

第二章 程序设计语言设计概述

- 2.1 表示与抽象
- 2.2 设计目标
- 2.3 设计准则
- 2.4 规格说明

计算机语言定义

- 表示法(或符号)系统——LRM
- 可以编制软件
- 机器可识别
- 可执行（应用）

2.1 表示与抽象

- 表示是人为制造的符号组合以表达我们需要表达的意思。
- 程序是程序设计语言表示的计算
 - `float n;` //n 是浮点数变量
 - `sqrt(n);` //对n取平方根
- 同一程序的高级语言表示、经翻译后的汇编码表示、机器码表示就是该程序在不同抽象层次上的表示。

2.1 表示与抽象

- 程序在不同抽象层次表示的关系
- 例： $x = x + 1$ 在机器码上就有两种方法。

从内存代表 x 的地址中取出
值放在运算器中。

加1，将结果放于某临时单元。

将临时单元内容做类型检查
(必要时转换) 并放入 x 中。

从内存代表 x 的地址中取出
值放在运算器中。

加1，将结果放入 x 地址中。

2.1 表示与抽象

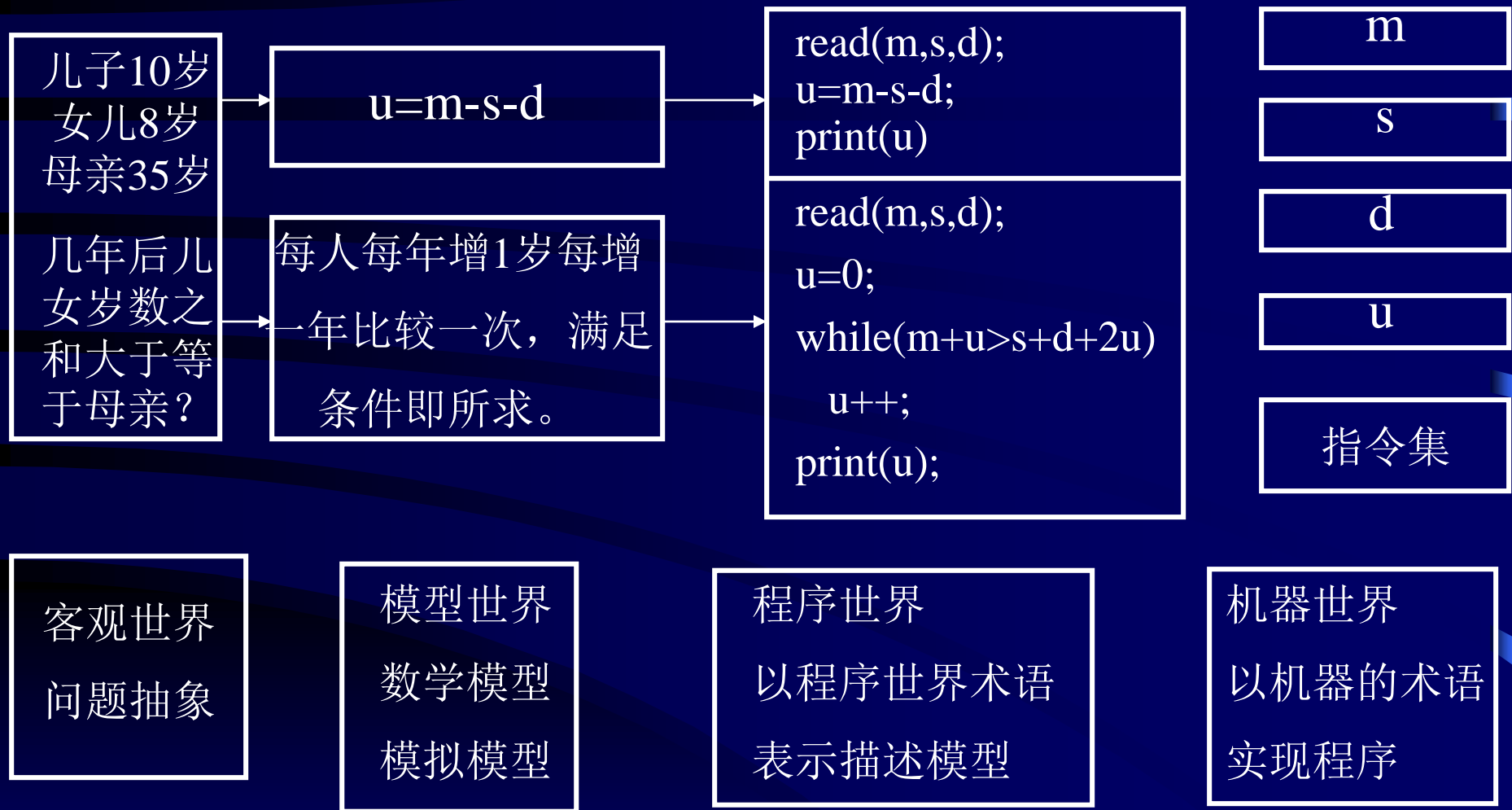


图2-1 计算机解题的四个世界

2.2 PL设计目标

定义一组能表示某种范型的特征集，每个特征有严格定义并可在机器上高效实现，程序员可灵活运用这些特征表达它所希望的任何计算。

什么是好PL

定义一组能表示某种范型的特征集，每个特征有严格定义并可在机器上高效实现，程序员可灵活运用这些特征表达它所希望的任何计算。

- 模型有力 Model Power
- 语义清晰 Semantic Clarity
- 移植性好 Portability
- 可读性好 Readability
- 程序质量 Quality
- 安全性
- 并发
- 方便 Convenience
- 简单 Simplicity
- 高效 Efficiency
- 灵活性 Flexibility
- 可扩充性 Extensible
- 可重用性 Reusable

什么是好PL

- 透明性、简单性和统一性
- 正交性
- 应用的自然性
- 对抽象性的支持
- 程序的易验证性
- 编程环境
- 程序的可移植性
- 使用代价

程序执行的代价、翻译的代价、程序创建测试和使用的代价、程序维护的代价

好PL的评价标准与矛盾

- 可读性：如果一个程序算法和数据结构能够明显地从程序文本中观察出来，那么这个程序就是可读的
- 可写性：使用简洁的、整齐的语法结构将会增强程序的可写性
- 易检验性：能够使用数学方法证明程序的正确性
- 易翻译性：关键是结构的规范化
- 无二义性：为每一种程序员可能写出的语法构造提供独一无二的解释

语言设计：目标演化

- Fortran 设计中最主要的考虑是易用性（与机器和汇编语言比较）和**高效实现**，特别关注程序能翻译成高效执行的代码，因为这样才可能被接受（今天 Fortran 仍高效）。
- 随着计算机变得越来越快，越来越便宜，效率问题虽然还是很重要，但重要性已大大下降。**易用性**方面的考虑仍非常明显：
 - 提高程序设计工作的效率
 - 帮助人们提高程序（软件）的质量，可靠性
 - 设法支持某些高级的软件设计技术

语言设计：目标演化

- 语言最主要作用是用于描述所需要的计算过程。为此需要：
 - **清晰，简洁**的形式（例子：C, Pascal, APL）
 - **清晰简单的语义**（易理解，易验证）
 - **正交性**（避免重复的可相互替代的特征，人们对此有些不同意见）
 - **可读性**（人容易阅读理解的东西，计算机也容易处理）

语言设计：目标演化

- 随着计算机应用发展，程序变得越来越复杂，开发程序变成代价高昂的工作。为支持复杂程序开发，提高开发工作的效率，语言设计有了许多新目标：

1. 支持对基本语言的**扩充**，提供各种扩充定义和抽象机制，过程、函数定义机制，扩充语言的基本操作，数据类型定义机制（及OO机制），扩充数据描述方式和功能。

例：C++ 在语言机制扩充方面有许多考虑（如运算符重载），可扩充语言（Extendable Languages），允许程序形式的改变（Lisp）

语言设计：目标演化

2. 提供支持复杂程序所需的高级组织的机制，支持大型程序开发模块机制（信息隔离和屏蔽），支持分别编译的机制，支持程序的物理组织

3. 支持软件重用，包括软件中的部分的重用，支持通用的基本程序库。

Pascal 失败之处之一就是忽略了库的开发。**C/Fortran** 都做得很好。**Ada**、**C++** 和 **Java** 的设计都特别考虑了对库的支持。许多新语言定义了功能非常丰富的标准程序库。

- 支持库开发：库是最基本的重用方式。是否支持库开发，决定了语言能否大范围使用。支持用户和第三方供应商开发各种扩充的和专用的库
- 支持某些层次或者方式的软件部件概念

问题：库开发或者部件是否需要本语言之外的功能？

OO 概念在上面许多方面起作用，因此成为复杂软件开发的重要方法

语言设计：目标演化

- 语言设计中需要考虑的另外一些重要问题：
 - 正常处理的**异常/错误处理**的良好集成（在产品软件的程序里，处理错误和各种特殊情况的代码占很大的比例，可能达**70%**）
 - 对于程序的**易修改可维护性**的支持
 - 对于**并发程序设计**的支持，用什么样的机制支持并发程序设计。
这方面的问题将长期成为语言研究和设计的热点问题
 - **安全性设计**：是否有助于程序员写出安全可靠的程序？这一问题
在未来许多年都会是语言设计的一个重要关注点
- 由于语言承载的功能越来越多，设计时需考虑的问题越来越多，新语言正在变得越来越复杂，语言的实现需要做的工作也越来越多（基本处理、对开发过程的支持、库等等），设计一种语言，支持所有需要变得越来越困难。

2.3 设计准则

- 频度准则：越常用越简单
- 结构一致：程序结构和计算的逻辑结构一致,可读、方便
- 局部性 Locality:
 - 只有全局变量 Basic
 - 不鼓励全局变量 Pascal, C
 - 无全局变量函数式 Java
- 词法内聚 Lexical Coherence : 变量在使用处就近声明 (Pascal 声明和语句严格分开)

```
((lambda (x y) (let ((x 3.5) (y (+ a 2)))  
  (+ (* x y) ((+ (* x y)  
  (- x y))) (- x y)))  
 3.5 (+ a 2))
```

$$\lambda x. \lambda y. ((x * y) + (x - y) \ 3.5 \ (a + 2))$$

续

- 语法一致性

GO TO (L1, L2, ..., Ln), I $I=\{1..n\}$

GO TO N, (L1, L2, ..., Ln)

ASSIGN Li TO N $N=\{L1...Ln\}$

- 安全性Security

语言—编译系统自动找出安全漏洞，不能弥补也要支持

安全性→强类型，即每个计算操作运算之前类型必须确定

C 留给程序员 过程参数不检查 一般不安全

续

- 正交性和正规性(Orthogonality & Regularity)

正交: 每个语言特征都是独立的, 增减不影响其它

正规: 每一约定或规则无一例外

不正规: 数组不能作返回值, 不能赋值

函数不能做参数

不正交→不正规

续

- 数据隐藏 (Data hiding)

封装,以名字封装内部数据设计者可见使用者不可见

局部性不一定封装, 如: Do 10 I=1, 10, 2

当I=7时 GOTO 20

...

10 CONTINUE

20

R=I 可以, 此时R=7.0

续

- 抽象表达

抽取因子、递归表达、高层模块名、

常量名=常量表达式（易于维护）

先抽象再修饰具体（如同自然语言）

```
static const int maxIndex=MAX_LENGTH_1
```

```
MATHLIB.TRIANG  COS(X)
```

- 可移植性 力图不依赖环境

预定义机制、预处理程序

显示表式和隐式表示

语言一开始就追求无二义性，都希望显式表示

—— 太繁

在不产生二义情况下用约定代替显示表示

—— 简化

```
PI: Constant FLOAT:= 3.14159;
```

```
PI: Constant:=3.14159;
```

聚合和分散表示

{	FORTH		C	
	+	单字长加	+	什么都表示
	D+	双字长加		要求编译有解析能力
	M+	混合加		

```
FORTRAN
  FUNCTION SUM(V, N)
  REAL V(N)
  SUM=0.0
  DO 200 I=1, N
  SUM=SUM+V(I)
200 RETURN
END
```

```
Ada
function SUM(V:VECTOR, N:INTEGER) return FLOAT;
```

2.4 程序设计语言规格说明

——语言参考手册

2.4.1 语法规格说明

形式语法：以形式结构规则的语言元素组合规则

微语法 词法 Lexicon

宏语法 定义特征的规则

语言的字符集

- 为了定义程序的形式，一个语言必须选定一个基本的字符集合
- 字符集：允许出现在语言的程序里出现的字符的全体
- 实例：
 - Java采用Unicode字符集，ASCII之外的字符可用于注释、标识符、字符和字符串字面量
 - C未规定字符集，要求基本字符集包含大小写字母、数字和29个特殊字符，其他字符可用在注释、字符和字符串常量里，效果由具体实现解释
- 程序语言中最低级的形式单元是词法元素
 - 词法元素就是程序语言里的“单词”
- 在字符序列观点向上一层，是把一个程序看着一些词法元素的序列
 - 由程序的字符序列得到词法元素序列的过程就是词法分析

词法元素

- **标识符**：文字形式的词法对象，用于表示
 - 语言里的关键字
 - 程序对象的名字
- **关键字**：语言规定了特殊意义的标识符，如 C 中的 if, while, for
- **保留字**：语言中规定了特殊意义，而且不允许程序员用于其他用途的标识符。C 语言的关键字都是保留字，Fortran 没有保留字
- **运算符**：有预定义意义的特殊字符或特殊字符序列（可能有标识符）
 - 运算符的语义就是语言定义的运算（操作），如 Java 的 new
 - 常规语言用运算符表示各种算术、逻辑运算
- **分隔符**：用于分隔程序里的不同词法元素的特殊符号或标识符
 - 空格，换行和制表符等，通常作为语法元素的分隔符

词法和词法分析

- 构成程序的基本词法元素包括标识符、运算符、字面量、注释等。复杂的词（标识符、各种字面量）需要明确定义的构词法，即词法
- 处理源程序的第一步是词法分析：
 - 编译器处理表示源程序的字符序列，根据词法规则做词法分析，将源程序切分为符合词法的段，识别出源程序的有意义单词（**token**）
 - 词法分析得到一个（表达被分析的源程序的）单词流，流中各单词标明了词法类别（是“标识符”，“整数”，“加号”，“左括号”，等等）
 - 词法分析中抛弃所有无保留价值的分隔符（如空白符）和噪声词

例：对 **int main (void) { return 0; }** 做词法分析得到的单词序列是：

"int" "main" "(" "void" ")" "{" "return" "0" ";" "}" 共10个单词

类别：标识符，左/右圆括号，左/右花括号，整数，分号

词法分析

- 词法分析通常采用**最长可能原则**，确定可能成为单词的最长字符串。

例：**staticint**是一个标识符，而不是关键字**static**和**int**

x+++y（C语言）的分析结果是：**x**，**++**，**+**，**y**；而不是**x**，**+**，**++**，**y**

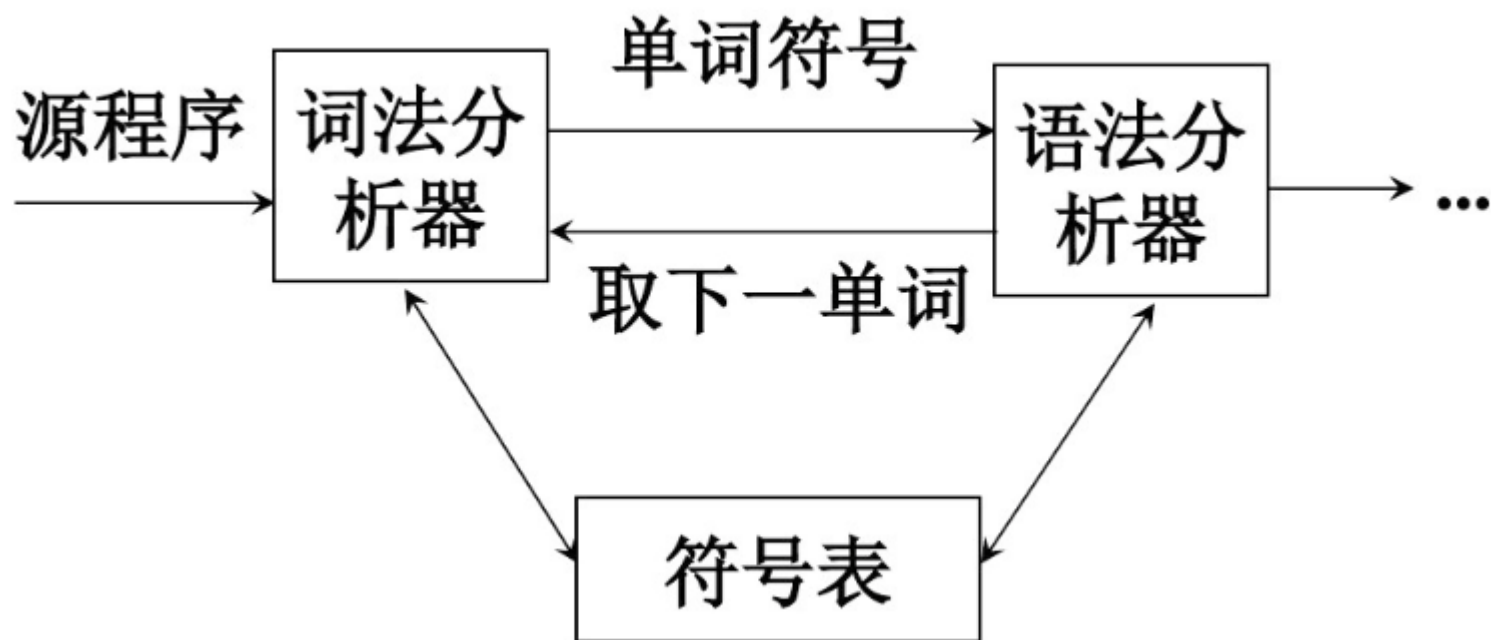
x+++++y是**x**，**++**，**++**，**+**，**y**（语法错，**++**的运算对象非左值）

早期的**Fortran**中存在复杂的词法问题。例如：**DO 10 I = 1.5**
意思类似于C语言的**DO10I = 1.5**

而**DO 10 I = 1,5**是枚举循环头部，
类似于C的**for (I = 1; I < 5; ++I) ...**

后来的语言都避免了这类问题，保证单词很容易识别（能局部识别）
人们已经开发了许多帮助构造词法分析器的自动工具，如**lex/flex**等

词法分析器



语法

- 语法规定位于词法层次之上的程序结构。例如：
 - 表达式的合法形式
 - 基本语句的形式，组合语句的构成方式
 - 更上层的结构，如子程序、模块等的构成形式
- 语法用于确定一个输入序列是否合法的程序。但什么是“合法”？
- 程序存在多个不同层次的合法性问题：
 - 局部结构 例：C程序里的if 之后是不是左括号，括号是否配对
 - 上下文关系 例：变量使用前是否有定义，使用是否符合类型的要求
 - 深层问题 例：使用变量的值之前，变量是否已经初始化

语言的语法定义通常只描述**1**（程序形式中的上下文无关部分）

编译程序通常检查条件**1**和**2**，有人称**2**为“静态语义”

2.4.1.1 文法

文法 产生符合语法的语言（句子集合）规则，如：

$$G=(S,N,T,P) \quad S \in N, T \cap N = \Phi^*$$

- T 是终结符号串的有限集。 N 是非终结符号串的有限集。
- $T \cap N = \Phi$ ，即它们是不相交的。 S 是起始符号串， $S \in N$ 。
- P 是产生式，一般形式是：

$$\alpha \rightarrow \beta \quad \alpha, \beta \in (T \cup N)^*$$

- “ \rightarrow ”表示左端可推导出右端，
- 如 $\alpha \rightarrow \beta$ ， $\alpha \rightarrow Y$ ， $\alpha \rightarrow \delta$ 则可写为： $\alpha \rightarrow \beta | Y | \delta$
- 如果产生式将语言的非终结符中的每一个标记都推得为终结符号，则这一组产生式集即为该语言的全部文法。

续

整数的产生式表示法:

设 $\langle \text{digit} \rangle \rightarrow 0|1|2|3|4|5|6|7|8|9$

则 $\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle$

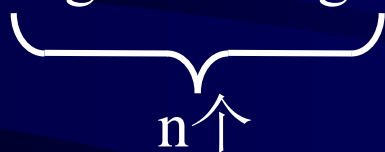
一位数是整数

$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle$

两位数也是

$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle \dots \langle \text{digit} \rangle$

n位数也是


n个

这势必造成产生式臃肿, 如果写成:

$\langle \text{Integer} \rangle \rightarrow \langle \text{digit} \rangle | \langle \text{Integer} \rangle \langle \text{digit} \rangle$
 $| \langle \text{digit} \rangle \langle \text{Integer} \rangle$

续

Chomsky的四种文法

产生式左符号集 \rightarrow 右符号集 由左符号集推导出右符号集

- 0型文法

$\alpha \rightarrow \beta \quad \alpha \in (N \cup T)^+, \beta \in (N \cup T)^*$

递归可枚举语言 图灵机可以识别

- 1型文法

$\alpha A \beta \rightarrow \alpha B \beta \quad \alpha, \beta \in (N \cup T)^*, A \in N, B \in (N \cup T)^+$

上下文相关文法 上下文敏感语言 线性有界自动机可识别

- 2型文法

$A \rightarrow \alpha \quad \alpha \in (N \cup T)^*, A \in N$

上下文无关文法语言 非确定下推自动机可识别

续

- 3型文法 $A \rightarrow \alpha B \mid B\alpha \quad \alpha \in T^*, A, B \in N$

正则文法 正则语言 有限自动机可以识别

可消除右端非终法符 P可以成为终结符表达式

例: $N = \{S, R, Q\}, T = \{a, b, c\}$

$P = \{S \rightarrow Ra, S \rightarrow Q, R \rightarrow Qb, Q \rightarrow c\}$

则有 $S \rightarrow Ra \rightarrow Qba \rightarrow cba \mid S \rightarrow Q \rightarrow c$

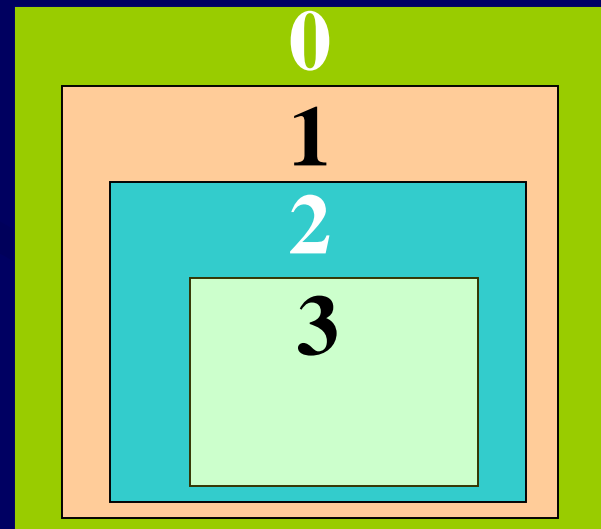
$R \rightarrow Qb \rightarrow cb$

$Q \rightarrow c$

简单语言用 3型, 汇编, 词法子语言

最常用 2型

0、1型无法判定二义性, 目前无法实现



2.4.1.2 上下文无关文法

文法的递归表示法

$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{integer} \rangle \rightarrow \langle \text{digit} \rangle$

一位数

$\langle \text{integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle$

二位数

$\langle \text{integer} \rangle \rightarrow \underbrace{\langle \text{digit} \rangle \dots \langle \text{digit} \rangle}_{n \text{ 个}}$

n 位数

n个

$\langle \text{integer} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{integer} \rangle \langle \text{digit} \rangle$ 左递归

$\langle \text{digit} \rangle \langle \text{Integer} \rangle$ 右递归尾递归

2.4.1.3 BNF 和EBNF

BNF: ::=代替 \rightarrow BNF表达能力同EBNF
<> 指示非终结 终结符直接写出(或黑体)
| 或者

有扩充: [] 括号内容是可选的
{ } 括号内容可重复0至多次

或扩充: C+ ‘Kleene加’ C可重复1至多次
C* ‘Kleene星’ C可重复0至多次

续

BNF示例

$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle$
 $\quad \quad \quad | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$

$\langle \text{integer} \rangle ::= +\langle \text{unsigned integer} \rangle$
 $\quad \quad \quad | -\langle \text{unsigned integer} \rangle$
 $\quad \quad \quad | \langle \text{unsigned integer} \rangle$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle$
 $\quad \quad \quad | \langle \text{idenfitier} \rangle \langle \text{digit} \rangle$
 $\quad \quad \quad | \langle \text{identifier} \rangle \langle \text{letter} \rangle$



$\langle \text{integer} \rangle ::= [+ | -] \langle \text{unsigned integer} \rangle$
 $\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{digit} \rangle | \langle \text{letter} \rangle \}$
 $\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle^+$
 $\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{digit} \rangle | \langle \text{letter} \rangle \}$

续

EBNF: 左端取消<>, 空白加 ‘-’
减少递归表示再加 ‘(’, ‘)’, ‘.’,
尽量用正则表达式
终结符号加 ‘ ’ 号或黑体

续

program ::= <program-heading> ';' ' <program-block> '.

program-heading ::= 'program' <identifier>
['(' <program-parameters> ')']

program-parameters ::= <identifier-list>

identifier-list ::= <identifier> {', ' <identifier>}

program-block ::= <block>

block ::= <label-declaration-part> <constant-declaration-part>
<type-declaration-part><variable-declaration-part>
<procedure-and-function-declaration-part><statement-part>.

variable-declaration-part ::= ['var' <variable-declaration> '; ']
{<variable-declaration> '; ' }

variable-declaration ::= <identifier-list> '; ' <type-denoter>

statement-part ::= compound-statement.

续

compound-statement ::= 'begin' <statement-sequence> 'end'

statement-sequence ::= <statement> {'; ' <statement>}

statement ::= [<label> ': '](<simple-statement> | <structured-statement>)

**simple-statement ::= <empty-statement> | <assignment-statement> |
<procedure-statement> | <goto-statement>**

**structured-statement ::= <compound-statement> | <conditional-statement>
| <repetitive-statement> | <with-statement>**

2.4.1.4 语法图

同EBNF

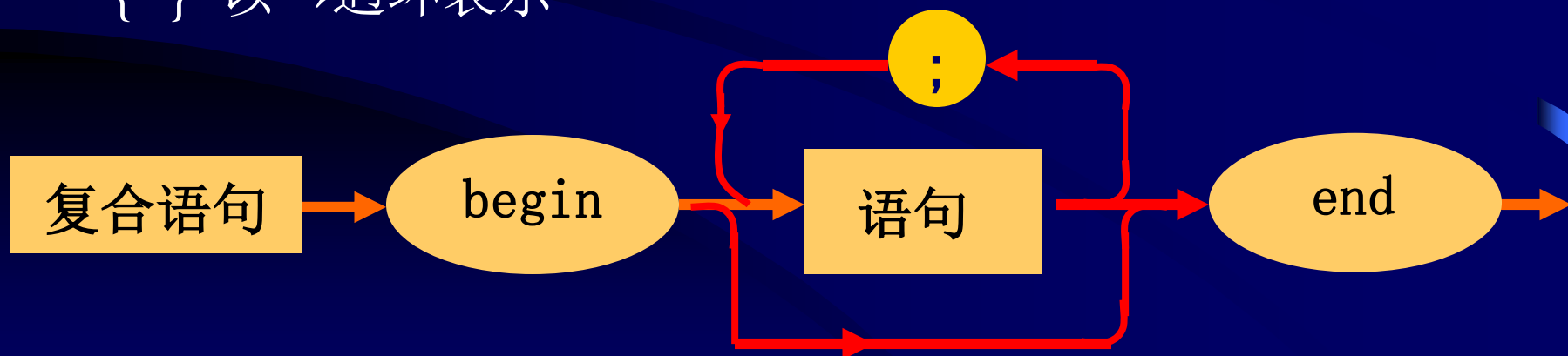
非终结符

终结符

终结符

走向

取消[] 以→短路表示
{ } 以→迴环表示



2.4.1.5 语法分析

- 语法规格说明定义了该语言程序合法的特征和语句。语言处理器则通过语法分析接受合法的程序，这就叫做程序释义（Parsing a Program），释义过程是产生式生成句子的逆过程。
- 语法分析的算法可归为两类：
 - “自顶向下” 释义则从文法的起始符开始，按可能产生的表达式寻找语句相同的结构匹配。每一步都产生下一个可能的源符号串，找到再往下走。
 - “由底向上” 释义则相反，它先查找源代码的各个符号串，看它能否匹配归结为产生式左边的非终结符，如果有含混则向前多读入k个符号串，为此归约下去，一个短语一个短语，最后到达起始符号串，归约的过程就形成了释义树。

Begin x ; = 17 ; writeln (x) end

标识符

无符号整数

过程标识符

标识符

变量标识符

无符号数

变量标识符

完整变量

无符号常量

完整变量

变量访问

因子

变量访问

项

因子

简单表达式

项

表达式

简单表达式

表达式

赋值语句

write参数

简单语句

writeln参数表

语句

过程语句

简单语句

语句

语句序列

复合语句

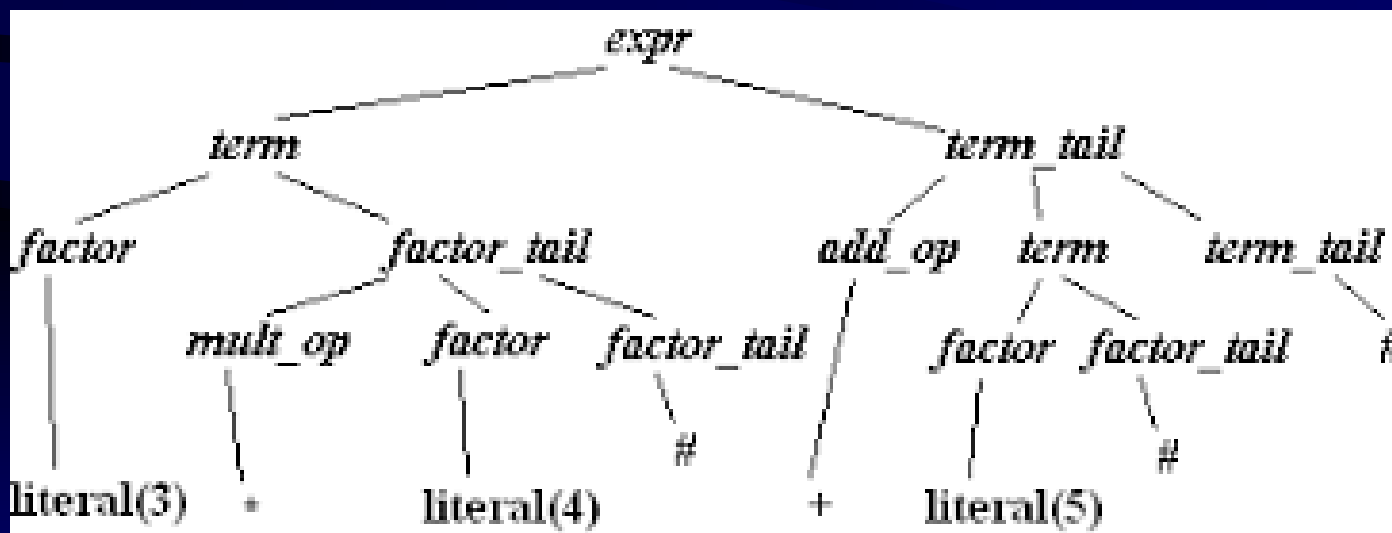
2.4.1.5 语法分析

语法分析判断输入是否为合语法的程序，并识别出源程序的结构

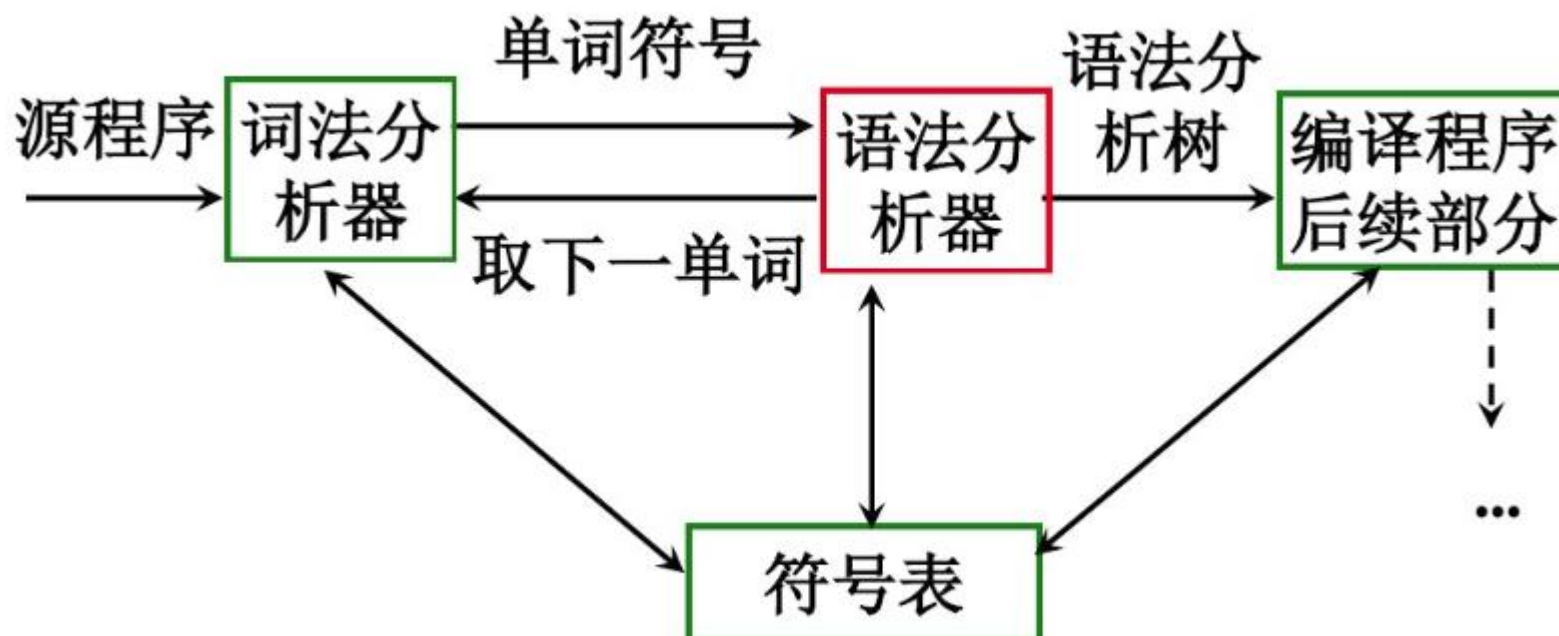
- 输入：来自词法分析器的单词序列和各单词的词法类别信息
- 输出：某种清晰地表达了源程序的内部构造的表达形式

一种常用的清晰直观的语法结构表达形式是语法分析树

例， $3 * 4 + 5$ 的语法分析树：



语法分析器



2.4.1.5 语法分析

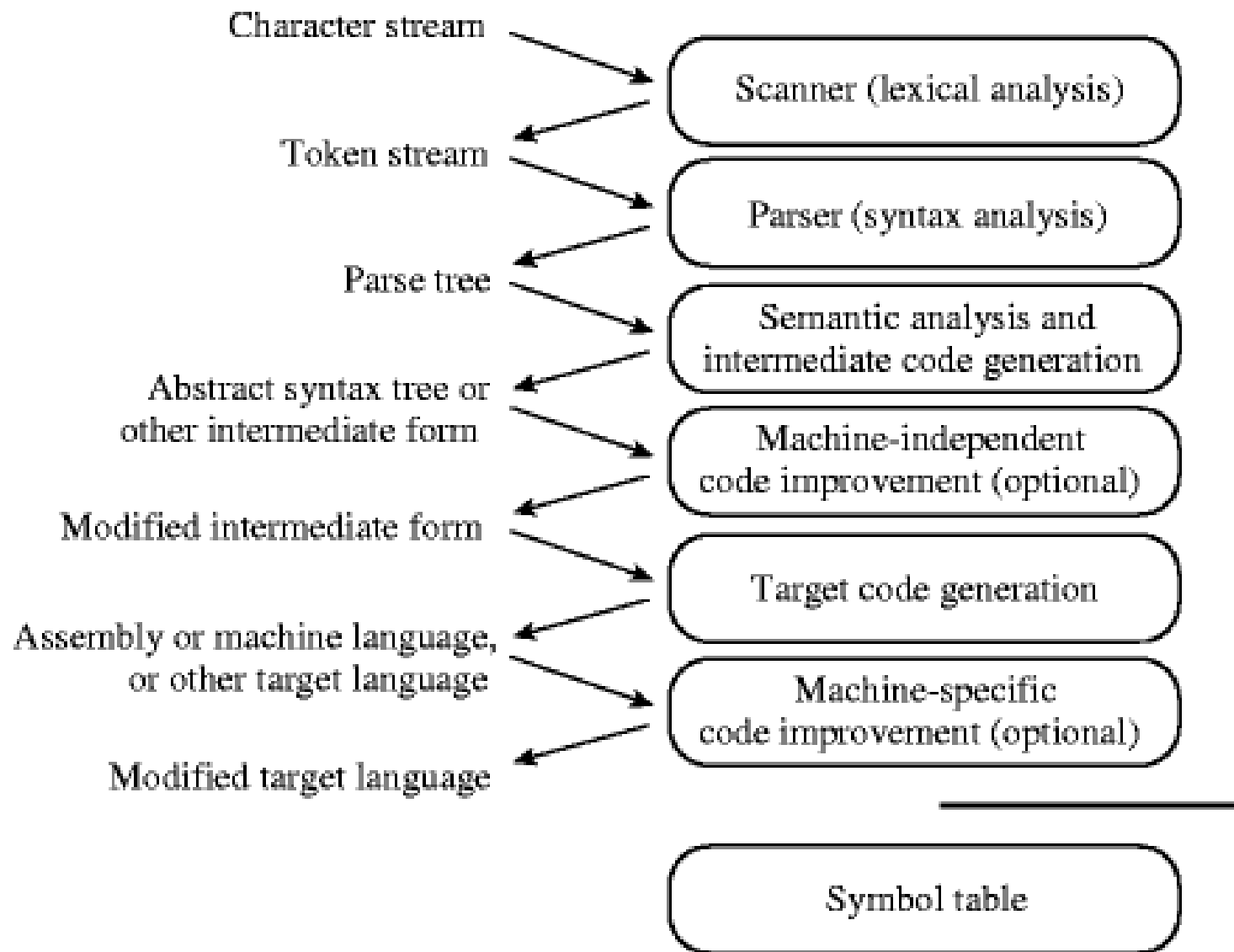
语法分析是语言翻译过程的一个重要阶段。语法分析技术是“编译原理”或“编译技术”课程的重要内容

本课程不准备详细介绍语法分析的技术，词法分析和语法分析过程并不仅用在编译系统里。

许多计算机软件的用户输入都需要做词法分析和简单的语法分析今天的大部分语法分析器都是用工具生成的。例**yacc/bison**。

这些工具接受某种类似**BNF**的语法描述，生成对应的语法分析器人们为各种主要语言（**C/C++/Java**等）开发了支持做词法和语法分析的工具或库，有些脚本语言本身就带有词法/语法分析库，可直接使用

常规的语言编译过程



词法分析

语法分析

语义分析和中间代码生成

与机器无关的优化

目标代码生成

机器特定的优化

2.4.2 语义定义与规格说明

语义定义：目标是赋予每个合法的程序一个明确的意义

- 精确定义程序执行的效果（行为，结果……）
- 要求编译程序（等）生成这种效果
- 写程序的人可以依赖于这种效果，据此写自己的程序

语义定义的基本方法是基于语言的语法结构：

- 定义语言中各种基本元素的意义
- 定义语言中各种语法结构的意义

基于相应结构的成分的意义，定义整个结构的意义一些具体情况：

- 表达式的意义是其怎样求值，求出什么值
- 语句的意义是其执行时的产生的效果

2.4.2 语义定义与规格说明

Algol 60 修订报告

用**BNF** 定义语言的语法，

采用自然语言形式的说明和实例的方式，非形式地定义语言的语义

Backus 在讨论**Algol 60** 时说：

现在我们已经有了办法严格地定义语言的语法了，希望今后不久就能严格地定义语言的语义。

很遗憾，至今的语言手册仍一直沿用上述方式。因为：

- 语义远比语法复杂，至今计算机工作者还没有开发出一种完美、功能强大、易理解、易使用，适用于定义一切语言中一切结构的语义描述方式
- 但是，对于程序语义的研究已经得到许多成果。有关程序语言语义的研究是计算机科学的一个重要研究领域

1、静态语义：类型的理论

类型是程序设计语言的一个重要问题

程序语言需要有清晰定义的类型系统，以确定程序里

- 各表达式的类型（**typing**）。注意，表达式可能嵌套
- 各语句是否类型良好的（**well-typed**），或称为良类型的
- 处理程序中与类型有关的各种问题：类型等价、类型相容、类型转换（自动转换规则，手工转换规则）等

有关类型的研究形成了计算机科学中一个重要的研究领域：类型理论

1、类型环境和类型断言

类型判断的根据：

- 每个文字量都有自己的确定类型
- 变量、函数等的声明提出了相关的类型要求
- 对类型合适的参数，运算符、函数将给出特定类型的结果（注意：重载的运算符也有运算结果和运算对象之间的关系）

类型检查需要知道程序里所有变量的类型信息。当前所有可见变量的类型信息的全体构成了当前的类型环境。一个变量的类型信息用变量名和类型的对表示，如 x 具有类型 T ：

$$x : T$$

类型环境就是这种对的序列，如：

$$x_1 : T_1, x_2 : T_2, \dots, x_n : T_n$$

下面用 Γ （可能加下标）表示类型环境

1、类型推导

我们用形式化的记法表示有关类型的断言。类型断言有两种形式：

$\Gamma \vdash e : T$ 在特定类型环境 Γ 下，我们确定了表达式 e 的结果类型是 T （表达式：常量、变量、有结构成分的复杂表达式）

$\Gamma \vdash c : \text{ok}$ 在环境 Γ 下命令（语句） c 是类型良好的（基本语句如赋值，各种复合语句和更复杂的程序结构）

为了作出程序中各种结构是否类型良好的判断，确定有类型结构（表达式）的具体类型，要定义一套做类型推断的规则（类型规则），说明

- 如何得到程序中基本表达式的类型
- 基于表达式的部分的类型推出整个表达式的类型
- 然后基于表达式的类型，推断基本语句是否类型良好；
- 基于基本语句的良好类型性得到复合语句的良好类型性
- 不能推导出类型的表达式是类型错误的表达式，对语句也一样

1、类型规则

下面用一个简单语言介绍类型规则的一些定义。假定语言里有
int 和 **double** 类型

变量声明，赋值语句和结构语句

类型规则包括（例子）：

基本规则（1）

```
⊢ 0 : int
.....
⊢ 0.0 : double
.....
Γ, x : T ⊢ x : T
```

基本规则（2）

$$\frac{\Gamma \vdash e : T \quad y \text{ is not in } e}{\Gamma, y : T_1 \vdash e : T}$$

1、类型规则

表达式的类型规则（一组）：

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

.....

语句的类型规则（一组）：

$$\frac{\Gamma \vdash x : \text{int} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash x := e ; : \text{ok}}$$
$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash s_1 : \text{ok} \quad \Gamma \vdash s_2 : \text{ok}}{\Gamma \vdash \text{if } (b) \text{ then } s_1 \text{ else } s_2 : \text{ok}}$$

.....

变量声明：

$$\frac{\Gamma \vdash T \text{ } x ; : \text{ok} \quad \Gamma, x : T \vdash c : \text{ok}}{\Gamma \vdash T \text{ } x ; c : \text{ok}}$$

其他规则可以类似写出

2、语义基础：环境和状态

考虑程序的（动态）语义（程序执行的语义）

- 希望用清晰严格的方式描述程序执行中各种动作的效果

对程序动态行为的理解和严格描述依赖于环境的概念

- 程序执行过程中存在着一些实体（变量、子程序等）
- 可能存在与这些实体相关的信息（例如，变量的值，函数的定义） 程序里动作的执行可能改变与一些实体相关的信息，如改变变量的值
- 环境记录与各种有效实体相关的信息
- 因此，动作的执行可能改变环境

讨论语义时关心的是程序的动态环境（运行中的环境）

编译（和类型检查）工作中维护的符号表是支持语言处理的“静态环境”

最简单的环境模型：从合法名字集合到“值”的映射（是一个有限函数）

2、环境和状态

程序执行的动作可能依赖于环境，或导致环境的变化：

- 需要某变量的值时，从当时环境里取出它的值
- 一些操作修改环境中变量的值，如赋值、输入等
- 进入或退出作用域（如进入/退出子程序或内部嵌套的复合语句）
可能改变当前有定义的变量集合（改变环境的定义域）
- 程序开始运行前，需要建立一个初始全局环境，其中包含着所有具有全局定义的名字及与之相关的默认信息

下面把环境建模为变量名到其取值的简单的有限函数

- 如果研究的是更复杂的语言，这种简单形式就可能不够了，需要考虑更复杂的环境模型
- 例如包含指针和动态存储分配/释放，或面向对象的语言（OO 语言），就需要为更复杂的环境模型

在程序运行中，当时环境中所有可用变量的取值情况构成了运行时的状态

2、环境和状态

例：设当时环境中可见的变量是 x, y, z ，取值情况是 x 取值3， y 取值5， z 取值0
这个状态就是有限函数：

$$\{x \mapsto 3, y \mapsto 5, z \mapsto 0\}$$

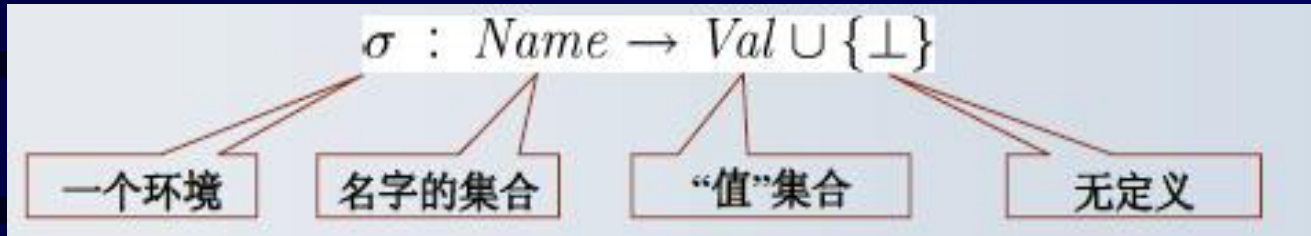
随着程序的执行，特别是赋值语句的执行，环境的状态就可能改变
经过给 x 赋值4，给 y 赋值1的两个操作，状态将变成：

$$\{x \mapsto 4, y \mapsto 1, z \mapsto 0\}$$

理解一个（或一段）程序的意义，就是理解它在有关环境下的意义，以及它的执行对环境的作用

2、环境和状态

严格些说，一个环境是一个“部分函数”：



为描述状态变化，在环境上定义了一种覆盖操作：

$$\sigma \oplus \sigma'(x) = \begin{cases} \sigma'(x) & \text{if } x \in \text{dom } \sigma' \\ \sigma(x) & \text{otherwise} \end{cases}$$

例：

$$\{x \mapsto 1, y \mapsto 2, z \mapsto 3\} \oplus \{x \mapsto 0\} \\ = \{x \mapsto 0, y \mapsto 2, z \mapsto 3\}$$

2、环境和状态

对于常量（如整数），任何环境都给出它们的字面值
环境确定一组变量的值，就确定了基于这些变量构造的表达式的意义
假定

$$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

我们有：

$$\sigma(x) = 1 \quad \sigma(y) = 2 \quad \sigma(z) = 3$$

对于一般表达式：

$$\begin{aligned}\sigma(e_1 + e_2) &= \sigma(e_1) + \sigma(e_2) \\ \sigma(e_1 * e_2) &= \sigma(e_1) \times \sigma(e_2) \\ \dots\dots\end{aligned}$$

由于环境可以确定表达式的意义（确定表达式的值）。我们也可以把环境看成从合法表达式到值集合的映射，

$$\sigma : Exp \rightarrow Val \cup \{\perp\}$$

有些表达式的值无定义，例如其中出现了在环境取值为无定义的变量，或者求值中出现无定义运算结果（如除数为0）等

3、语义规格说明

我们把程序的意义定义为它们对环境的作用。为定义语义需要：

- 定义基本语句的意义
- 定义各种结构的意义如何由它们的组成部分得到

有了这些定义，就可以确定由这些语言结构构成的复杂程序的意义
一个程序的执行导致环境的变化，问题是如何严格定义这种变化

人们提出了多种不同的形式化的语义定义方式（技术）：

- 操作语义：把程序运行看成执行了一系列改变环境的操作，用一组描述环境变化的规则定义程序中各种结构的意义
- 公理语义：用谓词描述状态，用公理和推理规则描述各种基本操作和组合结构的语义
- 指称语义：把程序的意义映射到某种的数学结构，这种结构能清晰地表达程序对环境的作用和环境的变化
- 代数语义：考虑程序之间的等价关系，间接定义程序的意义

操作语义：格局和规则

首先需要定义格局（**configuration**）。这里定义两种格局：

- 终止格局，就是一个环境，表示代码运行完成时的状态
- 非终止格局： p, σ 表示程序运行中的一个“状态”

一段程序代码（待执行代码段）和一个环境

结构化操作语义（**SOS**）用一组格局变迁规则定义语言的语义。可能包括一些无前提的规则和一些有前提的规则

赋值语句的语义规则（无前提规则）：

$$x := e, \sigma \rightsquigarrow \sigma \oplus \{x \mapsto \sigma(e)\}$$

复合语句的语义规则（有前提规则）：

$$\frac{p_1, \sigma \rightsquigarrow \sigma'}{p_1; p_2, \sigma \rightsquigarrow p_2, \sigma'}$$

操作语义：续

if 条件语句的语义规则（两条）：

$$\frac{\sigma(b) = \text{true} \quad p_1, \sigma \rightsquigarrow \sigma'}{\text{if } b \text{ then } p_1 \text{ else } p_2, \sigma \rightsquigarrow \sigma'}$$
$$\frac{\sigma(b) = \text{false} \quad p_2, \sigma \rightsquigarrow \sigma'}{\text{if } b \text{ then } p_1 \text{ else } p_2, \sigma \rightsquigarrow \sigma'}$$

while 循环语句的语义规则（两条）：

$$\frac{\sigma(b) = \text{true} \quad p, \sigma \rightsquigarrow \sigma'}{\text{while } b \text{ do } p, \sigma \rightsquigarrow \text{while } b \text{ do } p, \sigma'}$$
$$\frac{\sigma(b) = \text{false}}{\text{while } b \text{ do } p, \sigma \rightsquigarrow \sigma}$$

上面这组规则严格地定义了一个理想的小语言的语义
注意这个小语言的特点。它与实际的程序语言还是有很大差距

公理语义

R. Floyd 在1967年考虑如何说明一个程序完成了所需工作时，提出用逻辑公式描述程序环境的状态的思想。这种描述环境的逻辑公式称为断言

逻辑公式：

$$a_1 : x = 1 \wedge y = 2 \wedge z = 3$$

描述了状态：

$$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$$

说明环境满足公式，记为：

$$\sigma \models a_1$$

实际上，这个状态还满足另外的许多逻辑公式，如：

$$a_2 : x = 1$$

$$a_3 : x > 0 \wedge y = 2 \wedge z \geq 2$$

反过来看，一个逻辑公式可以被许多状态满足
因此可以认为，一个公式描述了一个状态集合。例如，上面第二个公式，
描述了所有的其中 x 值是1 的环境

公理语义：断言

把断言写在程序里某个特定位置，就是想提出一个要求：

- 程序执行到这个位置，当时的环境状态必须满足断言
- 如果满足，程序就正常向下执行（就像没有这个断言）
- 如果不满足，程序就应该（非正常地）终止

实例：

C 语言标准库提供了断言宏机制，可以在程序里的写

```
assert(x > 0);
```

Stroustrup 在《**The C++ Programming Language**》里特别讨论了如何定义断言类的问题

有些语言本身提供了断言机制，如**Eiffel**。见《**Design by Contract**》，中译本：邮电出版社

断言的概念在实际程序开发中起着越来越重要的作用

公理语义：前条件和后条件

如何借助于逻辑公式描述一段程序的意义？

Hoare 提出的方法是用一对公式，放在相应程序段之前和之后，分别称为该程序段的前条件（**precondition**）和后条件（**postcondition**）

pre P post

意思是：如果在程序**P**执行之前条件*pre*成立（当时的环境满足*pre*），那么在**P**执行终止时的环境中，条件*post*一定成立

我们可以用一对公式(*pre, post*)描述一段程序的语义，也可以把这样的公式对(*pre, post*)看作是对一段程序的语义要求，看作程序的规范

前后条件的两种基本用途：

- 作为评判程序是否正确的标准
- 作为待开发的程序的规范，研究如何从这种规范得到所需的程序

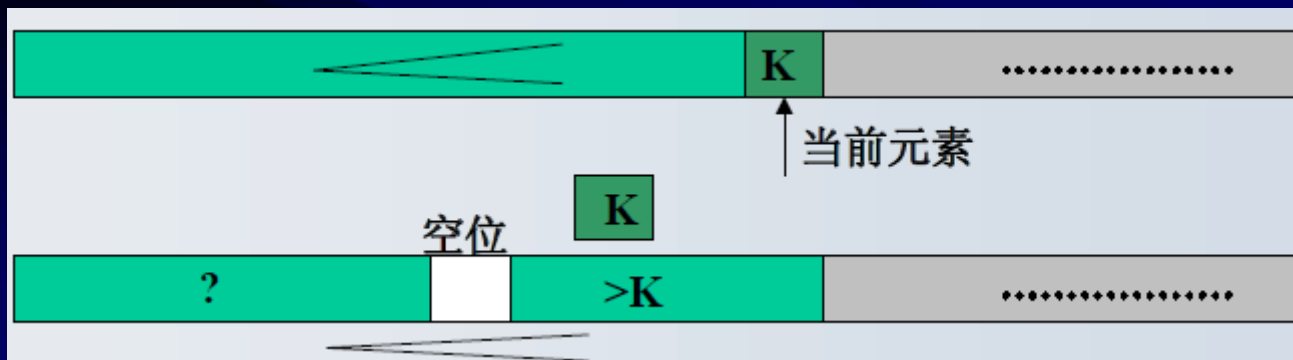
公理语义：不变式

不变式（不变量，**invariant**）的概念在许多领域中有重要地位

- 程序理论中的不变式，指写在程序里特定位置的逻辑公式，要求程序执行中每次达到某个（某些）情况时，这个公式都必须成立
- 其实，不变式并不特殊，因为写在程序里的断言，也就是位于代码中相应特定位置的不变式，“情况”就是执行达到这个位置

程序理论里最重要的不变式之一是循环不变式（**loop invariant**），要求一个循环的每次迭代开始之前成立，在描述和推导程序的意义时有特殊价值

直接插入排序算法两重循环的不变式（图示）：



公理语义：程序的意义

Floyd 最早提出用断言描述程序的意义，通过逻辑推导证明程序的正确性。他称自己提出的方法为“断言法”。基本方法：

- 在流程图程序里的每条边上放一条断言
- 设法证明从一个动作框的入边上的断言出发，执行相应动作产生的效果能保证该动作框的出边上的断言成立
- 对分支控制框也有特殊的规则（请自己考虑）
- 如果对每个动作框都能证明上述事实成立，那么标注在流程图中各条边上的断言就形成了该程序的一个（一套）语义描述
- 特别的，标在程序入口和出口上的断言表示了这段程序的整体效果

Hoare 总结了**Floyd**的工作，针对结构化的程序控制结构提出了一套逻辑推理规则，这就是有名的**Hoare** 逻辑

公理语义：Hoare逻辑

Hoare 逻辑是对有关程序意义的逻辑描述的整理和系统化

Hoare 逻辑中的逻辑公式形式为（称为**Hoare** 三元组）：

$$\{p\} S \{q\}$$

其中**p** 和**q** 是谓词逻辑公式，**S** 是一段程序

直观意义：如果在**S** 执行之前公式**p** 成立，在**S** 执行终止时**q** 就成立

Hoare 的最重要贡献是提出了一套推理规则，通过这些规则，可以把证明一个**Hoare** 公式（程序**S** 相对于**p** 和**q** 的正确性）的问题归结为证明一组普通一阶谓词逻辑公式的问题

Dijkstra 在此基础上提出了最弱前条件（**weakest precondition**）的概念，借助于这一概念，可以

- 把程序的正确性证明工作进一步规范化
- 用于做程序的推导（从规范出发推导程序）

公理语义：Hoare逻辑

公理：

[SKIP]	$\{p\} \text{ skip } \{p\}$
[ASSIGN]	$\{p[e/x]\} x := e \{p\}$
[COMP]	$\frac{\{p\} S_1 \{q\} \quad \{q\} S_2 \{r\}}{\{p\} S_1; S_2 \{r\}}$
[COND]	$\frac{\{p \wedge b\} S_1 \{q\} \quad \{p \wedge \neg b\} S_2 \{q\}}{\{p\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{q\}}$
[LOOP]	$\frac{\{I \wedge b\} S \{I\}}{\{I\} \text{ while } b \text{ do } S \{I \wedge \neg b\}}$
[IMPLY]	$\frac{p \Rightarrow p' \quad \{p'\} S \{q'\} \quad q' \Rightarrow q}{\{p\} S \{q\}}$

规则[**LOOP**]里的***I***是循环不变式

公理语义：Hoare逻辑

可以用**Hoare** 逻辑证明程序的正确性，也就是说，证明三元组中的程序语句“符合”前后条件的要求。其意义是：

- 若在前条件满足的情况下执行语句且语句执行终止，那么后条件满足
 - 这个证明并不保证语句终止，如果语句的执行不终止，什么后条件都可以证明。因此，这样证明的正确性称为“部分正确性”
 - 程序终止性需要另外证明，主要是需要证明其中循环的执行必然终止
 - 如果同时证明程序的部分正确性和终止性，这一程序就是“完全正确的”
- 程序正确性证明中的一个关键点是为各个循环提供适当的不变式

就像做数学归纳法证明中需要合适的归纳假设，过强或过弱都不行人们已证明，循环不变式不可能自动生成（无完全的算法，但有许多研究）。

公理语义为我们提供了许多有助于理解程序行为的概念，为设计程序时思考其行为提供了重要的依据和线索。

指称语义

指称语义学（**denotational semantics**）有坚实的数学基础，它的基本想法由**C.Strachey** 提出，**D. Scott** 完成了它的基础理论并因此获图灵奖

指称语义的基本思想是定义一个语义函数（指称函数），把程序的意义映射到某种意义清晰的数学对象（就像用中文解释英文）

作为被指的数学对象可以根据需要选择

对简单的程序语言，一种自然的选择是把程序的语义定义为从环境到环境的函数（作为指称）。定义语义就是指定每个程序对应的函数

环境的集合：

$$\Sigma \quad \sigma \in \Sigma$$

语义映射：

$$\llbracket \cdot \rrbracket$$

表达式的语义：

$$\llbracket e \rrbracket \in \Sigma \rightarrow \mathbb{Z}$$

假设是整型表达式

命令的语义：

$$\llbracket S \rrbracket \in \Sigma \rightarrow \Sigma$$

指称语义

表达式的语义定义:

$$\begin{aligned}\llbracket n \rrbracket(\sigma) &\hat{=} n \quad n \in \mathbb{Z} \\ \llbracket x \rrbracket(\sigma) &\hat{=} \sigma(x) \\ \llbracket e_1 + e_2 \rrbracket(\sigma) &\hat{=} \llbracket e_1 \rrbracket(\sigma) + \llbracket e_2 \rrbracket(\sigma) \\ &\dots\end{aligned}$$

语句的语义定义:

$$\begin{aligned}\llbracket x := e \rrbracket(\sigma) &\hat{=} \sigma \oplus \{x \mapsto \llbracket e \rrbracket(\sigma)\} \\ \llbracket S_1; S_2 \rrbracket &\hat{=} \llbracket S_1 \rrbracket \circ \llbracket S_2 \rrbracket \\ \llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket(\sigma) &\hat{=} \begin{cases} \llbracket S_1 \rrbracket(\sigma) & \text{if } \llbracket b \rrbracket(\sigma) = \text{true} \\ \llbracket S_2 \rrbracket(\sigma) & \text{if } \llbracket b \rrbracket(\sigma) = \text{false} \end{cases} \\ &\dots\end{aligned}$$

代数语义

代数语义把语义模型的集合看成是一个代数结构，模型簇对应为代数系统。用代数方法描述数据类型 $A=(V, Op)$

specification LISTS

sorts

List

NATURALS

formal sort Component

Operations

empty_list : List

cons(,) : Component, List \rightarrow List

headof_ : List \rightarrow Component

tailof_ : List \rightarrow List

lengthof_ : List \rightarrow NATURALs

variables c: Component

l: List

equations

headof cons (c,l)=c

tailof cons (c,l)=l

tailof empty_list = empty_list

lengthof empty_list = 0

lengthof cons (c,l)=succ (length of l)

end specification

语法、语义和语用

语法：规定合法程序的形式，清晰定义语言中各种结构的形式。上下文关系采用自然语言说明。

语义：规定合法程序的意义，程序执行时所产生的效果。形式语义学探讨形式化定义程序的语义。

语用：程序设计技术，具体语言的使用技术和惯用方法。程序设计技术、技巧和设计模式等。

作业

- 2.4 指出以下说法的正误
- (a)程序设计语言越简短越好。
- (b)复杂的语言可读性都差。
- (c)已有程序设计语言(Pascal, C, Ada, Lisp, Prolog 等)早已实用不存在二义性。
- (d)高级语言由于硬件速度提高快不用追求效率。
- (e)当今语言没有一个在语言的层次上就能保证可移植。
- (f)有副作用的函数百害无一利。

- 2.5 将以下BNF表示的Algol60部分产生式画成语法图
- $\langle \text{unsigned integer} \rangle : : = \langle \text{digit} \rangle$
- $\quad \quad \quad | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$
- $\langle \text{integer} \rangle : : = +\langle \text{unsigned integer} \rangle$
- $\quad \quad \quad | -\langle \text{unsigned integer} \rangle$
- $\quad \quad \quad | \langle \text{unsigned integer} \rangle$
- $\langle \text{decimal fraction} \rangle : : = . \langle \text{unsigned integer} \rangle$
-
- $\langle \text{exponent part} \rangle : : = 10 \langle \text{integer} \rangle$ //10为下标。
-
- $\langle \text{decimal number} \rangle : : = \langle \text{unsigned integer} \rangle$
- $\quad \quad \quad | \langle \text{decimal fraction} \rangle$
- $\quad \quad \quad | \langle \text{unsigned integer} \rangle \langle \text{decimal fraction} \rangle$
- $\langle \text{unsigned number} \rangle : : = \langle \text{decimal number} \rangle$
- $\quad \quad \quad | \langle \text{exponent part} \rangle$
- $\quad \quad \quad | \langle \text{decimal number} \rangle \langle \text{exponent part} \rangle$
-
- $\langle \text{number} \rangle : : = +\langle \text{unsigned number} \rangle$
- $\quad \quad \quad | -\langle \text{unsigned number} \rangle$
- $\quad \quad \quad | \langle \text{unsigned number} \rangle$

作业

- 2.6 将下面的EBNF转换为BNF:
- $S \rightarrow A \{ bA \}$
- $A \rightarrow a [b] A$
-
- 2.7 考虑下列文法:
- $\langle S \rangle \rightarrow \langle A \rangle a \langle B \rangle b$
- $\langle A \rangle \rightarrow \langle A \rangle b \mid b$
- $\langle B \rangle \rightarrow a \langle B \rangle \mid a$
- 下面的哪些句子属于这些文法所产生的语言?
- baab
- bbbab
- bbaaaaa
- bbaab

作业

- 2.9 下面的Pascal程序有错误，识别每一个错误并将它们按语法错、语义错、上下文约束错归类：

- program p;
- a: array 10 of char;
- b: Integer;
- begin
- a [0]:=b;
- c:='*'
- end.

- 2.12 在下面几种情况下，你能举出例子说明哪些二义性是人们希望的，哪些是可容许的？哪些是不希望的？
- (a)人们谈话中的二义性。
- (b)某本书或文章中的二义性。
- (c)交互式查询语言中的二义性。
- (d)程序设计语言中的二义性。