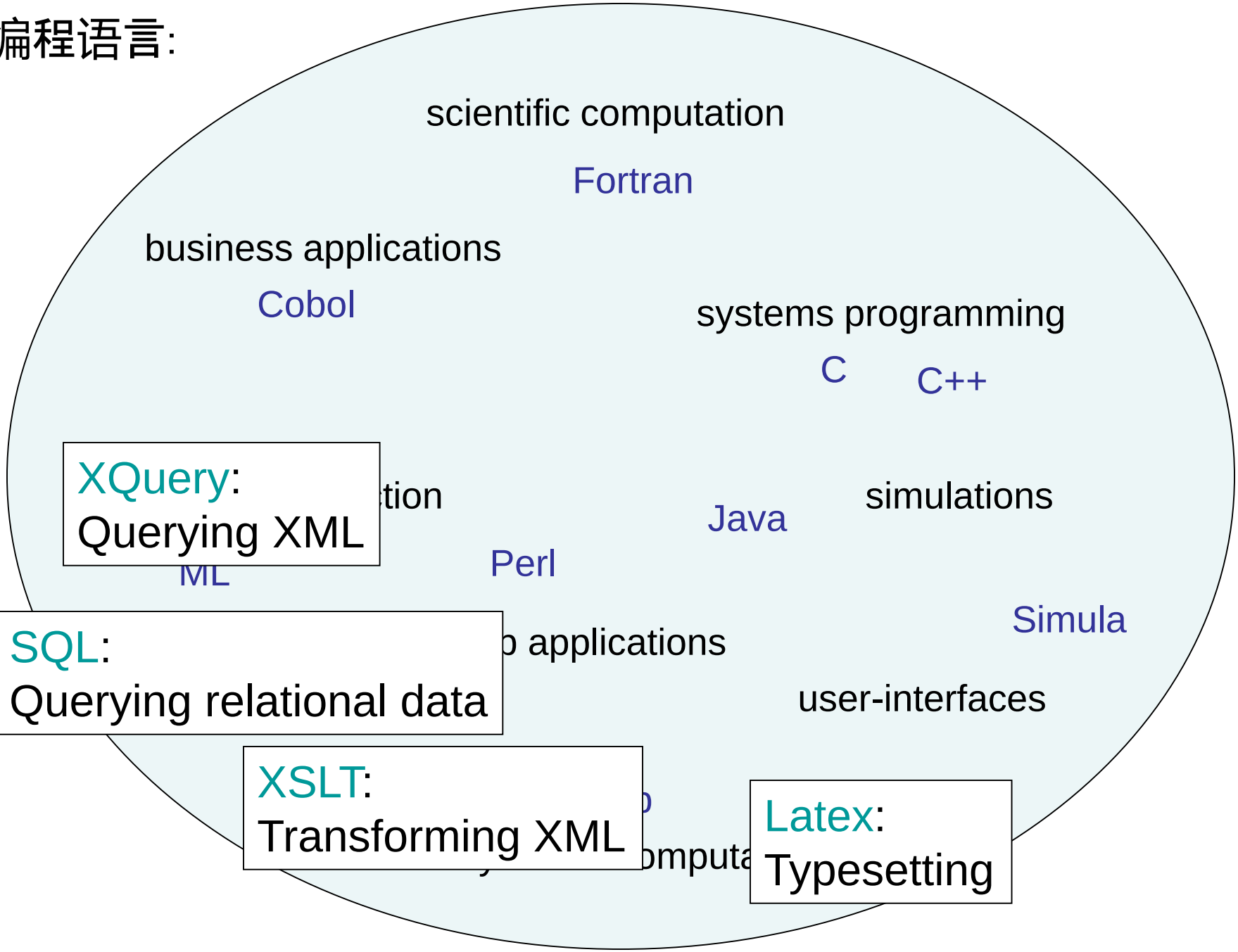
编程语言:


scientific computation
Fortran
business applications
Cobol
systems programming
C
C++
XQuery:
Querying XML
tion
simulations
Java
Perl
ML
SQL:
Querying relational data
o applications
Simula
user-interfaces
Lisp
symbolic computation

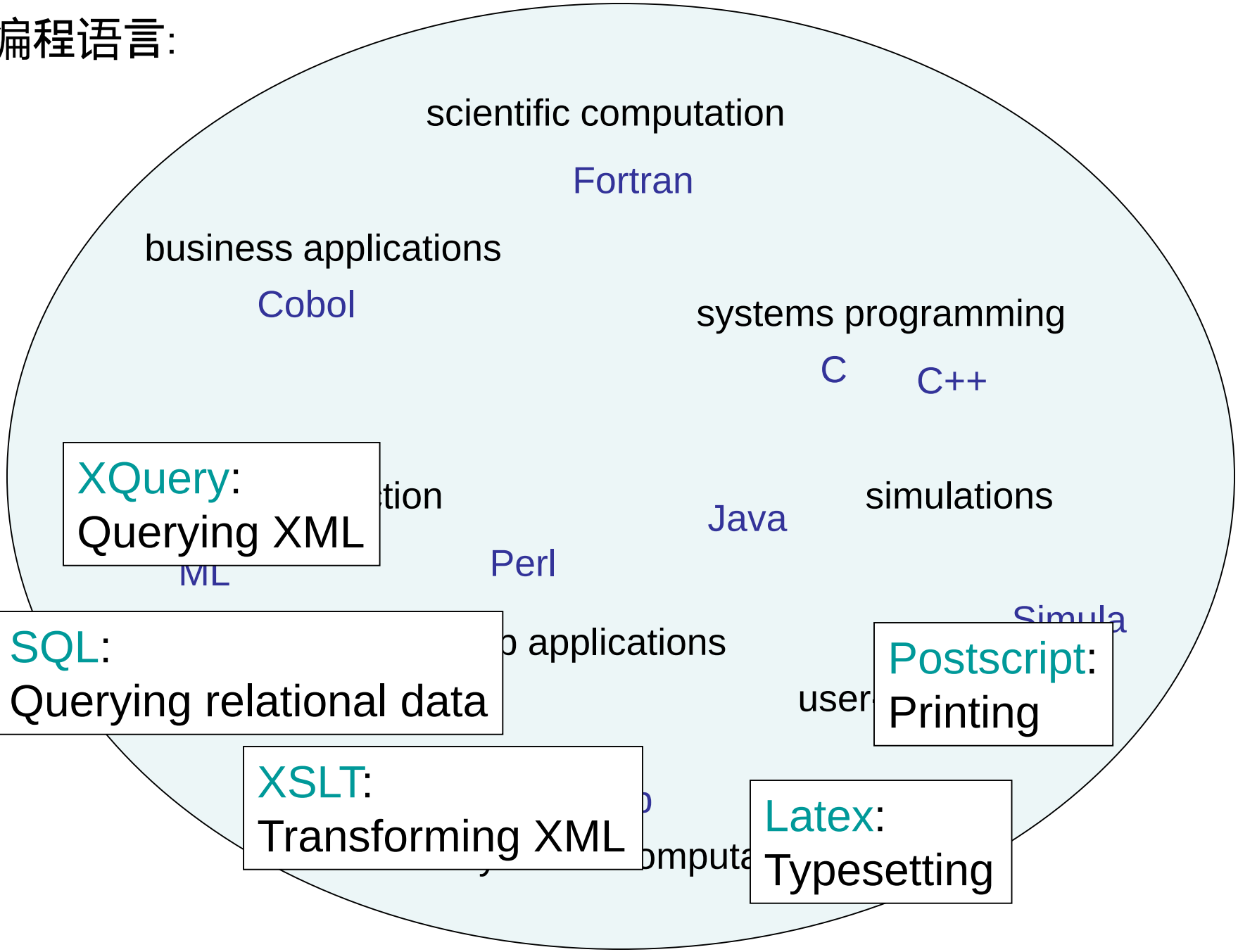
编程语言:


scientific computation
Fortran
business applications
Cobol
systems programming
C
C++
XQuery:
Querying XML
tion
simulations
Java
ML
Perl
Simula
SQL:
Querying relational data
o applications
user-interfaces
XSLT:
Transforming XML
omputation

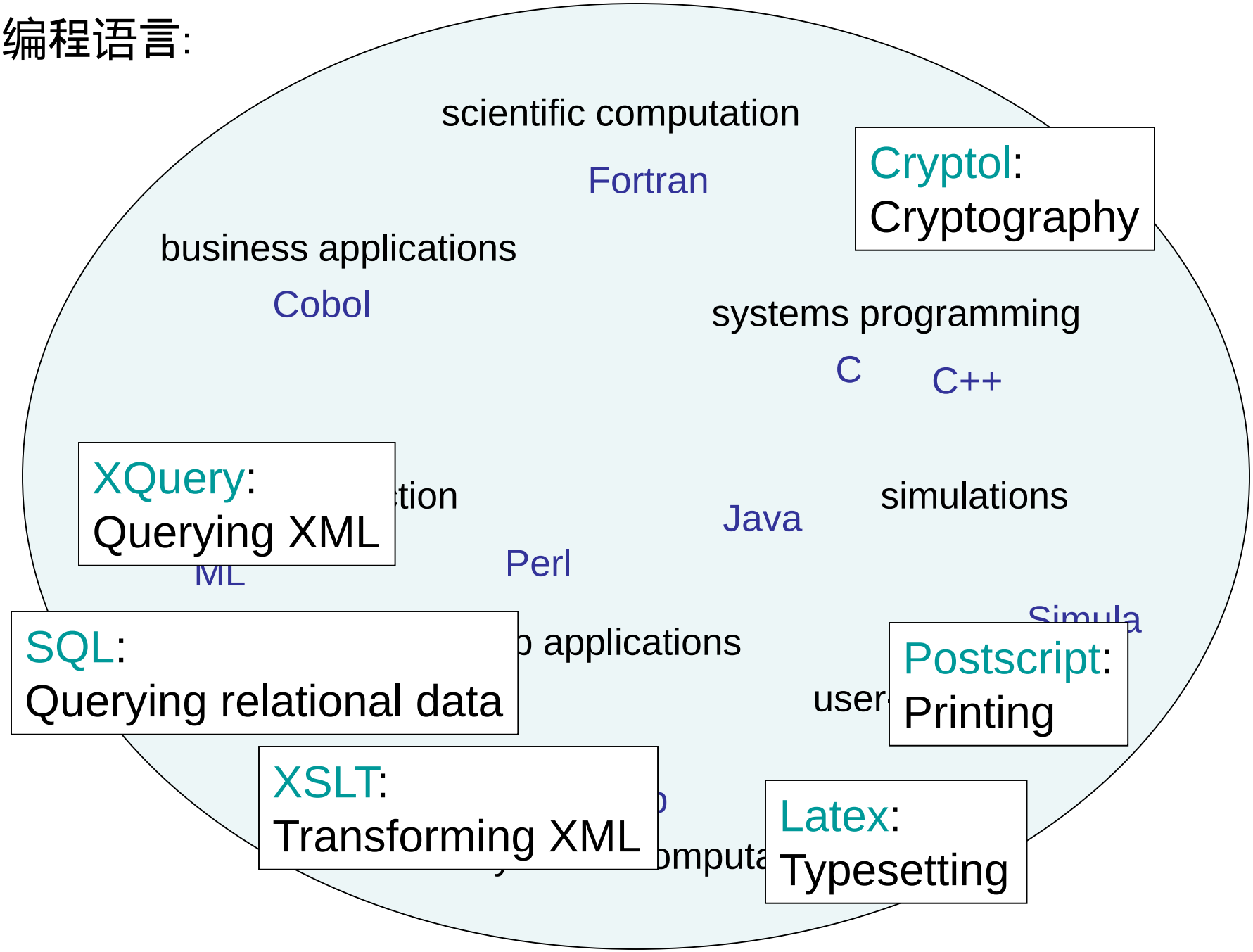
编程语言:


scientific computation
Fortran
business applications
Cobol
systems programming
C
C++
XQuery:
Querying XML
tion
simulations
Java
ML
Perl
Simula
SQL:
Querying relational data
o applications
user-interfaces
XSLT:
Transforming XML
Latex:
Typesetting

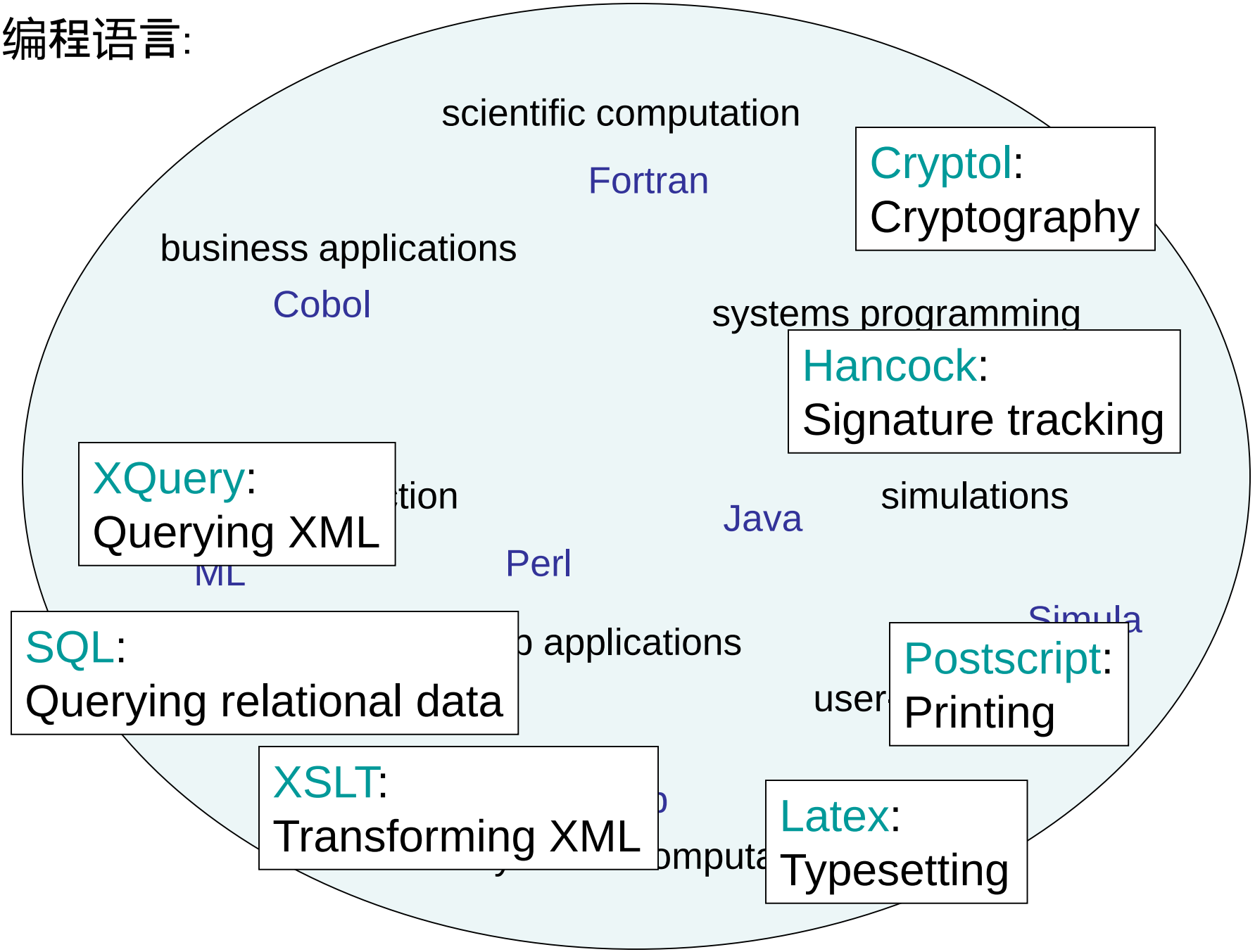
编程语言:


scientific computation
Fortran
business applications
Cobol
systems programming
C
C++
XQuery:
Querying XML
simulations
Java
Perl
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

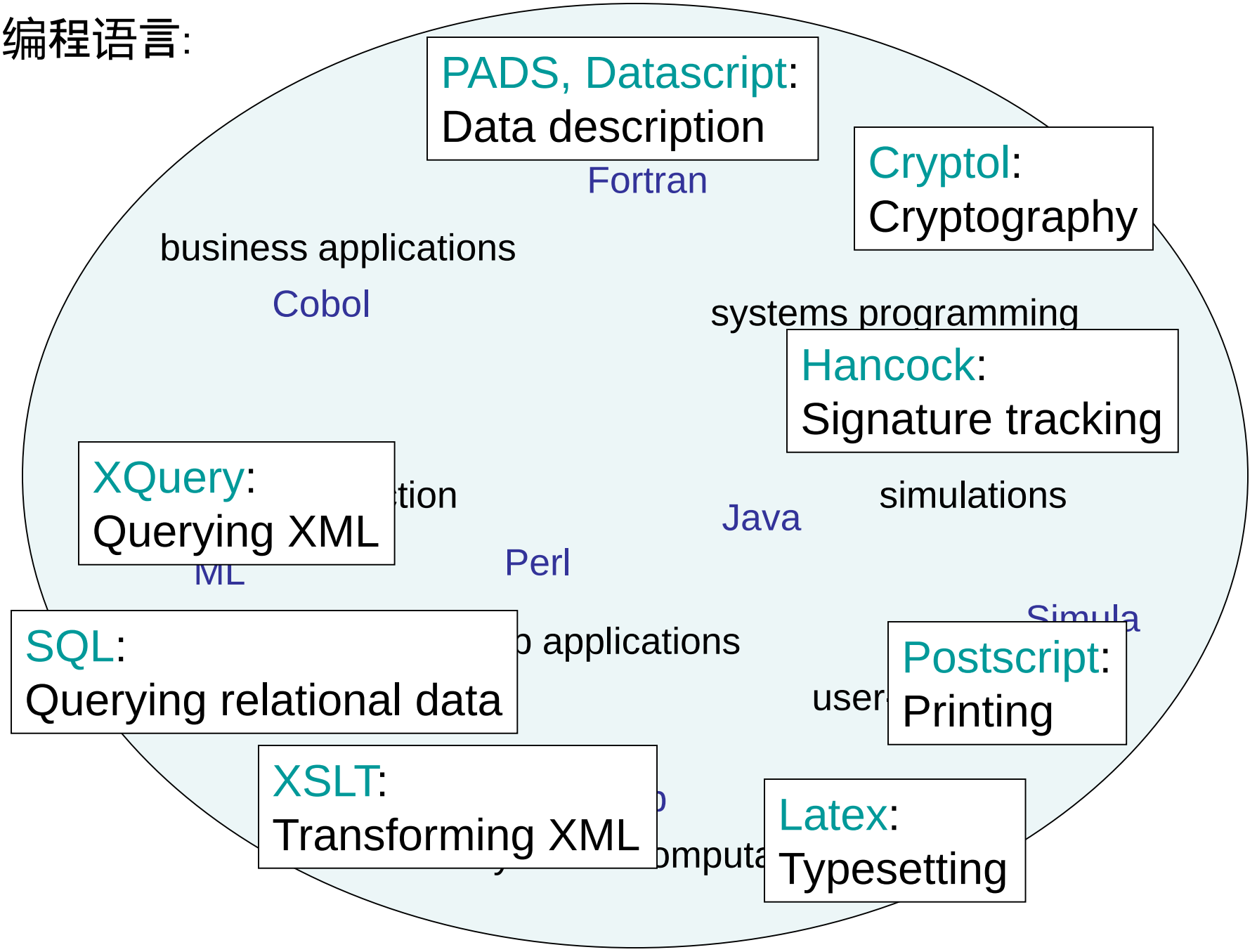
编程语言:


scientific computation
Fortran
Cryptol:
Cryptography
business applications
Cobol
systems programming
C
C++
XQuery:
Querying XML
simulations
Java
Perl
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

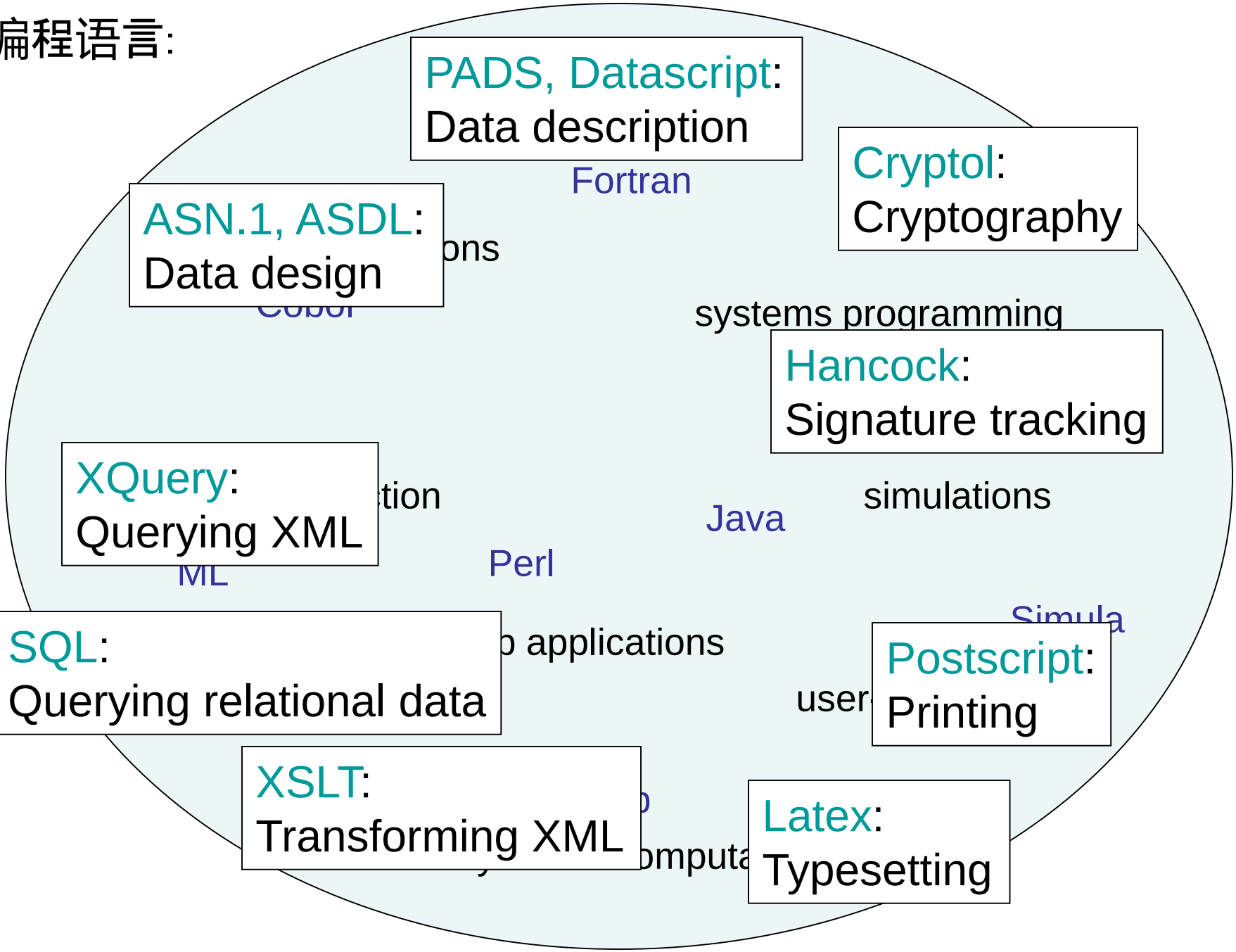
编程语言:


scientific computation
Fortran
Cryptol:
Cryptography
business applications
Cobol
systems programming
Hancock:
Signature tracking
XQuery:
Querying XML
simulations
Java
Perl
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

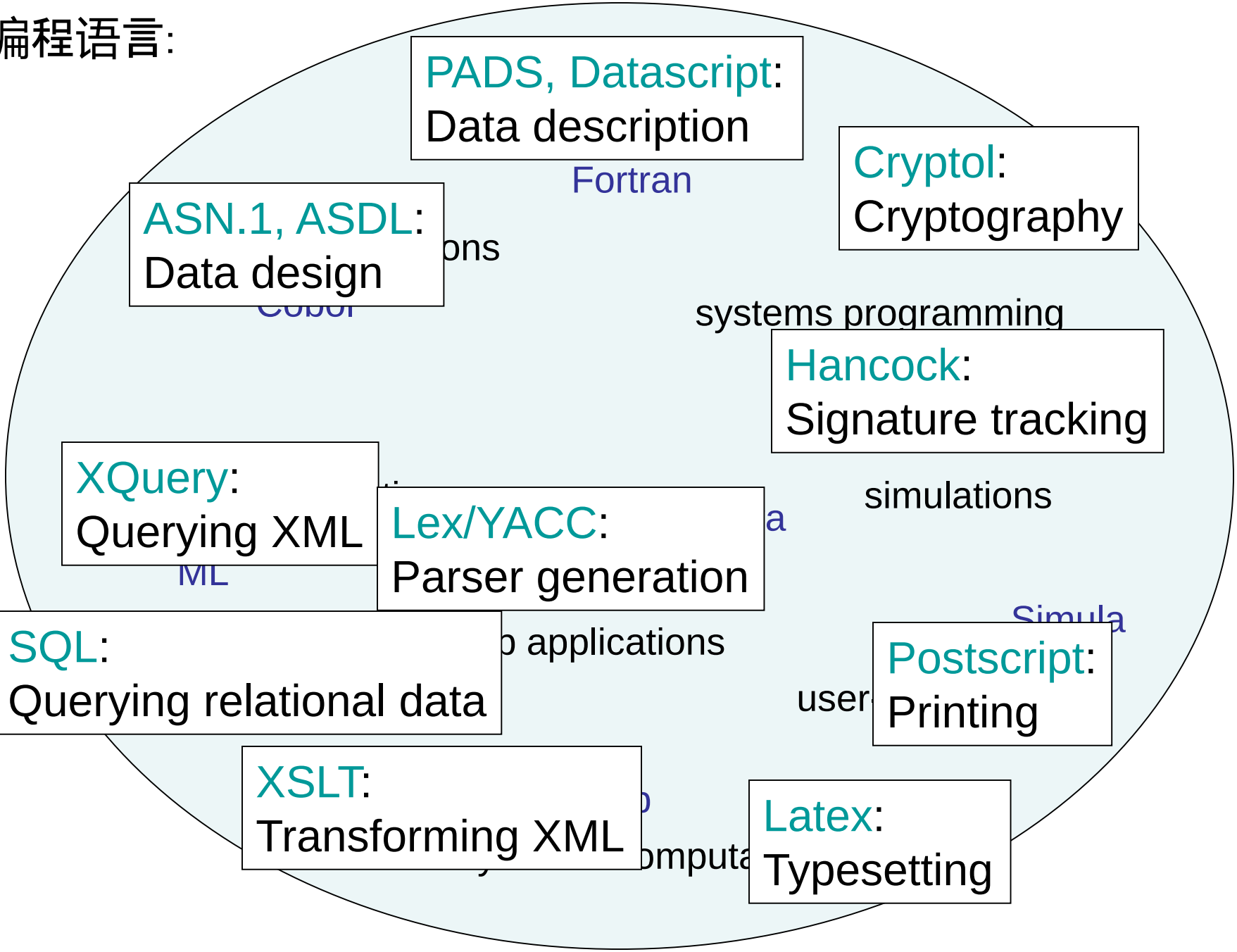
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
business applications
Cobol
systems programming
Hancock:
Signature tracking
XQuery:
Querying XML
simulations
Java
Perl
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

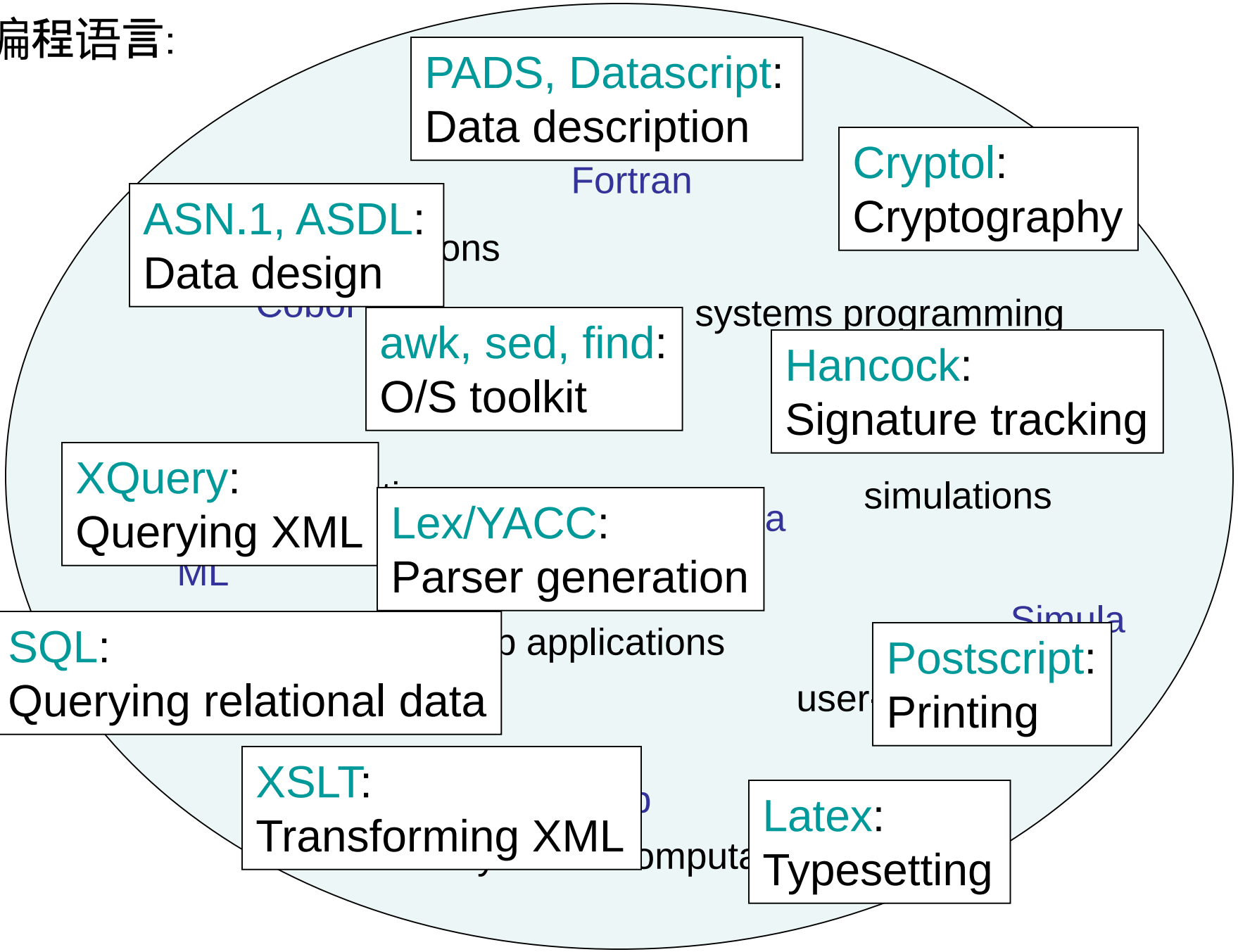
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
ons
systems programming
Hancock:
Signature tracking
XQuery:
Querying XML
tion
simulations
Java
Perl
ML
SQL:
Querying relational data
o applications
Simula
Postscript:
Printing
user
XSLT:
Transforming XML
Latex:
Typesetting
omputa

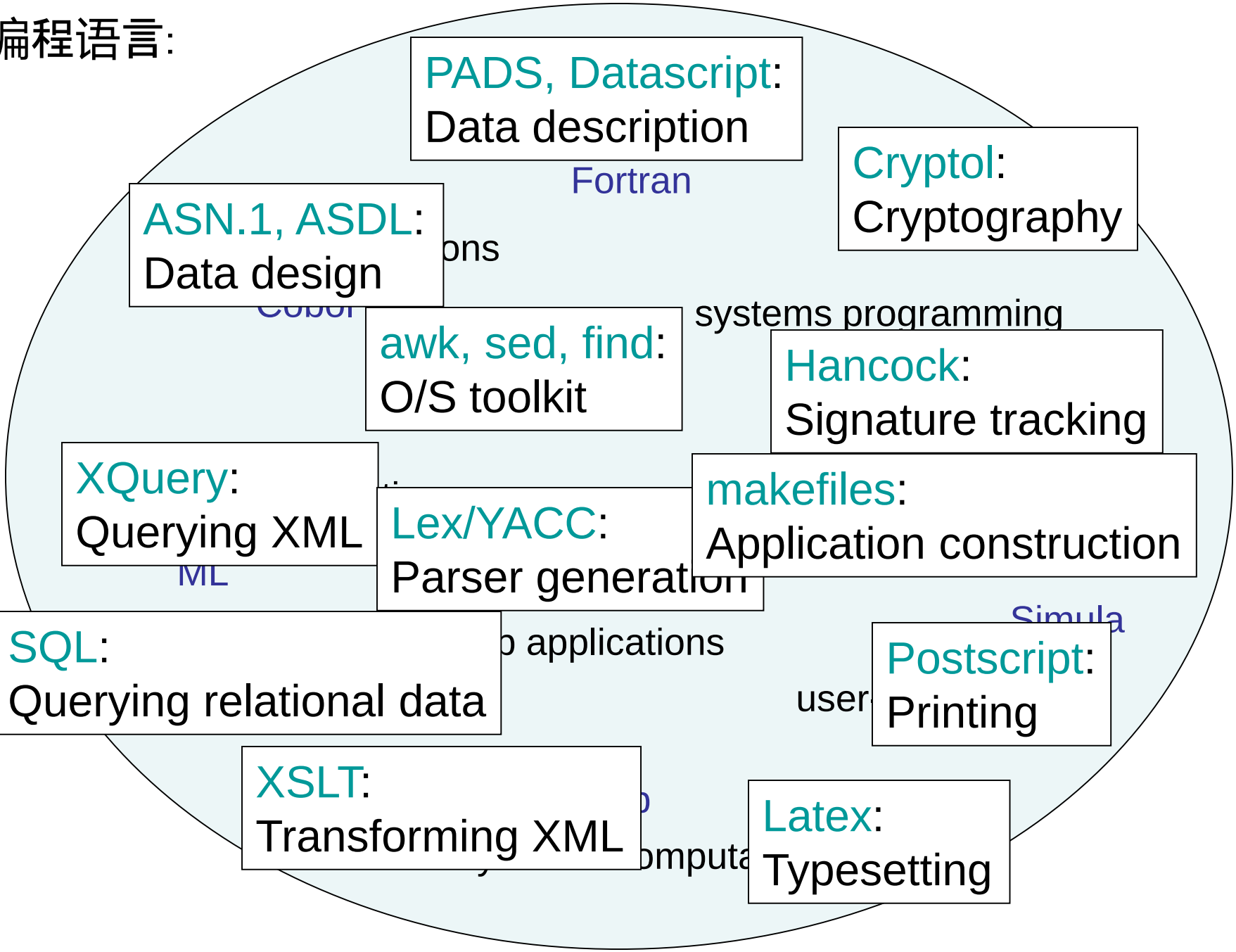
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
systems programming
Hancock:
Signature tracking
XQuery:
Querying XML
simulations
Lex/YACC:
Parser generation
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

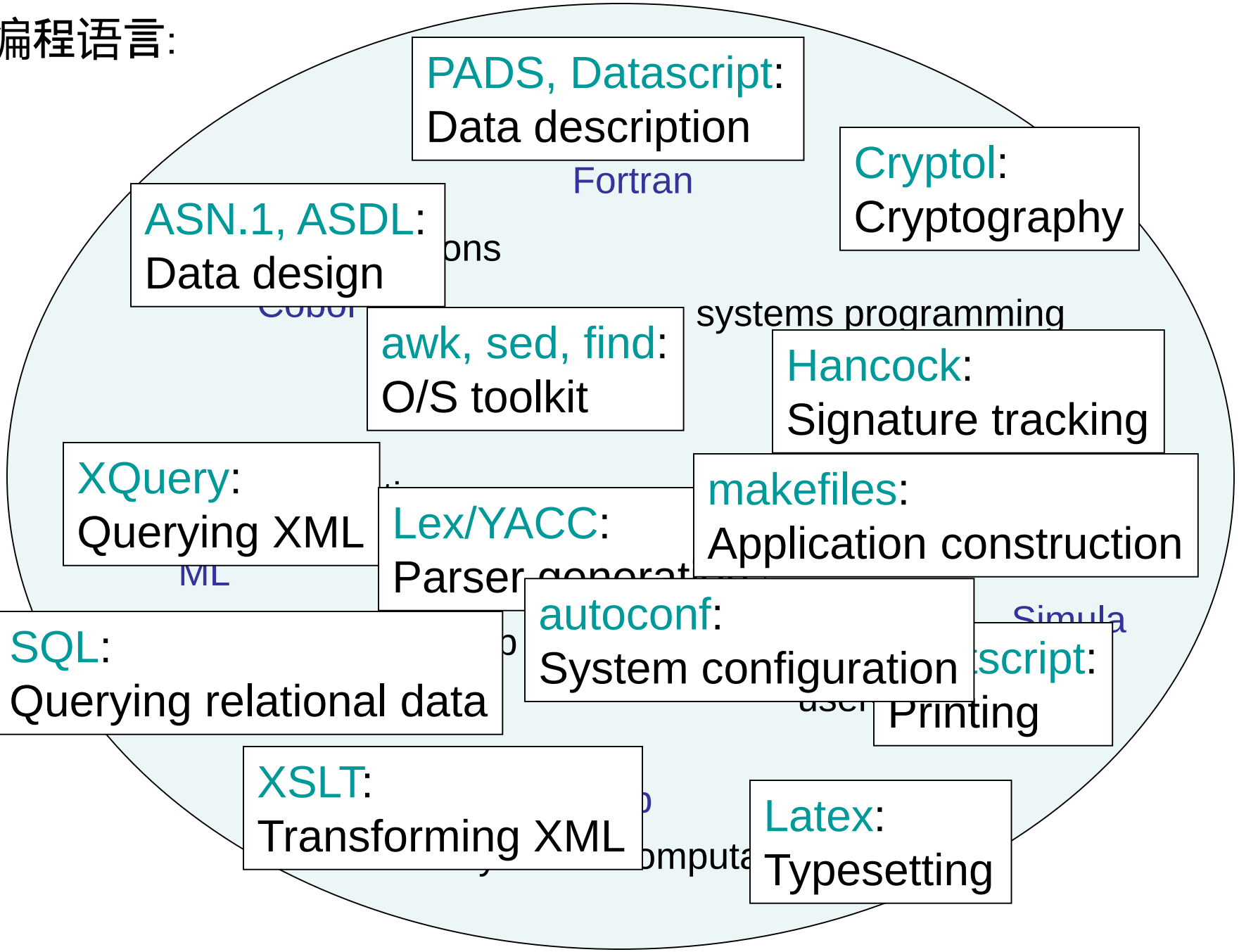
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
systems programming
awk, sed, find:
O/S toolkit
Hancock:
Signature tracking
XQuery:
Querying XML
simulations
Lex/YACC:
Parser generation
SQL:
Querying relational data
Postscript:
Printing
XSLT:
Transforming XML
Latex:
Typesetting

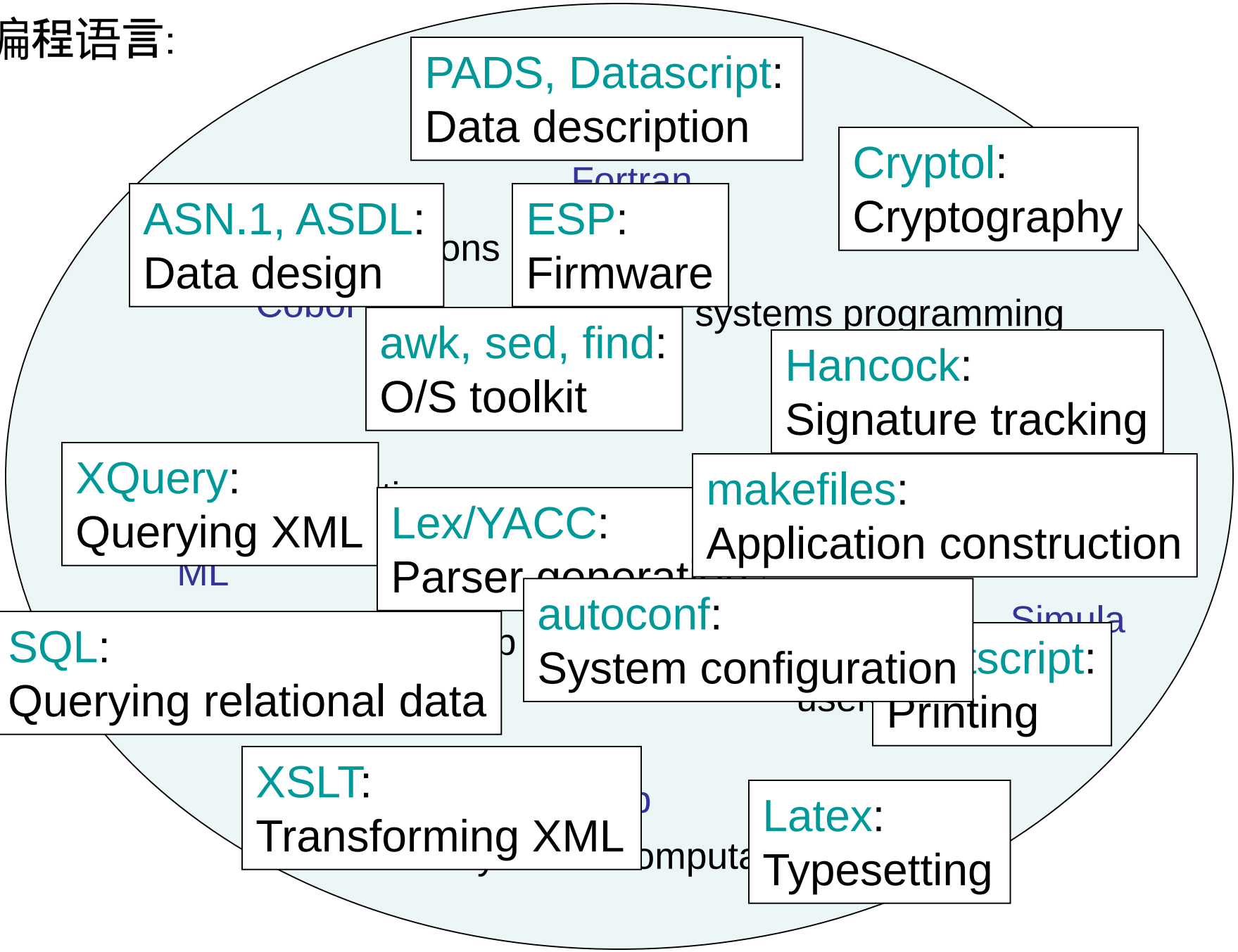
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
ons
systems programming
awk, sed, find:
O/S toolkit
Hancock:
Signature tracking
XQuery:
Querying XML
ML
Lex/YACC:
Parser generation
makefiles:
Application construction
Simula
SQL:
Querying relational data
o applications
Postscript:
Printing
user
XSLT:
Transforming XML
Latex:
Typesetting

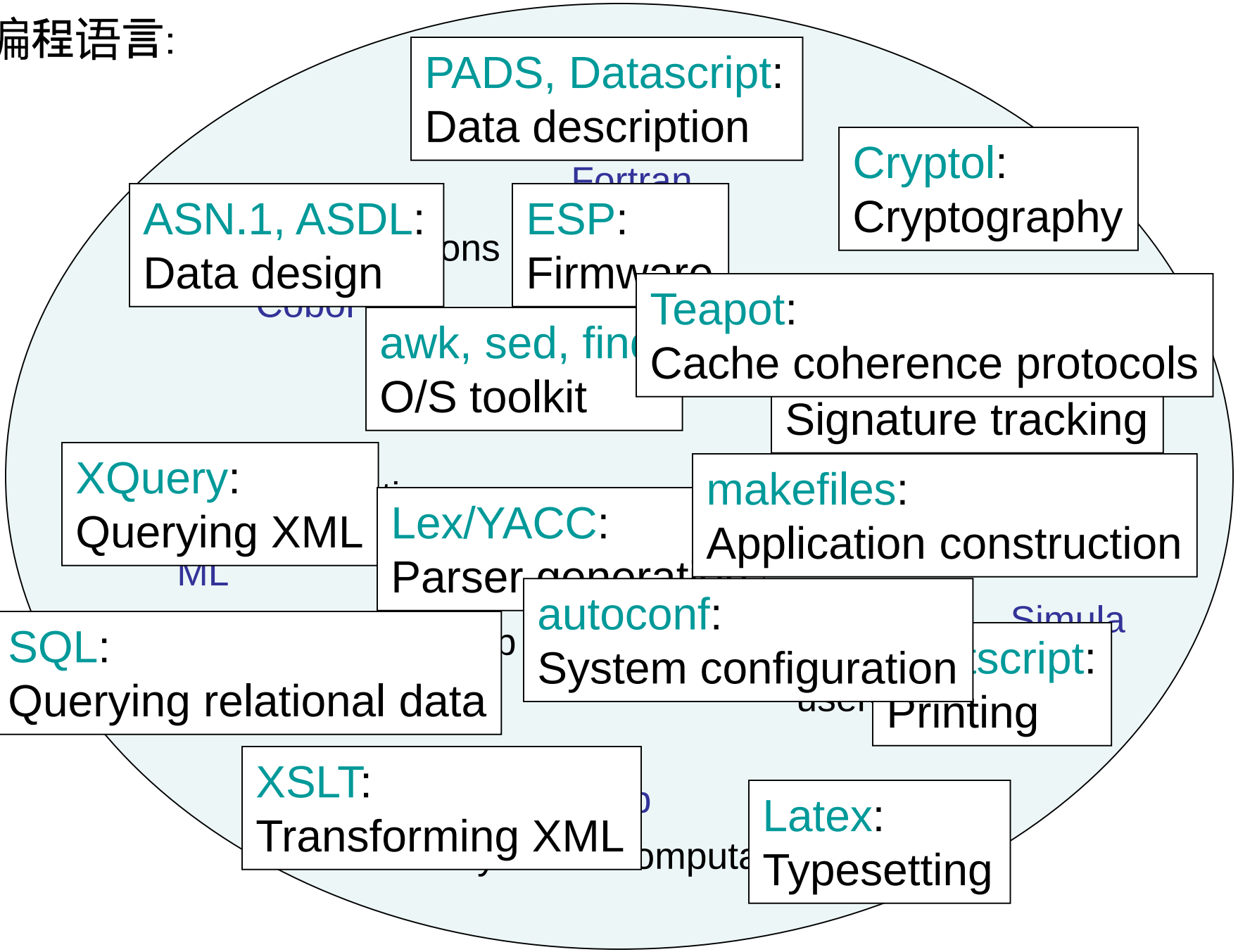
编程语言:


PADS, Datascript:
Data description
Fortran
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
systems programming
awk, sed, find:
O/S toolkit
Hancock:
Signature tracking
XQuery:
Querying XML
Lex/YACC:
makefiles:
Application construction
autoconf:
System configuration
SQL:
Querying relational data
Printing
XSLT:
Transforming XML
Latex:
Typesetting

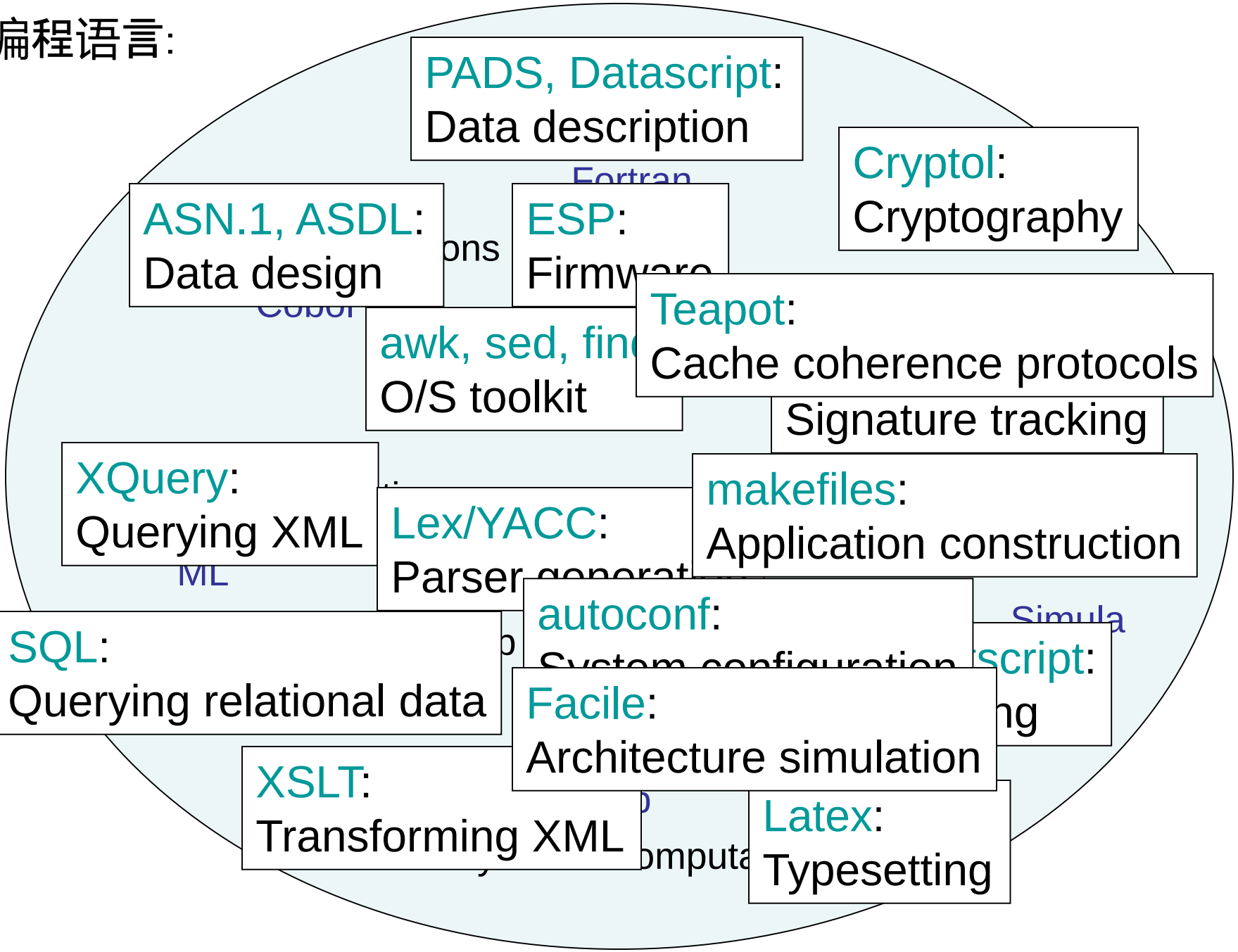
编程语言:


PADS, Datascript:
Data description
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
ESP:
Firmware
systems programming
awk, sed, find:
O/S toolkit
Hancock:
Signature tracking
XQuery:
Querying XML
Lex/YACC:
makefiles:
Application construction
autoconf:
System configuration
SQL:
Querying relational data
Printing
XSLT:
Transforming XML
Latex:
Typesetting

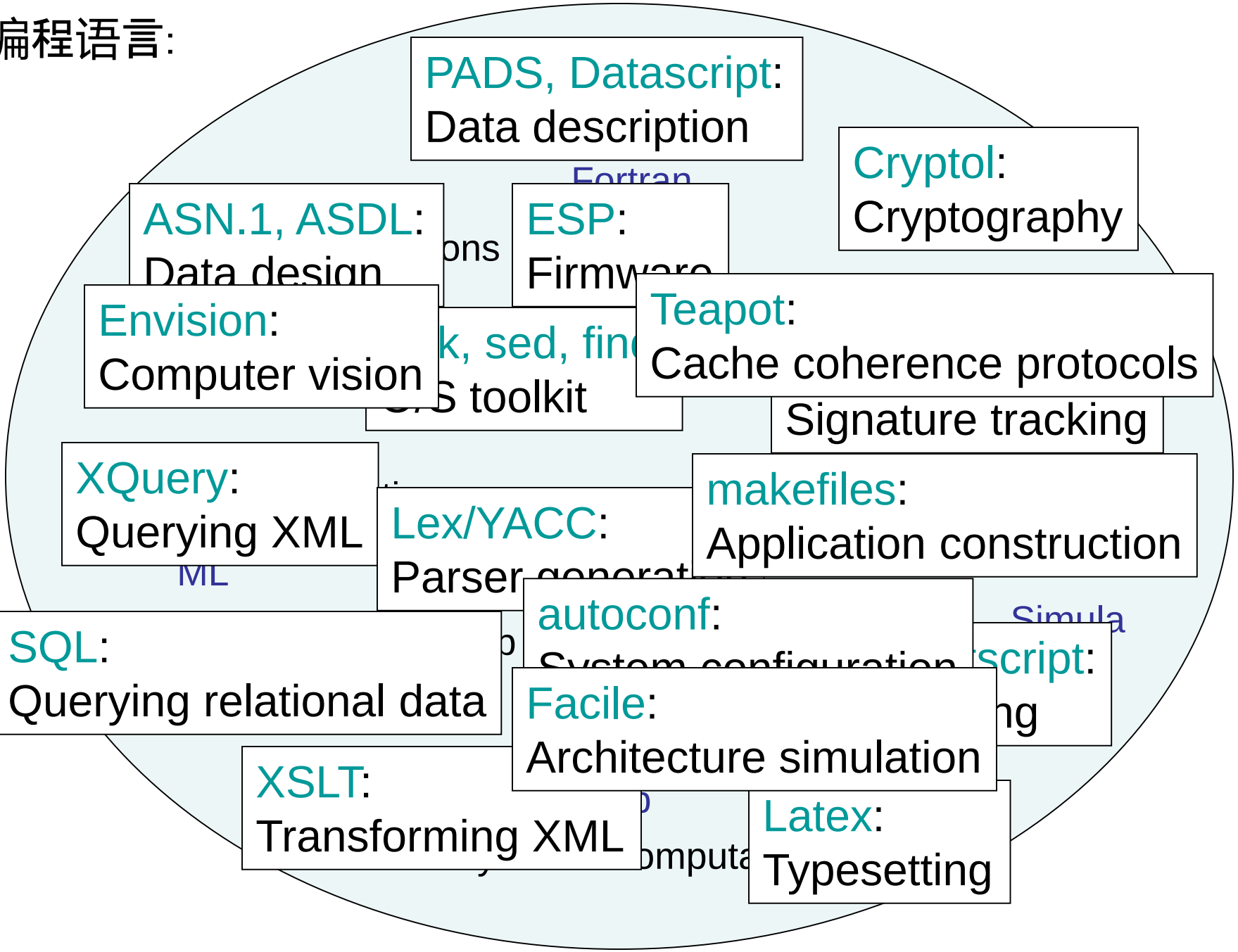
编程语言:


PADS, Datascript:
Data description
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
ESP:
Firmware
Teapot:
Cache coherence protocols
awk, sed, find
O/S toolkit
Signature tracking
XQuery:
Querying XML
Lex/YACC:
makefiles:
Application construction
autoconf:
System configuration
SQL:
Querying relational data
Printing
XSLT:
Transforming XML
Latex:
Typesetting

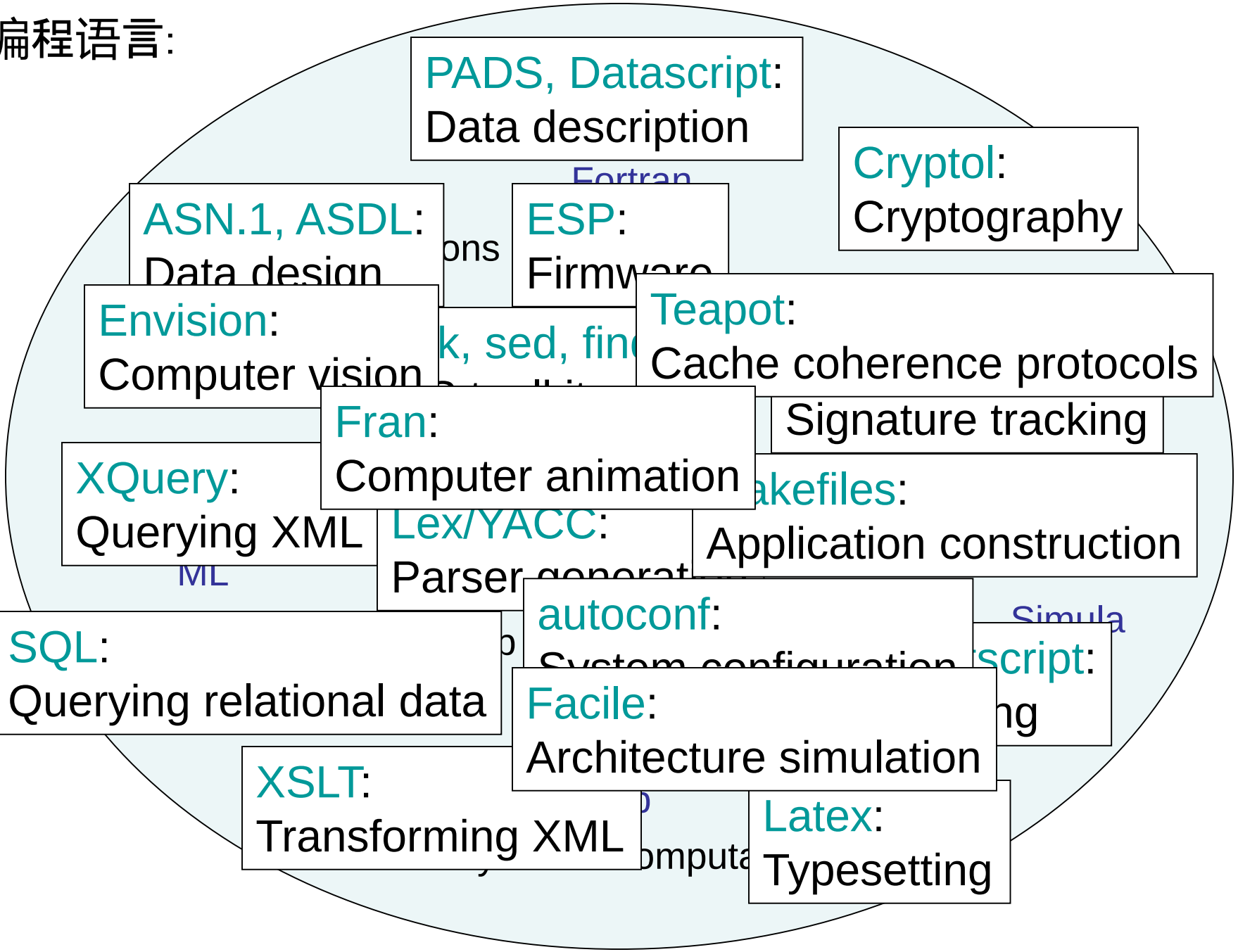
编程语言:


PADS, Datascript:
Data description
Cryptol:
Cryptography
ASN.1, ASDL:
Data design
ESP:
Firmware
Teapot:
Cache coherence protocols
awk, sed, find
O/S toolkit
Signature tracking
XQuery:
Querying XML
Lex/YACC:
makefiles:
Application construction
autoconf:
SQL:
Querying relational data
Facile:
Architecture simulation
XSLT:
Transforming XML
Latex:
Typesetting

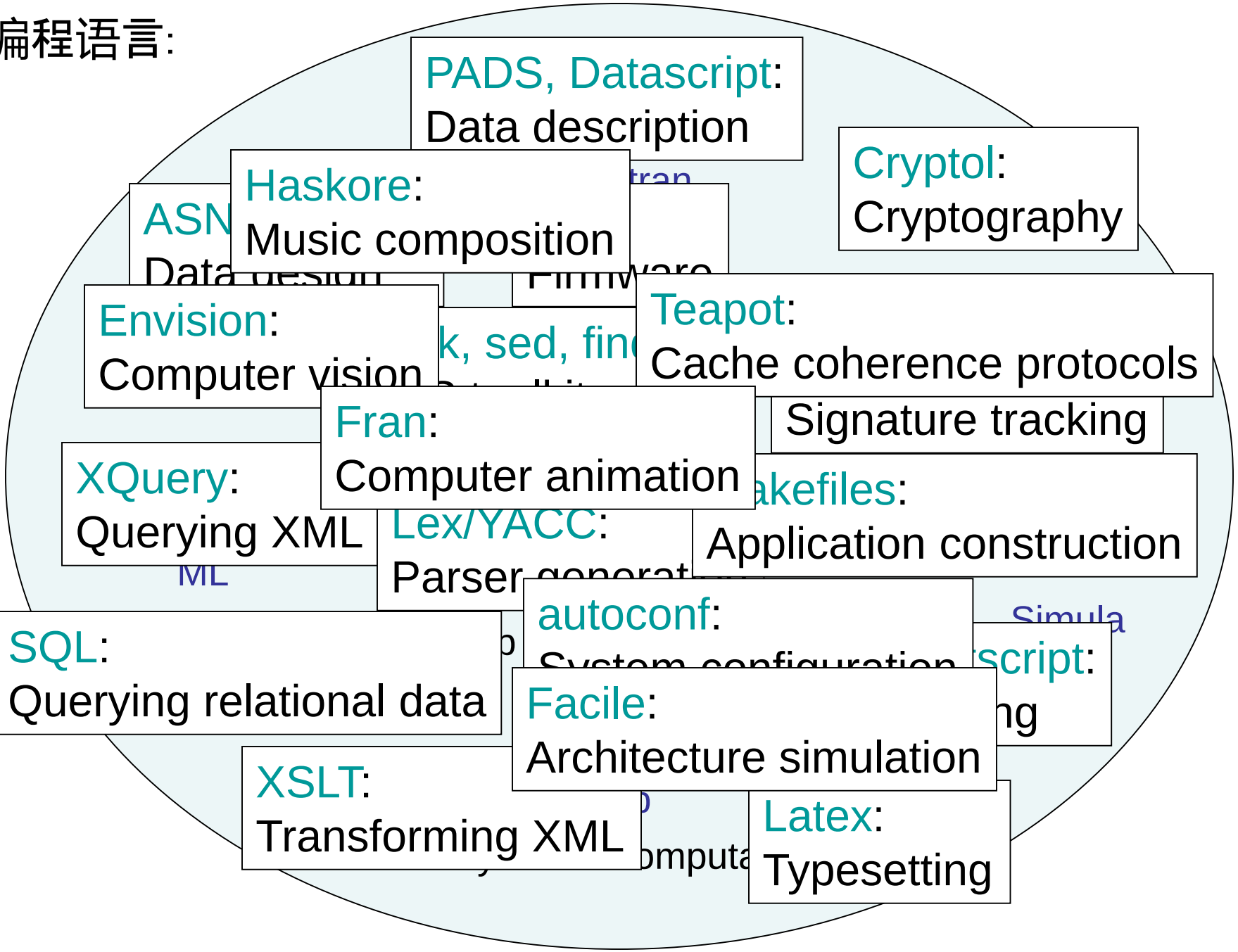
编程语言:


PADS, Datascript:
Data description
Cryptol:
Cryptography
ASN.1, ASDL:
ESP:
Envision:
Computer vision
Teapot:
Cache coherence protocols
Signature tracking
XQuery:
Querying XML
Lex/YACC:
makefiles:
Application construction
autoconf:
SQL:
Querying relational data
Facile:
Architecture simulation
XSLT:
Transforming XML
Latex:
Typesetting

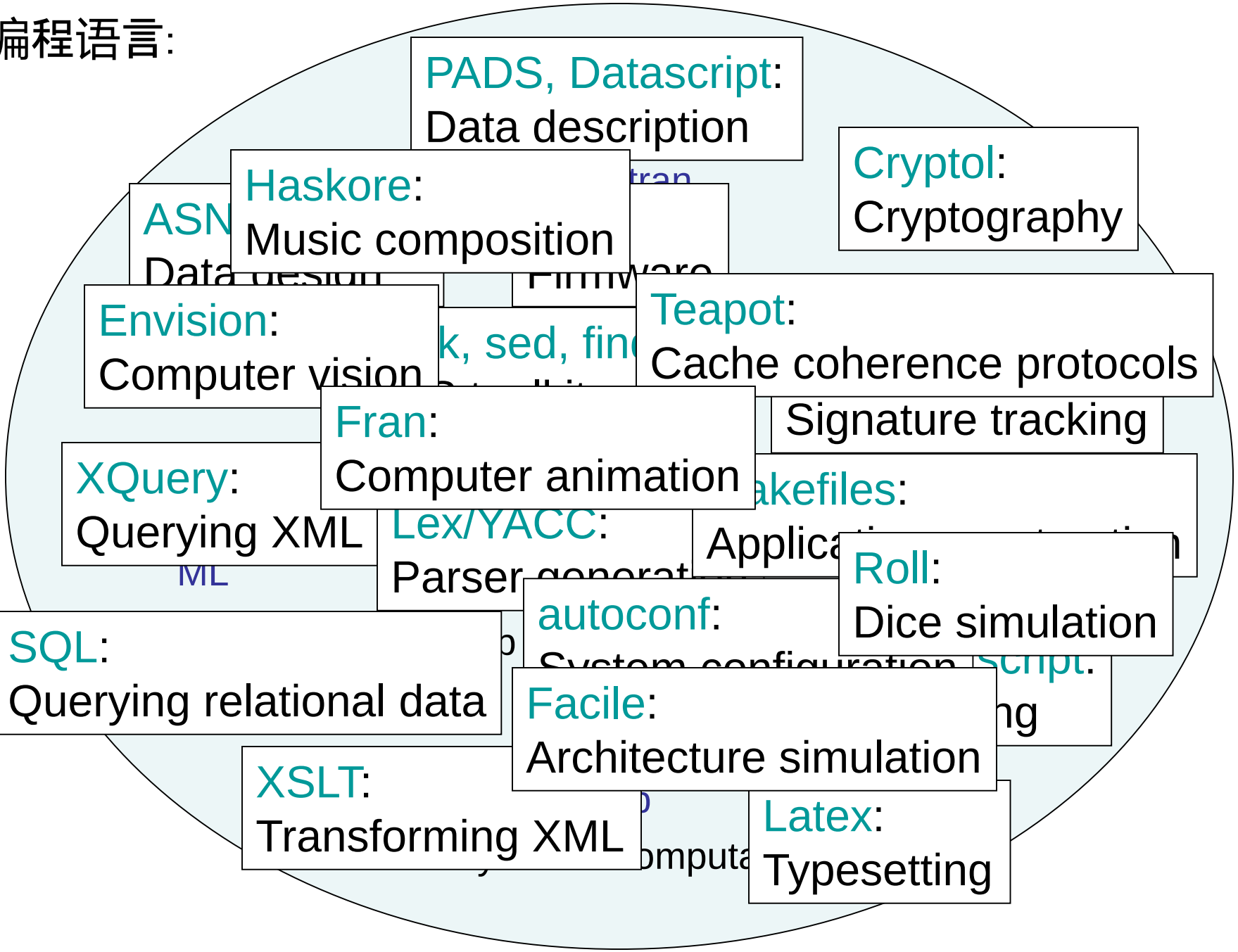
编程语言:


PADS, Datascript:
Data description
Cryptol:
Cryptography
ASN.1, ASDL:
ESP:
Envision:
Teapot:
Cache coherence protocols
Signature tracking
Fran:
Computer animation
XQuery:
Querying XML
Lex/YACC:
akefiles:
Application construction
autoconf:
SQL:
Querying relational data
Facile:
Architecture simulation
XSLT:
Transforming XML
Latex:
Typesetting

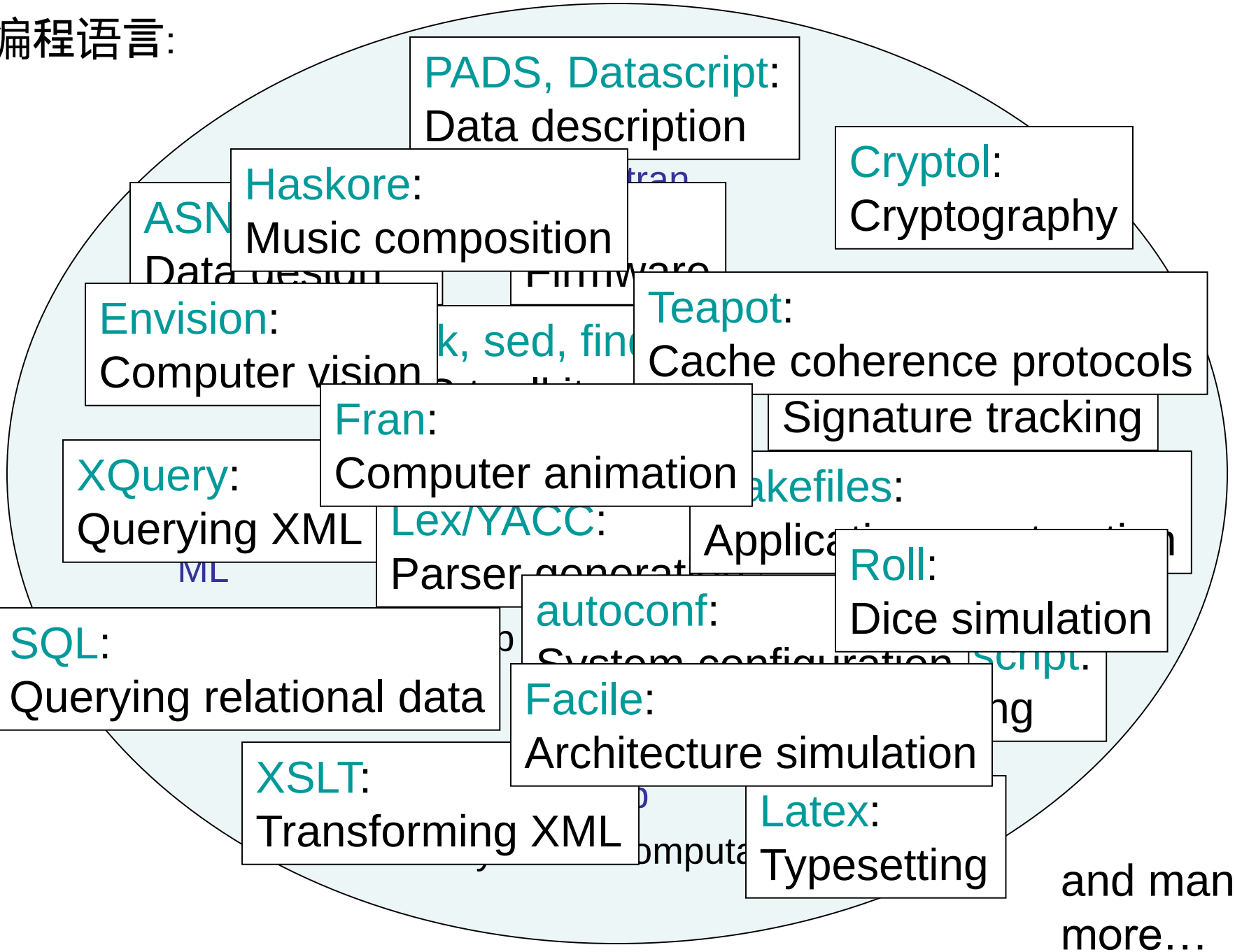
编程语言:


PADS, Datascript:
Data description
Haskore:
Music composition
Cryptol:
Cryptography
Envision:
Computer vision
Teapot:
Cache coherence protocols
Signature tracking
Fran:
Computer animation
XQuery:
Querying XML
Lex/YACC:
Application construction
SQL:
Querying relational data
autoconf:
Facile:
Architecture simulation
XSLT:
Transforming XML
Latex:
Typesetting

编程语言:


PADS, Datascript:
Data description
Haskore:
Music composition
Cryptol:
Cryptography
Envision:
Computer vision
Teapot:
Cache coherence protocols
Signature tracking
Fran:
Computer animation
XQuery:
Querying XML
Lex/YACC:
Roll:
Dice simulation
SQL:
Querying relational data
autoconf:
Facile:
Architecture simulation
XSLT:
Transforming XML
Latex:
Typesetting

编程语言:



and many more…

为什么使用 DSL?

- 首先，为什么需要一个语言?
  - 因为语言为计算机提供了丰富的接口

```
#!/sw/bin/ruby
require 'uri'; require  'net/http'

uri= URI.parse(ARGV[0])
h=Net::HTTP.new(uri.host,80)

resp,data = h.get(uri.path)
hwk = {}
if resp.message == "OK"
  data.scan(/Homework (\d*) \(due (\d*)\/(\d*)\)/)\
                    {|x,y,z| hwk[x] = Time.local(2005,y,z)}
end

hwk.each{| assignment, duedate|
 if duedate < (Time.now - 60 * 60 * 24)
   puts "Hwk #{assignment} was due on #{duedate.strftime("%A, %B %d")}."
 else
   puts "Hwk #{assignment} is due on #{duedate.strftime("%A, %B %d")}."
 end
}
```

vs

  - 因为语言可以直接提供一个计算域的模型

DSL具有针对领域裁剪的抽象性

- 易于被领域专家使用
- 提升了可靠性
  - 程序更加简短.
  - 通过编译器生成冗长的样板代码(boilerplate code)
- 允许程序充当实时文档

少即是多

- 限制表达方式有利于在领域级别上验证和优化
  - SQL 程序一定会结束（不会出现死循环）
  - 用YACC 描述的语言规范一定能编译成下推自动机（PDA，Push Down Automata）
  - CryptoI 程序一定只需要有限空间

举例：SQL

SQL是查询关系型数据库的语言

Students

ID	NAME
01	Harry Potter
02	Hermione Granger
03	Ronald Weasley

Potions

ID	GRADE
01	Satisfactory
02	Outstanding
03	Satisfactory

```
SELECT Students.NAME,
       Potions.GRADE
FROM Students, Potions
WHERE Students.ID = Potions.ID
```

NAME	GRADE
Harry Potter	Satisfactory
Hermione Granger	Outstanding
Ronald Weasley	Satisfactory

举例：SQL（续）

- SQL 程序会被编译成带有选择、映射和逻辑算子的关系代数
- 查询引擎基于数据索引和其它统计信息选择相应的物理算子
- 数据分析师（领域专家）只需定义问题，不必关心底层实现细节

DSL的理想实现

- 针对特定领域
  - 对于语言使用者易于理解
  - 简化的表达、检测和复用
- 可执行
  - 可以通过测试和调试来验证正确性
  - 可以生成测试例
- 声明式编程
  - 无特定的实现细节，程序紧凑
  - 多种使用方式-测试、生成、建模等
  - 易于迁移到任何架构下
- 含义清晰
  - 有形式化基础
  - 精确的语法和语义
  - 独立于底层机器模型

为什么不使用函数库呢?

- 有些DSL实际上就是函数库
  - 比如: Haskore 是一个用来作曲的语言
- 但是:
  - 复杂的函数库可能难以使用
  - 难以使用领域知识

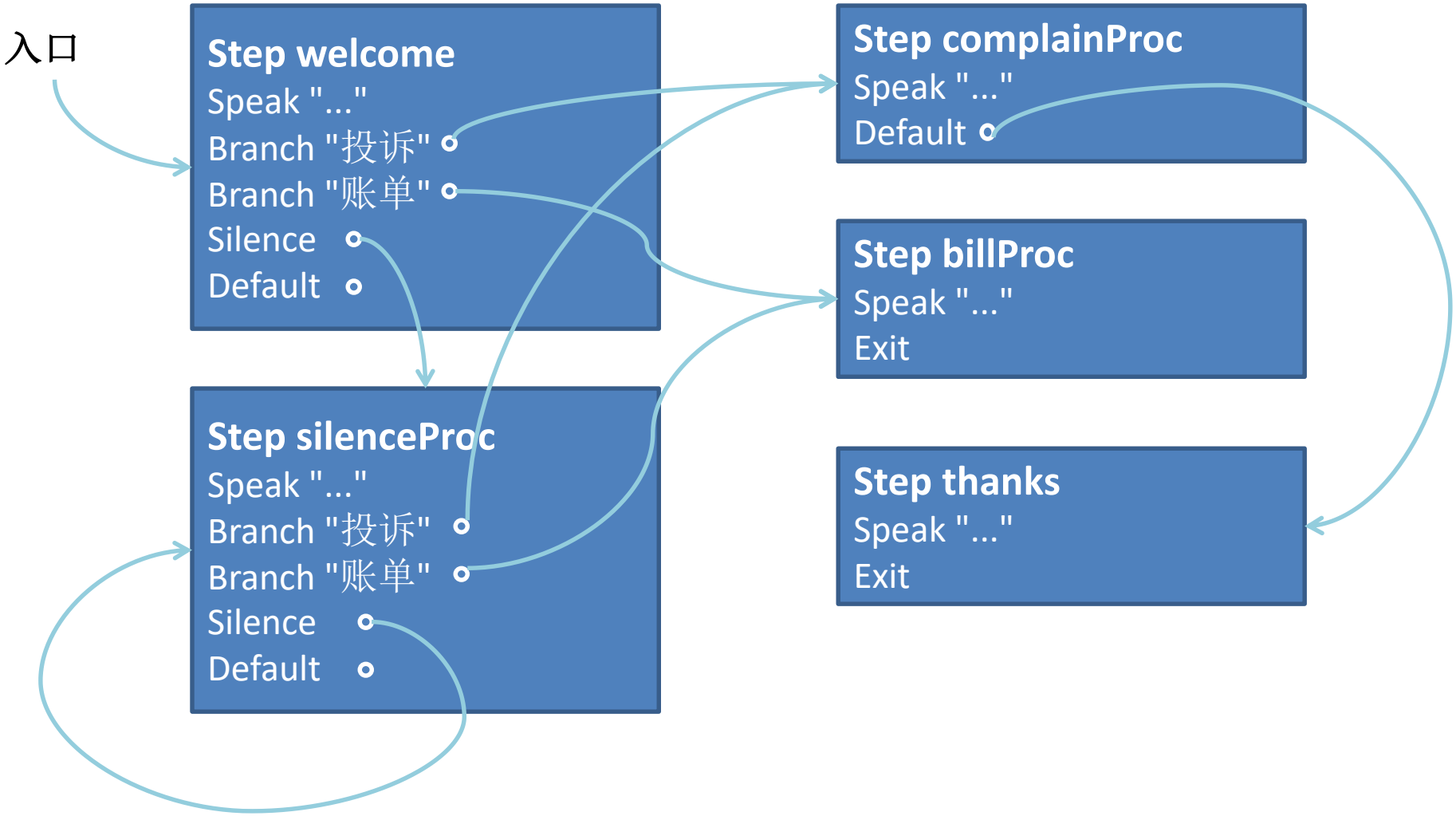
DSL的缺陷

- 用户需要学习一门新语言
- 实现和维护一个DSL令人生畏，特别是对于一个狭小专业的领域
- 缺乏工具支撑:
  - 调试器
  - 性能分析器
  - 集成开发环境
  - …

客服逻辑脚本示例

```
Step welcome
  Speak $name + "您好，请问有什么可以帮您?"
  Listen 5, 20
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silence
  Default defaultProc
Step complainProc
  Speak "您的意见是我们改进工作的动力，请问您还有什么补充?"
  Listen 5, 50
  Default thanks
Step thanks
  Speak "感谢您的来电，再见"
  Exit
Step billProc
  Speak "您的本月账单是" + $amount + "元，感谢您的来电，再见"
  Exit
Step silenceProc
  Speak "听不清，请您大声一点可以吗"
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silenceProc
  Default defaultProc
Step defaultProc
  ....
```

脚本的含义


入口
Step welcome
Speak "..."
Branch "投诉"
Branch "账单"
Silence
Default
Step complainProc
Speak "..."
Default
Step billProc
Speak "..."
Exit
Step silenceProc
Speak "..."
Branch "投诉"
Branch "账单"
Silence
Default
Step thanks
Speak "..."
Exit

脚本的语义动作

Step: 完整表示一个步骤的所有行为

Speak:
计算表达式合成一段文字
调用媒体服务器进行语音合成并播放

Listen:
调用媒体服务器对客户说的话录音，并进行语音识别
语音识别的结果调用“自然语言分析服务”分析客户的意愿

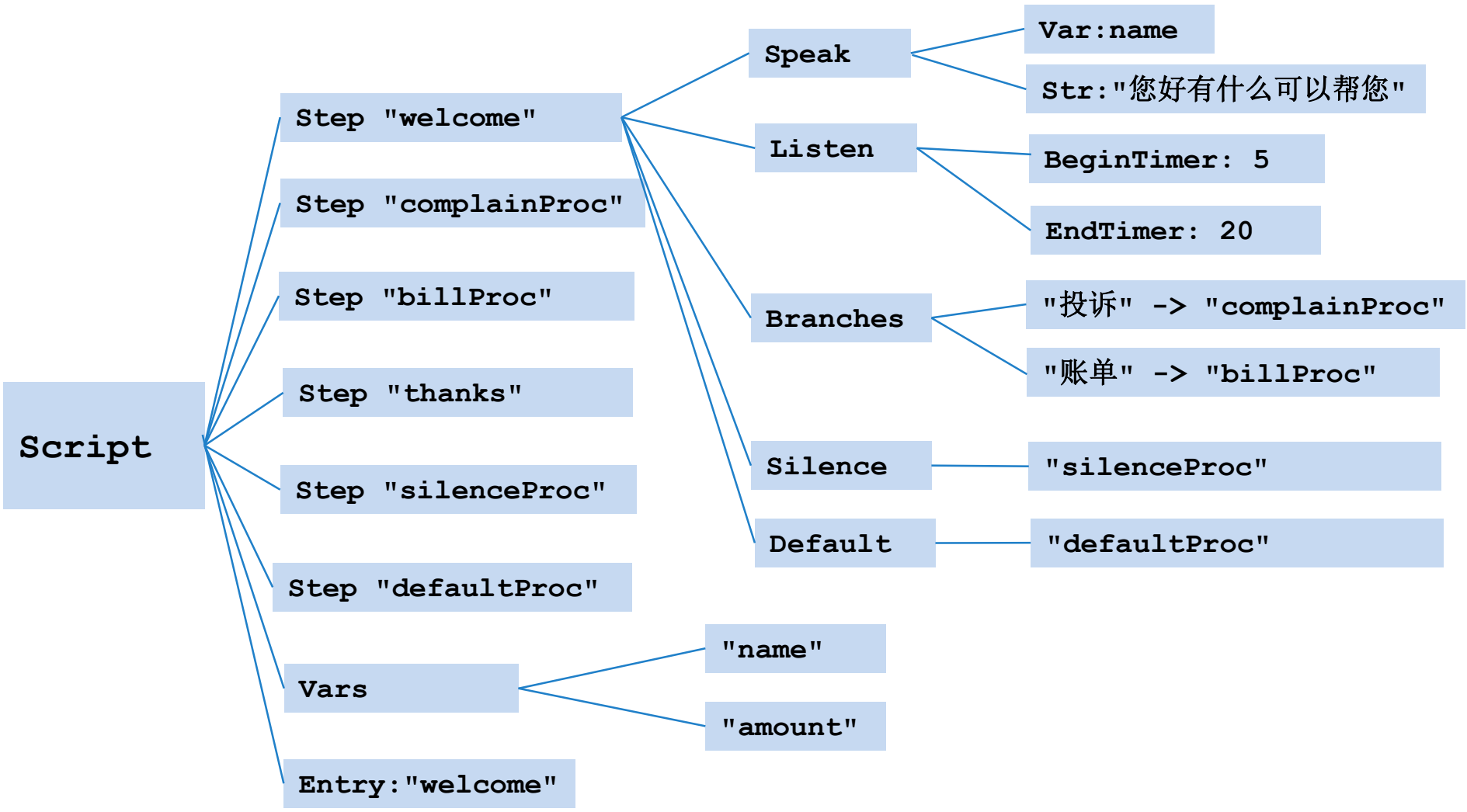
Branch:
对客户的意愿进行分支处理，不同的意愿，跳转到不同的Step

Silence: 如果用户不说话，应该跳转到哪个Step

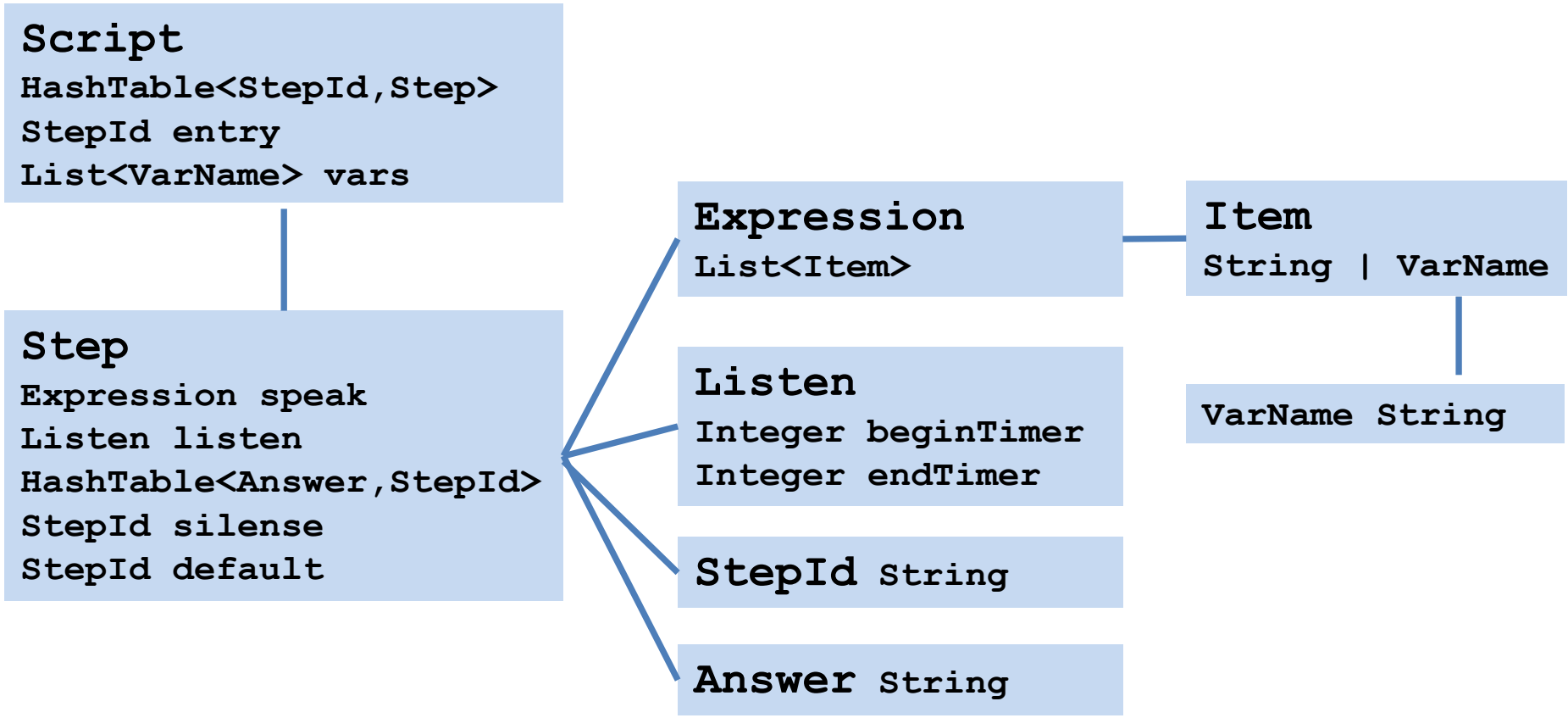
Default: 如果客户意愿没有相应匹配，应该跳转到哪个Step

Exit: 结束对话

将脚本的语法元素抽象为树形结构



存储语法树的数据结构


Script
HashTable<StepId,Step>
StepId entry
List<VarName> vars
Step
Expression speak
Listen listen
HashTable<Answer,StepId>
StepId silense
StepId default
Expression
List<Item>
Item
String | VarName
VarName String
Listen
Integer beginTimer
Integer endTimer
StepId String
Answer String

```
Step welcome
  Speak $name + "您好，请问有什么可以帮您?"
  Listen 5, 20
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silence
  Default defaultProc
Step complainProc
  Speak "您的意见是我们改进工作的动力，请问您还有什么补充?"
  Listen 5, 50
  Default thanks
Step thanks
  Speak "感谢您的来电，再见"
  Exit
Step billProc
  Speak "您的本月账单是" + $amount + "元，感谢您的来电，再见"
  Exit
Step silenceProc
  Speak "听不清，请您大声一点可以吗"
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silenceProc
  Default defaultProc

Step defaultProc
  ....
```

脚本文本

Parser
语法分析

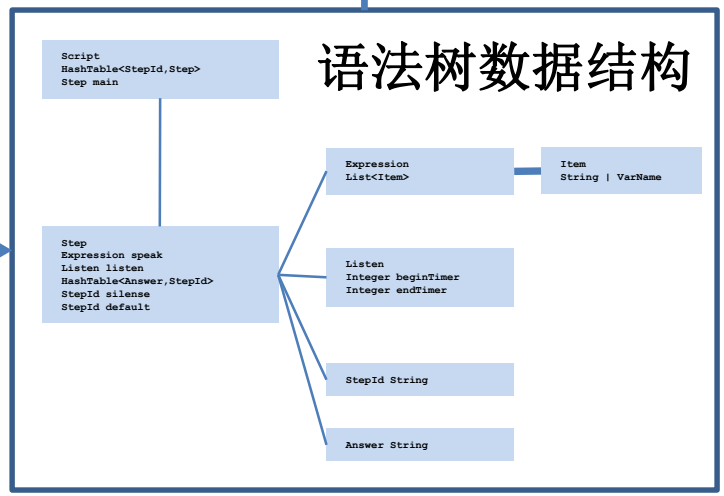
Interpreter
解释执行

呼叫
控制

媒体
控制

自然
语言
分析

语法树数据结构



```
Step welcome
  Speak $name + "您好，请问有什么可以帮您?"
  Listen 5, 20
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silence
  Default defaultProc
Step complainProc
  Speak "您的意见是我们改进工作的动力，请问您还有什么补充?"
  Listen 5, 50
  Default thanks
Step thanks
  Speak "感谢您的来电，再见"
  Exit
Step billProc
  Speak "您的本月账单是" + $amount + "元，感谢您的来电，再
见"
  Exit
Step silenceProc
  Speak "听不清，请您大声一点可以吗"
  Branch "投诉", complainProc
  Branch "账单", billProc
  Silence silenceProc
  Default defaultProc

Step defaultProc
  ....
```

脚本文本

这里简化一下

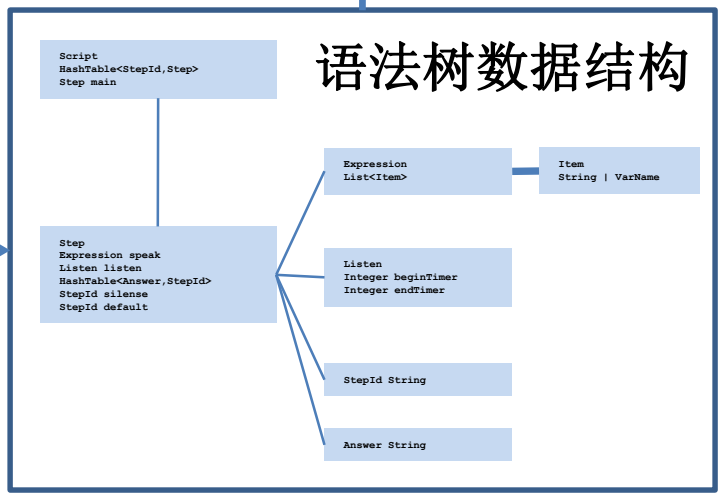
Interpreter
解释执行

显示Speak内容

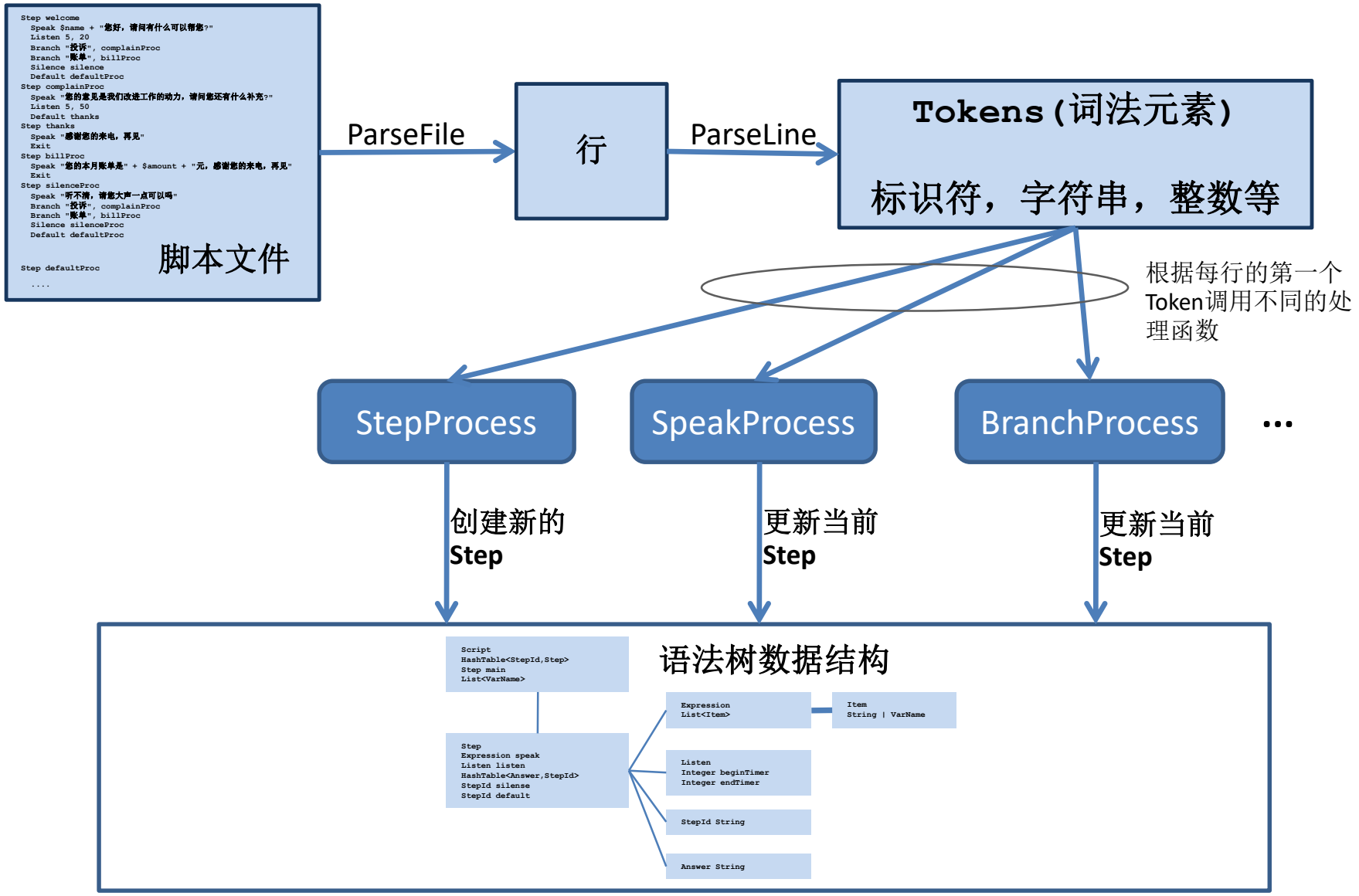
输入客户意愿

Parser
语法分析

语法树数据结构



Parser的实现


Step welcome
Speak $name + "您好，请问有什么可以帮您?"
Listen 5, 20
Branch "投诉", complainProc
Branch "账单", billProc
Silence silence
Default defaultProc
Step complainProc
Speak "您的意见是我们改进工作的动力，请问您还有什么补充?"
Listen 5, 50
Default thanks
Step thanks
Speak "感谢您的来电，再见"
Exit
Step billProc
Speak "您的本月账单是" + $amount + "元，感谢您的来电，再见"
Exit
Step silenceProc
Speak "听不清，请您大声一点可以吗"
Branch "投诉", complainProc
Branch "账单", billProc
Silence silenceProc
Default defaultProc
Step defaultProc
....
脚本文件
ParseFile
行
ParseLine
Tokens（词法元素）
标识符，字符串，整数等
根据每行的第一个Token调用不同的处理函数
StepProcess
SpeakProcess
BranchProcess
...
创建新的Step
更新当前Step
更新当前Step
Script
HashTable<StepId,Step>
Step main
List<VarName>
语法树数据结构
Expression
List<Item>
Item
String | VarName
Step
Expression speak
Listen listen
HashTable<Answer,StepId>
StepId silense
StepId default
Listen
Integer beginTimer
Integer endTimer
StepId String
Answer String

Parser的实现

```
ParseFile(fileName):
        打开文件
        读取文件的每一行line:
                line.trim()删除行首空白
                忽略空行
                忽略'#'开头的注释行
                ParseLine(line)处理一行
        关闭文件
```

```
ParseLine(line):
        读取一行中空白分割的每一个token:
                遇到'#'开头的token则处理结束（忽略行尾注释）
                获得标识符，字符串或者操作符几类token
                将token加入到List中
        ProcessTokens(token[])
```

Parser的实现

```
ProcessTokens(token[]):
        对List中的每一个token进行处理
        根据token[0]分情况处理:
                Step: ProcessStep(token[1])
                Speak: ProcessSpeak(token+1)
                Listen: ProcessListen (token[1], token[2])
                Branch: ProcessBranch(token[1], token[2])
                Silence: ProcessSilence(token[1])
                Default: ProcessDefault (token[1])
                Exit: ProcessExit ()
                如果不是上述token则报错
```

```
ProcessStep(stepId):
        Script创建一个新的Step，标识为stepId
        设置当前Step为新创建的Step
        如果这是第一个Step，则设置当前Step为Script的mainStep
```

Parser的实现

ProcessSpeak(token[]):

token[]是一个表达式，每个token可能是字符串，变量或者'+'

ProcessExpression(token[])得到Expression

将Speak以及对应的表达式存入当前的Step

ProcessExpression(token[]):

这么简单的表达式...

忽略掉加号，其它token追加到Expression中的List<Item>

将变量名存入Script的List<VarName>中

ProcessListen(startTimer, stopTimer):

构造Listen(startTimer, stopTimer)存入当前Step

ProcessBranch(answer, nextStepId):

将answer和nextStepId插入当前Step的HashTable

Parser的实现

ProcessSilence(nextStepId):
将当前Step的silence变量设置成nextStepId中的值

ProcessDefault(nextStepId):
将当前Step的default变量设置成nextStepId中的值

ProcessExit():
将当前Step设置为终结Step

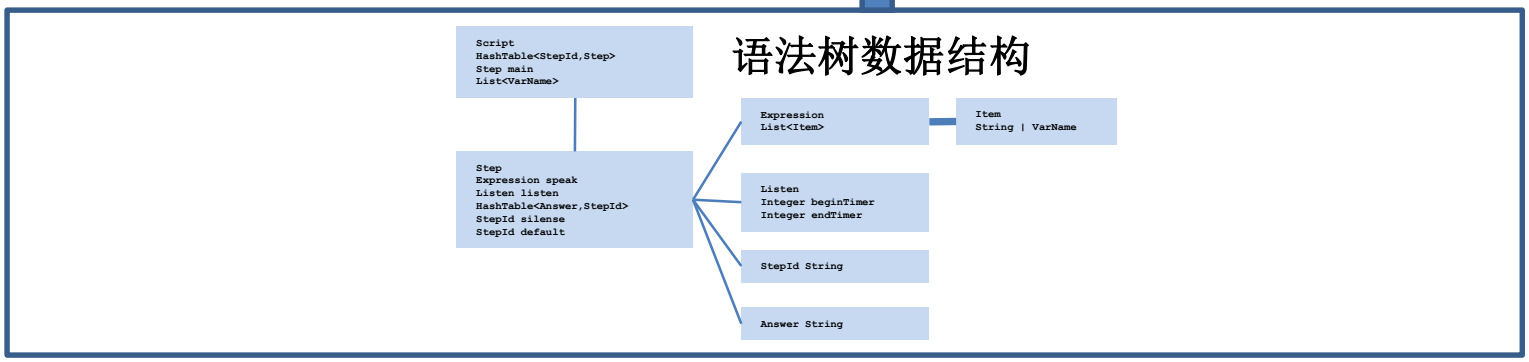
Interpreter的实现

执行环境：

- 变量表
- 当前Step
- ...

解释程序：

接通电话，连接媒体服务器，获取脚本语法树，创建执行环境
当前Step置为entryStep
循环针对当前Step做：
执行Speak（语音合成，语音播放）
如果本步骤是终结步骤，则结束循环，断开通话
执行Listen（录音，语音识别，自然语言理解）
获得下一个StepId：
如果用户沉默，则获得Silence的StepId
根据用户意向查找HashTable，获得StepId
如果查不到则获得Default的StepId
将当前Step置为刚才获得的StepId对应的Step



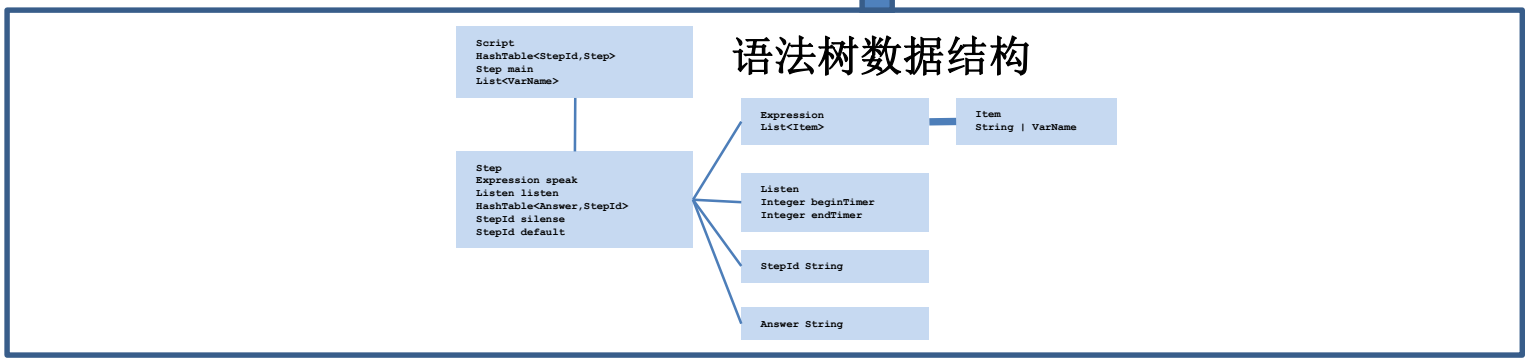
Interpreter的实现（简化版）

执行环境：

- 变量表
- 当前Step
- ...

解释程序（简化版）：

获取脚本语法树，创建执行环境
当前Step置为entryStep
循环针对当前Step做：
执行Speak（输出到标准输出）
如果本步骤是终结步骤，则结束循环，断开通话
执行Listen（直接从标准输入读入用户意愿）
获得下一个StepId：
如果用户沉默，则获得Silence的StepId
根据用户意向查找HashTable，获得StepId
如果查不到则获得Default的StepId
将当前Step置为刚才获得的StepId对应的Step



Interpreter的执行环境

当两个用户同时和机器人对话，解释器需要两个线程同时运行，每个线程服务一个用户。思考一下，哪些东西还是一份？那些东西需要两份？

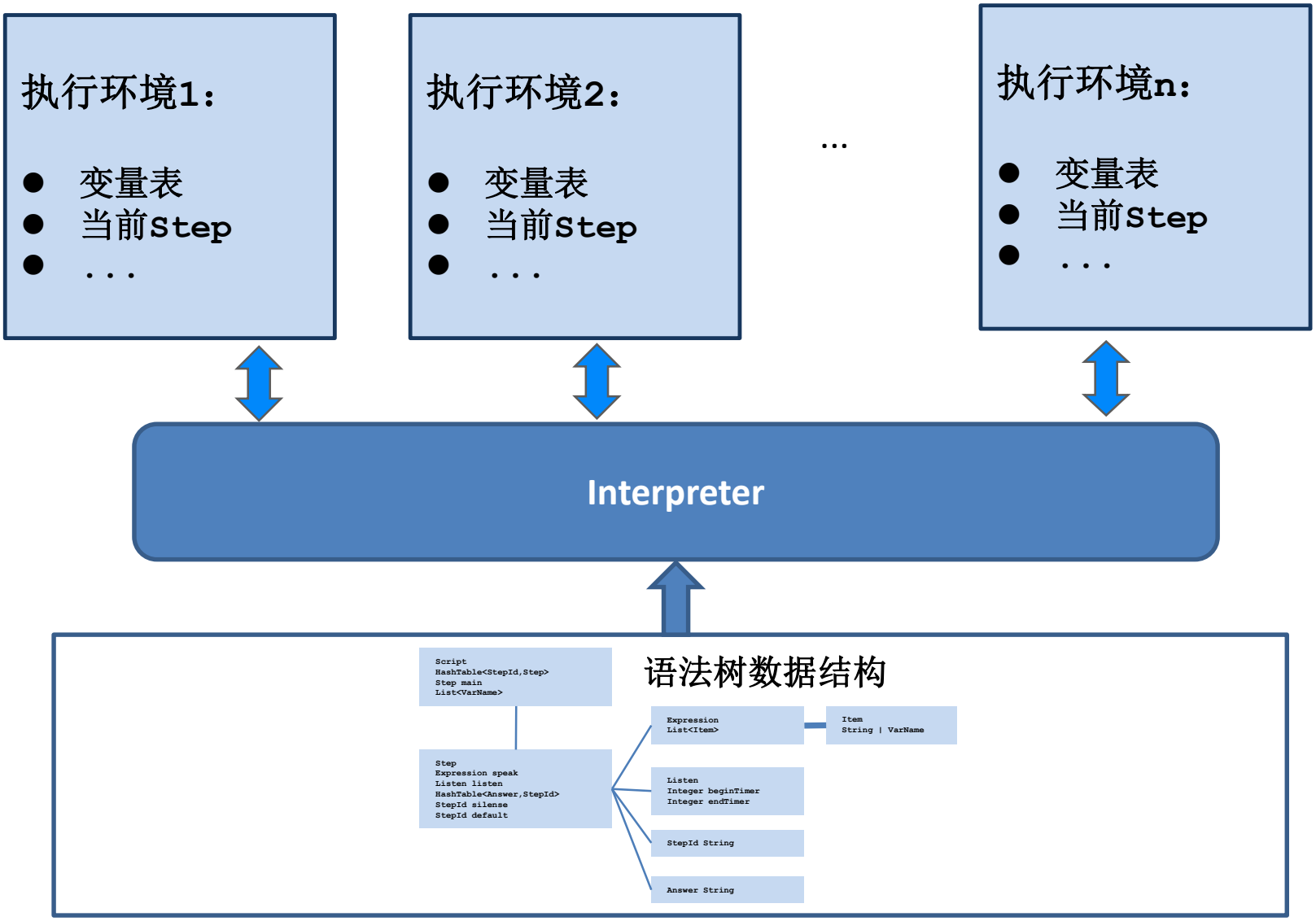
脚本文件：同一份（两个用户执行的是同一个脚本）
脚本语法树：同一份（可以调用一次Parser形成，多个线程公用）

但是，不同的用户肯定有不同的姓名，不同的账单，所以表示姓名和账单的变量表，肯定是两份。

两个用户有先有后，有快有慢，脚本执行的当前状态（例如当前step）肯定是两份。

简单来说，上述这些有“两份”的数据，也就是Interpreter的每次执行都需要一个新的实例的数据，我们称之为Interpreter的执行环境，需要单独的数据结构存放。

Interpreter的执行环境



思考题

执行环境：

- 变量表 ？
- 当前Step

执行环境中的变量表应该是什么数据结构？如何构建？如何使用？