编译原理

Compiler Construction Principles

朱 青

信息学院计算机系，

中国人民大学，

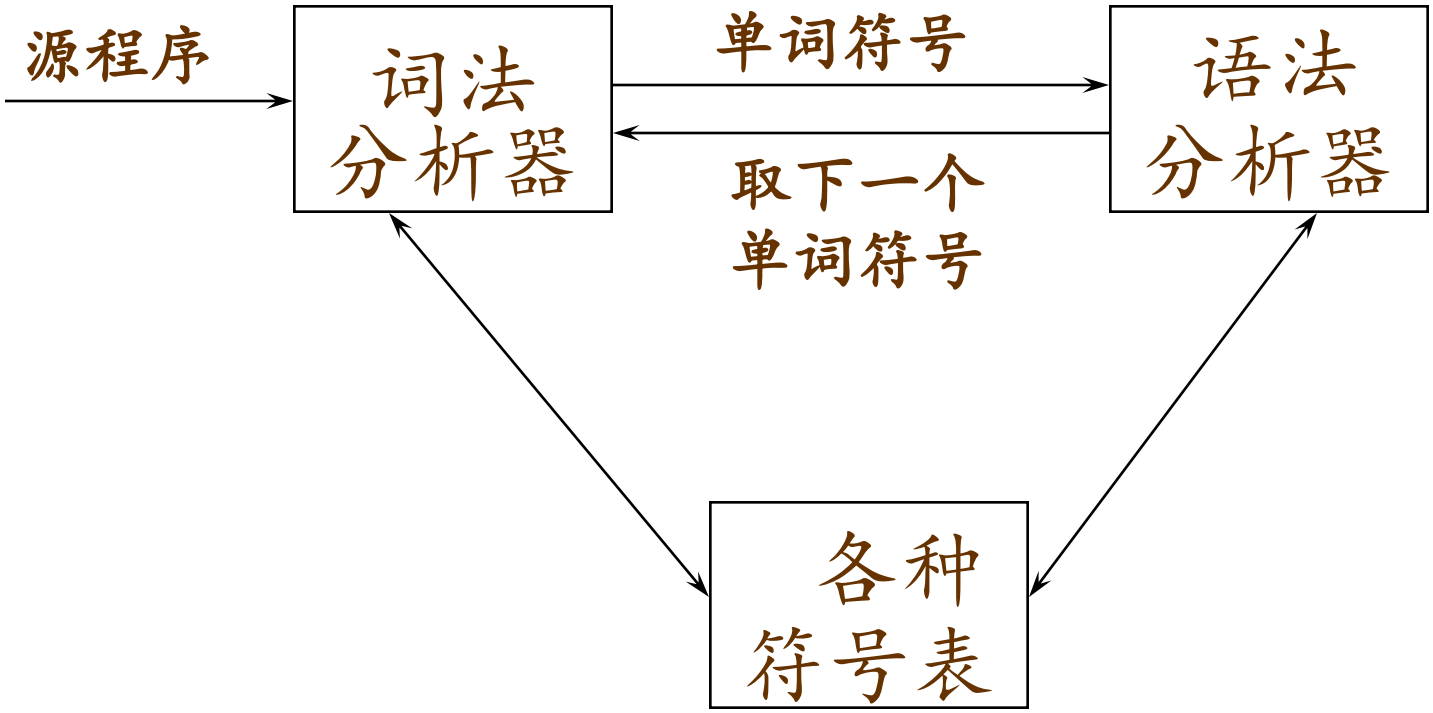
zqruc2012@aliyun.com

第2章:词法分析 (Lexical Analysis)

- 2.1 词法分析程序的功能
- 2.2 词法分析器的设计
- 2.3 正规表达式 (Regular Expression)
- 2.4 有限自动机
- 2.5 词法分析器的自动生成

2.1 词法分析程序的功能

词法分析是编译的基础。



设计与实现

- 设计问题：
  - 分清词法问题和语法问题
  - 确定单词种别和属性设置
  - 目的：方便算法的实现
- 实现问题
  - 单词的分割、识别、拼写合法

2.1 词法分析程序的功能

- 2.1.1 词法分析器的功能和输出形式
- 2.1.2 词法分析器作为一个独立子程序

2.1 词法分析程序的功能

源程序由单词符号组成。如下面的PASCAL程序段（例如2.1-1):

```
while   i  <>  j  do
  if  i  > j  then  i  :=  i  -  j
              else   j :=  j  -  i
```

单词符号是最小的语义单位 。

2.1.1 词法分析器的功能和输出形式

功能： 输入源程序，做词法分析，

输出单词符号及其属性。

单词符号分五种：

1）基本字（保留字，关键字） IF，DO

2）标识符 变量名，函数名

3）常数 125 0.718 TRUE

4）运算符 + - * /

5）界符 , ; （ ）

输出单词符号的形式 ：

二元式：

（单词种别编码，单词自身的值）

说明：单词符号的机内表示可以是多种多样的，依具体情况（需处理的语言、处理的方法）而定。

编码方式:

1）基本字

总归成一种，但一字一种较方便。

2）标识符

可以是一种，也可以按类型分种。

3）常数

按类型分种。

4）运算符

一符一种，或一类符号一种。

5）界符

一符一种，种别编码本身可以代表单词自身的值。

例2.1-1 PASCAL源程序段

经词法分析器分析后,将转换为如下的二元式的单词符号序列:

(WHILE , -----) \\ while i<>j do
(IDENTIFIER , I-PTR)
\\ 指向i的符号表入口的指针
(RELATION_OP, NE)
(IDENTIFIER , J-PTR)
\\ 指向j的符号表入口的指针
(DO , -----)

(IF , -----) \\ if i>j then i:=i-j

(IDENTIFIER , I-PTR)

(RELATION_OP, GT)

(IDENTIFIER , J-PTR)

(THEN,-----)

(IDENTIFIER , I-PTR)

(ASSIGN-OP,-----)

(IDENTIFIER , I-PTR)

(MINUS_OP,-----)

(IDENTIFIER , J-PTR)

(ELSE, -----) \\ else j:=j-i

(IDENTIFIER , J-PTR)

(ASSIGN-OP,-----)

(IDENTIFIER , J-PTR)

(MINUS_OP,-----)

(IDENTIFIER , I-PTR)

说明：对单词符号的种别与属性值是事先设定好的。

例2.1-2 FORTRAN源程序段中的语句

- IF (5.EQ.M) GOTO 100
- 解：（单词种别用整数表示）
  - IF ( )
  - (( )
  - 5 ( )
  - .EQ. ( )
  - M ( )
  -) ( )
  - GOTO ( )
  - 100 ( )

例 2.1-3: 单词符号序列

```
while(*pointer!='\0'){pointer++;}
```

while	**(WHILE, _)**
(	**(SLP, _)**
*****	**(FETCH, _)**
pointer	**(IDN, 符号表入口指针)**
!=	**(RELOP, NE)**
'\0'	**(CONST, 0)**
)	**(SRP, _)**
{	**(LP, _)**
pointer	**(IDN, 符号表入口指针)**
++	**(INC, _)**
;	**(SEMI, _)**
}	**(RP, _)**

2.1.2 词法分析器作为一个独立子程序

- 好处：使编译程序的结构更简洁，清晰和条理化。
- 词法分析器不一定作为单独的一遍,常作为语法分析器的子程序，当需要一个单词时，就调用词法分析器。
- 词法分析器作为单独的一遍时，需将转换的单词符号建立一个单词符号文件，语法分析器通过该文件取单词。

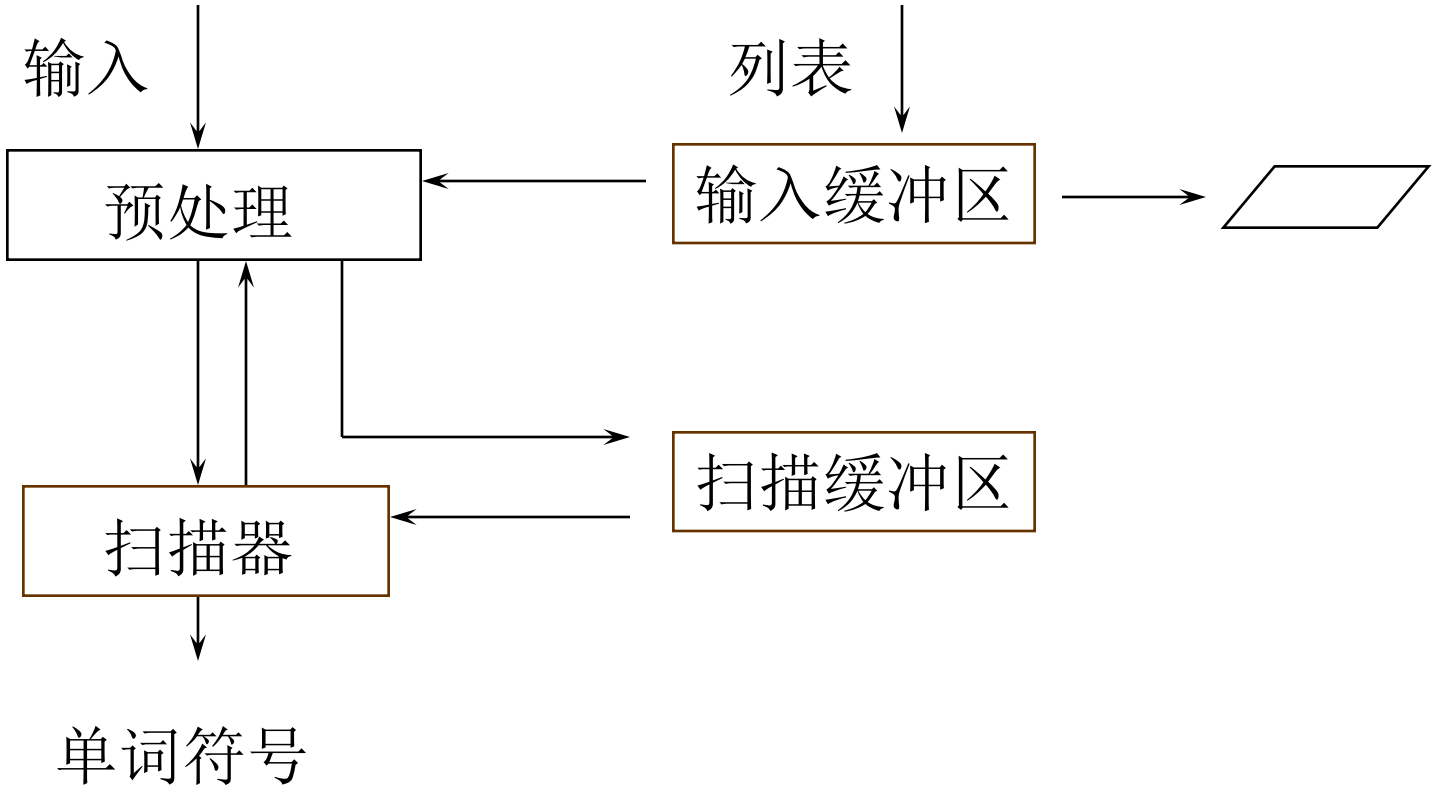
第2章:词法分析 (Lexical Analysis)

- 2.1 词法分析程序的功能
- 2.2 词法分析器的设计
- 2.3 正规表达式 (Regular Expression)
- 2.4 有限自动机
- 2.5 词法分析器的自动生成

2.2 词法分析器的设计

- 2.2.1 词法分析器的结构图
- 2.2.2 单词符号的识别
- 2.2.3 状态转换图 (Transition diagram)
- 2.2.4 状态转换图的实现

2.2.1 词法分析器的结构图



输入，预处理：

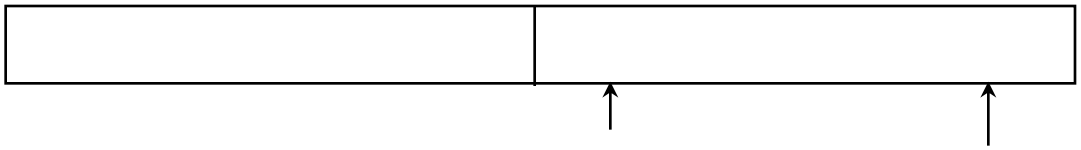
输入： 源程序文本⟶ 输入缓冲区

预处理：（EOF 结束）

— 剔除无用的空白，跳格，回车和换行等。

— 出错信息的列表打印。

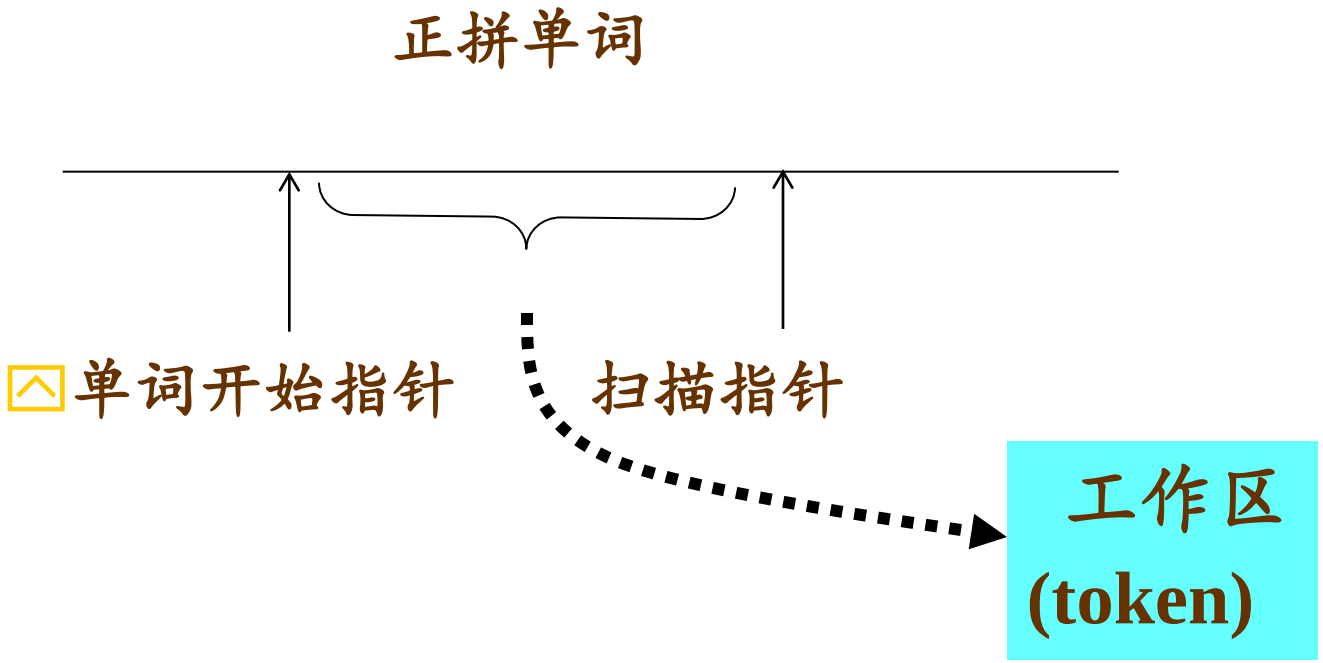
扫描缓冲区：N个字符（1024 或 4096 ）



开始指针　向前指针

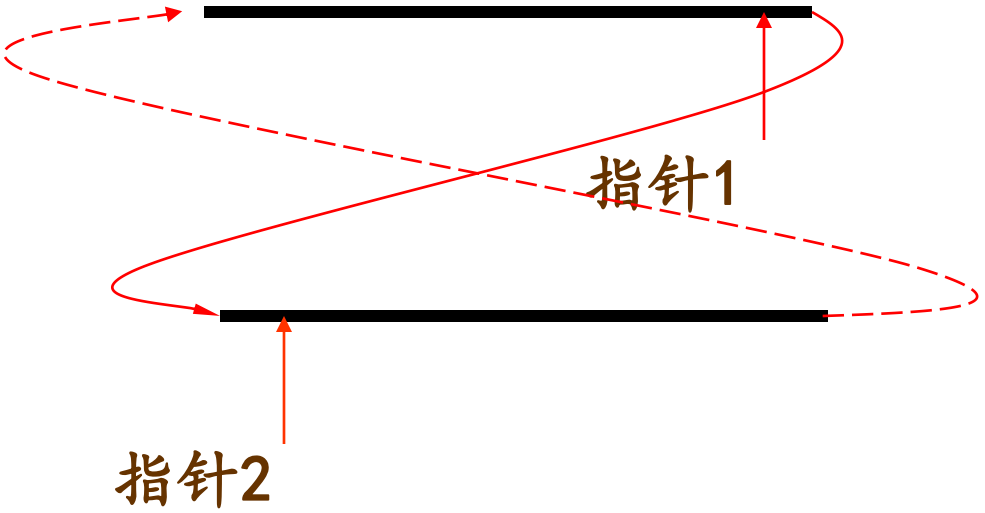
相关问题

- 词法分析器可以作为一遍独立的扫描来安排。
- 输入缓冲区



相关问题

⌘双缓冲区问题



- 功能1：并行
- 功能2：捻接

- ？如何设计和实现扫描器

2.2 词法分析器的设计

- 2.2.1 词法分析器的结构图
- 2.2.2 单词符号的识别
- 2.2.3 状态转换图 (Transition diagram)
- 2.2.4 状态转换图的实现

2.2.2 单词符号的识别

单词符号识别的简单方法-----超前搜索

- 基本字的识别：　　（超前搜索）

例：FORTRAN 的正确语句：

1　　DO99K=1，10

2　　IF(5.EQ.M) GOTO55

3　　DO99K=1.10

4　　IF(5)=55

- 标识符的识别：

  标识符后跟算符或界符.

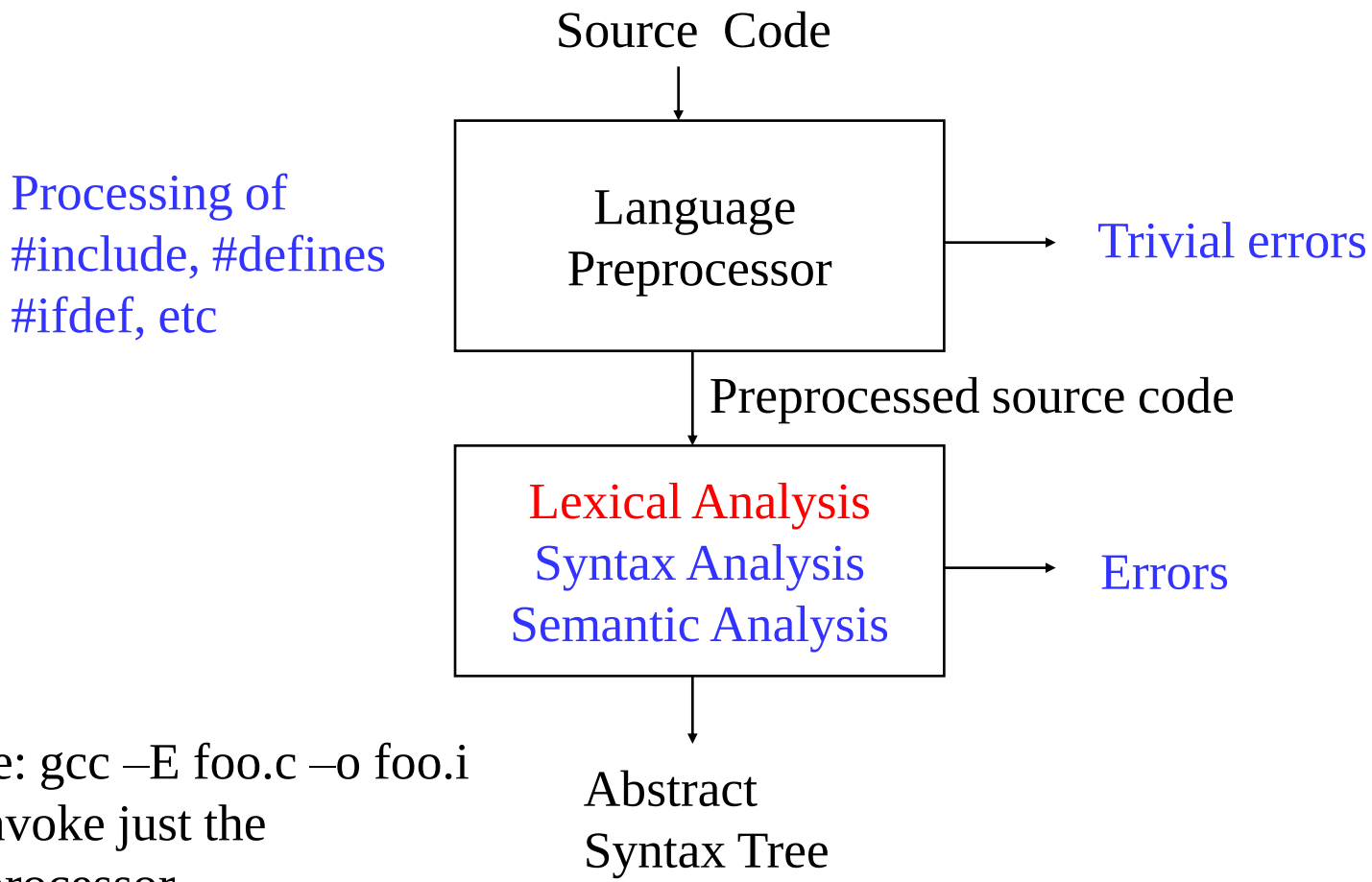
- 常数的识别：

  算术常数　逻辑常数　串常数

- 算符和界符的识别：

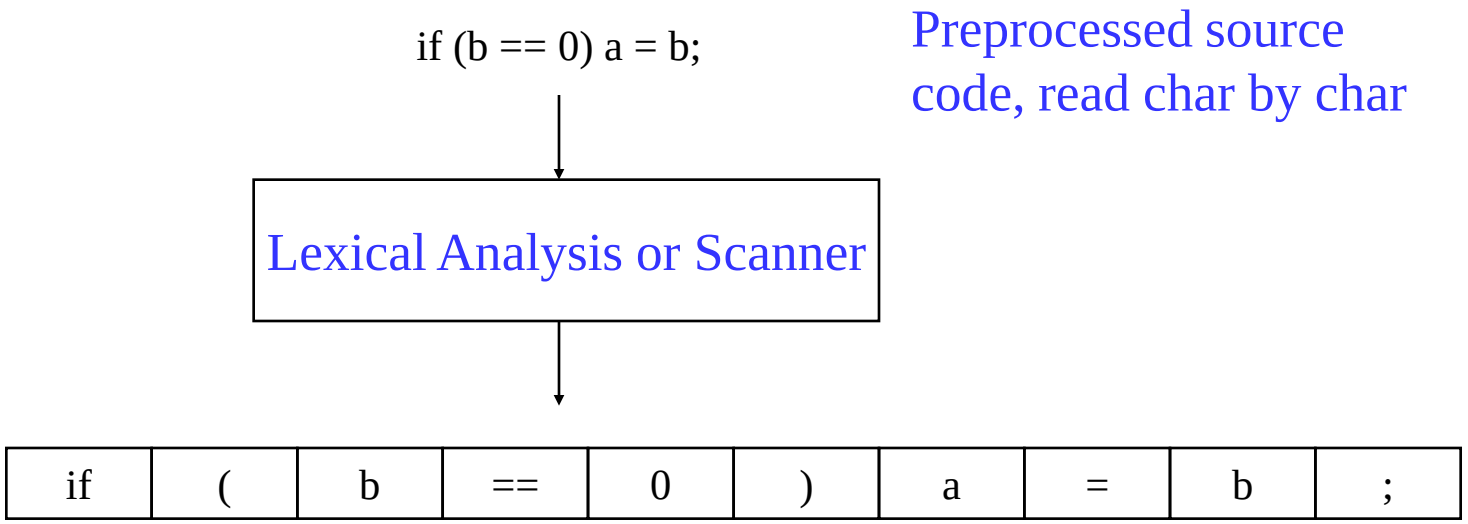
  特殊：　:=　　**　　(*　　*)

Frontend Structure



Note: gcc –E foo.c –o foo.i to invoke just the preprocessor

Lexical Analysis Process



Preprocessed source code, read char by char

Lexical analysis

- Transform multi-character input stream to token stream
- Reduce length of program representation (remove spaces)
-

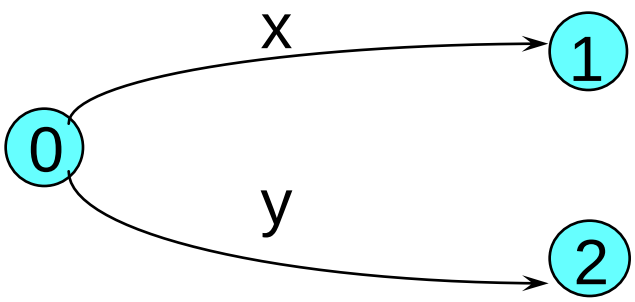
Tokens

- Identifiers: x y11 elsex
- Keywords: if else while for break
- Integers: 2 1000 -20
- Floating-point: 2.0 -0.0010 .02 1e5
- Symbols: + * { } ++ << < <= [ ]
- Strings: “x” “He said, \”I love EECS 483\””
- Comments: /* bla bla bla */

2.2 词法分析器的设计

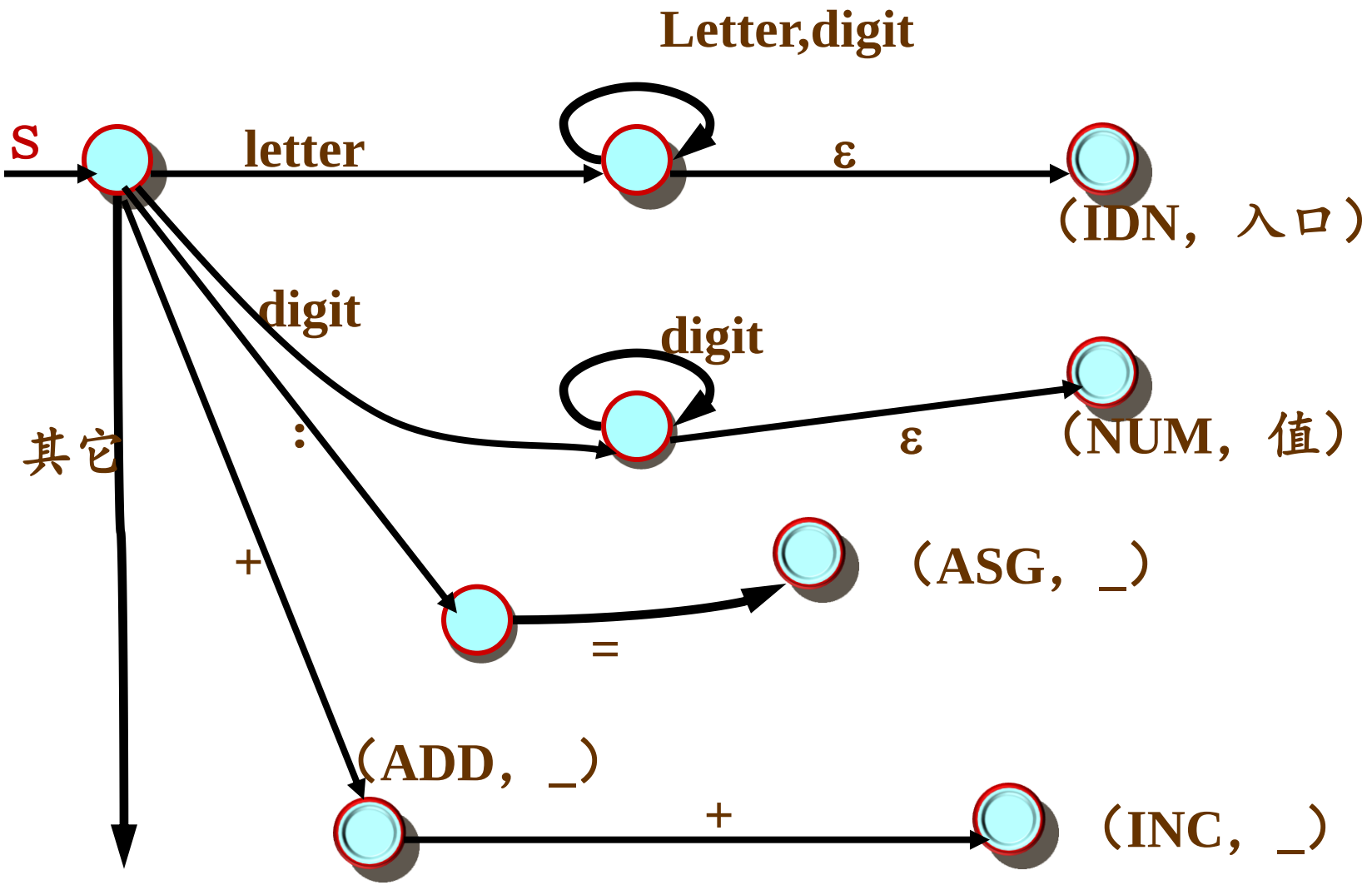
- 2.2.1 词法分析器的结构图
- 2.2.2 单词符号的识别
- 2.2.3 状态转换图 (Transition diagram)
- 2.2.4 状态转换图的实现

2.2.3 状态转换图(Transition diagram)



功能:用于识别(或接受)一定的字符串.

举例2.2-1：状态图


Letter,digit
S
letter
ε
（IDN，入口）
digit
digit
其它
:
ε
（NUM，值）
+
（ASG，_）
=
（ADD，_）
+
（INC，_）

利用状态转换图识别单词

1. 从初态出发
2. 读入一字符
3. 按当前字符转入下一状态
4. 重复 2,3 直到无法继续转移

§. 在遇到读入的字符是Token的分割符时，若当前状态是终止状态，说明读入的字符组成一单词；否则，说明输入不符合词法规则。

说明:状态转换图是一张有限方图:

结点---状态 (圆圈)

箭弧　初态　终态(双圆圈)

例如 2.2-2: 状态图举例 (P41 图3.2)

例 2.2-3 C语言无符号整数的识别

1、 定义式描述

八进制数: （OCT，值）

- oct→0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)*

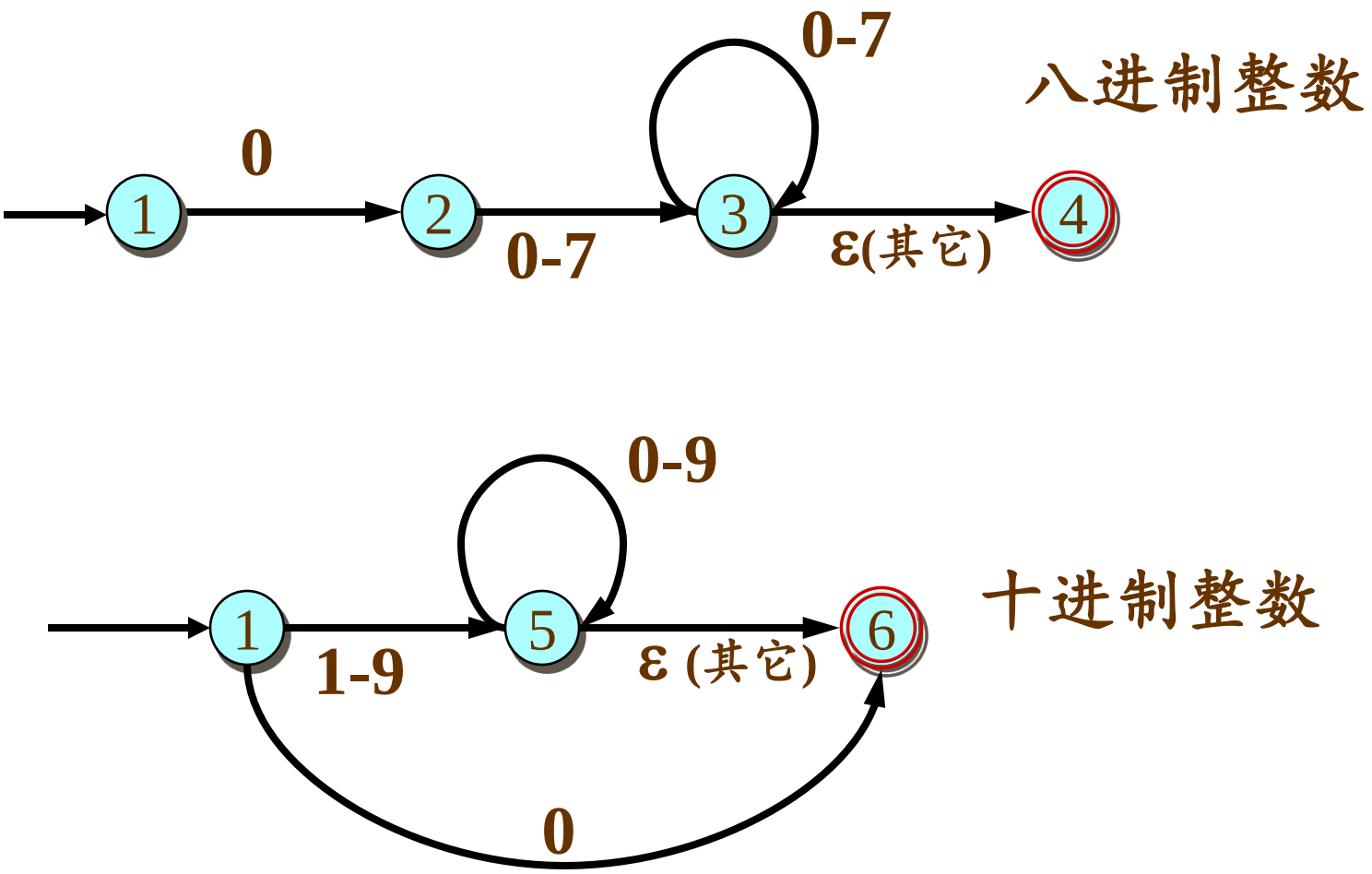
十进制数: （DEC，值）

- dec→(1|...|9)(0|...|9)* |0

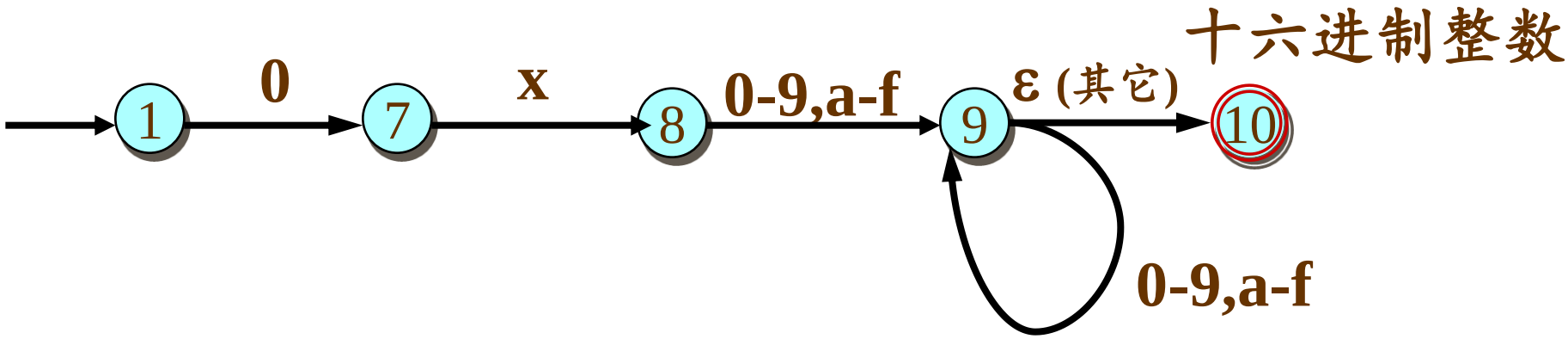
十六进制数: （HEX，值）

- hex→0x(0|1|...|9|a|...|f|A|…|F)(0|...|9|a|...|f |A|…|F)*

2、识别不同进制数的状态图



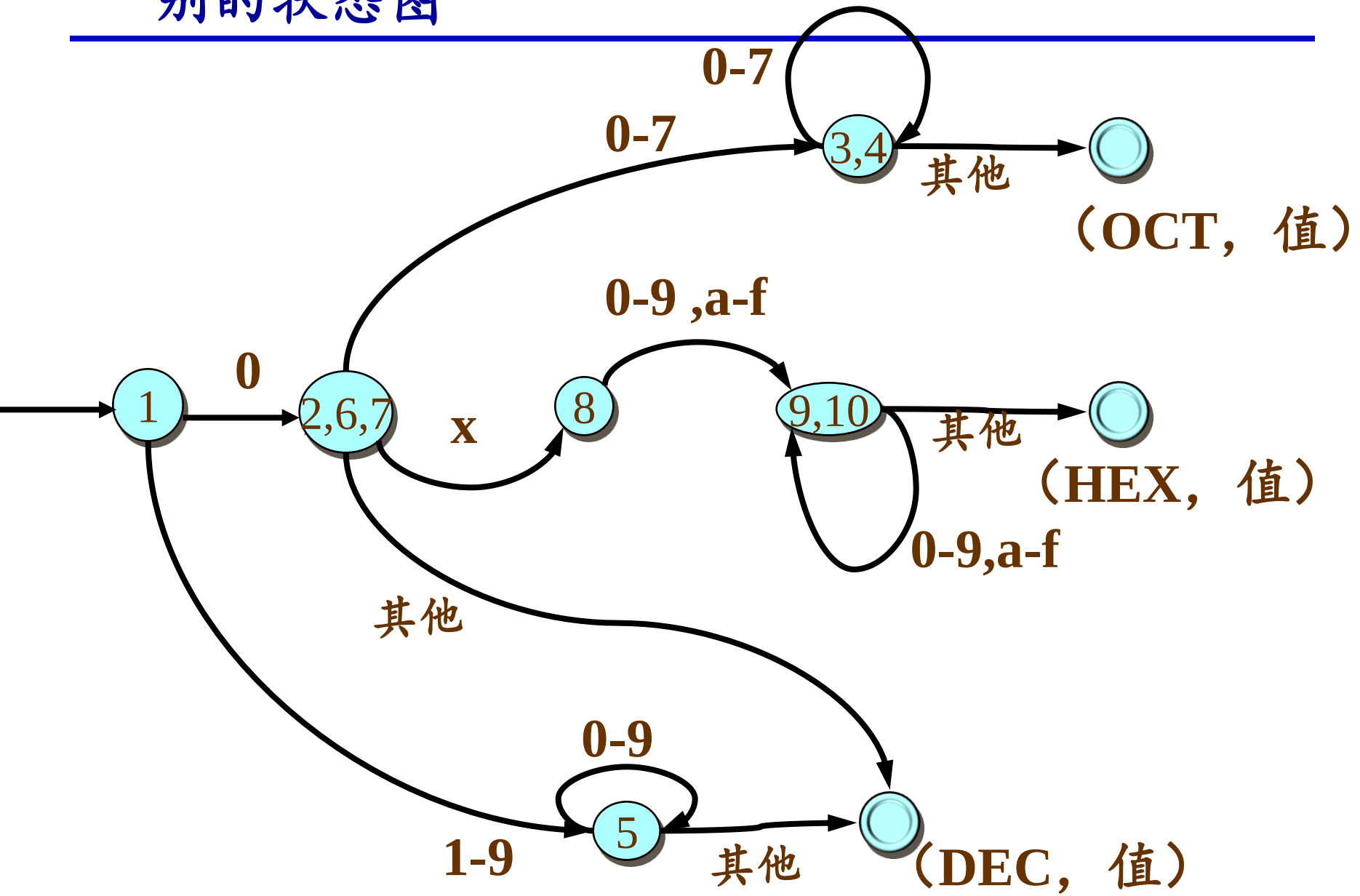
2、识别不同进制数的状态图



1、从开始状态出发；
2、选择输入符号，构成目标状态集
3、从新状态集出发，重复1、2

} 状态图合并

3.C语言无符号整数识别的状态图



2.2 词法分析器的设计

- 2.2.1 词法分析器的结构图
- 2.2.2 单词符号的识别
- 2.2.3 状态转换图 (Transition diagram)
- 2.2.4 状态转换图的实现

2.2.4 状态转换图的实现

程序实现:每个状态结对应一段程序.

1) 不含回路的分叉结:

CASE IF--THEN--ELSE

2) 含回路的分叉结:

WHILE IF

3) 终点结:

RETURN(C,VAL)

例如：状态转换图的实现 。（P43图3.3)

举例：词法分析程序的设计与实现

词法的扫描器的实现:

- 子程序 scan( )
  - 输入：字符流
  - 输出：
    - Symbol:单词种别
    - Attr:属性（全局变量attr）。

数据结构与子例程

- 数据结构
  - ch　　当前输入字符
  - token　输入缓冲区(字符数组)
  - symbol　单词种别（子程序的返回值）
  - attr　　属性（全局变量attr）
- 子例程
  - isKeyword(token):判别 token是关键字？返回关键字种别或 -1
  - Lookup(token):将 token 存入符号表，返回入口指针
  - getchar():从输入缓冲区中读入一个字符放入ch

例2.2-1 状态图的实现算法

```
1. getchar()
2.    WHILE ch 是空格                                    //跳过空格
2.1       DO getchar();
3.    CASE ch OF
4.      isdigit(ch) :
4.1       ch→token; getchar();
4.2       WHILE isdigit(ch) DO
            ch→token; getchar();
4.3       输入指针回退一个字符;
4.4       将token中的字符串变成数值→attr
4.5       返回 NUM
```

5. isalpha(ch) :

5.1 ch→token; getchar();

5.2 WHILE isalpha(ch) OR isdigit(ch)
DO ch→token; getchar();

5.3 输入指针回退一个字符;

5.4 key = isKeyword(token);

5.5 IF key≥0 THEN 返回 key

5.6 Lookup(token)→attr;

5.7 返回 IDN

6 ':' : getchar();

6.1 IF ch等于'=' THEN 返回 ASG

6.2 出错处理

```
7     '+' : 返回 ADD
8     '-' : 返回 SUB
9     '*' : 返回 MUL
10    '/' : 返回 DIV
11    '=' : 返回 EQ
12    '>' : 返回 GT
13    '<' : 返回 LT
14    '(' : 返回 LP
15    ')' : 返回 RP
16    ';' : 返回 SEMI
17    其它 : 出错处理
18  END OF CASE
```

第2章:词法分析 (Lexical Analysis)

- 2.1 词法分析程序的功能
- 2.2 词法分析器的设计
- 2.3 正规表达式 (Regular Expression)
- 2.4 有限自动机
- 2.5 词法分析器的自动生成

2.3 正规表达式 (Regular Expression)

举例：pascal语言的标识符表示为：

（1）letter (letter | digit)*

其中： letter - 字母

digit - 数字

| - 或

* - 零次或多于零次的引用。

（1）是正规式； 标识符是正规集

常用符号和表示方式：

Σ ----有限字母表

Σ 上的字----- Σ 中的字符构成的有穷序列。

ε--空字，不含任何字符的序列。

φ---空集，不含任何字的集合。{　}

{ε}---仅含空字的集合。

Σ^*--- Σ 上的所有字的集合，包括空字。

例： $\Sigma=\{a,b\}$

$\sum^{*}=\{\varepsilon, a,b,aa,ab,ba,bb,aaa,\ldots\}$

正规式可以详细说明单词符号的结构，可以精确地定义集合（正规集）。

例：正规式**a|b** 表示的集合**{a,b}.**

正规式**a*** 表示的集合$\{\varepsilon, a,aa,aaa,\ldots\}$

(连接)积：Σ^*的子集U和V的积(union)定义为

$$UV=\{\ \alpha\beta|\alpha \in U\ \ \&\ \ \beta \in V\ \}$$

$$(\ U\ V\ )\ W = U\ (\ V\ W\ )$$

$$V^n = V\ V \ldots V \qquad /*n个*/$$

闭包：

(Kleene closure) $V^* = V^0\ V^1\ V^2\ V^3 \ldots$

(Positive closure) $V^+ = V\ V^* \qquad V^0 = \{\ \varepsilon\ \}$

正规式与 正规集

定义(Definition)：

1. ε 和 Φ 都是 Σ 上的正规式 ， 其正规集为{ ε} 和 Φ ；

2. 任何a∈Σ, a是Σ的一个正规式 ，正规集为{ a} ；

3. 假定 U 和 V 都是 Σ 上的正规式，正规集为 L(U) 和 L(V) ,那么，

(U|V),(U.V),(U)*也都是S上的正规式(Regular Expression) , 其正规集(Regular Set)为 L(U)UL(V) , L(U)L(V) 和 L(U)*;

仅由有限次使用上述三步骤而定义的表达式才是 S上的正规式 ,仅由这些正规表达式所表示的字集才是S 上的正规集。

例: r = letter(letter|digit)* 表示的语言

$$L(r)=L(letter)(L(letter) \cup L(digit))^*$$
$$=\{A,\ldots,Z,a,\ldots,z\}(\{A,\ldots,Z,a,\ldots,z,0,\ldots,9\})^*$$

- 运算优先级和结合性:
  - * 高于 "连接" 高于 |
  - | 具有交换律、结合律
  - “连接” 具有结合律、分配律
  - ( ) 指定优先关系

- 算符的优先顺序为： * . |
- 若两个 正规式所表示的正规集相同，则 认为二者等价。记为：U=V。

例： b(ab)* b,bab,babab,...

(ba)*b b,bab,babab,...

则 b(ab)*= (ba)*b

例： （a|b)*=(a*b*）*

例题2.3-1：正规式与正规集.(P47)

令： $\Sigma=\{A,B,0,1\}$, **下面是正规式与正规集。**

正规式（A|B)(A|B|0|1)*

正规集 Σ**上的“标识符”的全体。**

正规式 (0|1)(0|1)*

正规集 Σ**上的“数”的全体。**

例题2.3-2 ：正规式与正规集

令：S={a,b}, 下面是正规式与正规集。

a|b	**{a,b}**
(a|b)(a|b)	**{aa,ab,ba,bb}**
a*	**{ε,a,aa,aaa,…)**
(a|b)*	**{a,b}*//包含零个或若干个a或b的所有串的集合。**
a|a*b	**{a,b,ab,aab,aaab,…}**
a|ba*	**{a,b,ba,baa,baaa,…}**

令：U、V和W均为正规式，下列关系普遍成立：

1. 交换律　　$U|V = V|U$

2. 结合律　　$U|(V|W) = (U|V)|W$

3. 结合律　　$U(V\ W) = (U\ V)\ W$

4. 分配律　　$U(V|W) = UV|UW$

　　$(V|W)U = VU|WU$

5.　　$\varepsilon U = U\varepsilon = U$

正规定义式与正规文法

- 正规表达式与正规文法有相同的表达能力。

定义（正规定义式）：

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

。。。

$d_n \rightarrow r_n$

r_i 是正规表达式，r_i 不能含有 $d_i, d_{i+1}, \ldots, d_n$.

例如 2.3-3：标识符的正规定义式。

letter → A|B|…|Z|a|b|…|z

digit → 0|1|2|…|9

id → letter(letter|digit)*

例如 2.3-4：PASCAL的无符号数的正规定义式

如：5280，39.27 ,6.36E4或 1.894E-4.

解：

digit ⟶ 0|1|2|…|9

digits ⟶ digit digit*

op_fra ⟶ . digits|ε

op_exp ⟶ (E(+|-|ε)Digits)|ε

num ⟶ digits op_fra op_exp

*组成：整数部分、可选小数部分、可选指数部分。

例 2.3-5 C语言无符号整数的正规定义式

八进制数:　　(OCT，值)

⌘oct→0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)*

十进制数:　　(DEC，值)

⌘dec→(1|...|9)(0|...|9)* |0

十六进制数:　　(HEX，值)

⌘hex→0x(0|1|...|9|a|...|f|A|...|F)(0|...|9|a|...|f |A|...|F)*

- 定义（正规文法）：

1） 如果文法G =（V_T，V_N，S，P）中的每一个产生式的形式为

$$A \rightarrow aB \quad 或 \quad A \rightarrow a$$

其中，$A, B \in V_N$，$a \in V_T \cup \{\varepsilon\}$，

则称 G 是右线性文法 。

2） 若文法G中的每一个产生式的形式为

$$A \rightarrow Ba \quad 或 \quad A \rightarrow a$$

则称 G是左线性文法。

右线性文法和左线性文法都称为正规文法

（3型文法）。它所产生的语言都称为正规语言或3型语言。

正规式转换到正规文法

- 引入非终结符（开始符号），构造
  - 〈开始符号〉→ 正规式
- 引入非终结符，逐级分解“连接”的正规式
- 用多个产生式表示“或”关系
- 用非终结符的递归表示，改写“重复”的正规式

例 标识符定义的转换

⌘ 引入 id

$id \rightarrow let\ (let \mid dig)^*$

⌘ 引入 rid 消除连接

rid

$\rightarrow (\ let \mid dig\ )^*$

$\rightarrow \varepsilon \mid (let|dig)^+$

$\rightarrow \varepsilon \mid (let|dig)\ (let|dig)^*$

$\rightarrow \varepsilon \mid let(let|dig)^* \mid dig(let|dig)^*$

$\rightarrow \varepsilon \mid let\ rid \mid dig\ rid$

得到的正规文法

- 得到的产生式集合 P

  id → let rid

  rid → ε| let rid | dig rid

  - 如果把 let 和 dig 看做终结符， 已构成右线性文法:

- G ＝（{let,dig}，{id,rid}，P，id）

3）词法的描述

例：某简易语言的词法

词法规则	单词种别	属性
⌘ <标识符> → <字母> (<字母>|<数字>)*	IDN	符号表入口
⌘ <无符号整数>→ <数字> (<数字>)*	NUM	数值
⌘ <赋值符> → :=	ASG	无
⌘ 其他单词 → 字符本身	单词名称	无

改写为正规文法

id → LETTER rid　　标识符

rid → ε| LETTER rid | DIGIT rid

num → DIGIT rnum　　整数

rnum → ε| DIGIT rnum

asg → : eq　　赋值

add → +　　加法

eq → =　　相等

…

（其他算术运算、关系运算、分号、括号等）

How to Describe Tokens

- Use regular expressions to describe programming language tokens!
- A regular expression (RE) is defined inductively
  - a — ordinary character stands for itself
  - ε — empty string
  - R|S — either R or S (alteration), where R,S = RE
  - RS — R followed by S (concatenation)
  - R* — concatenation of R 0 or more times (Kleene closure)

Language

- A regular expression R describes a set of strings of characters denoted L(R)
- L(R) = the language defined by R
  - L(abc) = { abc }
  - L(hello|goodbye) = { hello, goodbye }
  - L(1(0|1)*) = all binary numbers that start with a 1
- Each token can be defined using a regular expression

Example

- L_1 = {a,b,c,d} L_2 = {1,2}
- L_1L_2 = {a1,a2,b1,b2,c1,c2,d1,d2}
- $L_1 \cup L_2$ = {a,b,c,d,1,2}
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon, 0, 00, 000, 0000, ...\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

.

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter $\rightarrow$ A | B | ... | Z | a | b | ... | z

  digit $\rightarrow$ 0 | 1 | ... | 9

  id $\rightarrow$ letter (letter | digit) *

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit $\rightarrow$ 0 | 1 | ... | 9

  digits $\rightarrow$ digit $^+$

  opt-fraction $\rightarrow$ (. digits) ?

  opt-exponent $\rightarrow$ (E (+|-)? digits) ?

  unsigned-num $\rightarrow$ digits opt-fraction opt-exponent