Chap. 3 Lexical Analysis

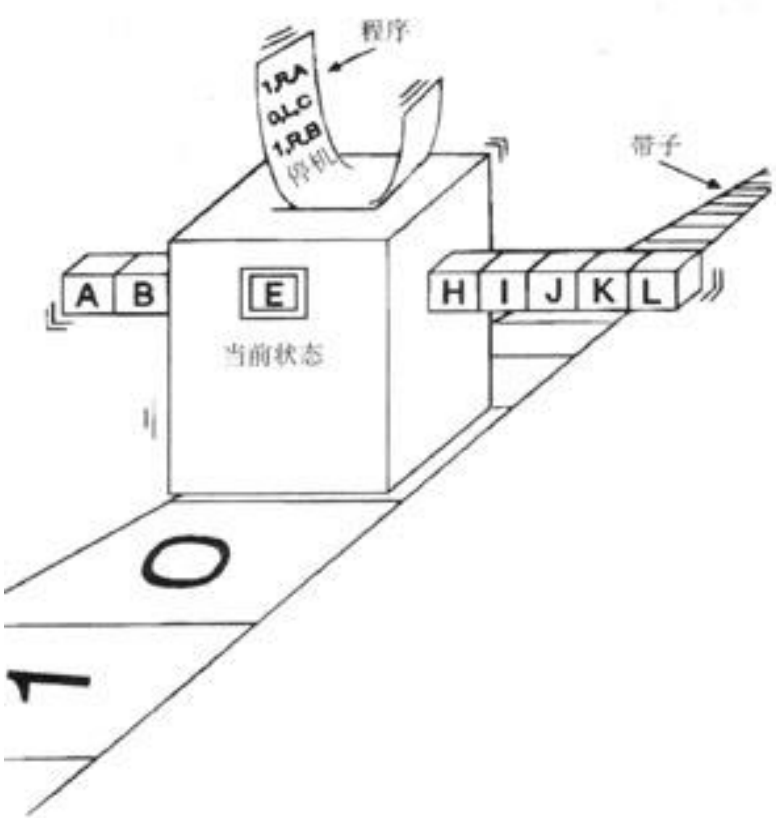
Chaokun Wang
IISE@Tsinghua

Outline

- 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

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* : $S \times \Sigma \cup \{\varepsilon\} \rightarrow Pow(S)$**
- **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]].

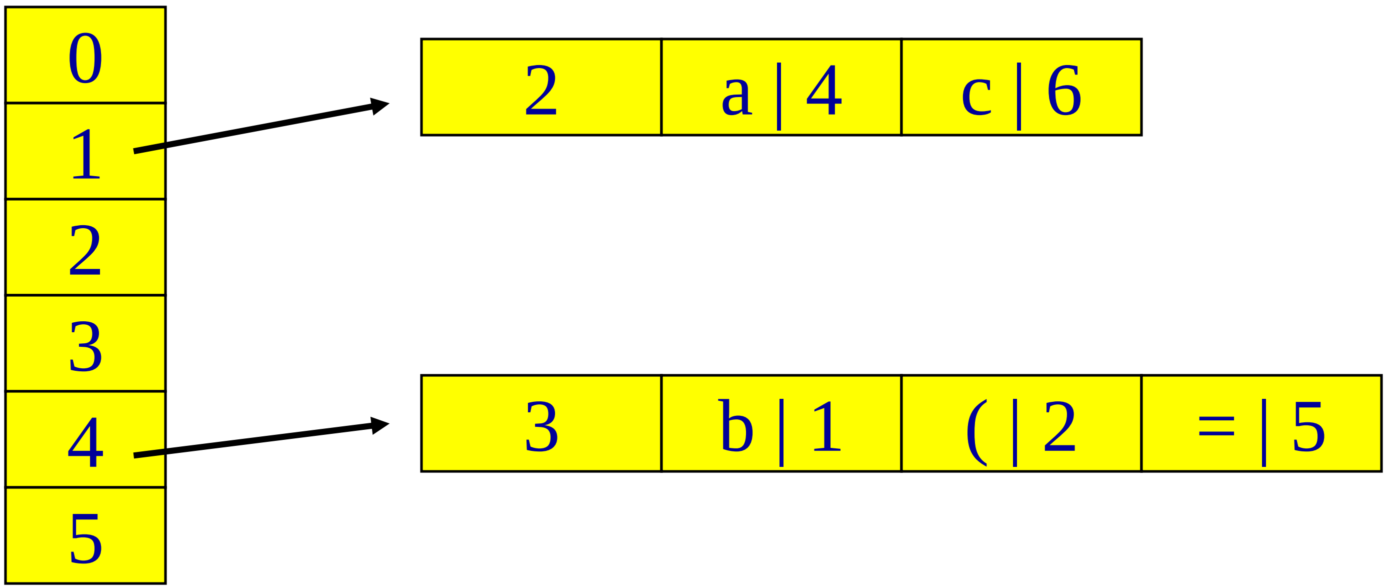
a b c …. 1 2 3 …. =

1 1 1….. 2 2 2 ….. 3

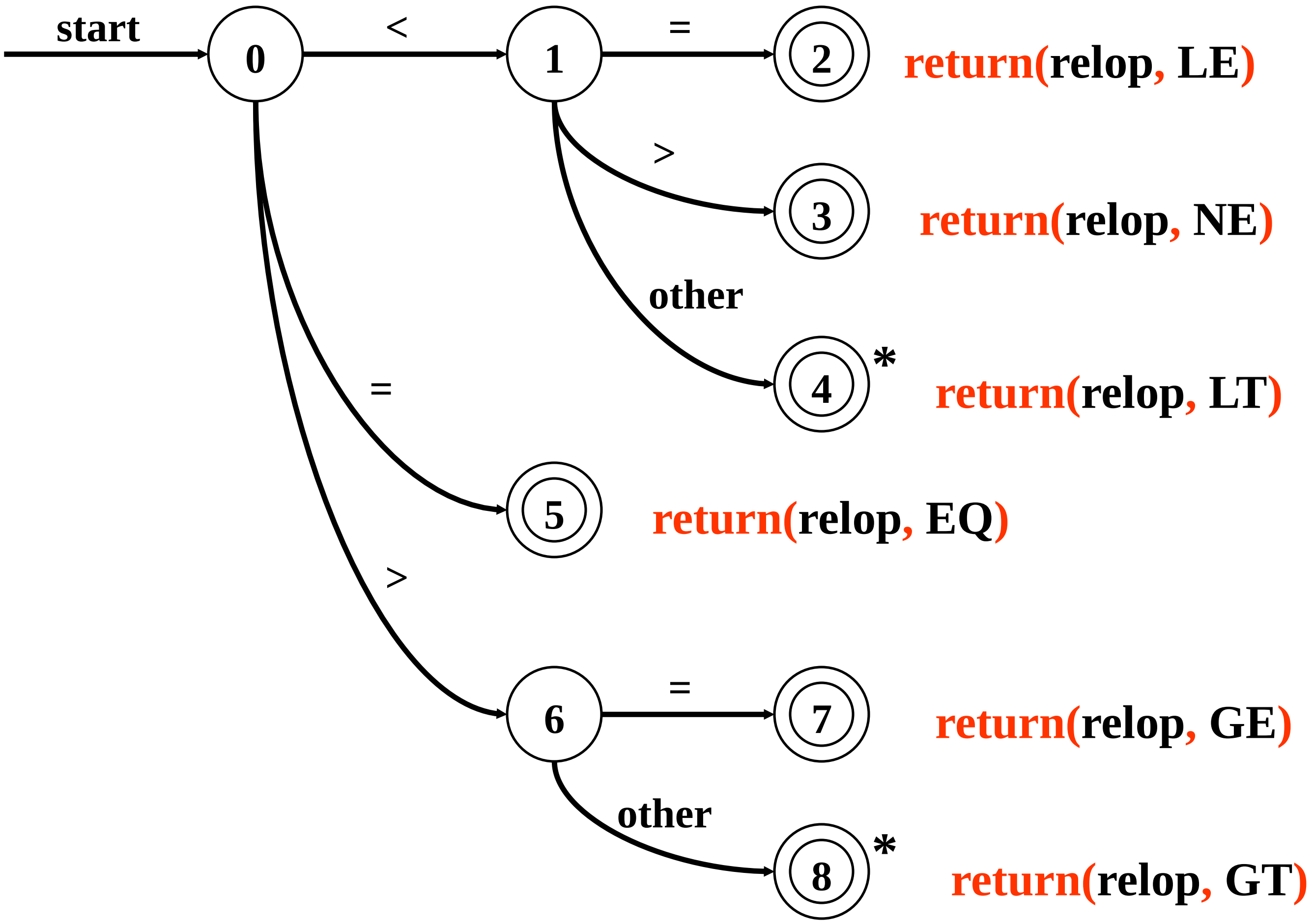
1	1
2	1
3	2
4	3
5	4
6	3
7	5
8	5

1	2	5
2	3	4
3	4	4
2	4	7
4	3	6

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

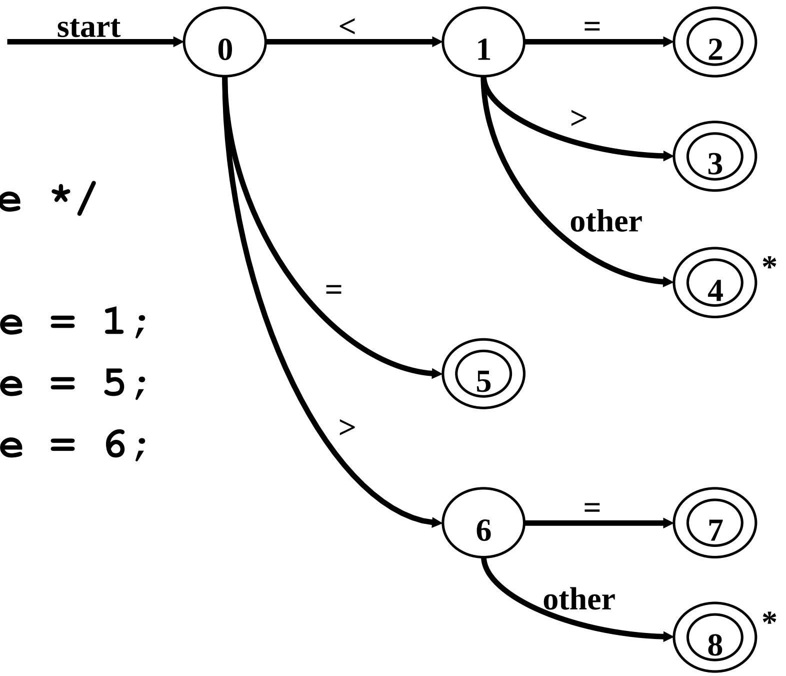
FUNCTIONS USED

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

Implementing Transition Diagrams

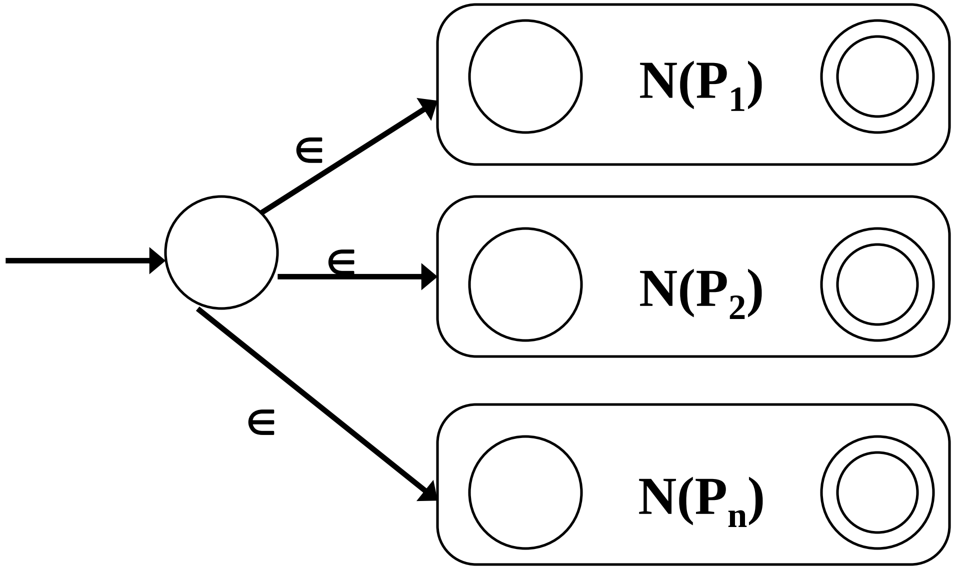

```
int state = 0, start = 0
lexeme_beginning = forward;
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c is lookahead character */
            if (c== blank || c==tab || c== newline) {
                state = 0;
                lexeme_beginning++;
                  /* 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, … , P_n be Lexer patterns
  (regular expressions for valid tokens in prog. lang.)
- Construct $N(P_1)$, $N(P_2)$, … $N(P_n)$
- Note: accepting state of $N(P_i)$ will be marked by P_i
- Construct NFA:



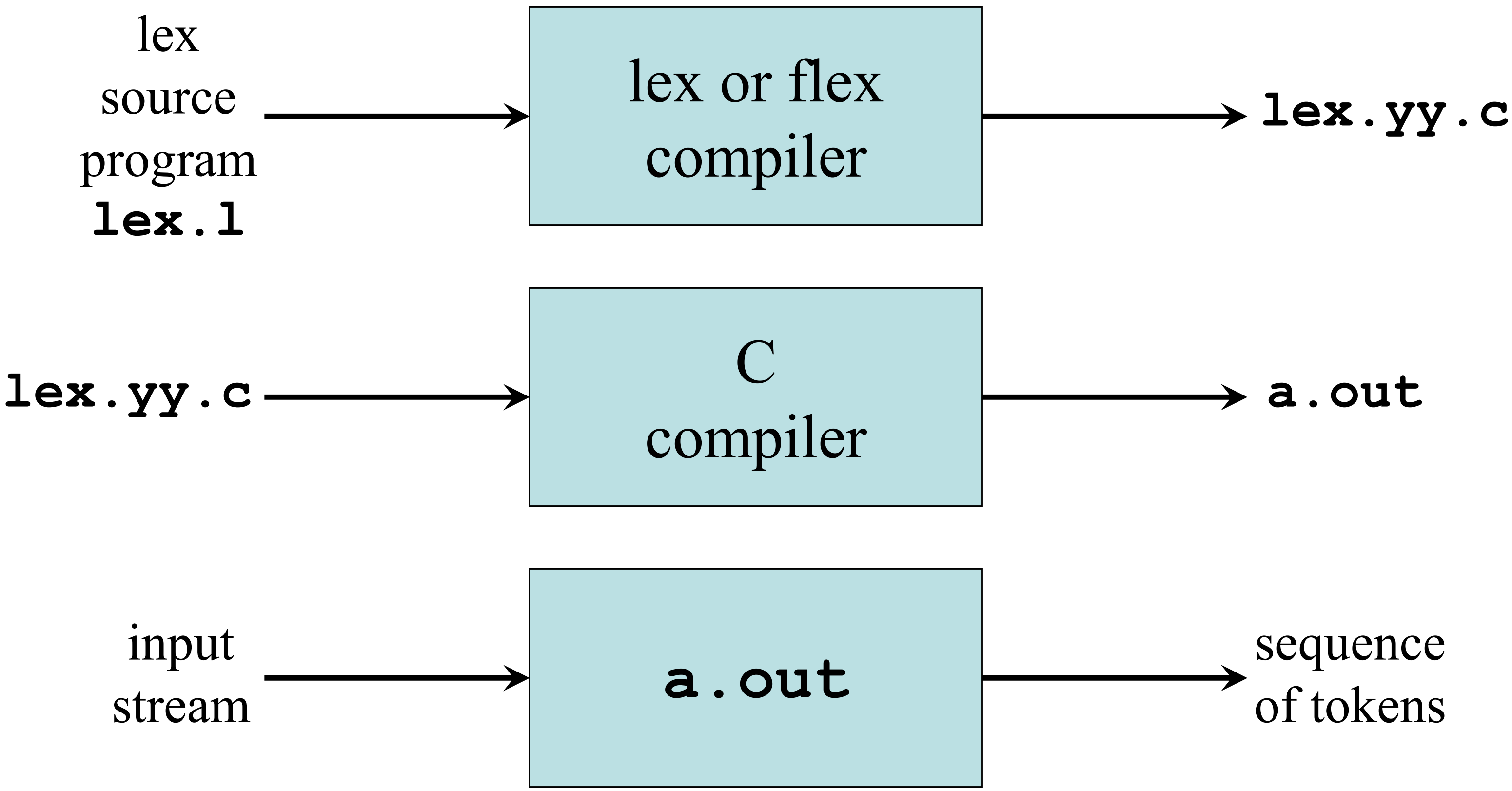
- Lexer applies conversion algorithm to construct DFA that is equivalent!

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

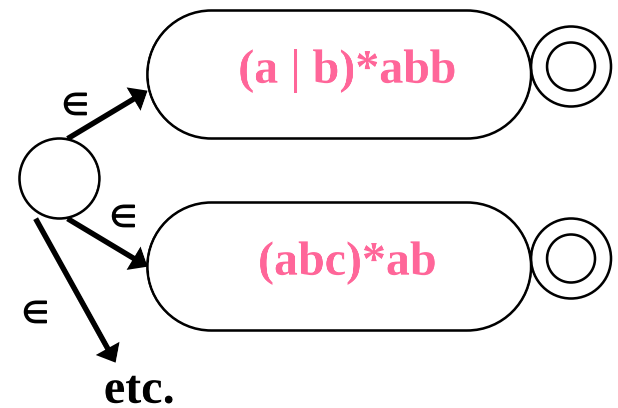
  NFA → DFA conversion

  DFA simulation for lexical analyzer

- **Recall Lex Structure**

Pattern	**Action**
Pattern	**Action**
…	**…**

e.g.



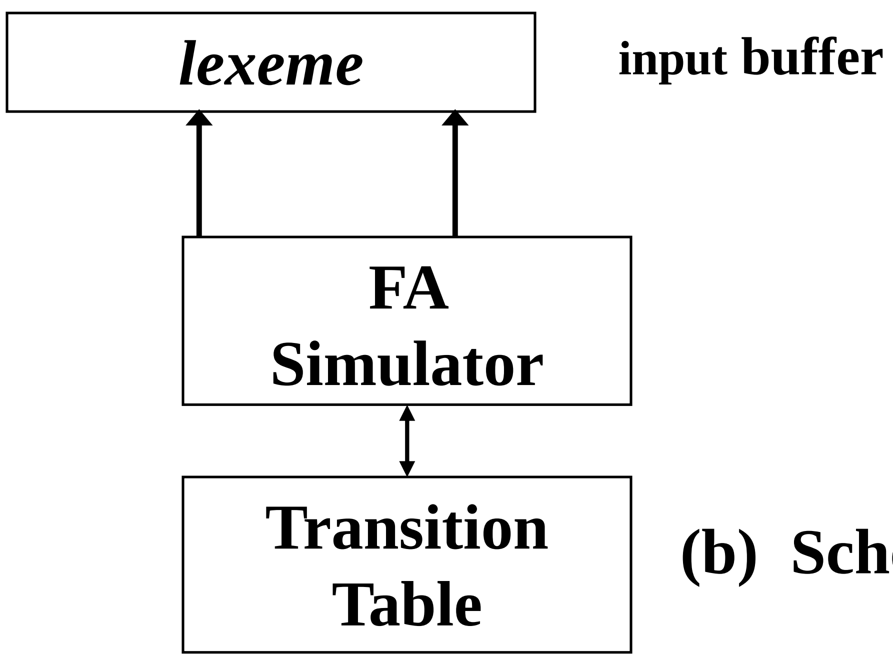
Recognizer!

- Each pattern recognizes lexemes

- Each pattern described by regular expression

Pictorially

(a) Lex Compiler



(b) Schematic lexical analyzer

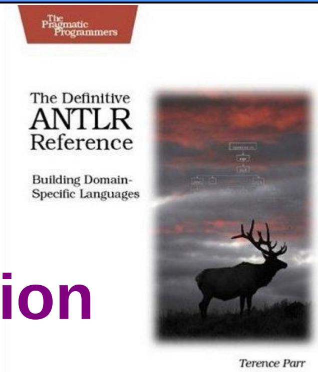
ANTLR

- 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 generates 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**

Chapter 4
Syntax Analysis

王朝坤
IISE@Tsinghua

Outline

- ✧ 语法分析的基本思想*
- ✧ 带回溯的自顶向下分析
- ✧ 预测分析
- ✧ LL(1)文法**
  - ✧First集；Follow集
- ✧ 递归下降LL(1)分析*
- ✧ 表驱动LL(1)分析*
- ✧ 文法变换*
- ✧ 错误处理

基本思想

- ✧ 语法分析程序（*Parser*）的作用
  - – 分析源程序的单词流是否符合语言的语法规则
  - – 报告语法错误
  - – 产生源程序的语法分析结果，
    以语法分析树或与之等价的形式体现出来
- ✧ 语法规则描述工具
  - – 通常是一种上下文无关文法
  - – 语法分析的核心即为针对上下文无关文法的句型分析

基本思想

✧ 语法分析

– 句型分析

对任意上下文无关文法$G = (V，T，P，S)$和任意$w \in T^*$，是否有$w \in L(G)$？若成立，则给出分析树或（最左）推导步骤；否则，进行报错处理。

– 三种实现途径

通用分析（*Cocke-Younger-Kasami算法*）
自顶向下（*top-down*）分析
自底向上（*bottom-up*）分析

基本思想

✧ 自顶向下分析思想

- 从文法开始符号出发进行推导；每一步推导都获得文法的一个句型；直到产生出一个句子，恰好是所期望的终结符串
- 每一步推导是对当前句型中剩余的某个非终结符进行扩展，即用该非终结符的一个产生式的右部替换该非终结符
- 如果不存在任何一个可以产生出所期望的终结符串的推导，则表明存在语法错误

基本思想（例）

✧ 自顶向下分析举例

– 单词序列 aaab 的一个自顶向下分析过程

文法 G（S）：

$S \rightarrow AB$

$A \rightarrow aA \mid \varepsilon$

$B \rightarrow b \mid bB$

$$
\begin{array}{ll}
\quad S & (S \rightarrow AB) \\
\Rightarrow AB & (A \rightarrow aA) \\
\Rightarrow aAB & (B \rightarrow b) \\
\Rightarrow aAb & (A \rightarrow aA) \\
\Rightarrow aaAb & (A \rightarrow aA) \\
\Rightarrow aaaAb & (A \rightarrow \varepsilon) \\
\Rightarrow aaab &
\end{array}
$$

带回溯的自顶向下分析

- 一般方法
  - 两类非确定性

    在每一步推导中，选择哪一个非终结符、哪一个产生式都可能是非确定的

    分析成功的结果：得到一个推导

✧ 举例

– 单词序列 aaab 的一个自顶向下分析过程

文法 G（S）：

（1）$S \rightarrow AB$
（2）$A \rightarrow aA$
（3）$A \rightarrow \varepsilon$
（4）$B \rightarrow b$
（5）$B \rightarrow bB$

S	（1）
$\Rightarrow AB$	（2）
$\Rightarrow aAB$	（5）
$\Rightarrow aAbB$	（2）
$\Rightarrow aaAbB$	（2）
$\Rightarrow aaaAbB$	（3）
$\Rightarrow aaabB$	（回溯）

……

复杂度很高
失败条件较复杂

带回溯的自顶向下分析

✧ 改进的方法

– 仅有产生式选择是非确定的

在每一步推导中，总是对最左边的非终结符进行展开，但选择哪一个产生式是非确定的

分析成功的结果：得到一个最左推导

原理：每个合法的句子都存在至少一个起始于开始符号的最左推导；一个终结符串，只要存在一个起始于开始符号的最左推导，它就是一个合法的句子

从左向右扫描输入单词，失败条件较简单

✧ 改进的方法举例

– 单词序列 aaab 的一个自顶向下分析过程

文法 G（S）：

（1） $S \rightarrow AB$
（2） $A \rightarrow aA$
（3） $A \rightarrow \varepsilon$
（4） $B \rightarrow b$
（5） $B \rightarrow bB$

S	（1）
$\Rightarrow AB$	（2）
$\Rightarrow aAB$	（3）
$\Rightarrow aB$	（回溯）
$\Rightarrow aAB$	（2）
$\Rightarrow aaAB$	（2）
$\Rightarrow aaaAB$	（3）
$\Rightarrow aaaB$	（5）
$\Rightarrow aaabB$	（回溯）
$\Rightarrow aaaB$	（4）
$\Rightarrow aaab$	（成功）

复杂度降低
失败条件简化

预测分析

✧ 确定的自顶向下分析

– 非终结符选择和产生式选择都是确定的

在每一步推导中，总是对最左边的非终结符进行展开，且选择哪一个产生式是确定的，因此是一种无回溯的方法

从左向右扫描，可能向前查看（lookahead）确定数目的单词

分析成功的结果：得到唯一的最左推导

分析条件：对文法需要有一定的限制

✧ 举例（向前查看 2 个单词）

– 单词序列 $a^n b^m$（$n \geq 0, m > 0$）的预测分析过程

文法 G（S）：

（1） $S \rightarrow AB$
（2） $A \rightarrow aA$
（3） $A \rightarrow \varepsilon$
（4） $B \rightarrow b$
（5） $B \rightarrow bB$

只要向前查看 2 个单词，就可预测分析L(G)中所有句子

S	（1）
$\Rightarrow AB$	（2）
$\Rightarrow aAB$	（2）
……	……
$\Rightarrow a^n AB$	（3）
$\Rightarrow a^n B$	（5）
$\Rightarrow a^n bB$	（5）
……	……
$\Rightarrow a^n b^{m-1} B$	（4）
$\Rightarrow a^n b^m$	（成功）

✧ 左递归带来的问题

– 考虑下列文法识别 ba^n 的分析过程

文法 G（S）：

（1）$S \rightarrow Sa$

（2）$S \rightarrow b$

$$
\begin{array}{ll}
\quad S & (1) \\
\Rightarrow Sa & (1) \\
\Rightarrow Saa & (1) \\
\Rightarrow Saaa & (1) \\
\ldots\ldots & \ldots\ldots \\
\Rightarrow Sa^n & (2) \\
\Rightarrow ba^n &
\end{array}
$$

需要向前查看n+1个单词，才能确定这样的推导序列

但是：无论向前查看的单词数确定为多少，都无法满足预测分析L(G)中所有句子的需求

预测分析

✧ 要求文法不含左递归

– 例：直接左递归

$P \rightarrow Pa$
$P \rightarrow b$
……

– 例：间接左递归

$P \rightarrow Aa$
$A \rightarrow Pb$
……

– 可以通过文法变换消除左递归

专门讨论

✧ 左公因子带来的问题

– 如下文法需要向前查看 3 个单词来预测分析 L(G)中的句子

文法 G（S）：$S \rightarrow abA \mid abB$
$A \rightarrow a$
$B \rightarrow b$

文法 G'（S）：$S \rightarrow aAb \mid aAc$
$A \rightarrow a \mid aA$

– 对于文法G' 无法确定需要向前查看多少个单词来预测分析 L(G) 中的句子

预测分析

- 通常要求文法不含左公因子
  - 可以通过文法变换消除左公因子
    专门讨论

预测分析

- 应用较普遍的预测分析是LL（1）分析
  - 要求文法一定是LL（1）文法

    专门讨论
  - LL（1）分析程序既可以手工构造，也可以自动构造

LL（1） 分析

✧ LL（1）的含义

– 第一个“L”, 代表从左（Left）向右扫描单词

– 第二个“L”, 代表产生的是最左（Leftmost）推导

– “1”代表向前查看（lookahead）一个单词

LL（1）分析

- 对文法的限制
  - 要求文法是LL（1）的
  - 什么是LL（1）文法?
- 两个重要概念
  - First 集合
  - Follow 集合

An Example:
S =>* abcAde;
a ∊ first(S);
d ∊ follow(A).

LL（1）分析

✧ First 集合

– 定义

设 $G = (V_T, V_N, P, S)$ 是上下文无关文法

对 $\alpha \in (V_T \cup V_N)^*$,

$\text{First}(\alpha) = \{ a \mid \alpha \overset{*}{\Rightarrow} a\beta, a \in V_T, \beta \in (V_T \cup V_N)^* \}$

若$\alpha \overset{*}{\Rightarrow} \varepsilon$则规定$\varepsilon \in \text{First}(\alpha)$

✧ 计算 First 集合

设 $\alpha = X \in V_N \cup V_T$, 则First(X) 可按如下步骤计算:

- 若 $X \in V_T$,则First(X)={X}
- 若X→ε 也是一个产生式,则把 ε 也加到First(X)中；

若 $X \in V_N$,且有产生式X→a…， $a \in V_T$, 则把a加入到First(X)中;

若 $X \rightarrow Y_1Y_2 \ldots Y_K$ 是一个产生式, $Y_1, Y_2, \ldots, Y_k \in V_N \cup V_T$,

1) 对于任何 j:1≤j≤i-1, 1≤i ≤k, First(Y_j)都含有ε, 但 First(Y_i)不含ε, 则把 First(Y_j)中的所有非ε元素和First(Y_i)中的所有元素都加到First(X)中;

2) 特别是,对于任何 j:1≤j≤k，First(Y_j)都含ε，则除First(Y_j)中的非ε元素外，把ε也加到First(X)中.

LL（1）分析

✧ 计算 First 集合

设$\alpha = X_1X_2 \ldots X_n$,则First($\alpha$) 可按如下步骤计算:

– 若对于任何j:1≤j≤i-1<n, 有

$\varepsilon \in First(X_j) \wedge \varepsilon \notin First(X_i)$,

则 $First(\alpha) = \bigcup_{j=1}^{i-1} First(X_j) \cup First(X_i) - \{\varepsilon\} = \bigcup_{j=1}^{i} First(X_j) - \{\varepsilon\}$

– 若所有的j,1≤j ≤n, 都有$\varepsilon \in First(X_j)$,

则 $First(\alpha) = \bigcup_{j=1}^{n} First(X_j)$

S → ES'
S' → ε | +S
E → number | (S)

FIRST(S) = {number, (}
FIRST(S') = {ε, + }
FIRST(E) = { number, (}

LL（1）分析

✧ Follow 集合

– 定义

设 $G=(V_T, V_N, P, S)$ 是上下文无关文法

对 $A \in V_N$,

$$\mathrm{Follow}(A)=\{a \mid S \overset{*}{\Rightarrow} \alpha A\beta \text{ 且 } a \in \mathrm{First}(\beta),\ \alpha \in (V_T \cup V_N)^*,\ \beta \in (V_T \cup V_N)^+\}$$

若 $S \overset{*}{\Rightarrow} \alpha A\beta$, 且 $\beta \overset{*}{\Rightarrow} \varepsilon$, 则规定 $\# \in \mathrm{Follow}(A)$

（# 代表输入单词序列右边的结束符）

✧ 计算 Follow 集合

- 对于文法的开始符号*S*，置#于Follow(*S*) 中;
- 若$A \rightarrow \alpha B\beta$是一个产生式,则把 First($\beta$)–{ε}
  加至 Follow(*B*) 中;
- 若 $A \rightarrow \alpha B$ 是一个产生式, 或$A \rightarrow \alpha B\beta$是一个
  产生式而$\beta \stackrel{*}{\Rightarrow}$ ε （即ε∈First(β)），
  则把 Follow(*A*)加至Follow(B)中.

S → ES'
S' → ε | +S
E → number | (S)

FOLLOW(S) = { #,) }
FOLLOW(S') = FOLLOW(S)={ #,) }
FOLLOW(E) = (FIRST(S') -{ε}) ∪FOLLOW(S)
= { +, # ,) }

✧ 例：计算 First 和 Follow 集合

文法 G（S）：

（1）S → AB
（2）A → Da | ε
（3）B → cC
（4）C → aADC | ε
（5）D → b | ε

First(D) ={b, ε}
First(C) ={a, ε}
First(B) ={c}
First(A) ={b,a, ε}
First(S) ={b,a,c}

Follow(S) = {#}
Follow(C) = {#}
Follow(A) = {c,b,a, #}
Follow(B) = {#}
Follow(D) = {a,#}

LL（1）分析

✧ 定义： LL（1）文法

文法G是LL（1）的，当且仅当对于G的每个非终结符A的任何两个不同产生式$A\to\alpha\mid\beta$，下面的条件成立：

- First(α)∩First(β)=ϕ，即 α 和 β 推导不出以同一个单词为首的符号串，也不会同时推导出 ε
- 假若$\beta\overset{*}{\Rightarrow}\varepsilon$，那么First(α)∩Follow(A)＝ϕ,即α所能推出的串的首符号不应在Follow(A）中.

LL（1）分析

✧ 举例：判断如下文法G（S）是否是LL(1)文法

文法 G（S）：

（1）$S \rightarrow AB$

（2）$A \rightarrow Da \mid \varepsilon$

（3）$B \rightarrow cC$

（4）$C \rightarrow aADC \mid \varepsilon$

（5）$D \rightarrow b \mid \varepsilon$

First(D) = {b, ε}

First(C) = {a, ε}

First(B) = {c}

First(A) = {b,a, ε}

First(S) = {b,a,c}

Follow(S) = {#}

Follow(C) = {#}

Follow(A) = {c,b,a, #}

Follow(B) = {#}

Follow(D) = {a,#}

For A：first(Da) ∩ follow(A)={b,a} ∩{c,b,a,#} ={b,a}

∴G(S)不是 LL(1)文法

LL（1） 分析

✧ LL(1)文法的性质

LL(1)文法是无二义的

LL(1)文法是无左递归的

LL(1)文法是无左公因子的

除了利用定义外，有时可以利用这些性质判定某些文法不是LL(1)的

LL（1） 分析

✧ LL（1）分析的实现

– 递归下降 LL（1）分析

（递归下降分析：非终结符 ⇔子程序）

– 表驱动 LL（1）分析

借助于预测分析表和一个下推栈

✧ 递归下降LL(1)分析程序

- 工作原理

  每个非终结符都对应一个子程序。该子程序的行为根据语法描述来明确：

  - 每遇到一个终结符，则判断当前读入的单词是否与该终结符相匹配，若匹配，再读取下一个单词继续分析；不匹配，则进行出错处理
  - 每遇到一个非终结符，则调用相应的子程序

递归下降 LL（1）分析

- ✧ 非终结符对应的递归下降子程序
  - – 例 对于下列文法 （其中 function，identifier，parameter_list 和 statement 是非终结符）

function → FUNC identifier (parameter_list) statement

```
void ParseFunction( )
{
    MatchToken(T_FUNC);      //匹配FUNC
    ParseIdentifier( );
    MatchToken(T_LPAREN);  // 匹配 （
    ParseParameterList( );
    MatchToken(T_RPAREN);  // 匹配 ）
    ParseStatement( );
}
```

递归下降LL（1）分析

✧ 非终结符对应的递归下降子程序

– 例 续上页

```
void MatchToken(int expected)
{
  if (lookahead != expected)  //判别当前单词是否与
  {                            //期望的终结符匹配
    printf("syntax error \n");
    exit(0);
  }
  else        // 若匹配,消费掉当前单词并读入下一个
    lookahead = nexttoken();   //调用词法分析程序
}
```

递归下降 LL（1）分析

- 非终结符对应的递归下降子程序
  - 一般结构

设 *A* 的产生式:

$$A \rightarrow u_1 \mid u_2 \mid \ldots,$$

相对于非终结符A的递归下降子程序 **ParseA**的一般结构如右图所示

```
void ParseA()
{
    switch (lookahead) {
      case First(u1):
          /* code to recognize u1 */
          break;
      case First(u2):
          /* code to recognize u2 */
          break;
      ...
      case Follow(A): /* when A ⇒* ε */
          /* usually do nothing here */
          break;
      default:
          printf("syntax error \n");
          exit(0);
    }
}
```

✧ 递归下降LL(1)分析程序举例

– 例 对于下列文法

G(S):

$$S \rightarrow AaS \mid BbS \mid d$$
$$A \rightarrow a$$
$$B \rightarrow \varepsilon \mid c$$

First（S）= {a, *b*, *c* , *d* }

First（A）= {a}

First（B）= {ε, *c* }

Follow（S）= {#}

Follow（A）= {a }

Follow（B）= {b}

因为 First(*AaS*)={a}，First(*BbS*)={b, c}，以及 First(*d*)={d} 之间两两互不相交，同时 Follow(B) = {b} 与 First(*c*)={ c}不相交，所以，G(S)是LL(1)文法

– 接上例 针对文法G(S)构造的递归下降分析程序

G(S): $S \to AaS \mid BbS \mid d$

$A \to a$

$B \to \varepsilon \mid c$

First（S）= {a, *b*, *c* , *d* }

Follow（S）= {#}

```
void ParseS( )
{
    switch (lookahead)  {
        case a:
            ParseA( );
            MatchToken(a);
            ParseS( );
            break;
        case b,c:
            ParseB( );
            MatchToken(b);
            ParseS( );
            break;
        case d:
            MatchToken(d);
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
}
```

– 接上例 针对文法G(S) 构造的递归下降分析程序

G(S): $S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

First（A）= {a}
First（B）= {ε, c }
Follow（S）= {#}
Follow（A）= {a }
Follow（B）= {b}

```
void ParseB( )
{
    if (lookahead==c)  {
        MatchToken(c);
    }
    else if (lookahead==b) {
        }
        else {
            printf("syntax error \n");
            exit(0);
        }
}
```

```
void ParseA( )
{
    if (lookahead==a)  {
        MatchToken(a);
    }
    else {
        printf("syntax error \n");
        exit(0);
    }
}
```

✧ 表驱动LL(1)分析程序

– 工作原理 利用预测分析表和一个下推栈实现

（0）初始化，将符号#入栈；

（1）文法开始符号入栈；

（2）若栈顶为 终结符，则判断当前读入的单词是否与该终结符相匹配，

(a) 若匹配，再读取下一 单词继续分析；

(b) 不匹配，则进行出错处理；

（3）若栈顶为非终结符，则根据该非终结符和当前输入单词查预测分析表，

(a) 若相应表项中是产生式（唯一的），则将此非终结符出栈，并把产生式右部符号从右至左入栈；

(b) 若表项为空，则进行出错处理；

（4）重复（2）和（3），直到栈顶为 # 同时输入也遇到结束符 # 时，分析结束

✧ 预测分析表

- 表驱动分析程序需要的二维表*M*
- 表的每一行*A* 对应一个非终结符
- 表的每一列a 对应某个终结符或输入结束符#
- 表中的项*M*(*A*,a) 表示栈顶为*A*，下一个输入符号为a时，可选的产生式集合
- 对于LL（1）文法，可以构造出一个*M*(*A*,a) 最多只包含一个产生式的预测分析表，可称之为LL（1）分析表
- *M*(*A*,a) 不含产生式时，对应一个出错位置

✧ 预测分析表的构造算法

– 对文法G的每个产生式 $A \rightarrow \alpha$ 执行如下步骤：

（1）对每个 a ∈ First(α)，把 $A \rightarrow \alpha$ 加入$M[A,a]$

（2）若 ε ∈ First(α)，则对任何 b ∈ Follow(A)，把 $A \rightarrow \alpha$ 加至$M[A,b]$中

– 把所有无定义的$M[A,a]$标上“出错标志”

可以证明: 一个文法G的预测分析表不含多重入口，当且仅当该文法是LL(1)的

表驱动 LL（1）分析

✧ 预测分析表的构造举例

– 对于下列文法**G(S)**:

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

First（S）= {*a*, *b*, *c* , *d* }
First（A）= {*a*}
First（B）= {*ε*, *c* }
Follow（S）= {#}
Follow（A）= {*a* }
Follow（B）= {*b*}

可构造如下预测分析表:

	a	b	c	d	#
S	$S \to AaS$	$S \to BbS$	$S \to BbS$	$S \to d$	
A	$A \to a$				
B		$B \to \varepsilon$	$B \to c$		

✧ 表驱动预测分析程序分析算法

初始时 '#'入栈，然后文法开始符号入栈；首个输入符号读进 **a** ；

```
flag =TRUE;
while (flag)  do   {
    栈顶符号出栈并放在X中；
    if (X∈V_T)  {
        if  (X==a)
            把下一个输入符号读进a;
        else ERROR;
     }
     else if  (X=='#'）{
        if  (a=='#')  flag = FALSE;
        else ERROR;
     }
     else if (M[X,a] == {X→X1X2…Xk}) Xk,XK-1,…,X1依次进栈;
     else  ERROR;
}
/*分析成功，过程完毕* /
```

✧ 表驱动预测分析过程举例

– 对于下列文法**G(*S*)**:

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

剩余的输入串

aabd#

栈
S
#

分析输入串 ***aabd*** 的过程:

	a	b	c	d	#
S	$S \to AaS$	$S \to BbS$	$S \to BbS$	$S \to d$	
A	$A \to a$				
B		$B \to \varepsilon$	$B \to c$		

✧ 表驱动预测分析过程举例

– 对于下列文法**G(S)**:

$$S \rightarrow AaS \mid BbS \mid d$$
$$A \rightarrow a$$
$$B \rightarrow \varepsilon \mid c$$

剩余的输入串

aabd#

A
a
S
#

分析输入串 ***aabd*** 的过程:

	a	b	c	d	#
S	$S \rightarrow AaS$	$S \rightarrow BbS$	$S \rightarrow BbS$	$S \rightarrow d$	
A	$A \rightarrow a$				
B		$B \rightarrow \varepsilon$	$B \rightarrow c$		

表驱动预测分析过程举例

– 对于下列文法**G**(***S***)：

$$S \rightarrow AaS \mid BbS \mid d$$
$$A \rightarrow a$$
$$B \rightarrow \varepsilon \mid c$$

剩余的输入串

aabd#

a
a
S
#

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	$S \rightarrow AaS$	$S \rightarrow BbS$	$S \rightarrow BbS$	$S \rightarrow d$	
A	$A \rightarrow a$				
B		$B \rightarrow \varepsilon$	$B \rightarrow c$		

✧ 表驱动预测分析过程举例

– 对于下列文法**G(S)**:

$$S \rightarrow AaS \mid BbS \mid d$$
$$A \rightarrow a$$
$$B \rightarrow \varepsilon \mid c$$

剩余的输入串

abd#

栈：

a
S
#

分析输入串 ***aabd*** 的过程:

	a	b	c	d	#
S	$S\rightarrow AaS$	$S\rightarrow BbS$	$S\rightarrow BbS$	$S\rightarrow d$	
A	$A\rightarrow a$				
B		$B\rightarrow\varepsilon$	$B\rightarrow c$		

表驱动 LL（1）分析

✧ 表驱动预测分析过程举例

– 对于下列文法*G*(*S*):

$S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

剩余的输入串

bd#

S
#

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	*S*→*AaS*	*S*→*BbS*	*S*→*BbS*	*S*→*d*	
A	*A*→*a*				
B		*B*→*ε*	*B*→*c*		

✧ 表驱动预测分析过程举例

– 对于下列文法**G(S)**:

$$S \rightarrow AaS \mid BbS \mid d$$
$$A \rightarrow a$$
$$B \rightarrow \varepsilon \mid c$$

剩余的输入串

bd#

分析栈：**B** **b** **S** **#**

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	S→AaS	S→BbS	S→BbS	S→d	
A	A→a				
B		B→ε	B→c		

✧ 表驱动预测分析过程举例

– 对于下列文法*G*(*S*)：

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

剩余的输入串
bd#

b
S
#

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	$S\to AaS$	$S\to BbS$	$S\to BbS$	$S\to d$	
A	$A\to a$				
B		$B\to\varepsilon$	$B\to c$		

✧ 表驱动预测分析过程举例

– 对于下列文法G(S)：

$$S \to AaS \mid BbS \mid d$$
$$A \to a$$
$$B \to \varepsilon \mid c$$

剩余的输入串

d#

S
#

分析输入串 *aabd* 的过程：

	a	b	c	d	#
S	$S \to AaS$	$S \to BbS$	$S \to BbS$	$S \to d$	
A	$A \to a$				
B		$B \to \varepsilon$	$B \to c$		

✧ 表驱动预测分析过程举例

– 对于下列文法**G(S)**:

$S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

剩余的输入串

d#

d
#

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	$S\rightarrow AaS$	$S\rightarrow BbS$	$S\rightarrow BbS$	$S\rightarrow d$	
A	$A\rightarrow a$				
B		$B\rightarrow \varepsilon$	$B\rightarrow c$		

✧ 表驱动预测分析过程举例

– 对于下列文法**G(*S*)**：

$S \rightarrow AaS \mid BbS \mid d$

$A \rightarrow a$

$B \rightarrow \varepsilon \mid c$

剩余的输入串

#

☺

#

分析输入串 ***aabd*** 的过程：

	a	b	c	d	#
S	S→AaS	S→BbS	S→BbS	S→d	
A	A→a				
B		B→ε	B→c		

文法变换

✧ 文法变换：消除左递归、提取左公因子

- LL(1)文法不含左递归和左公因子
- 许多文法在消除左递归和提取左公因子后可以变换为LL(1)文法
- 但不含左递归和左公因子的文法不一定都是LL(1)文法

✧ 左递归消除规则

– 消除直接左递归

对形如

$$P \rightarrow P\alpha \mid \beta$$

的产生式，其中α非ε，α，β不以 P 打头，

可改写为：

$$P \rightarrow \beta Q$$

$$Q \rightarrow \alpha Q \mid \varepsilon$$

其中Q为新增加的非终结符

✧ 左递归消除规则

– 消除直接左递归

对更一般的形如

$$P \rightarrow P\alpha_1 \mid P\alpha_2 \mid \ldots \mid P\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

的一组产生式，其中α_i（$1 \le i \le m$）不为ε，β_j（$1 \le j \le n$）不以 P 打头，

可改写为：

$$P \rightarrow \beta_1 Q \mid \beta_2 Q \mid \ldots \mid \beta_n Q$$

$$Q \rightarrow \alpha_1 Q \mid \alpha_2 Q \mid \ldots \mid \alpha_m Q \mid \varepsilon$$

其中Q为新增加的非终结符

✧ 左递归消除举例

原文法 G[E]:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a$$

消除左递归后的文法 G'[E]:

(1) $E \rightarrow TE'$　　(2) $E' \rightarrow + TE'$　　(3) $E' \rightarrow \varepsilon$

(4) $T \rightarrow FT'$　　(5) $T' \rightarrow * FT'$　　(6) $T' \rightarrow \varepsilon$

(7) $F \rightarrow (E)$　　(8) $F \rightarrow a$

✧ 左递归消除规则

– 消除一般左递归

对无回路($A \overset{+}{\Rightarrow} A$)、无ε-产生式的文法，通过下列步骤可消除一般左递归（包括直接和间接左递归）：

（1）以某种顺序将文法非终结符排列$A_1, A_2, \ldots, A_n$

（2）for i = 1 , …, n do {

for j=1, …, i-1 do {

用$A_j \to \alpha_1 \mid \alpha_2 \ldots \mid \alpha_k$替换形如 $A_i \to A_j \gamma$ 的规则的右部中的首个A_j，得到 $A_i \to \alpha_1\gamma \mid \alpha_2\gamma \ldots \mid \alpha_k\gamma$，其中

$A_j \to \alpha_1 \mid \alpha_2 \ldots \mid \alpha_k$ 是关于A_j 的全部产生式;

}

消除A_i规则的直接左递归;

}

（3）化简由（2）得到的文法

✧ 左递归消除举例

原文法 G[S]: $S \to PQ \mid a$

$P \to QS \mid b$

$Q \to SP \mid c$

非终结符排序为 S、P、Q，按照消除一般左递归的方法，进行如下变换:

$Q \to SP \mid c$

$\Rightarrow Q \to PQP \mid aP \mid c$

$\Rightarrow Q \to QSQP \mid bQP \mid aP \mid c$

$\Rightarrow Q \to bQPR \mid aPR \mid cR$

$R \to SQPR \mid \varepsilon$

结果:

$S \to PQ \mid a$

$P \to QS \mid b$

$Q \to bQPR \mid aPR \mid cR$

$R \to SQPR \mid \varepsilon$

Conclusions

- ✧ Relating FA to Lexical Analysis
- ✧ Lexical Analyzer Generators
- ✧ 语法分析的基本思想
- ✧ 带回溯的自顶向下分析；预测分析
- ✧ LL(1)文法：First/Follow集
- ✧ 递归下降LL(1)分析，表驱动LL(1)分析
- ✧ 文法变换：消除左递归

推荐教学资料

- ✧ § 3.4 - 3.8
- ✧ § 4.1 - 4.2
- ✧ § 4.3.1 - 4.3.3
- ✧ § 4.4.1 - 4.4.4
- ✧ 熟悉Flex及ANTLR

Thank you!