Outline

Chapter 2.

- 文法
- **CFG**
- 文法化简
- 文法推断

Chapter 3

- 词法分析器
- 自动机
- 词法分析器的自动生成器

Chap. 2　编译示例及形式语言简顾

王朝坤
chaokun@tsinghua.edu.cn

文法的类型

通过对产生式施加不同的限制，**Chomsky**将文法分为四种类型：

0型文法：对任一产生式 $\alpha \rightarrow \beta$，都有 $\alpha \in (V_N \cup V_T)^+$，　$\beta \in (V_N \cup V_T)^*$

1型文法：对任一产生式 $\alpha \rightarrow \beta$，都有 $|\beta| \geqslant |\alpha|$，　仅仅 $S \rightarrow \varepsilon$ 除外

2型文法：对任一产生式 $\alpha \rightarrow \beta$，都有 $\alpha \in V_N$

3型文法：任一产生式 $\alpha \rightarrow \beta$ 的形式都为A→aB或A→a，其中 $A \in V_N$，$B \in V_N$，$a \in V_T$ *

0型文法

- 无限制文法
- 形式语言谱系中最大的文法类
- 对生成式$\alpha \rightarrow \beta$不作特殊限制

$$\alpha \neq \varepsilon$$

1型文法

例：1型（上下文有关）文法

文法G[S]：

S→CD

C→aCA

C→bCB

AD→aD

BD→bD

Aa→bD

Ab→bA

Ba→aB

Bb→bB

C→a

D→b

β要至少和α一样长

2型文法

例：2型（上下文无关）文法

文法G[S]：S→AB

A→BS|0

B→SA|1

α必须是非终结符

3型文法

例：3型（正则）文法

G[S]:

S→0A|1B|0

A→0A|1B|0S

B→1B|1|0

G[I]:

I → Tl

I → l

T → Tl

T → Td

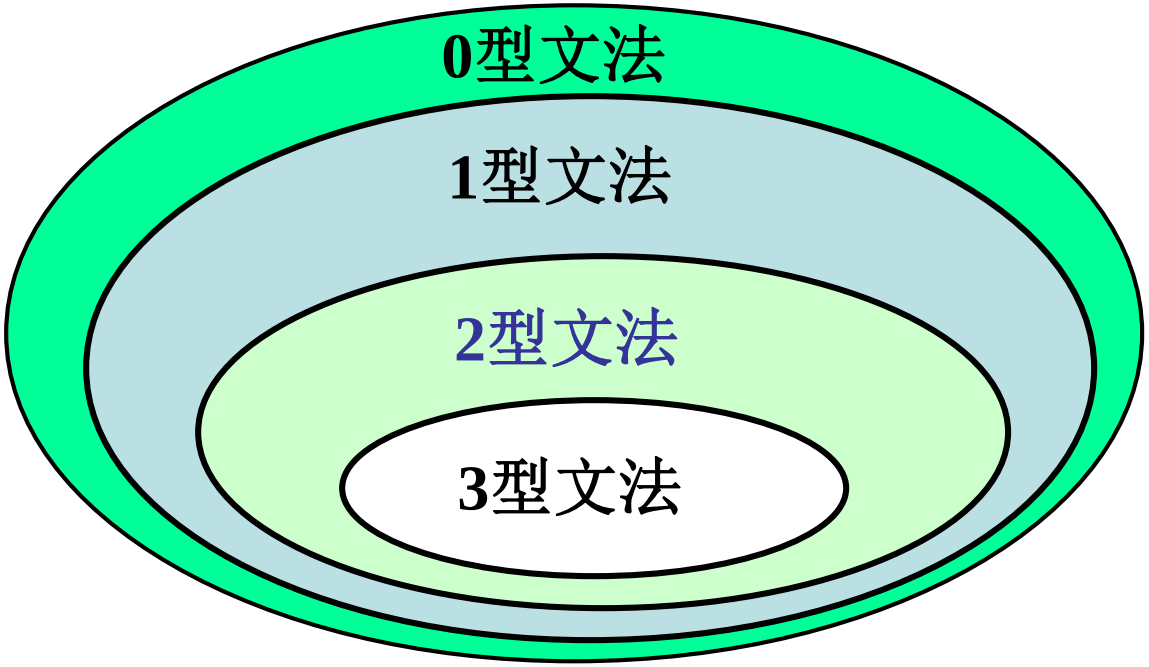
T → l

T → d

右线性文法/左线性文法

文法的关联

四类文法之间的逐级“包含”关系



在不考虑空字符串时，i型语言都真包含i＋1型语言 (for i＝0，1，2)

文法和语言

- 0型文法产生的语言称为0型语言
- 1型文法或上下文有关文法（ CSG ）产生的语言称为1型语言或上下文有关语言（ CSL ）
- 2型文法或上下文无关文法（ CFG ）产生的语言称为2型语言或上下文无关语言（ CFL ）
- 3型文法或正则（正规）文法（ RG ）产生的语言称为3型语言正则（正规）语言（ RL ）

文法和语言

- 四种文法之间的关系：
  - 将产生式做进一步限制而定义的
- 语言之间的关系依次：
  - 存在不是上下文有关语言的0型语言
  - 存在不是上下文无关语言的1型语言
  - 存在不是正则语言的上下文无关语言

文法和识别系统

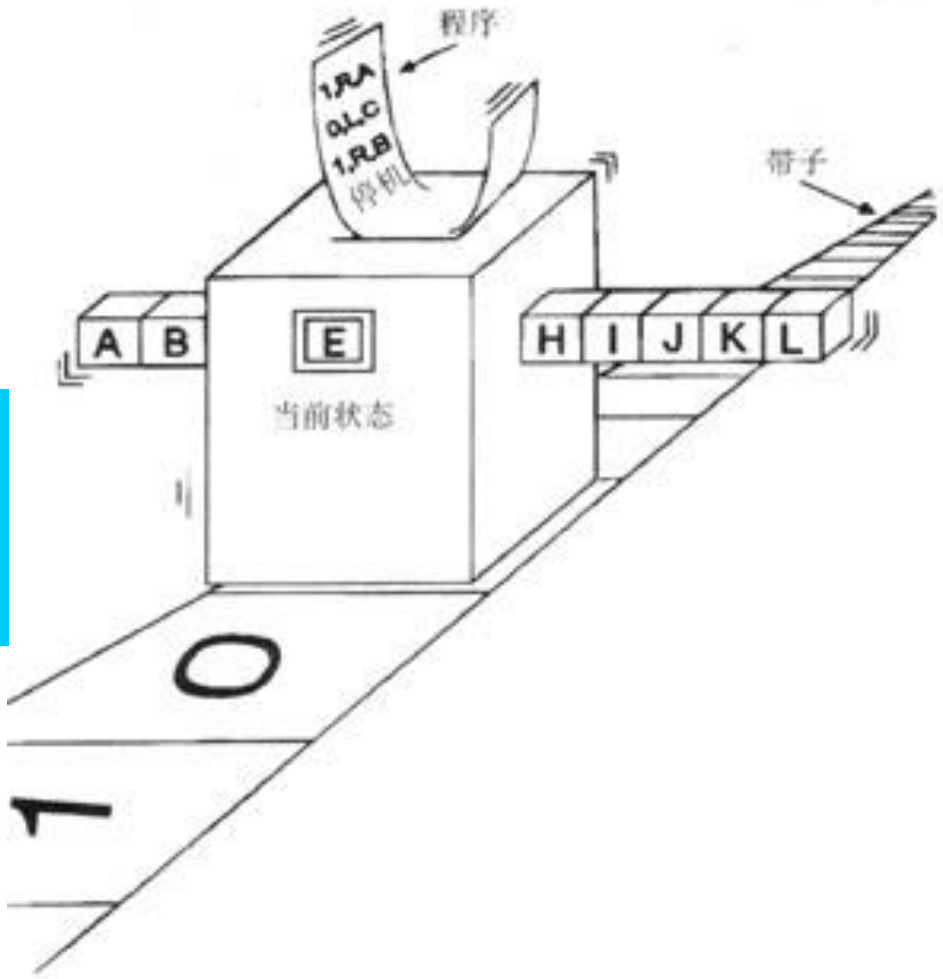
- 根据形式语言理论，文法和识别系统间有这样的关系
- 0型文法（短语结构文法）
  - 能力相当于图灵机
  - 可以表征任何递归可枚举集
  - 任何0型语言都是递归可枚举的

- 任何能用图灵机描述的计算都能机械实现
- 任何能在现代计算机上实现的计算都能用图灵机描述

图灵机

- 理论计算机——一切计算机的抽象模型
  - 一个无限长的带子
  - 一个读写头
  - 内部状态
  - 控制程序

根据程序的命令以及它的内部状态进行带子的读写、移动



文法和识别系统(cont'd)

- **1型文法（上下文有关文法）**
  - 产生式的形式为$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$
  - 只有A出现在α_1和α_2的上下文中时，才允许β取代A
  - 其识别系统是非确定型线性有界自动机

- **2型文法（上下文无关文法CFG）**
  - 产生式的形式为$A \rightarrow \beta$
  - β取代A时与A的上下文无关
  - 其识别系统是不确定的下推自动机

- **3型文法（正规文法RG）**
  - 产生的语言是有穷自动机（FA）所接受的集合
  - 正则表达式

4. 上下文无关文法及其语法树

上下文无关文法有足够的能力描述程序设计语言的语法结构

语法树---*句型推导*的直观表示

句型推导（a+a*a）

G[E]： $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid a$

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$
$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+a*a$
$\Rightarrow T+a*a \Rightarrow F+a*a \Rightarrow a+a*a$

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T+T*F \Rightarrow F+T*F \Rightarrow F+F*F$
$\Rightarrow a+F*F \Rightarrow a+F*a \Rightarrow a+a*a$

规范推导　规范句型

句型推导的分类

最左（最右）推导：

— 在推导的任何一步$\alpha \Rightarrow \beta$，其中α、β是句型，都是对α中的最左（右）非终结符进行替换

最右推导被称为规范推导。

由规范推导所得的句型称为规范句型

句型推导的直观表示

语法树

语法树

设G=(V_N,V_T,P,S)为一cfg，若一棵树满足下列4个条件，则此树称作G的语法树/推导树/派生树：

1. 根结点的标号是文法的开始符号S
2. 每个内部结点的标号为A，且A∈V_N
3. 每个叶结点标号为一个终结符号或 ε
4. 如果结点n有标记A,其子结点从左到右的次序是n_1，n_2，…，n_k，对应的标号分别为A_1，A_2，…，A_k，那么A→A_1A_2，…，A_k一定是P中的一个产生式。特例：若A→ε是一个产生式，则标号为A的结点可以只有一个标号为ε的子结点。

语法树的结果：

从左到右读出叶子的标号构成一个句子

语法树示例

- 句型aabbaa的可能推导序列和语法树

例: G[S]:

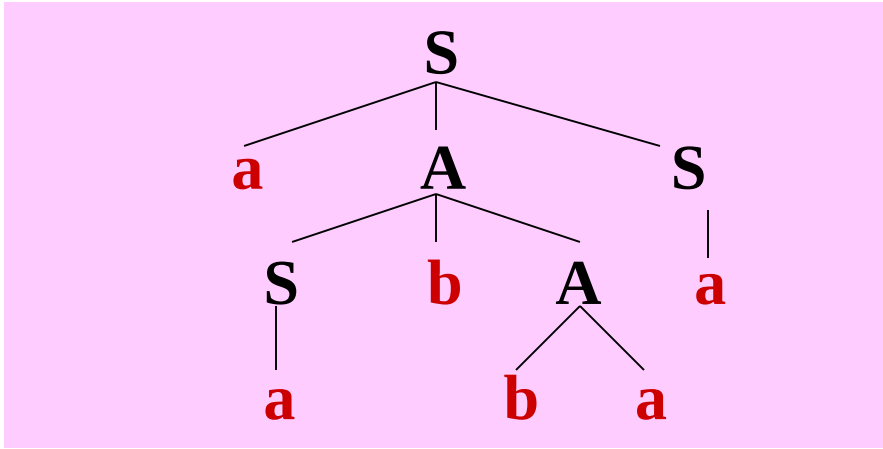
S→aAS

A→SbA

A→SS

S→a

A→ba



S⇒aAS⇒aAa⇒aSbAa⇒aSbbaa⇒aabbaa

S⇒aAS⇒aSbAS⇒aabAS⇒aabbaS⇒aabbaa

S⇒aAS⇒aSbAS⇒aSbAa⇒aabAa⇒aabbaa

语法树性质

给定文法$G=(V_N,V_T,P,S)$，对于G的任何句型都能构造与之关联的语法树(推导树)吗？

定理：

G为上下文无关文法，

对于$\alpha\neq\varepsilon$，有$S =>^* \alpha$，当且仅当

文法G有以 α 为结果的一棵语法树(推导树)

一棵语法树表示了一个句型的种种可能的(但未必是所有的)不同推导过程，包括最左(最右)推导。但是，一个句型是否只对应唯一的一棵语法树呢?一个句型是否只有唯一的一个最左(最右)推导呢?

句型 i*i+i

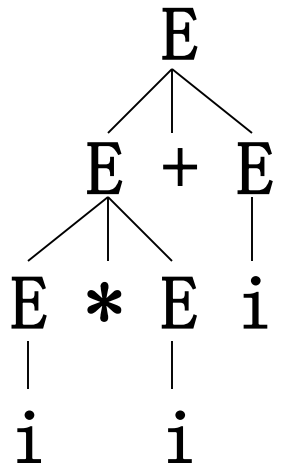
例：G'[E]:

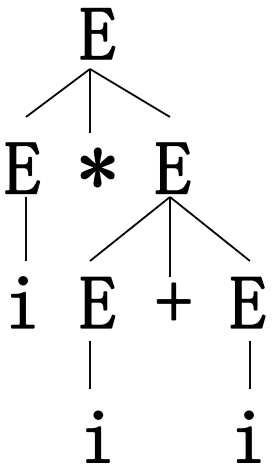
E → i

E → E+E

E → E*E

E → (E)





句型 i*i+i 的两个不同的最左推导：

推导1：E ⇒ E+E ⇒ E*E+E ⇒ i*E+E ⇒ i*i+E ⇒ i*i+i

推导2：E ⇒ E*E ⇒ i*E ⇒ i*E+E ⇒ i*i+E ⇒i*i+i

二义性

自然语言中存在二义性

“走火”

“打酱油”

“你是个好人.....”

程序设计语言中存在“类似情况”吗？

```
i+i*i
```

如何避免二义性→如何刻画二义性

语言与文法的关系→语言的二义性、文法的二义性

文法的二义性和语言的二义性

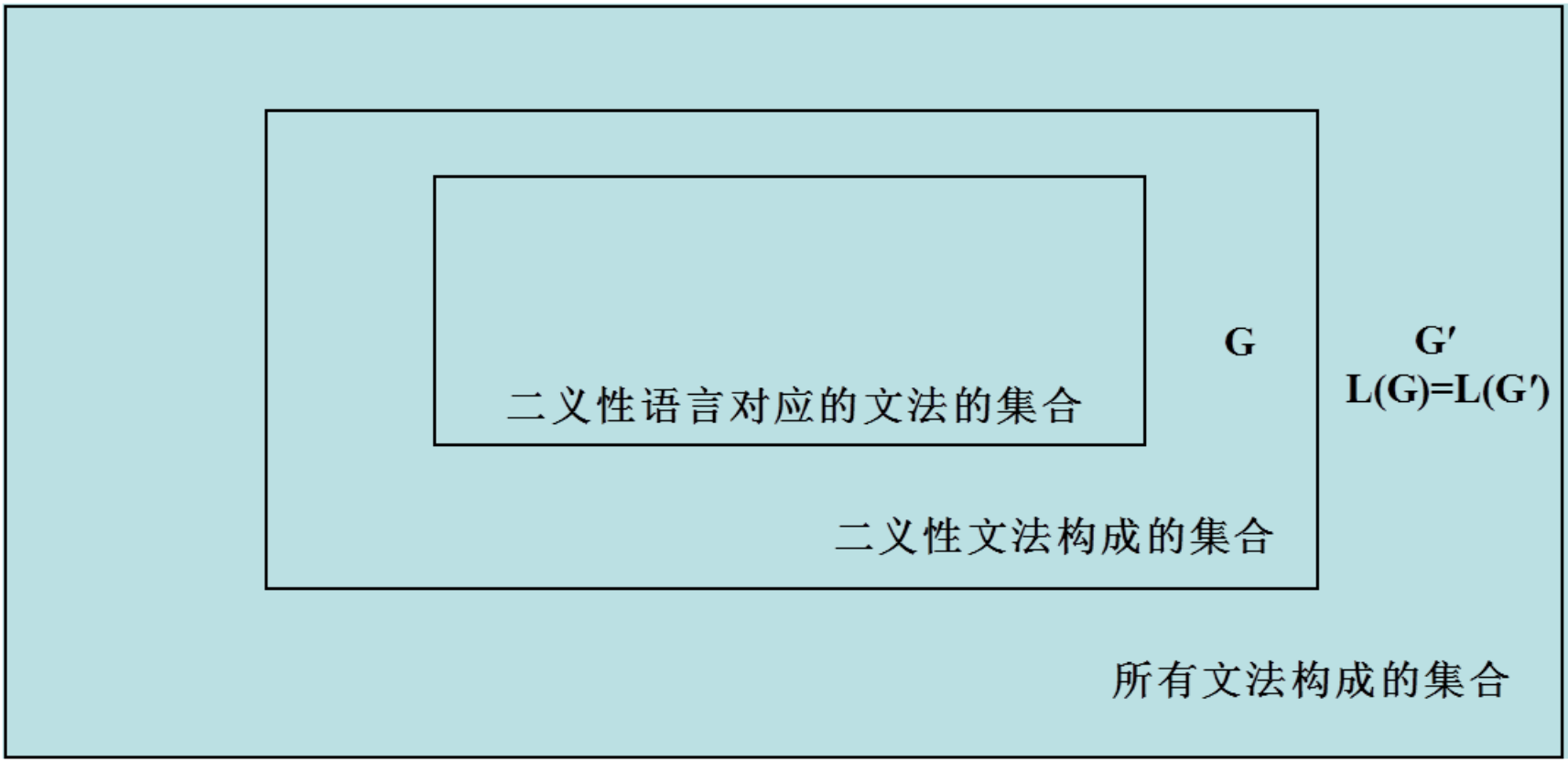
文法的二义性和语言的二义性是两个不同的概念

若一个文法存在某个句子对应两棵不同的语法树，则称这个文法是二义的

或者，若一个文法存在某个句子有两个不同的最左（右）推导，则称这个文法是二义的

如果产生上下文无关语言的每一个文法都是二义的，则说此语言是先天二义的。

二义性CFG的范围



可能有两个不同的文法G和G′，其中G是二义的，但是却有L(G)=L(G′)，也就是说，这两个文法所产生的语言是相同的。

二义性与程序设计语言

判定任给的一个上下文无关文法是否二义，或它是否产生一个先天二义的上下文无关语言，这两个问题是递归不可解的，但可以为无二义性寻找一组充分条件

对于一个程序设计语言来说，常常希望它的文法是无二义的，因为希望对它的每个语句的分析是唯一的。

二义文法改造为无二义文法

G[E]: E → i
E → E+E
E → E*E
E → (E)

G′ [E]: E → T|E+T
T → F|T*F
F → (E) |i

规定算符优先性和结合性

（上下文无关文法）句型的分析

*句型分析*就是识别一个符号串是否为某文法的句型，是某个推导的构造过程。

在语言的编译实现中，把完成句型分析的程序称为*分析程序*或*识别程序*。分析算法又称*识别算法*。

从左到右的分析算法，即总是从左到右地识别输入符号串，首先识别符号串中的最左符号，进而依次识别右边的一个符号，直到分析结束。

句型的分析算法分类

分析算法可分为：

自上而下分析法：

从文法的开始符号出发，反复使用文法的产生式，寻找与输入符号串匹配的推导，或者说，为输入串寻找一个最左推导。

自下而上分析法：

从输入符号串开始，逐步进行归约，直至归约到文法的开始符号。

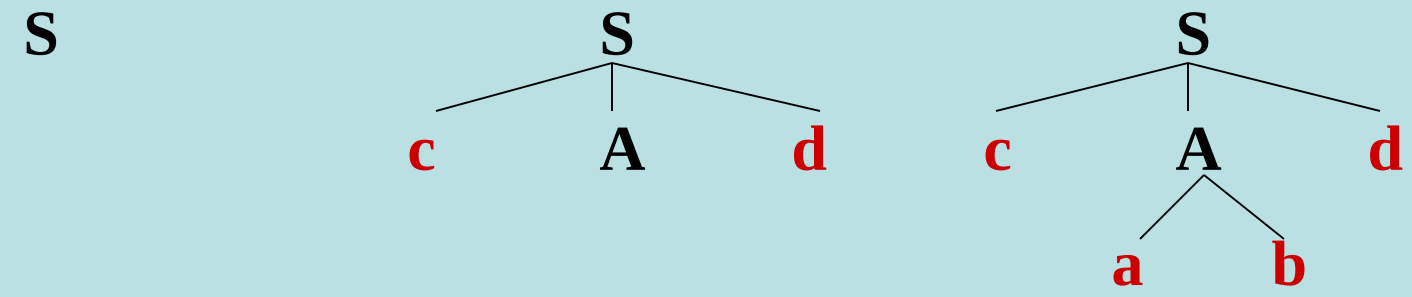
自上而下的语法分析

例：文法G： S → cAd

A → ab

A → a

识别输入串w=cabd是否为该文法的句子



推导过程：S ⇒ cAd　　cAd ⇒ cabd

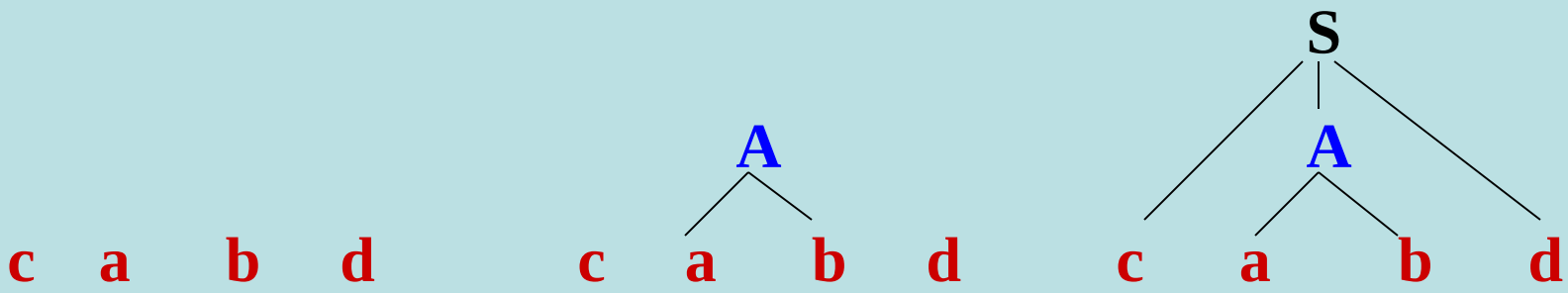
自下而上的语法分析

例：文法G： S → cAd

A → ab

A → a

识别输入串w=cabd是否为该文法的句子



归约过程构造的推导： cAd ⇒ cabd　　S ⇒ cAd

自上而下的语法分析

(1) S →cAd (2) A →ab (3) A → a

识别输入串w=cad是否为该文法的句子

若S ⇒ cAd 后选择(2)扩展A，S ⇒ cAd ⇒ cabd

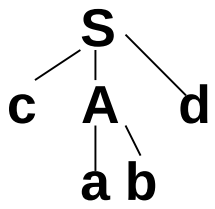
那将会？

w的第二个符号可以与叶子结点a得以匹配，但第三个符号却不能与下一叶子结点d匹配

？宣告分析失败（其意味着，识别程序不能为串cad构造语法树，即cad不是句子）

-显然是错误的结论。

导致失败的原因是在分析中对A的选择不是正确的。



这时应该回朔，把A为根的子树剪掉，扫描过的输入串中的a吐出来，再试探用产生式（3）

自下而上的语法分析

(1)S → cAd (2) A → ab (3)A → a

识别输入串w=cabd是否为该文法的句子

对串cabd的分析中，如果不是选择ab用产生式(2),而是选择a用产生式(3)将a归约到了A，那么在c A b d中无法找到一个可归约串了，最终就达不到归约到S的结果，因而也无从知道cabd是一个句子

c a b d

c A b d

a

文法使用中的一些说明

限制化简文法

文法中不含有*有害规则*和*多余规则*

有害规则：形如U→U的产生式。会引起文法的二义性

多余规则：指文法中任何句子的推导都不会用到的规则

文法中不含有不可到达和不可终止的非终结符

1）文法中某些非终结符不在任何规则的右部出现，该非终结符称为不可到达。

2）文法中某些非终结符，由它不能推出终结符号串，该非终结符称为不可终止。

对于文法G[S]，为了保证任一非终结符A在句子推导中出现，必须满足如下两个条件：

1. A必须在某句型中出现

即有S =>* αAβ，其中α，β属于V*

2. 必须能够从A推出终结符号串t来

即A =>* t，其中$t \in V_T^*$

化简文法

- 例：G[S]：
  1) S→Be
  2) B→Ce
  3) B→Af
  4) A→Ae
  5) A→e
  6) C→Cf
  7) D→f

D为不可到达 (产生式7)

C为不可终止 (产生式6)

产生式 2），6），7）为多余规则应去掉。

CFG 中的ε规则

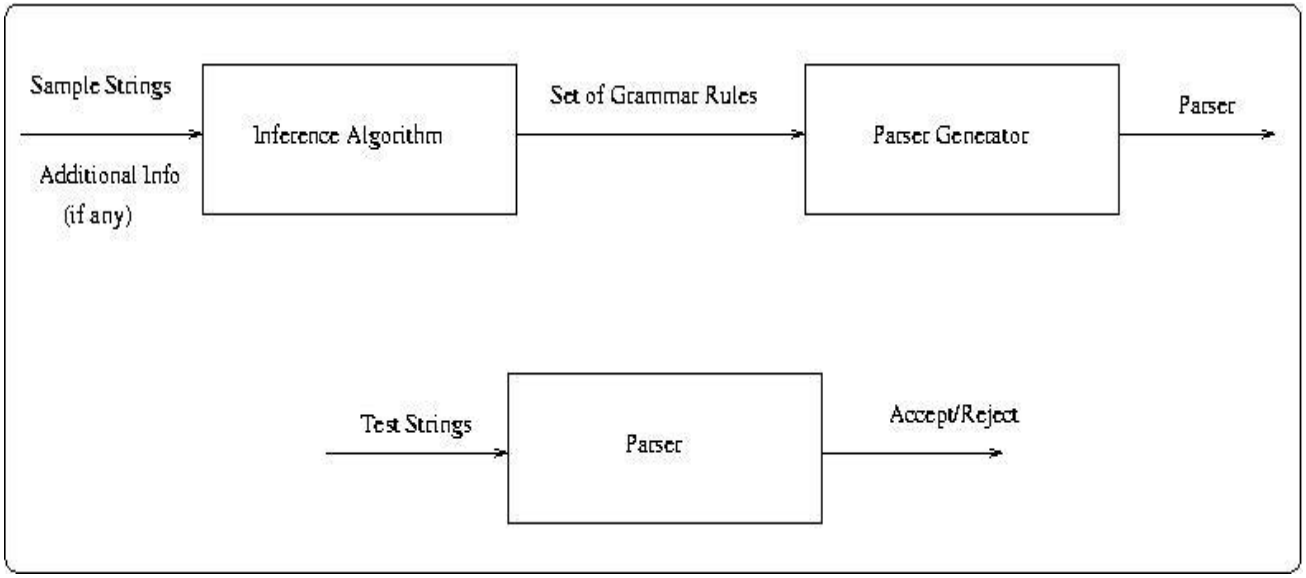
上下文无关文法中某些规则可具有形式A→ε，称这种规则为ε规则

因为ε规则会使得有关文法的一些讨论和证明变得复杂,有时会限制这种规则的出现

两种定义的唯一差别是ε句子在不在语言中

如果语言L有一个有穷的描述，则L_1=L∪｛ε｝也同样有一个有穷的描述。并且可以证明，若L是上下文有关语言、上下文无关语言或正规语言，则L∪｛ε｝和L-｛ε｝分别是上下文有关语言、上下文无关语言和正规语言。

Grammatical Inference （文法推断）



文法推断指的是 从语言的有限信息(输入)出发，通过一个归纳推断过程，
最终得到语言的文法描述(输出)。

输入信息一般包含

正例样本集，反例样本集，其它附加信息。

ICGI 2012, ICGI 2014, ICGI 2016，ICGI 2018

...

ICGI2018 Accepted Papers

Makoto Kanazawa and Tobias Kappé. Decision problems for Clark-congruential languages
Florent Avellaneda and Alexandre Petrenko. Inferring DFA without Negative Examples
Roland Groz, Nicolas Bremond and Adenilso Simao. Using Adaptive Sequences for Learning Non-Resettable FSMs
Wojciech Wieczorek, Olgierd Unold and Łukasz Strąk. Suffix Classification Trees
Joshua Moerman. Learning Product Automata
Hossep Dolatian and Jeffrey Heinz. Learning reduplication with 2-way finite-state transducers
Stéphane Ayache, Rémi Eyraud and Noé Goudian. Explaining black boxes on sequential data using weighted automata
Philip Amortila and Guillaume Rabusseau. Learning Graph Weighted Models on Pictures
Mateusz Pyzik, François Coste and Witold Dyrka. How to measure the topological quality of protein parse trees?
Roman Manevich and Sharon Shoham. Inferring Program Extensions from Traces
François Coste and Jacques Nicolas. Learning local substitutable context-free languages from positive examples in polynomial time and data

Chap. 3 Lexical Analysis

Chaokun Wang

IISE@Tsinghua

Outline

- **Lexical Analyzer***
  - **RE, RE~RG**
- **Finite Automata Concepts**
  - **Finite Automata**
  - **Non-Deterministic and Deterministic FA**
  - **Conversion Process**
    - **Regular Expressions to NFA**
    - **NFA to DFA**
    - **Minimizing the Number of States of a DFA**
    - **From a RE to a DFA****
- **Lexical Analyzer Generators***
  - **Lex/ANTLR**

What the Lexical Analyzer Sees


```
int gcd(int a, int b)
{
      while (a != b) {
            if (a > b) a -= b;
                  else b -= a;
      }
      return a;
}
```

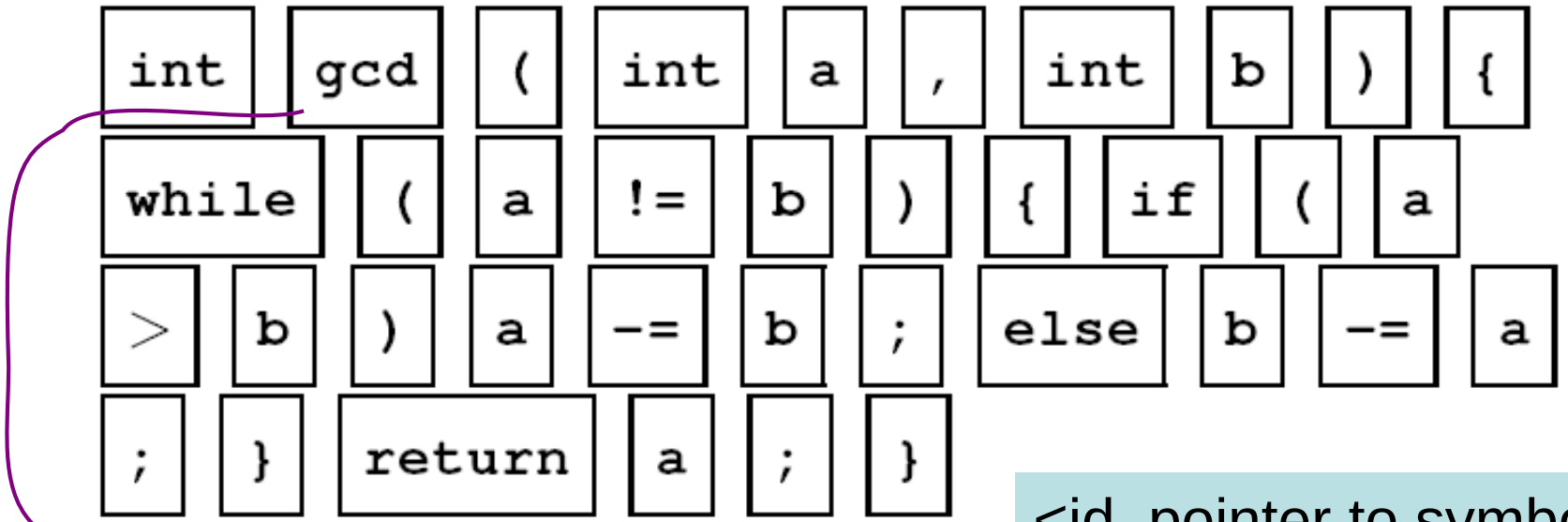
```
i n t sp g c d ( i n t sp a , sp i n t sp b ) nl {
nl sp sp w h i l e sp ( a sp ! = sp b ) sp { nl sp
sp sp sp i f sp ( a sp > sp b ) sp a sp - = sp b
; nl sp sp sp sp e l s e sp b sp - = sp a ; nl sp sp
} nl sp sp r e t u r n sp a ; nl } nl
```

Text file is a sequence of characters

Lexical Analysis Gives Tokens

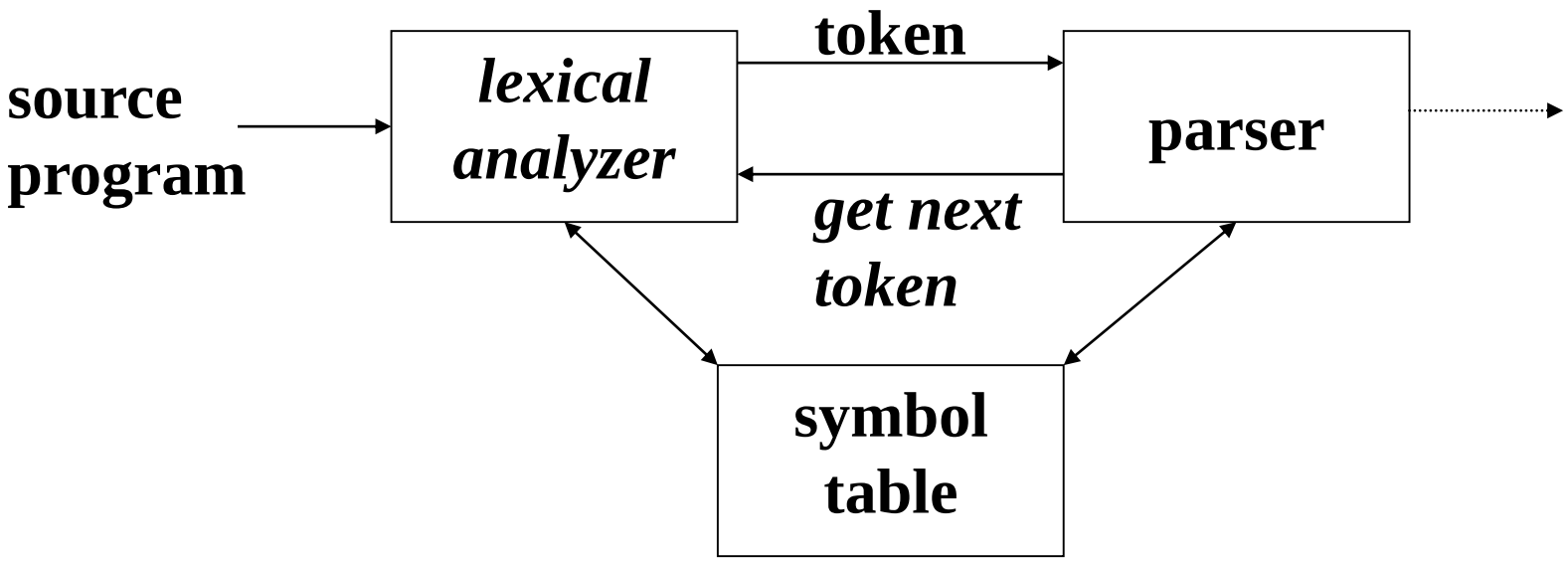

```
i n t sp g c d ( i n t sp a , sp i n t sp b ) nl {
nl sp sp w h i l e sp ( a sp ! = sp b ) sp { nl sp
sp sp sp i f sp ( a sp > sp b ) sp a sp - = sp b
; nl sp sp sp sp e l s e sp b sp - = sp a ; nl sp sp
} nl sp sp r e t u r n sp a ; nl } nl
```

A stream of tokens.
Whitespace, comments removed.



<id, pointer to symbol-table entry for gcd>

Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser.

Lexical Analyzer in Perspective

- **LEXICAL ANALYZER**
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Read info from ST
  - Generate Errors
  - Send Tokens to Parser

- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Create Abstract Rep. of Source
  - Update Symbol Table Entries
  - Generate Errors
  - And More…. (We'll see later)

Lexical Analysis is a Separate Phase

- **Separation of Lexical Analysis From Parsing**
- **Presents a Simpler Conceptual Model**
  - **A parser embodying the conventions for comments and white space is significantly more complex than one that can assume comments and white space have already been removed by lexical analyzer.**
- **Separation Increases Compiler Efficiency**
  - **Specialized buffering techniques for reading input characters and processing tokens…**
- **Separation Promotes Portability.**
  - **Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.**

Tokens, Patterns, and Lexemes

- **A *token* (词法单元/记号) is a classification of lexical units**
  - **a pair consisting of**
    - **a token name (词法单元名)**
    - **an optional attribute value (可选属性值)**
  - **token names: id and num**

- ***Lexemes* (词素) *are the specific character strings that make up a token***
  - **For example: abc and 123**

- ***Patterns* (模式) *are rules describing the set of lexemes belonging to a token***
  - **For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"**

Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Attributes for Tokens

Tokens influence parsing decision; the attributes influence the translation of tokens.

Example: R = M + C ** 5

<id, pointer to symbol-table entry for R>

<assign_op, >

<id, pointer to symbol-table entry for M>

<plus_op, >

<id, pointer to symbol-table entry for C>

<exp_op, >

<num, integer value 5>

Lexical Errors

- Error Handling is very localized, with Respect to Input Source
- For example: whil (x = 0) do
  generates no lexical errors in PASCAL
- Possible error recovery actions:
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters

- Handling Lexical Errors
- Buffer pair

Regular Definitions

We may give names to regular expressions and to define regular expression using these names as if they were symbols.

Let Σ is an alphabet of basic symbols. The regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$\ldots$$

$$d_n \rightarrow r_n$$

Where,

each d_i is a distinct name, and

each r_i is a regular expression over the symbols in

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

Towards Token Definition

Regular Definitions: Associate names with Regular Expressions

For Example : PASCAL IDs

letter → A | B | C | … | Z | a | b | … | z

digit → 0 | 1 | 2 | … | 9

id → letter (letter | digit)*

Shorthand Notation:

"+" : one or more $r^* = r^+ \mid \varepsilon$ (Kleene) & $r^+ = r\, r^*$ (Positive)

"?" : zero or one r?=r | ε

[range] : set range of characters (replaces "|")

[A-Z] = A | B | C | … | Z

Example Using Shorthand : PASCAL IDs

id → [A-Za-z][A-Za-z0-9]*

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

What language construct are they used for ?

Given Tokens, What are Patterns ?

if → **if**

then → **then**

else → **else**

relop → < | <= | > | >= | = | <>

id → **letter (letter | digit)***

num → **digit** $^{+}$ **(. digit** $^{+}$ **) ? (E(+ | -) ? digit** $^{+}$ **) ?**

What does this represent ?

Grammar:

stmt → |if *expr* then *stmt*
|if *expr* then *stmt* else *stmt*
|ε

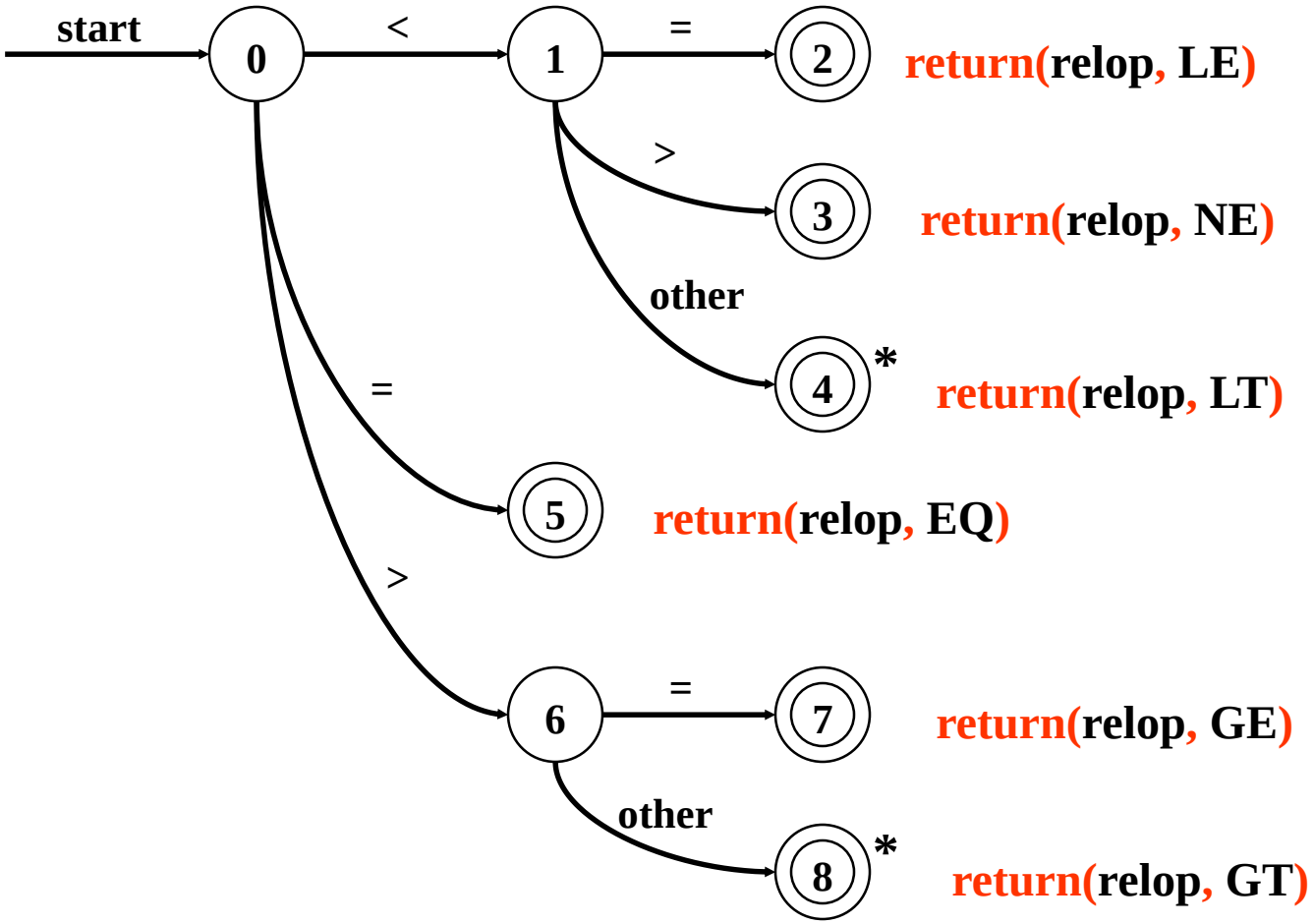
expr → *term* relop *term* | *term*

term → id | num

Constructing Transition Diagrams for Tokens

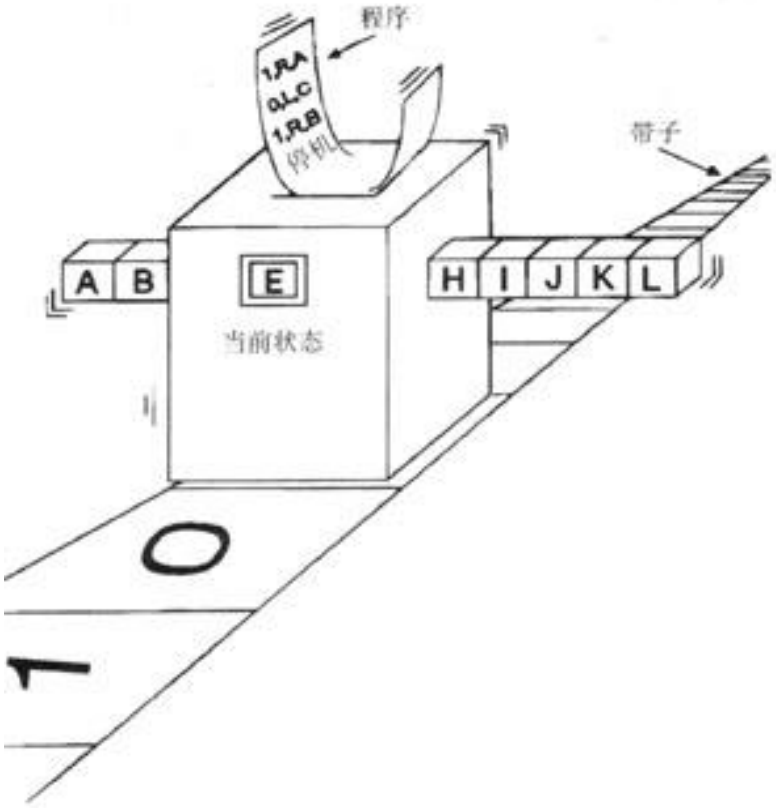
- **Transition Diagrams (TD) are used to represent the tokens**
- **As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern**
- **Each TD has:**
  - **States : Represented by Circles**
  - **Actions : Represented by Arrows between states**
  - **Start State : Beginning of a pattern (Arrowhead)**
  - **Final State(s) : End of pattern (Concentric Circles)**

Example : All RELOPs



Finite Automata (有限状态自动机)

Finite Automata : **A recognizer that takes an input string & determines whether it's a valid sentence of the language**



Non-Deterministic Finite Automata

An NFA is a mathematical model that consists of :

- **S, a set of states**
- **Σ, the symbols of the input alphabet**
- ***move*, a transition function.**
  - ***move*(state, symbol) → set of states**
  - ***move*** **: S × Σ∪{ε} → Pow(S)**
- **A state, $s_0 \in S$, the start state**
- **$F \subseteq S$, a set of final or accepting states.**

NFAs & DFAs

Non-Deterministic : **Has more than one alternative action for the same input symbol.**

Deterministic(确定) : **Has at most one action for a given input symbol.**

Both types are used to recognize regular expressions.

Non-Deterministic Finite Automata (NFAs) easily represent regular expression, but are somewhat less precise.

Deterministic Finite Automata (DFAs) require more complexity to represent regular expressions, but offer more precision.

Direct Simulation of an NFA


```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c);
   c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

DFA simulation

```
S ← ∈-closure({s0})
c ← nextchar;
while c ≠ eof do
   S ← ∈-closure(move(S,c));
   c ← nextchar;
end;
if S∩F≠∅ then return "yes"
         else return "no"
```

NFA simulation

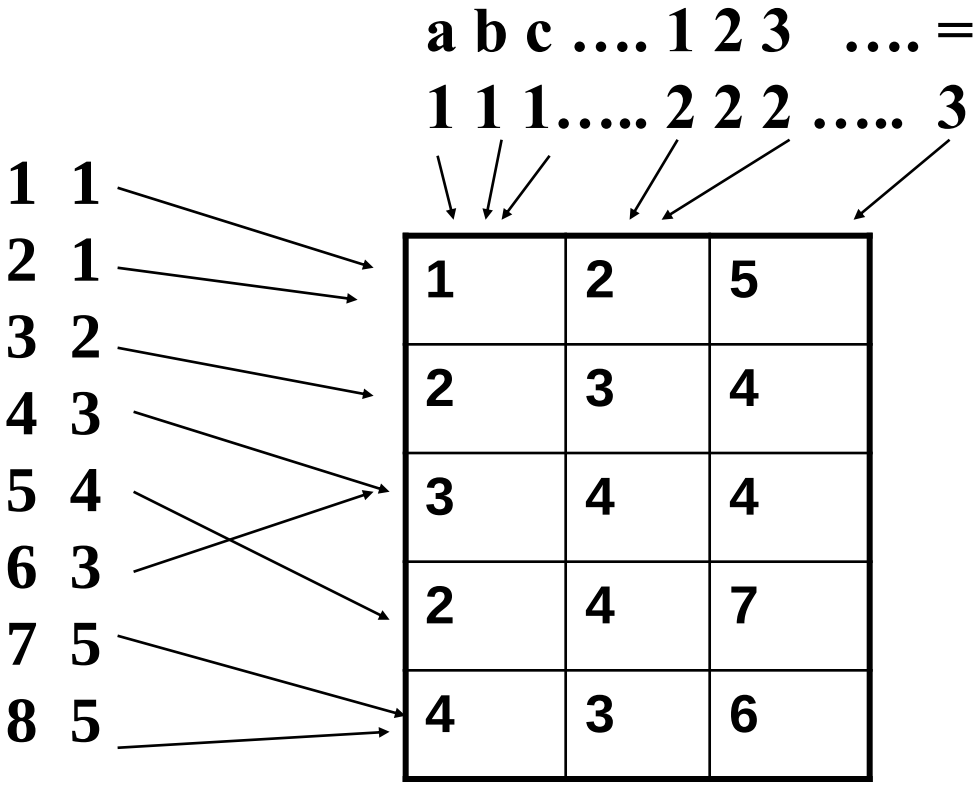
Optimizing Finite Automata

- Table Compaction
  - Two dimensional arrays provide fast access
  - Table size may be a concern (10KB to 100KB)
  - Table compression techniques
    - Compressing by eliminating redundant rows
    - Pair-compressed transition tables

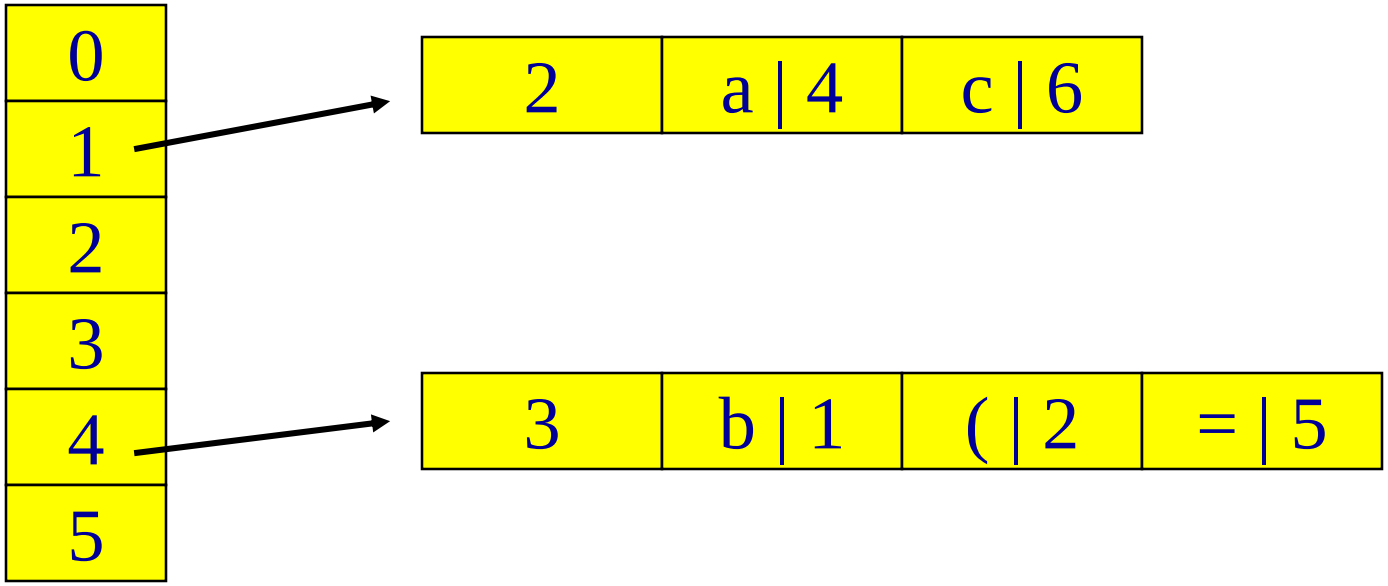
- **A typical transition table has many identical columns and some identical rows.**

	a	b	c	…	1	2	…	=
1	1	1	1		2	2		5
2	1	1	1		2	2		5
3	2	2	2		3	3		4
4	3	3	3		4	4		4
5	2	2	2		4	4		7
6	3	3	3		4	4		4
7	4	4	4		3	3		6
8	4	4	4		3	3		6

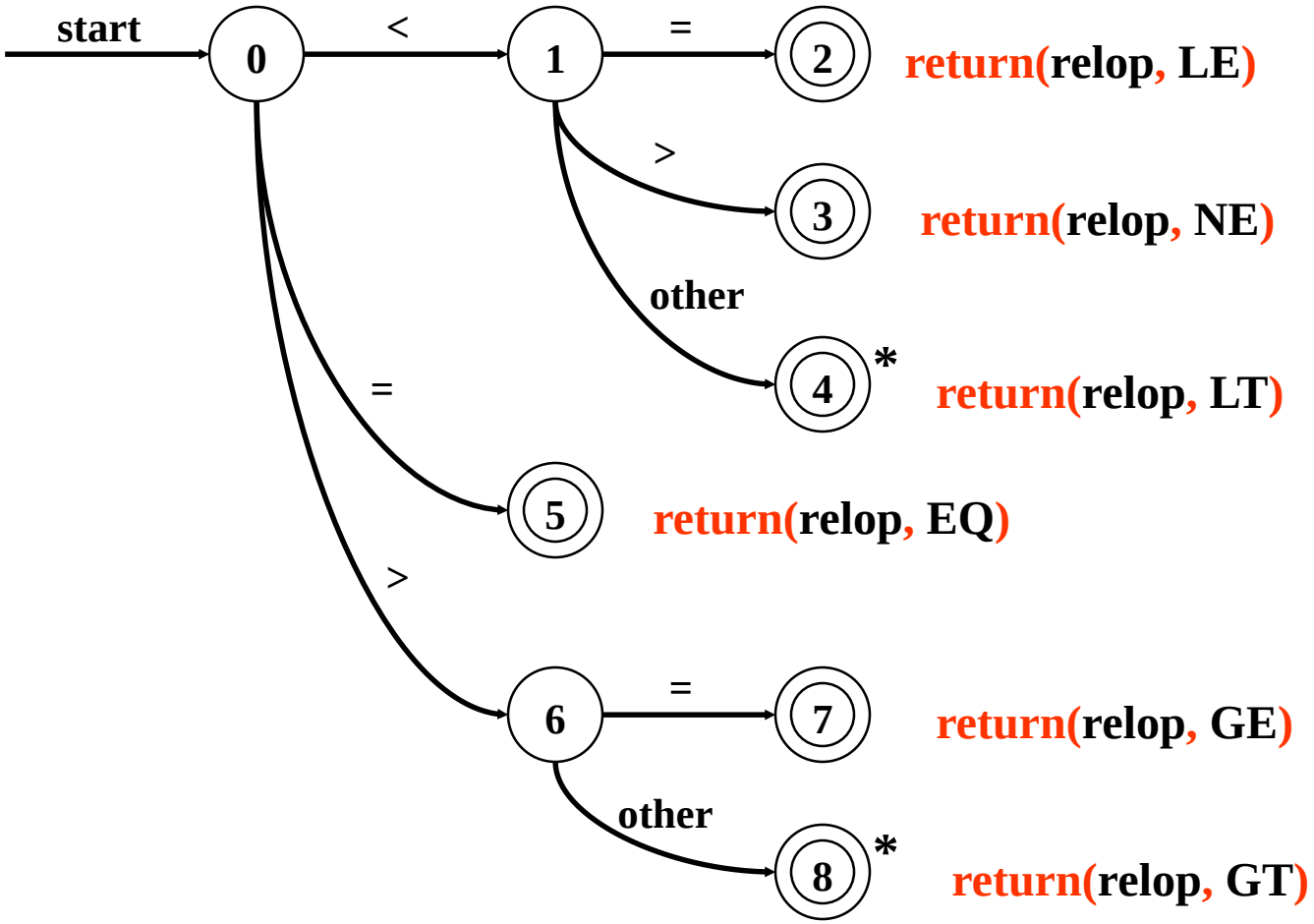
We may create a much smaller transition table with indirect row and column maps. Table is now accessed as T[rmap[s], cmap[c]].



Sparse table techniques



Example： All RELOPs （Review）



Implementing Transition Diagrams

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the grammar

Each state gets a segment of code

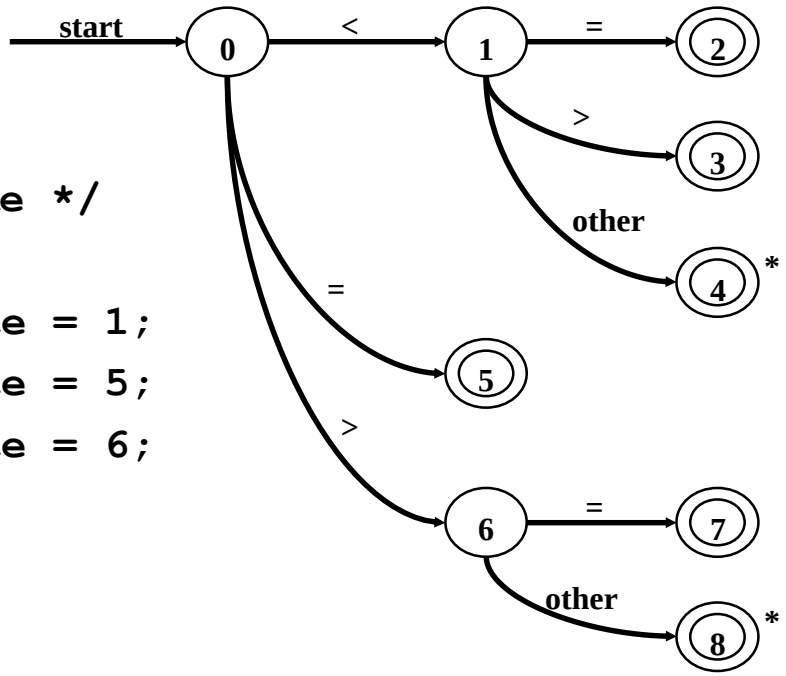
FUNCTIONS USED

- nextchar(),
- retract(),
- install_num(),
- install_id(),
- gettoken(),
- isdigit(),
- isletter(),
- recover()

Implementing Transition Diagrams

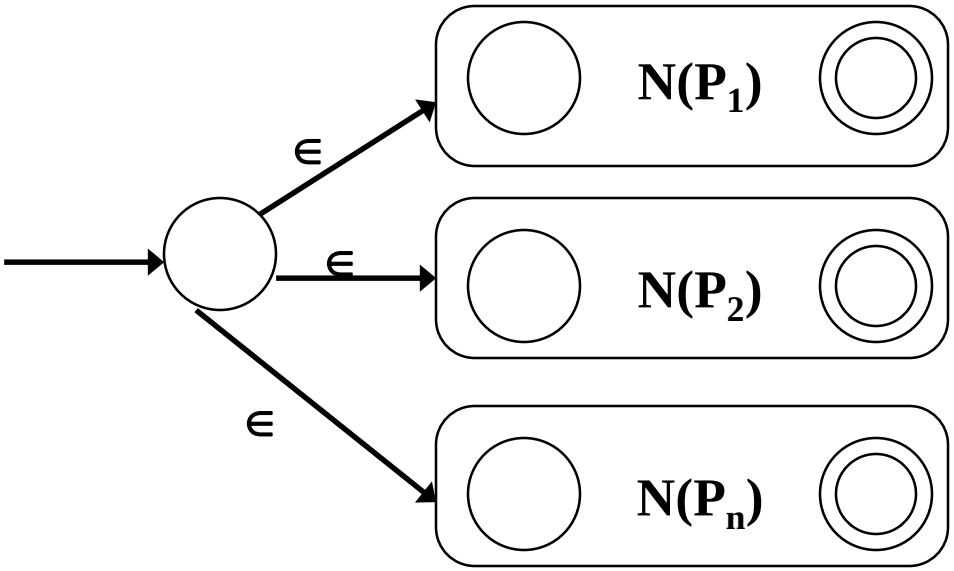

```
int state = 0, start = 0

lexeme_beginning = forward;
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
           /* c is lookahead character */
           if (c== blank || c==tab || c== newline) {
              state = 0;
              lexeme_beginning++;
                 /* advance
                   beginning of lexeme */
           }
           else if (c == '<') state = 1;
           else if (c == '=') state = 5;
           else if (c == '>') state = 6;
           else state = fail();
           break;
           … /* cases 1-8 here */
```

repeat until a "return" occurs



Pulling Together Concepts

- Let $P_1, P_2, \ldots, P_n$ be Lexer patterns
  (regular expressions for valid tokens in prog. lang.)
- Construct $N(P_1), N(P_2), \ldots N(P_n)$
- Note: accepting state of $N(P_i)$ will be marked by P_i
- Construct NFA:



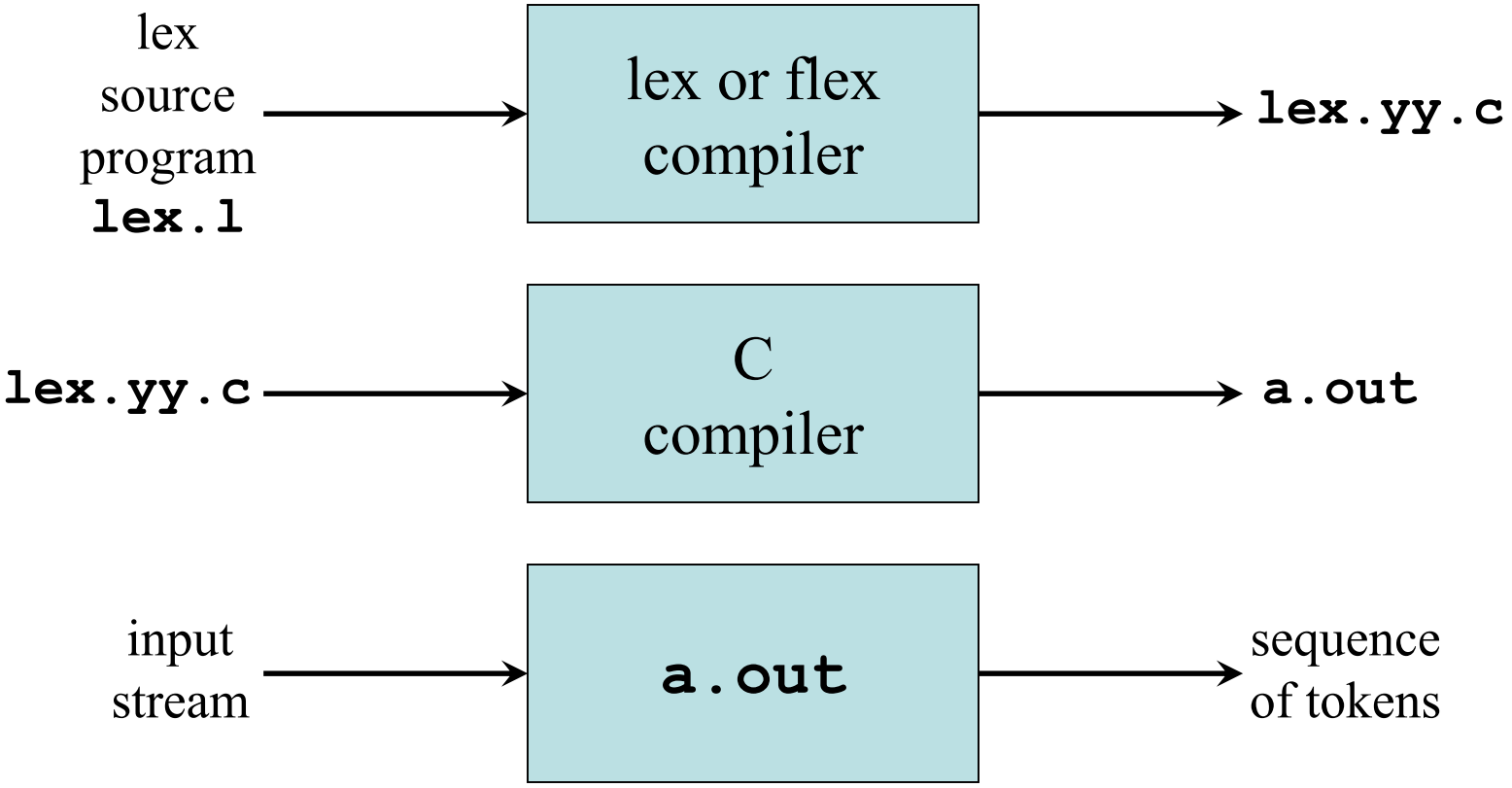
- Lexer applies conversion algorithm to construct DFA that is equivalent!

The Lex and Flex Scanner Generators

- Lex: a tool for automatically generating a lexer or scanner given a lex specification (.l file)
- A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens.
- For example, consider breaking a text file up into individual words.

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex/Flex



Lex Specification

- A *lex specification* consists of three parts:
  regular definitions, C declarations in %{ %}
  %%
  translation rules
  %%
  user-defined auxiliary procedures

- The *translation rules* are of the form:
  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

Regular Expressions in Lex

x	match the character **x**
\.	match the character **.**
"*string*"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
*r******	closure (match zero or more occurrences)
*r***+**	positive closure (match one or more occurrences)
*r***?**	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
r_1 **|** r_2	match r_1 or r_2 (union)
(*r* **)**	grouping
r_1 **/** r_2	match r_1 when followed by r_2
{*d***}**	match the regular expression defined by *d*

Example Lex Specification 1

Regular definitions

Translation rules

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Example Lex Specification 2


```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+|-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
"<>"       {yylval = NE; return RELOP;}
">"        {yylval = GT; return RELOP;}
">="       {yylval = GE; return RELOP `
%%
int install_id()
…
```

Return token to parser

Token attribute

Install yytext as identifier in symbol table

Lex Specification → Lexical Analyzer

- **Designing Lexical Analyzer Generator**

  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer

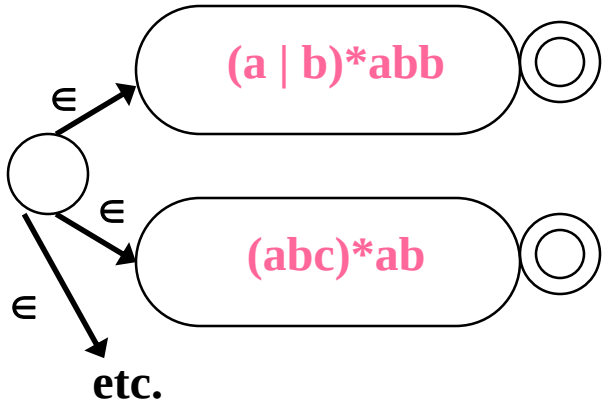
- **Recall Lex Structure**

  Pattern Action

  Pattern Action

  … …

e.g.



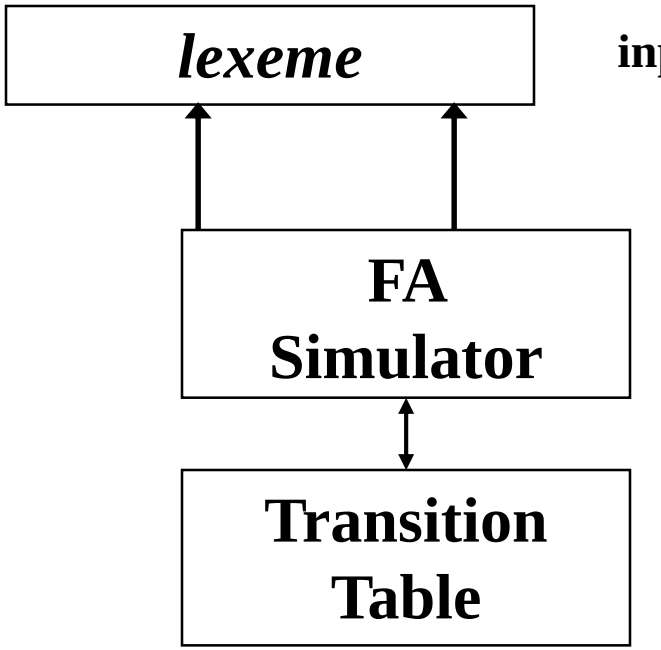
Recognizer!

- Each pattern recognizes lexemes

- Each pattern described by regular expression

Pictorially

(a) Lex Compiler



(b) Schematic lexical analyzer

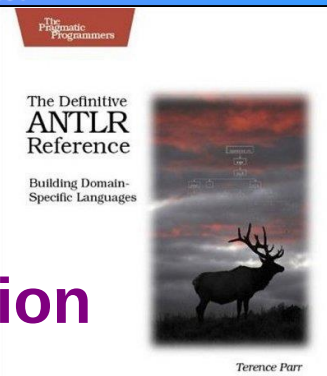
ANTLR

- ANother Tool for Language Recognition

- Terence Parr
- Prof. of Computer Science at the University of San Francisco

- Language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages
- Provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting

What does ANTLR do?



- Generates (the source code for) language processing tools from a grammatical description
- Commonly categorised as a *compiler generator* or *compiler compiler* in the tradition of tools.
- ANTLR can generate the source code for various tools that can be used to analyze and transform input in the language defined by the input grammar
- The basic types of language processing tools that ANTLR can generate are Lexers (a.k.a scanners, tokenizers), Parsers and TreeParsers (a.k.a tree walkers, *c.f.* visitors).

Tools

- **Lex / Yacc**
- **FLex / Bison**
- **JLex / CUP**
- **ANTLR**
- **JavaCC**

Conclusions

- ✧ Formal Lang.
- ✧ Grammar Simplification / Inference
- ✧ Lexical Analyzer
- ✧ Relating FA to Lexical Analysis
- ✧ LEX / ANTLR ...

推荐教学资料

- §3 Lexical Analysis
- §4.2 Context-Fee Grammars
- §2 Using Lex. Lex& Yacc. 2nd Edition.
- Part I: Introducing ANTLR and Computer Language. The Definitive ANTLR 4 Reference
- 熟悉Flex及ANTLR

Thank you!