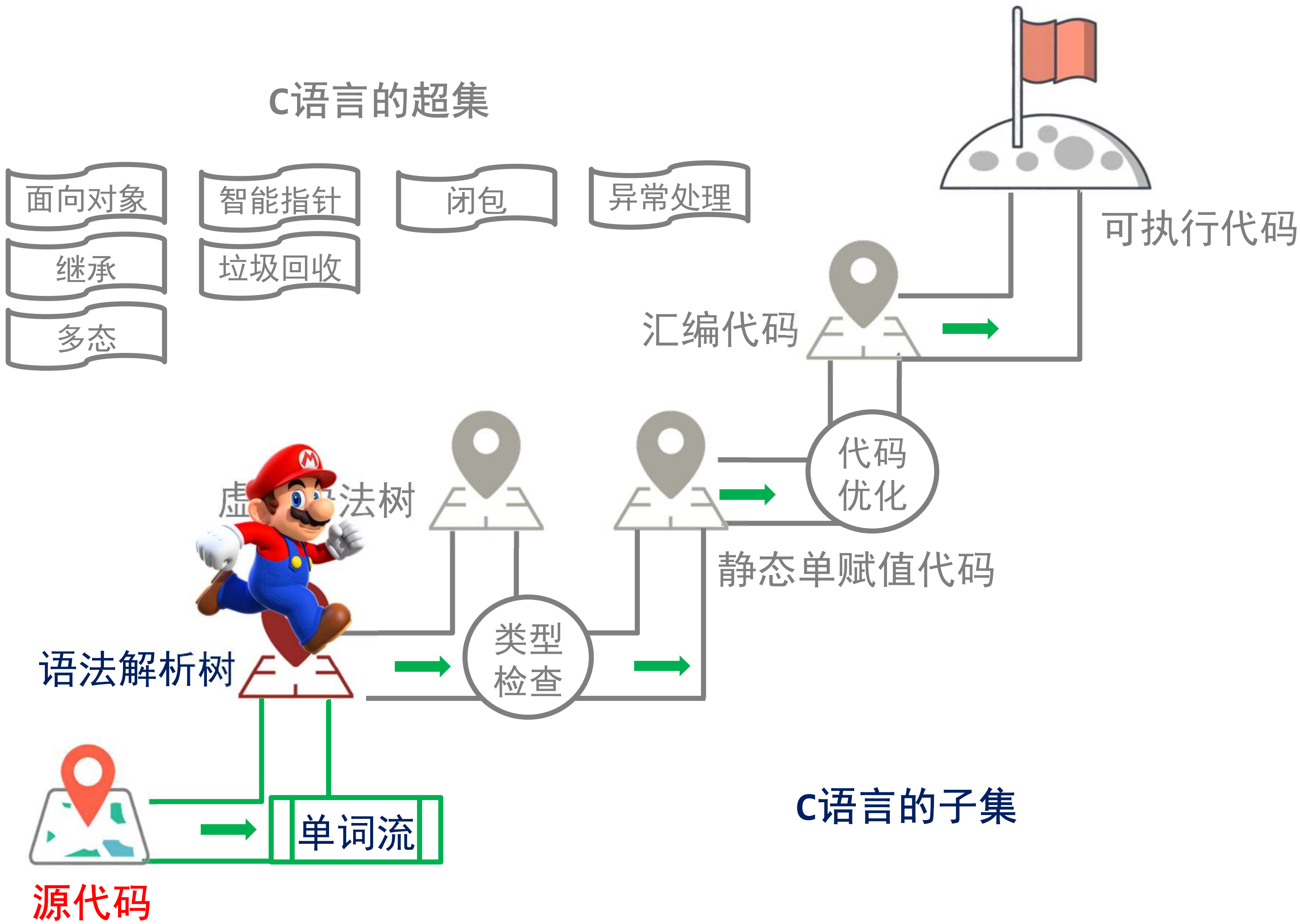
Lecture 3

句式分析

徐 辉

xuh@fudan.edu.cn

学习地图



C语言的超集
面向对象
智能指针
闭包
异常处理
继承
垃圾回收
多态
可执行代码
汇编代码
代码优化
静态单赋值代码
类型检查
语法解析树
单词流
源代码
C语言的子集

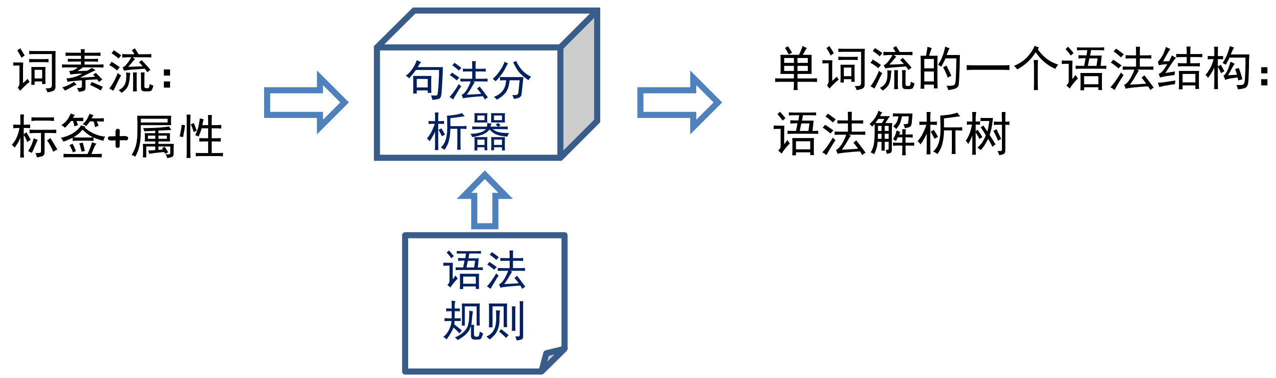
大纲

- 一、句式分析的基本概念
- 二、LLVM案例分析
- 三、自顶向下分析
- 四、自底向上分析
- 五、语法分析工具

一、句式分析的基本概念

问题定义

- 给定一个句子和语法规则，找到可生成该句子的一个语法推导。
- 通过词法分析已经将句子转换为了标签流。
- 语法规则（Grammar）定义了：
  - 什么是语法分析器（parser）可接受的标签组成，
  - 及其语法推导方式。



基本概念

- 一门语言（language）是多个句子（sentences）的集合。
- 句子（sentence）是由终结符（terminal symbols）组成的序列（sequence）。
- 字符串（string）是包含终结符和非终结符的序列。
  - 字符串符号：α, β, γ
  - 非终结符：X, Y, Z
  - 终结符（标签）：a, b, c
- 一条语法（grammar）包括一个开始符号S和多条推导规则（productions）
  - $X \to \beta$。

语法推导

- 语法G的语言$L(G)$是该语法可推导的所有句子的集合。
- 问题：下列语法是否可推导出句子$aaabbbccc$?

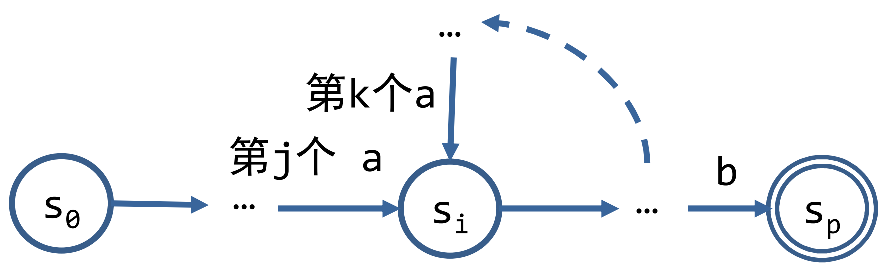
语法规则		推导	
[1]	$S \rightarrow aBSc$	[1]	$S \rightarrow aBSc$
[2]	$S \rightarrow abc$	[1]	$S \rightarrow aBaBScc$
[3]	$Ba \rightarrow aB$	[3]	$S \rightarrow aaBBScc$
[4]	$Bb \rightarrow bb$	[2]	$S \rightarrow aaBBabccc$
		[3]	$S \rightarrow aaBaBbccc$
		[3]	$S \rightarrow aaaBBbccc$
		[4]	$S \rightarrow aaaBbbccc$
		[4]	$S \rightarrow aaabbbccc$

语法表示：使用正则表达式？

- 正则表达式是否可识别四则运算？
  - $y = a \times x + b$
    - $(var|num)\big((+|-|\times|\div)\ (var|num)\big)^*$
  - $y = a \times (x + b)$
    - $('('|var|num)\big((+|-|\times|\div)\ ('('|var|num|')')\big)^*$
    - 可导致单词流被错误接收：
      - $y = (a \times (x + b)$
      - $y = (a \times (x + (b)$
- 正则表达式不能处理括号匹配问题：$(^*)^*$

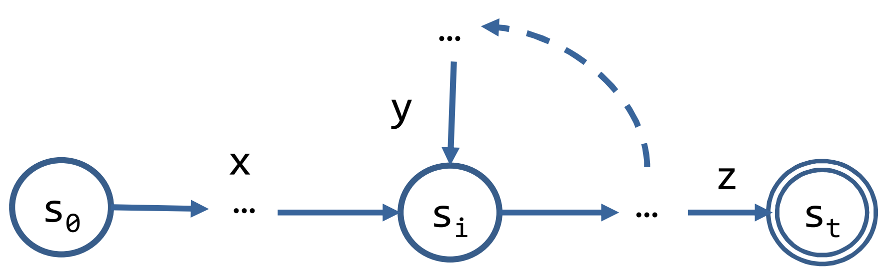
非正则语言

- 不能用正则表达式或有穷自动机表示的语言。
  - 正则语言不能计数，如$L = \{a^n b^n, n > 0\}$
  - 证明：
    - 假设DFA可识别该语言，其包含p个状态；
    - 假设某词素为$a^q b^q, q > p$。
    - 识别该词素需要经过某状态s_i至少两次，分别对应第j和第k个a；
    - 该DFA可同时接受$a^q b^q$和$a^{q+k-j} b^q$，推出矛盾。



正则语言的泵引理（Pumping Lemma）

- 词素数量有限的语言一定是正则语言。
- 词素数量无穷多的语言是否为正则语言？
- 某语言$L(r)$是正则语言的必要条件：
  - 任意长度超过p（泵长）的句子都可以被分解为xyz的形式
  - 其中x和z可为空，
  - 子句y被重复任意次（如$xyyz$）后得到的句子仍属于该语言。

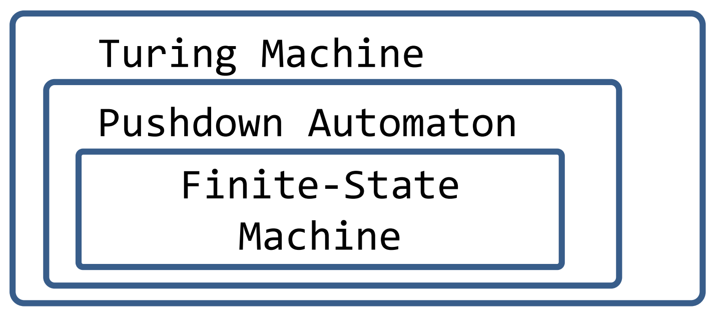


语言分析问题难度

- 通常来说，判断一个句子是否属于某个语言$w \in L(G)$是不能计算的。

Chomsky Hierarchy

Class	Languages	Automaton	Rules	Word Problem	Example
type-0	recursively enumerable	Turing machine	no restriction	undecidable	Post's corresp. problem
type-1	context sensitive	linear-bounded TM	$\alpha \to \gamma$ $\lvert\alpha\rvert \le \lvert\gamma\rvert$	PSPACE-complete	$a^n b^n c^n$
type-2	context free	pushdown automaton	$A \to \gamma$	cubic	$a^n b^n$
type-3	regular	NFA / DFA	$A \to a$ or $A \to aB$	linear time	$a^* b^*$



上线文无关语法和BNF范式

- 上下文无关语法（CFG/context-free grammar）是一个四元组(T, NT, S, P)
  - T：终结符
  - NT：非终结符
  - S：起始符号
  - P：产生式规则集合$X \rightarrow \gamma$，
    - X 是非终结符
    - γ 是可能包含终结符和非终结符的字符串
- BNF范式（Backus-Naur form）是传统的上下文无关语法表示方法。

$$\begin{aligned} \langle SheepNoise \rangle ::= &\ baa\ \langle SheepNoise \rangle \\ &|\ baa \end{aligned}$$

上线文无关语法举例

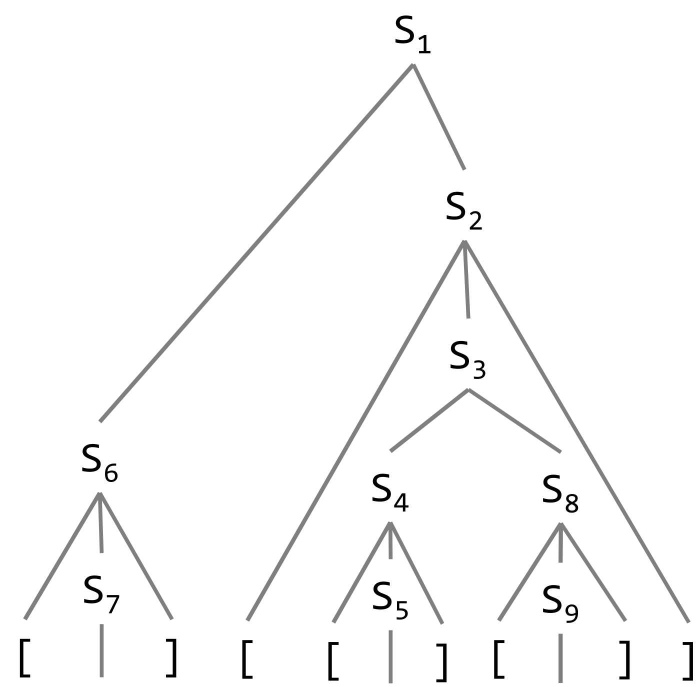
- 给定可生成所有匹配括号对的语法，[ ][[ ][ ]]是该语法的一个推导吗？

语法规则

[1]	$S \rightarrow \epsilon$
[2]	$\mid [S]$
[3]	$\mid SS$

推导

[3] $S \rightarrow SS$
[2] $S \rightarrow S[S]$
[3] $S \rightarrow S[SS]$
[2] $S \rightarrow S[S[S]]$
[1] $S \rightarrow S[S[\ ]]$
[2] $S \rightarrow S[[S][\ ]]$
[1] $S \rightarrow S[[\ ][\ ]]$
[2] $S \rightarrow [S][[\ ][\ ]]$
[1] $S \rightarrow [\ ][[\ ][\ ]]$



语法解析树

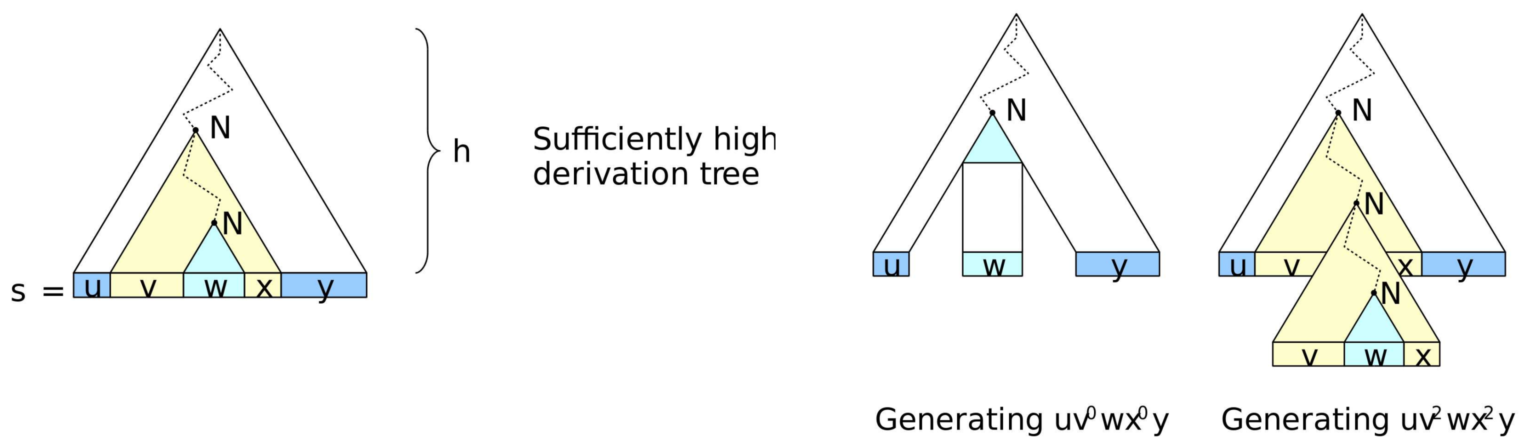
非CFG语言：上下文敏感语法

- $L = \{a^n b^n c^n, n > 0\}$不是CFG语言
- 无法用CFG定义
  - $X \to \gamma$
- 可以用上下文敏感语法定义
  - $\alpha A \beta \to \alpha \gamma \beta$
  - α和β不变，A展开

[1] $S \to aBC$
[2] $\quad | \, aSBC$
[3] $CB \to CZ$
[4] $\mathrm{CZ} \to WZ$
[5] $\mathrm{WZ} \to WC$
[6] $WC \to BC$
[7] $aB \to ab$
[8] $bB \to bb$
[9] $bC \to bc$
[10] $cC \to cc$

非CFG语言的泵引理

- CFG语言的泵引理（必要条件）：
  - 任意长度超过p（泵长）的句子可以被拆分为$uvwxy$，
  - 子句v和x被重复任意次后得到的新句子（如$uvvwxxy$）仍属于该语言。
- 正则属于CFG：$uv^nw\epsilon^n\epsilon$



澄清几个概念

- Regular language：用DFA/NFA可以计算
  - Regular expression
    - 狭义：其表示的都是正则语言，所有正则语言都可以用正则表达式表示（Flex工具）
    - 广义（regex）：字符串匹配工具。
- Context-free language：需要pushdown automaton计算
  - Context-free grammar
  - 正则语言可以用CFG表示：$StartSymbol \rightarrow regex$
    - 特性：右侧的非终结符均可替换为终结符
      - $S \rightarrow (0?1)^*$
      - $S \rightarrow 0S1S|1S0S|\epsilon$
- Context-sensitive language：需要图灵机计算

https://en.wikipedia.org/wiki/Regular_expression

下列语言是否为正则语言？

- 集合表示
  1) $L = \{a^n b^n\}$
  2) $L = \{a^n b^n | n \leq 100\}$
  3) $L = \{a^n | n \geq 1\}$
  4) $L = \{a^{2n} | n \geq 1\}$
  5) $L = \{a^p | \text{ p is prime}\}$
- Regex/CFG语法表示
  1) $S \to (0?1)^*$
  2) $S \to aT | \epsilon, T \to Sb$
  3) $S \to 0S1S | 1S0S | \epsilon$
  4) $S \to A | B$
     - $A \to E1A'E$
     - $A' \to A | \epsilon$
     - $B \to E0B'E$
     - $B' \to B | \epsilon$
     - $E \to 0S1S | 1S0S | \epsilon$

推导（Derivation）的优先级

暂不考虑优先级

$$
\begin{array}{lrl}
[1] & Expr & \rightarrow (Expr) \\
[2] & & \mid Expr\ Op\ num \\
[3] & & \mid num \\
[4] & Op & \rightarrow + \\
[5] & & \mid - \\
[6] & & \mid \times \\
[7] & & \mid \div
\end{array}
$$

(a + b) × 2

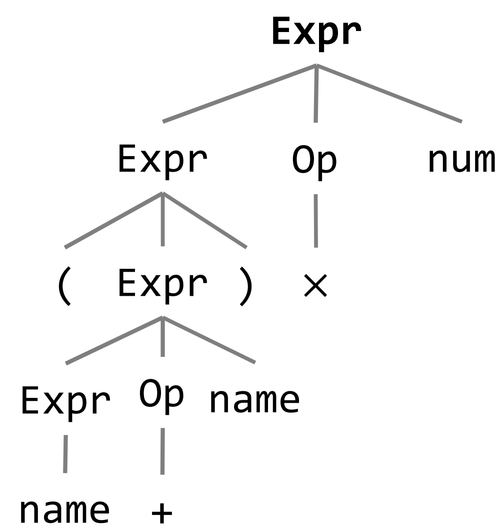
右侧优先推导
（Rightmost Derivation）

$$
\begin{array}{ll}
 & Expr \\
[2] & \underset{rm}{\Rightarrow} Expr\ Op\ num \\
[6] & \underset{rm}{\Rightarrow} Expr \times num \\
[1] & \underset{rm}{\Rightarrow} (Expr) \times num \\
[2] & \underset{rm}{\Rightarrow} (Expr\ Op\ num) \times num \\
[4] & \underset{rm}{\Rightarrow} (Expr + num) \times num \\
[3] & \underset{rm}{\Rightarrow} (num + num) \times num
\end{array}
$$

(a + b) × 2

$$
\begin{array}{ll}
 & Expr \\
[2] & \underset{lm}{\Rightarrow} Expr\ Op\ num \\
[1] & \underset{lm}{\Rightarrow} (Expr)\ Op\ num \\
[2] & \underset{lm}{\Rightarrow} (Expr\ Op\ num) Op\ num \\
[3] & \underset{lm}{\Rightarrow} (num\ Op\ num)\ Op\ num \\
[4] & \underset{lm}{\Rightarrow} (num + num)\ Op\ num \\
[6] & \underset{lm}{\Rightarrow} (num + int) \times num
\end{array}
$$

左侧优先推导
（Leftmost Derivation）



语法解析树完全相同

练习：语法推导

给定下列语法和字符串$aa+a*$

1） 写出左推导

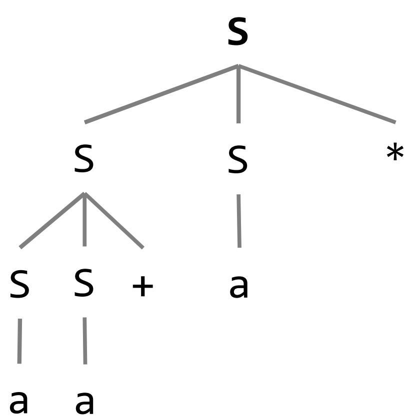
2） 写出右推导

3） 画出语法推导树

[1] $S \to SS+$
[2] $\quad\;\, | \, SS*$
[3] $\quad\;\, | a$

$$S \underset{lm}{\Rightarrow} SS* \underset{lm}{\Rightarrow} SS+S* \underset{lm}{\Rightarrow} aS+S* \underset{lm}{\Rightarrow} aa+S* \underset{lm}{\Rightarrow} aa+a*$$

$$S \underset{rm}{\Rightarrow} SS* \underset{rm}{\Rightarrow} Sa* \underset{rm}{\Rightarrow} SS+a* \underset{rm}{\Rightarrow} Sa+a* \underset{rm}{\Rightarrow} aa+a*$$



练习：语法设计

- 为下列语言设计语法规则。
  1) 所有0和1组成的字符串，每一个0后面紧跟着若干个1
  2) 所有0和1组成的字符串，0和1的个数相同
  3) 所有0和1组成的字符串，0和1的个数不相同

$$S \to (0?\,1)^*$$

$$S \to 0S1S|1S0S|\epsilon$$

$$\begin{aligned}
&S \to A|B \\
&A \to E1A'E \\
&A' \to A|\epsilon \\
&B \to E0B'E \\
&B' \to B|\epsilon \\
&E \to 0S1S|1S0S|\epsilon
\end{aligned}$$

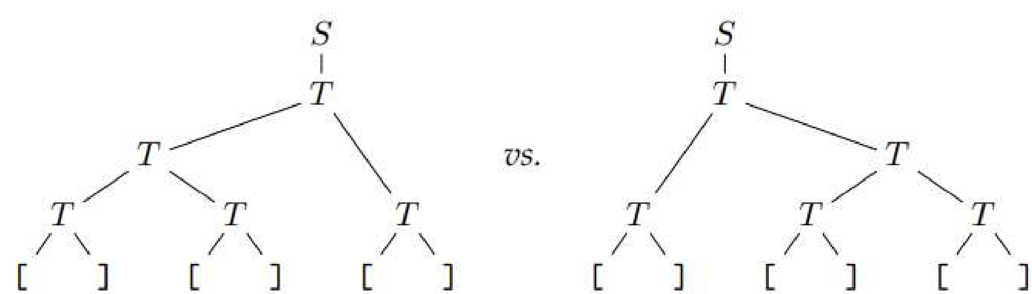
二义性（ambiguity）

- 如果L(G)中的某个句子有一个以上的最左（或最右）推导，那么语法G就有二义性。
  - 语法解析树不同
- 根据下列语法规则如何推导出[ ][ ][ ]?

[1]	$S \to \epsilon$
[2]	$\quad\mid T$
[3]	$T \to []$
[4]	$\quad\mid [T]$
[5]	$\quad\mid TT$

$$S \to T \to TT \to TTT \to [\,]TT \to T[\,][\,] \to [\,][\,][\,]$$

$$S \to T \to TT \to [\,]T \to [\,]TT \to [\,][\,]T \to [\,][\,][\,]$$



极端情况

- 存在无数棵语法解析树
  - 考虑循环的情况
- 根据下列语法规则如何推导出[ ][[ ][ ]]?

$$
\begin{array}{ll}
[1] & S \rightarrow \epsilon \\
[2] & \quad\;\; \mid [S] \\
[3] & \quad\;\; \mid SS
\end{array}
$$

$$S \rightarrow SS \rightarrow [S]S \rightarrow [\,]S \rightarrow [\,][S] \rightarrow [\,][S\ S] \rightarrow [\,]\big[[S]\ [S]\big] \rightarrow [\,]\big[[\,][\,]\big]$$

$$S \rightarrow SS \rightarrow S \rightarrow SS \rightarrow \cdots$$


If your DOG
does a POO
Please put it
in a litter bin.
Please help keep our
open spaces clean.

消除二义性

$$
\begin{array}{ll}
[1] & S \to \epsilon \\
[2] & \quad \mid [S] \\
[3] & \quad \mid SS
\end{array}
$$

$$
\begin{array}{ll}
[1] & S \to \epsilon \\
[2] & \quad \mid T \\
[3] & T \to [] \\
[4] & \quad \mid [T] \\
[5] & \quad \mid TT
\end{array}
$$

$$
\begin{array}{ll}
[1] & S \to \epsilon \\
[2] & \quad \mid T \\
[3] & T \to UT \\
[4] & \quad \mid U \\
[5] & U \to [\,] \\
[6] & \quad \mid [T]
\end{array}
$$

消除循环引起的二义性

$S \to SS \to S$

推导[ ][ ][ ]的例子

$T \to TT$ 左递归容易引起二义性（回溯语法）

将语义加入语法中：四则运算的例子

[1] $Expr \rightarrow Expr\ Bop\ Expr$
[2] $\quad\quad |\ num$
[3] $\quad\quad |\ (Expr)$
[4] $Bop \rightarrow +$
[5] $\quad\quad |\ -$
[6] $\quad\quad |\ \times$
[7] $\quad\quad |\ \div$

$3 + 4\ \times\ 5$的语义？

$Expr$
$\rightarrow Expr\ Bop\ Expr$
$\rightarrow Expr\ Bop\ Expr\ Bop\ Expr$
$\rightarrow Expr + Expr\ \times\ Expr$

$(3 + 4) \times 5$

$3 + (4 \times 5)$

将运算符特性加入到语法规则中：

- 优先级：( ) >×/÷>+/−
- 结合性：左结合

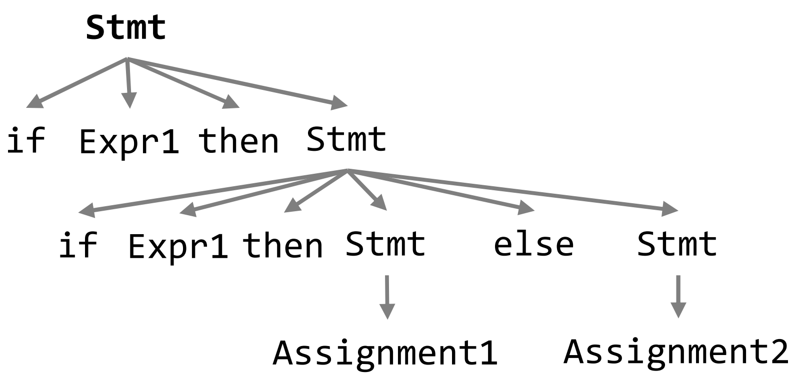
$Expr$
$\rightarrow Expr\ AMop\ Term$
$\rightarrow Term\ AMop\ Term$
$\rightarrow Term\ AMop\ Term\ MDop\ Factor$
$\rightarrow \cdots$

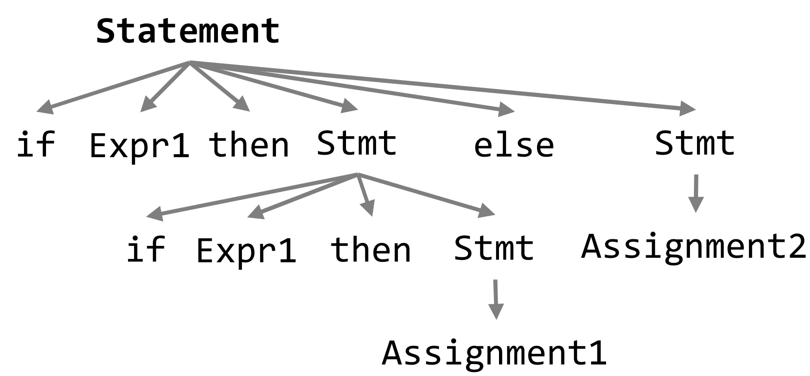
[1] $Expr \rightarrow Expr\ AMop\ Term$
[2] $\quad\quad |\ Term$
[3] $Term \rightarrow Term\ MDop\ Factor$
[4] $\quad\quad |\ Factor$
[5] $Facor \rightarrow (Expr)$
[6] $\quad\quad |\ num$
[7] $AMop \rightarrow +$
[8] $\quad\quad |\ -$
[9] $MDop \rightarrow \times$
[10] $\quad\quad |\ \div$

If-Else嵌套的二义性语法

[1] $Stmt \rightarrow$ if $Expr$ then $Stmt$ else $Stmt$
[2] $\qquad\qquad\;|$ if $Expr$ then $Stmt$
[3] $\qquad\qquad\;|$ $Assignment$
[4] $\qquad\qquad\;|$ …

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```






```
if Expr1 then
  if Expr2 then
    Assignment1
  else
    Assignment2
```

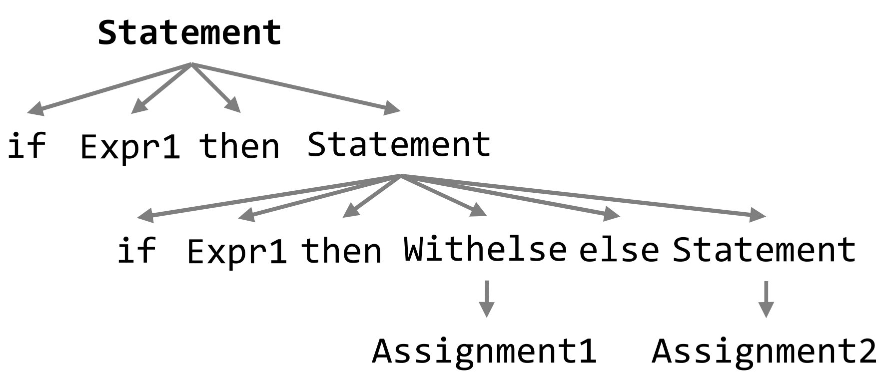
```
if Expr1 then
  if Expr2 then
    Assignment1
else
    Assignment2
```

消除If-Else语法的二义性

- 将语义编码加入到结构中
  - 要求else优先匹配内层if
    - 如果外层出现else，则内部嵌套的if语句一定有匹配的else

[1] $Stmt \rightarrow \text{if } Expr \text{ then } Withelse \text{ else } Stmt$	[1] 有else配套的if语句，可继续展开[4]
[2] $\quad\quad \mid \text{if } Expr \text{ then } Stmt$	[2] 无配套else的if语句
[3] $\quad\quad \mid Assignment$	[3]
[4] $Withelse \rightarrow \text{if } Expr \text{ then } Withelse \text{ else } Withelse$	[4] 内层有配套else的if语句
[5] $\quad\quad \mid Assignment$	[5]

```
if Expr1 then if Expr2 then Assignment1 else Assignment 2
```




```
if Expr1 then
  if Expr2 then
    Assignment1
  else
    Assignment2
```

不存在其它推导方式

练习：二义性分析

- 下列If-Else语法是否存在二义性？

[1] $Stmt \to \text{if } Expr \text{ then } Stmt$
[2] $\quad\quad\quad \mid matchedStmt$
[3] $matchedStmt \to \text{if } Expr \text{ then } matchedStmt \text{ else } Stmt$
[4] $\quad\quad\quad \mid Assignment$

```
if Expr1 then if Expr2 then Assignment1 else Assignment 2
```

```
if Expr1 then if Expr2 then Assignment1 else if Expr3 then Assignment2 else Assignment3
```

```
if Expr1 then
    if Expr2 then
        Assignment1
    else
        if Expr3 then
            Assignment2
        else
            Assignment3
```

```
if Expr1 then
    if Expr2 then
        Assignment1
    else
        if Expr3 then
            Assignment2
else
     Assignment3
```

练习

- 为描述正则语言的正则表达式语法设计一种CFG
  - 支持字符[A-Za-z0-9]
  - 支持连接、或|、闭包*运算
  - 支持()
- 检查语法是否有二义性？

[1] $<regex>::=<union>\mid<concat>\mid<closure>\mid<term>$
[2] $<union>::=<regex>\text{"|"}<regex>$
[3] $<concat>::=<regex><regex>$
[4] $<closure>::=<regex>*$
[5] $<term>::=<group>\mid<alphanum>$
[6] $<group>::=(<regex>)$
[7] $<alphanum>::=A|\ldots|Z|a|\ldots|z|0|\ldots|9$

- 主要问题：
  - 未考虑运算的优先级：比如解析ab|c存在歧义。
  - 一般按照运算符优先级由低到高依次展开

改写

$$
\begin{aligned}
&< regex >::=< union > \mid < concat > \mid < closure > \mid < term > \\
&< union >::=< regex > "|" < regex > \\
&< concat >::=< regex >< regex > \\
&< closure >::=< regex >* \\
&< term >::=< group > \mid < alphanum > \\
&< group >::= (< regex >) \\
&< alphanum >::= A|\dots|Z|a|\dots|z|0|\dots|9
\end{aligned}
$$

$$\Downarrow$$

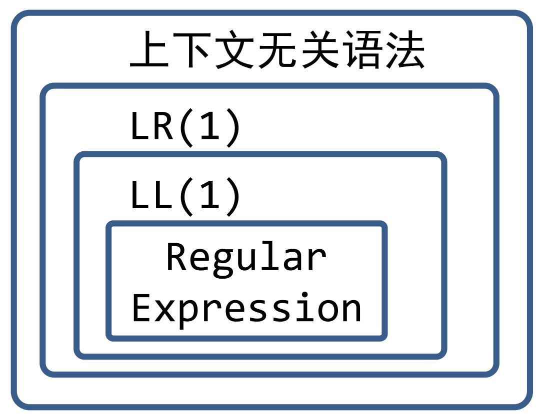
$$
\begin{aligned}
&< regex >::=< union > \mid < concat > \\
&< union >::=< regex > "|" < concat > \\
&< concat >::=< concat >< term > \mid < term > \\
&< term >::=< element >* \mid < element > \\
&< element >::= (< regex >) \mid < alphanum > \\
&< alphanum >::= A|\dots|Z|a|\dots|z|0|\dots|9
\end{aligned}
$$

思考

1) 用正则表达式可以定义所有的正则语言吗？
2) 用有穷自动机（正则表达式模拟器）可以解析任意正则表达式吗？
3) 用CFG可以定义任意正则语言吗？
4) 用CFG可以定义任意CFL语言吗？
5) 用pushdown automaton（CFG模拟器）可以解析任意正则表达式吗？
6) 用pushdown automaton可以解析任意CFG吗？
7) 用通用图灵机可以解析任意CFG吗？
8) 用通用图灵机可以解析任意程序吗？

编译器的任务：找到语法树推导

- 方法：
  - 自顶向下（top-down parser）
  - 自底向上（bottom-up parser）
- 语法难度：CFG>LR(1)>LL(1)>RE
  - 任意CFG需要花费更多时间进行语法分析
    - Earley/CYK算法复杂度$O(n^3)$
  - LL(1)是LR(1)的一个子集
    - Left-to-Right，Leftmost
    - 前瞻单词1个
    - 适合自顶向下分析
  - LR(1)是无歧义CFG的一个子集
    - Left-to-Right，Rightmost
    - 前瞻单词1个
    - 适合自底向上分析



二、LLVM案例分析

LR(0)

LR(1): SLR/LALR

Clang采用自顶向下分析算法

A single unified parser for C, Objective C, C++, and Objective C++

Clang is the "C Language Family Front-end", which means we intend to support the most popular members of the C family. We are convinced that the right parsing technology for this class of languages is a **hand-built recursive-descent parser**. Because it is plain C++ code, recursive descent makes it very easy for new developers to understand the code, it easily supports ad-hoc rules and other strange hacks required by C/C++, and makes it straight-forward to implement excellent diagnostics and error recovery.

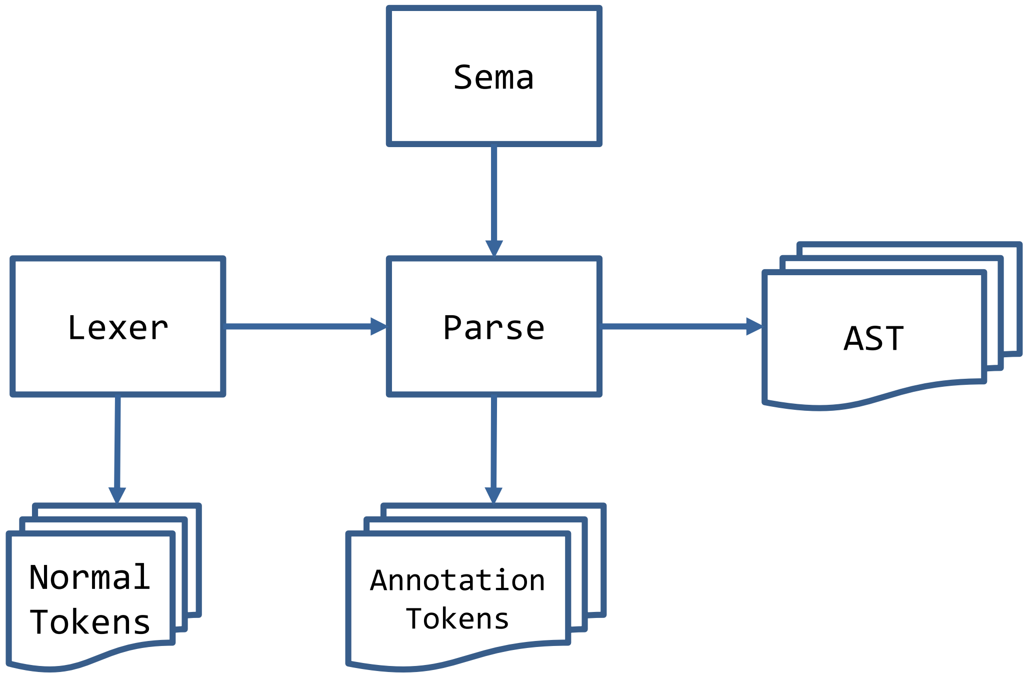
We believe that implementing C/C++/ObjC in a single unified parser makes the end result easier to maintain and evolve than maintaining a separate C and C++ parser which must be bugfixed and maintained independently of each other.

https://clang.llvm.org/features.html#unifiedparser

作业回顾

$$
\begin{array}{lcl}
<program> & ::= & <gdecl>^{*}<function>^{*} \\
<gdecl> & ::= & extern <prototype>; \\
<function> & ::= & <prototype><body> \\
<prototype> & ::= & <type><ident>(<paramlist>) \\
<paramlist> & ::= & \epsilon| <type><ident>[,<type><ident>]^{*} \\
<body> & ::= & \{<stmt>\} \\
<stmt> & ::= & <exp> \\
<exp> & ::= & (<exp>)| <const> | <ident> | \\
 & & <exp><binop><exp> | <callee> \\
<callee> & ::= & <ident>(\epsilon| <exp>[,<exp>]^{*}) \\
<ident> & ::= & [A-Z\_a-z][0-9A-Z\_a-z]^{*} \\
<const> & ::= & <intconst> | <doubleconst> \\
<binop> & ::= & + \mid \text{-} \mid * \mid < \\
<intconst> & ::= & [0-9][0-9]^{*} \\
<doubleconst> & ::= & <intconst>.<intconst> \\
<type> & ::= & int|double
\end{array}
$$

LLVM前端架构



Lexer::LexTokenInternal()

```
bool Lexer::LexTokenInternal(Token &Result, bool TokAtPhysicalStartOfLine) {
LexNextToken:
  // New token, can't need cleaning yet.
  Result.clearFlag(Token::NeedsCleaning);
  Result.setIdentifierInfo(nullptr);
  const char *CurPtr = BufferPtr;
  if (isHorizontalWhitespace(*CurPtr)) {
    do {
      ++CurPtr;
    } while (isHorizontalWhitespace(*CurPtr));
    BufferPtr = CurPtr;
    Result.setFlag(Token::LeadingSpace);
  }
  char Char = getAndAdvanceChar(CurPtr, Result);
  tok::TokenKind Kind;
  switch (Char) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
          MIOpt.ReadToken();
          return LexNumericConstant(Result, CurPtr);
      case …
```


https://github.com/llvm/llvm-project/blob/main/clang/lib/Lex/Lexer.cpp

根据首字符大致分类

0-9 → NumericConstant

A-Za-z_ → Identifier

特殊情况：
1） 保留字
2） u/U/L

'\' → CharConstant

'"' → StringLiteral

'?' → tok::question

'[' → tok::l_square

']' → tok::r_square

'(' → tok::l_paren

')' → tok::r_ paren

'{' → tok::l_brace

'}' → tok::r_brace

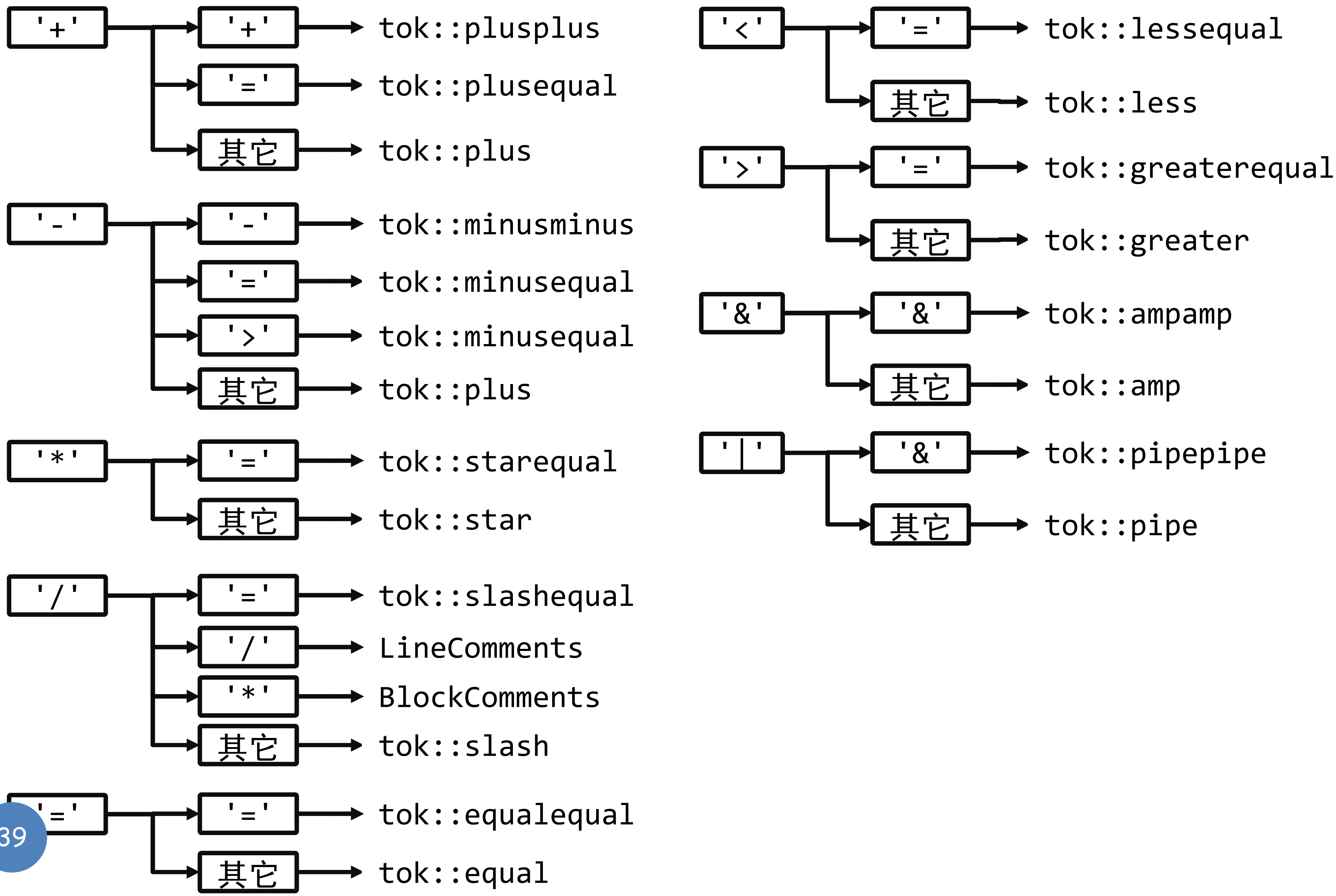
';' → tok::semi

',' → tok::comma

':' → ?

'.' → ?

根据前两个字符大致分类



Token类

- 单个token，供lexer和parser使用，AST中不存在。
- Sizeof(Token)是16字节
- 分为两类：
  - Normal tokens：lexer返回的结果
    - 标识符信息（指向用于检索的哈希值）
    - Token类型（参照TokenKinds.def中的定义）
    - 位置、长度、Flags
    - 缺少了什么？
      - 数值信息
  - Annotation tokens：parser处理后的结果，添加了语义信息

Parser::ParseStatementOrDeclarationAfterAttributes()

```
Parser::ParseStatementOrDeclarationAfterAttributes(
    StmtVector &Stmts, ParsedStmtContext StmtCtx,
    SourceLocation *TrailingElseLoc, ParsedAttributesWithRange &Attrs) {
    tok::TokenKind Kind = Tok.getKind();
    switch (Kind) {
        case tok::at: // May be a @try or @throw statement
        {
            AtLoc = ConsumeToken();  // consume @
            return ParseObjCAtStatement(AtLoc, StmtCtx);
        }
        case tok::identifier: {
           Token Next = NextToken();
           if (Next.is(tok::colon)) { // C99 6.8.1: labeled-statement
               return ParseLabeledStatement(Attrs, StmtCtx);
           }
           if (Next.isNot(tok::coloncolon)) {  … }
           default: {
               …
           }
        }
        …
    }
}
```


https://github.com/llvm/llvm-project/blob/main/clang/lib/Parse/ParseStmt.cpp

根据首字符大致分类

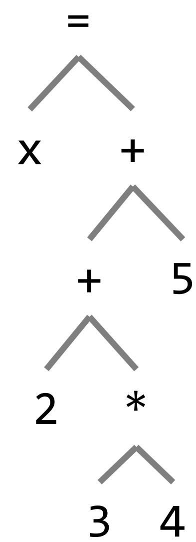
tok::identifier	→ ParseExprStatement
	→ ParseDeclaration
tok::l_brace:	→ ParseCompoundStatement
tok::kw_if:	→ ParseIfStatement
tok::kw_switch:	→ ParseSwitchStatement
tok::kw_case:	→ ParseCaseStatement
tok::kw_while:	→ ParseWhileStatement
tok::kw_do:	→ ParseDoStatement
tok::kw_for:	→ ParseForStatement
tok::kw_break:	→ ParseBreakStatement
tok::kw_default:	→ ParseDefaultStatement
tok::kw_continue:	→ ParseContinueStatement
tok::kw_goto:	→ ParseGotoStatement
tok::kw_return:	→ ParseReturnStatement

clang/lib/Parse/ParseStmt.cpp

如何解析表达式?

x = 2 + 3 * 4 + 5

- 根据运算符优先级,
- 假设*>+>=



Pratt Parsing

```
优先级
=: [1,2]
+: [3,4]
*: [5,6]

Parse(token, precedence) {
  left = token.next();
  if left.type != tok::num
    return -1;
  while true:
    op = token.peek();
    if op.tokentype != tok::binop
      return -1;
    lp, rp = Precedence[op];
    if lp < precedence
      break;
    token.next();
    right = Parse(token, rp)
    left = (op, left, right)
  return left
}
```

0 1 2 3 4 5 6 3 4 0

x = 2 + 3 * 4 + 5

Parse(start,0)

op: =,

Parse(=,2)

op: +

Parse(+,4)

op: *

Parse(*,+)

Return left = 4

left = *(3, 4)

0 1 2 3 4 ~~5 6~~ 3 4 0

x = 2 + ~~3 * 4~~ + 5

Return left = +(2, *(3, 4))

0 1 2 ~~3 4 5 6~~ 3 4 0

x = ~~2 + 3 * 4~~ + 5

Parse(pre5,+)

Return left = +(+(2, *(3, 4)), 5)

0 1 2 ~~3 4 5 6 3 4~~ 0

x = ~~2 + 3 * 4 + 5~~

Return left = = (x, +(+(2, *(3, 4)), 5))

三、自顶向下分析

Top-down

Recursive-descent

自顶向下构建语法解析树

如何解析(num+num)×num?

$$
\begin{aligned}
&[1]\ Expr \rightarrow Expr + Term \\
&[2]\ \quad\quad\ \ |\ Expr\ - Term \\
&[3]\ \quad\quad\ \ |\ Term \\
&[4]\ Term \rightarrow Term\ \times\ Factor \\
&[5]\ \quad\quad\ \ |\ Term \div Factor \\
&[6]\ \quad\quad\ \ |\ Factor \\
&[7]\ Facor \rightarrow (Expr) \\
&[8]\ \quad\quad\quad |\ num \\
&[9]\ \quad\quad\quad |\ id
\end{aligned}
$$

假设每次都能选对规则

word	cur	Rule	Stack
(	Expr	[3]	Term
(	Term	[4]	Term, ×, Factor
(	Term	[6]	Factor, ×, Factor
(	Factor	[7]	(, Expr,), ×, Factor
(	(	-	Expr,), ×, Factor
num	Expr	[1]	Expr,+, Term,), ×, Factor
num	Expr	[3]	Term,+, Term,), ×, Factor
num	Term	[6]	Factor,+, Term,), ×, Factor
num	Term	[8]	num,+, Term,), ×, Factor
num	num	-	+, Term,), ×, Factor
+	+	-	Term,), ×, Factor
num	Term	[6]	Factor,), ×, Factor
num	Factor	[8]	num,), ×, Factor
num	num	-	), ×, Factor
)	)	-	×, Factor
×	×	-	Factor
num	Factor	[8]	num
num	num	-	null
eof			

自动搜索语法树的算法…

```
输入: 程序单词流 seq;
       CFG语法 rules;
Output: accept: 语法解析树ptree,
        reject;
初始化:
let ptree = start symbol;
let ptr = root;
let st = stack();
st.push(null);

开始:
let word = seq.NextWord();
While (true) do:
  if (!ptr.nodetype().isTerminal())
    For each rule in {A → β1, …, βn; A → ⋯}
      ptr.children = (β1, …, βn);
      For 1 < i < n+1
        st.push(βn+2−i);
      ptr = β1;
  //ptr.nodetype()=terminal
  else if (word == ptr) //单词匹配成功
    word = seq.NextWord(); //下一个单词
    ptr = st.pop()
  else if (word == eof && cur == null)
    accept and return ptree;
  else
    backtrack(); //回溯
```

- 不考虑递归的情况：
  - $A \to \cdots \to A$
- 不考虑左递归的情况：
  - $A \to \cdots \to AX$
- 复杂度高：
  - 单词个数
  - 规则个数
  - 规则生产的符号数
  - 基于栈的回溯

左递归问题

- 对CFG的一个规则来说，其右侧的第一个符号与左侧符号相同或者能够推导出左侧符号。
- 主要问题：可使搜索算法无限递归下去，不终止。

$$\begin{array}{ll} [1] & Expr \rightarrow Expr + Term \\ [2] & \quad\quad\quad \mid Expr - Term \\ [3] & \quad\quad\quad \mid Term \end{array}$$

word	cur	Rule	Stack
(	Expr	[1]	*Expr*, +, *Term*
(	Expr	[1]	*Expr*, +, *Term*, +, *Term*
(	Expr	[1]	*Expr*, +, *Term*, +, *Term*, +, *Term*
			…

消除左递归

- 引入新的非终结符，基本规则：

$$\begin{aligned} E &\rightarrow E\ \alpha \\ &\mid \beta \end{aligned} \quad\Rightarrow\quad \begin{aligned} E &\rightarrow \beta\ E' \\ E' &\rightarrow \alpha\ E' \\ &\mid \epsilon \end{aligned}$$

$$\begin{aligned} E &\rightarrow E\ \alpha \\ &\mid \beta \\ &\mid \gamma \end{aligned} \quad\Rightarrow\quad \begin{aligned} E &\rightarrow \beta\ E' \mid \gamma E' \\ E' &\rightarrow \alpha\ E' \\ &\mid \epsilon \end{aligned}$$

举例：

$$\begin{aligned} &[1]\ Expr \rightarrow Expr + Term \\ &[2]\qquad\quad \mid Expr\ - Term \\ &[3]\qquad\quad \mid Term \end{aligned} \quad\Rightarrow\quad \begin{aligned} &[1]\ Expr \rightarrow Term\ Expr' \\ &[2]\ Expr' \rightarrow + Term\ Expr' \\ &[3]\qquad\quad \mid - Term\ Expr' \\ &[4]\qquad\quad \mid \epsilon \end{aligned}$$

$$\begin{aligned} &[4]\ Term \rightarrow Term \times Factor \\ &[5]\qquad\quad \mid Term \div Factor \\ &[6]\qquad\quad \mid Factor \end{aligned} \quad\Rightarrow\quad \begin{aligned} &[5] Term \rightarrow Factor\ Term' \\ &[6] Term' \rightarrow \times\ Factor\ Term' \\ &[7]\qquad\quad \mid \div\ Factor\ Term' \\ &[8]\qquad\quad \mid \epsilon \end{aligned}$$

$$\begin{aligned} &[7]\ Facor \rightarrow (Expr) \\ &[8]\qquad\quad \mid num \\ &[9]\qquad\quad \mid name \end{aligned} \quad\Rightarrow\quad \begin{aligned} &[9]\ Facor \rightarrow (Expr) \\ &[10]\qquad\quad \mid num \\ &[11]\qquad\quad \mid name \end{aligned}$$

更多例子

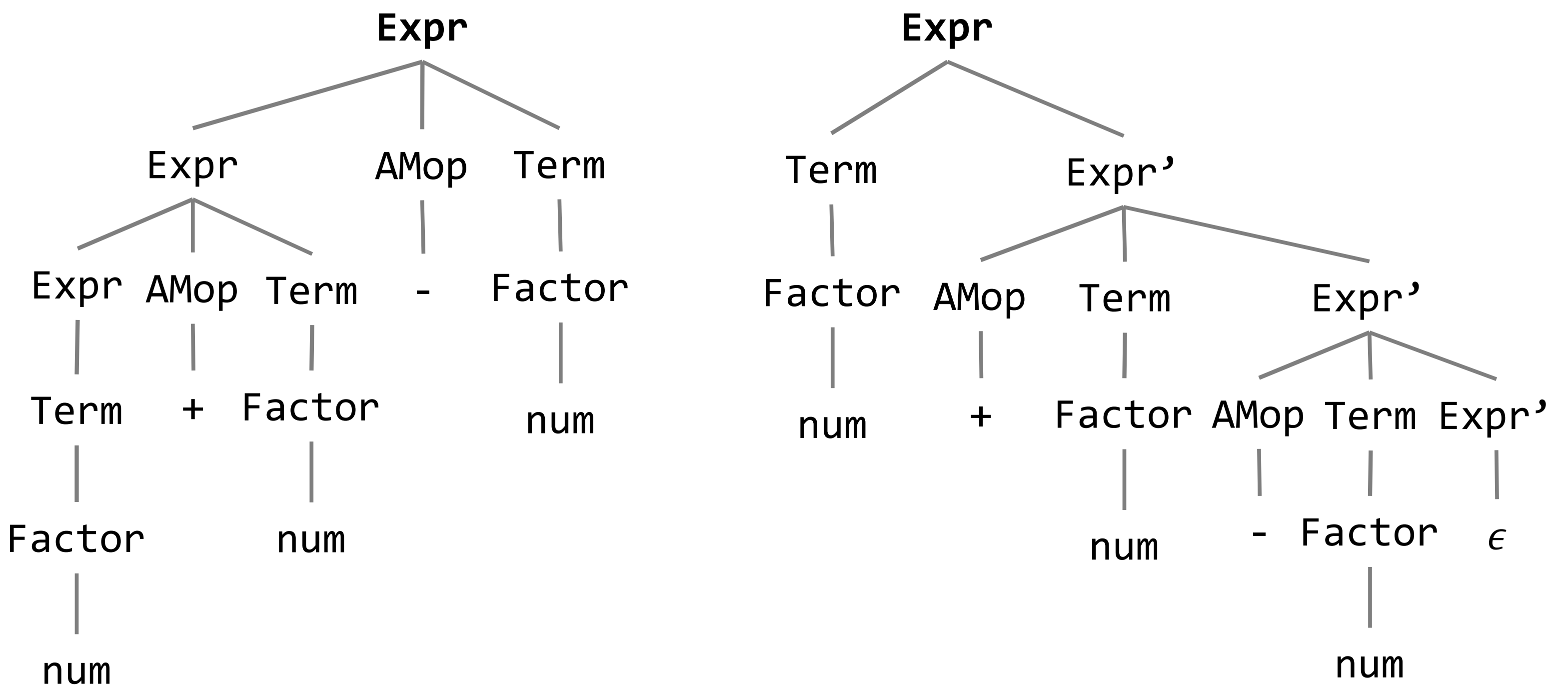
[1] $Expr \rightarrow Expr\ AMop\ Term$
[2] $\quad\quad\quad |\ Term$
[3] $Term \rightarrow Term\ MDop\ Factor$
[4] $\quad\quad\quad |\ Factor$
[5] $Facor \rightarrow (Expr)$
[6] $\quad\quad\quad |\ num$
[7] $AMop \rightarrow +$
[8] $\quad\quad\quad |\ -$
[9] $MDop \rightarrow \times$
[10] $\quad\quad\quad |\ \div$

$\Rightarrow$

[1] $Expr \rightarrow Term\ Expr'$
[2] $Expr' \rightarrow AMop\ Term\ Expr'$
[3] $\quad\quad\quad |\ \epsilon$
[4] $Term \rightarrow Factor\ Term'$
[5] $Term' \rightarrow MDop\ Factor\ Term'$
[6] $\quad\quad\quad |\ \epsilon$
[7] $Facor \rightarrow (Expr)$
[8] $\quad\quad\quad |\ num$
[9] $AMop \rightarrow +$
[10] $\quad\quad\quad |\ -$
[11] $MDop \rightarrow \times$
[12] $\quad\quad\quad |\ \div$

思考

- 是否会改变运算符结合性？如3+4-5。
  - 语法解析树和虚拟语法树的区别



间接左递归问题

$$\begin{array}{l} E \rightarrow \alpha \\ \alpha \rightarrow \beta + \\ \beta \rightarrow E \end{array} \Longrightarrow E \rightarrow E +$$

展开所有非终结符NT检测和消除间接左递归

输入: Grammar{T,NT}

开始:

for i=1 to n
  for j=1 to i-1
    if $\exists\, NT_i \rightarrow NT_j\, \gamma$
      展开 $NT_i \rightarrow NT_j\, \gamma$中的非终结符$NT_j$
  重写会造成NT_i左递归的规则

无回溯语法

- 目的：消除语法生成规则选择时的不确定性，避免回溯。
- 思路：如果对于每个非终结符的任意两个生成式，其产生的首个终结符号不同，则在前瞻一个单词的情况下，总能够选择正确的生成式规则。
  - [1] $NT_1 \rightarrow NT_i \rightarrow \cdots \rightarrow \mathrm{term}_1\ NT_p$
  - [2] $NT_1 \rightarrow NT_j \rightarrow \cdots \rightarrow \mathrm{term}_2\ NT_q$
- 预测解析（Predictive Parsing）：LL(1)语法
  - Left-to-Right, Leftmost, 前瞻一个字符

消除回溯：提取左因子

- 对一组产生式提取并隔离共同前缀

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma_1 | \dots | \gamma_j \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \alpha B \mid \gamma_1 | \dots | \gamma_j \\ B \rightarrow \beta_1 \mid \beta_2 | \dots | \beta_n \end{array}$$

应用举例：

[11]	$Factor \rightarrow name$
[12]	$\mid name\ [ArgList]$
[13]	$\mid name\ (ArgList)$
[14]	$ArgList \rightarrow Expr\ MoreArgs$
[15]	$MoreArgs \rightarrow , Expr\ MoreArgs$
[16]	$\mid \epsilon$

$\Rightarrow$

[11]	$Factor \rightarrow name\ Arguments$
[12]	$Arguments \rightarrow [ArgList]$
[13]	$\mid (ArgList)$
[14]	$\mid \epsilon$
[15]	$ArgList \rightarrow Expr\ MoreArgs$
[16]	$MoreArgs \rightarrow , Expr\ MoreArgs$
[17]	$\mid \epsilon$

无回溯语法的必要性质

$$First^+(A \to \beta) = \begin{cases} First(\beta), & if \epsilon \notin First(\beta) \\ First(\beta) \cup Follow(A), & otherwise \end{cases}$$

$$\forall 1 \le i, j \le n, First^+(A \to \beta_i) \cap First^+(A \to \beta_j) = \emptyset$$

- 同一非终结符A 的任意两个语法推导$(A \to \beta_i)$和$(A \to \beta_j)$ 所产生的的首个终结符不能相通。
- $First(\beta)$是从语法符号β推导出的每个子句的第一个终结符的集合，其值域是$T \cup \{\epsilon, \mathrm{eof}\}$。
- 如果$First(\beta)$是$\{\epsilon\}$，则计算紧随A之后出现的终结符的集合$Follow(A)$。

First集合计算

- 对于生成式$A \rightarrow \beta_1\beta_2 \ldots \beta_n$ 来说：
  - 如果 $\epsilon \notin First(\beta_1)$， 则$First(A) = First(\beta_1)$
  - 如果 $\epsilon \in First(\beta_1) \& \ldots \& \epsilon \in First(\beta_i)$, 则$First(A) = First(\beta_1) \cup \cdots \cup First(\beta_{i+1})$

[1] $Expr \rightarrow Term\ Expr'$
[2] $Expr' \rightarrow + Term\ Expr'$
[3] $\mid - Term\ Expr'$
[4] $\mid \epsilon$

[5] $Term \rightarrow Factor\ Term'$
[6] $Term' \rightarrow \times Factor\ Term'$
[7] $\mid \div Factor\ Term'$
[8] $\mid \epsilon$

[9] $Facor \rightarrow (Expr)$
[10] $\mid num$
[11] $\mid id$

	num	id	+	−	×	÷	(	)	ϵ
$Expr$	√	√					√		
$Expr'$			√	√					√
$Term$	√	√					√		
$Term'$					√	√			√
$Facor$	√	√					√		

Follow集合计算

- 紧随非终结符之后出现的所有可能的终结符

$$
\begin{aligned}
&[1]\ Expr \rightarrow Term\ Expr' \\
&[2]\ Expr' \rightarrow +\ Term\ Expr' \\
&[3]\ \qquad\quad |\ -\ Term\ Expr' \\
&[4]\ \qquad\quad |\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&[5]\ Term \rightarrow Factor\ Term' \\
&[6]\ Term' \rightarrow \times\ Factor\ Term' \\
&[7]\ \qquad\quad |\ \div\ Factor\ Term' \\
&[8]\ \qquad\quad |\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&[9]\ Facor \rightarrow (Expr) \\
&[10]\ \qquad\quad |\ num \\
&[11]\ \qquad\quad |\ id
\end{aligned}
$$

	num	id	$+$	$-$	$\times$	$\div$	(	)	ϵ	eof
$Expr$	√	√					√	⊙		⊙
$Expr'$			√	√				⊙	√	⊙
$Term$	√	√	⊙	⊙			√	⊙		⊙
$Term'$			⊙	⊙	√	√		⊙	√	⊙
$Facor$	√	√	⊙	⊙	⊙	⊙	√	⊙		⊙

First+集合计算

$$First^{+}(A \to \beta) = \begin{cases} First(\beta), & if \epsilon \notin First(\beta) \\ First(\beta) \cup Follow(A), & otherwise \end{cases}$$

	num	id	$+$	$-$	$\times$	$\div$	(	)	ϵ	eof
$Expr$	√	√					√	$\cancel{\odot}$		$\cancel{\odot}$
$Expr'$			√	√				$\odot$	√	$\odot$
$Term$	√	√	$\cancel{\odot}$	$\cancel{\odot}$			√	$\cancel{\odot}$		$\cancel{\odot}$
$Term'$			$\odot$	$\odot$	√	√		$\odot$	√	$\odot$
$Facor$	√	√	$\cancel{\odot}$	$\cancel{\odot}$	$\cancel{\odot}$	$\cancel{\odot}$	√	$\cancel{\odot}$		$\cancel{\odot}$

解析表构造：应用哪条规则可得到目标终结符?

	num	id	$+$	$-$	$\times$	$\div$	(	)	ϵ	eof
$Expr$	√	√					√			
$Expr'$			√	√				⊙	√	⊙
$Term$	√	√					√			
$Term'$			⊙	⊙	√	√		⊙	√	⊙
$Facor$	√	√					√			

[1] $Expr \rightarrow Term\ Expr'$
[2] $Expr' \rightarrow +\ Term\ Expr'$
[3] $\quad\quad\quad |\ -\ Term\ Expr'$
[4] $\quad\quad\quad |\ \epsilon$

[5] $Term \rightarrow Factor\ Term'$
[6] $Term' \rightarrow \times\ Factor\ Term'$
[7] $\quad\quad\quad |\ \div\ Factor\ Term'$
[8] $\quad\quad\quad |\ \epsilon$

[9] $Facor \rightarrow (Expr)$
[10] $\quad\quad\quad |\ num$
[11] $\quad\quad\quad |\ id$

	num	id	$+$	$-$	$\times$	$\div$	(	)	ϵ	eof
$Expr$	1	1					1			
$Expr'$			2	3				4		4
$Term$	5	5					5			
$Term'$			8	8	6	7		8		8
$Facor$	10	11					9			

应用解析表

	num	id	+	−	×	÷	(	)	*eof*
Expr	1	1					1		
Expr′			2	3				4	4
Term	5	5					5		
Term′			8	8	6	7		8	8
Facor	10	11					9		

(num+num)×num

[1] $Expr \rightarrow Term\ Expr'$
[2] $Expr' \rightarrow +\ Term\ Expr'$
[3] $\quad\quad |\ -\ Term\ Expr'$
[4] $\quad\quad |\ \epsilon$

[5] $Term \rightarrow Factor\ Term'$
[6] $Term' \rightarrow \times\ Factor\ Term'$
[7] $\quad\quad |\ \div\ Factor\ Term'$
[8] $\quad\quad |\ \epsilon$

[9] $Facor \rightarrow (Expr)$
[10] $\quad\quad |\ num$
[11] $\quad\quad |\ id$

word	cur	Rule	Stack
(	Expr	[1]	Term, Expr’
(	Term	[5]	Factor, Term’, Expr’
(	Factor	[9]	(, Expr,), Term’, Expr’
(	(	-	Expr,), Term’, Expr’
num	Expr	[1]	Term, Expr’,), Term’, Expr’
num	Term	[5]	Factor, Term’, Expr’,), Term’, Expr’
num	Factor	[10]	num, Term’, Expr’,), Term’, Expr’
num	num	-	Term’, Expr’,), Term’, Expr’
+	Term’	[8]	Expr’,), Term’, Expr’
+	Expr’	[2]	+, Term, Expr’,), Term’, Expr’
+	+	-	Term, Expr’,), Term’, Expr’
num	Term	[5]	Factor, Term’, Expr’,), Term’, Expr’
num	Factor	[10]	num, Term’, Expr’,), Term’, Expr’
num	num	-	Term’, Expr’,), Term’, Expr’
)	Term’	[8]	Expr’,), Term’, Expr’
)	Expr’	[4]	), Term’, Expr’
)	)	-	Term’, Expr’
+	Term’	[8]	Expr’
…	…	…	…

练习：

- 将正则表达式CFG改写为LL(1)语法并写出应用解析表

$$
\begin{aligned}
&< regex >::=< union > \mid < concat > \\
&< union >::=< regex > \text{"|"} < concat > \\
&< concat >::=< concat >< term > \mid < term > \\
&< term >::=< element >* \mid < element > \\
&< element >::= (< regex >) \mid < alphanum >
\end{aligned}
$$

解答：消除左递归

- 左递归问题一：
  - $< concat >::=< concat >< term > \mid < term >$
- 改写结果：
  - $< concat >::=< term >< concat' >$
  - $< concat' >::=< term >< concat' > \mid \epsilon$
- 和下列形式等价（但转换AST时要注意结合性）：
  - $< concat >::=< term >< concat > \mid < term >$

[1]	$< regex >::=< concat >< regex' >$
[2]	$< regex' >::= "\|" < concat >< regex' >$
[3]	$\mid \epsilon$
[4]	$< concat >::=< term >< concat' >$
[5]	$< concat' >::=< term >< concat' >$
[6]	$\mid \epsilon$
[7]	$< term >::=< element >*$
[8]	$\mid < element >$
[9]	$< element >::= (< regex >)$
[10]	$\mid < alphanum >$

- 左递归问题二（间接左递归）：
  - $< regex >::=< union > \mid < concat >$
  - $< union >::=< regex > "|" < concat >$
  - $\Rightarrow < regex >::=< regex > "|" < concat > \mid < concat >$
- 改写结果：
  - $< regex >::=< concat >< regex' >$
  - $< regex' >::= "|" < concat >< regex' > \mid \epsilon$

解答：消除回溯

- 回溯问题语法
  - $< term >::=< element >*$
  - $\qquad\qquad | < element >$
- 改写结果：
  - $< term >::=< element >< follow >$
  - $< follow >::=* \ |\epsilon$

```
[1]  <regex> ::= <concat><regex'>
[2]  <regex'> ::= "|" <concat><regex'>
[3]              |ε
[4]  <concat> ::= <term><concat'>
[5]  <concat'> ::= <term><concat'>
[6]              |ε
[7]  <term> ::= <element><follow>
[8]  <follow> ::= *
[9]              |ε
[10] <element> ::= (<regex>)
[11]             | <alphanum>
```

解答：构建LL(1)解析表

	(	)	$A-Z, a-z, 0-9$	|	*	ϵ	eof
regex	√[1]		√[1]				
regex'				√[1]		√	⊙[3]
concat	√[4]		√[4]				
concat'	√[5]		√[5]			√	⊙[6]
term	√[7]		√[7]				
element	√[10]		√[11]				
follow	⊙[9]		⊙[9]		√[8]	√	
alphanum			√[12]				

√:first()
⊙:follow()
[]:语法规则条目

通用自顶向下语法分析算法：Earley算法

- 三种基本操作：
  - 预测（Prediction）：对于每个状态$X \to \alpha \circ Y\beta$，根据语法规则预测$Y \to \circ\gamma$。
  - 扫描（Scanning）：如果下一个待处理的符号是a，并且存在状态 $X \to \alpha \circ a\beta$，则扫描该字符并且将状态变更为 $X \to \alpha a \circ \beta$ 。
  - 完成（Completion）：$Y \to \gamma\circ$完成了对Y的分析，进而更新 $X \to \alpha \circ Y\beta$为 $X \to \alpha Y \circ \beta$。

Earley
算法参考

```
DECLARE ARRAY S;

function INIT(words) {
    S ← CREATE-ARRAY(LENGTH(words) + 1)
    for k ← from 0 to LENGTH(words)
        S[k] ← EMPTY-ORDERED-SET
}
function EARLEY-PARSE(words, grammar) {
    INIT(words)
    ADD-TO-SET((γ → •S, 0), S[0])
    for k ← from 0 to LENGTH(words)
        for each state in S[k] { // S[k] can expand during this loop
            if not FINISHED(state) {
                if NEXT-ELEMENT-OF(state) is a nonterminal then
                    PREDICTOR(state, k, grammar)    // non-terminal
                else
                    SCANNER(state, k, words)        // terminal
            }
            else
                COMPLETER(state, k)
        }
    }
    return chart
}
procedure PREDICTOR((A → α•Bβ, j), k, grammar) {
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar)
        ADD-TO-SET((B → •γ, k), S[k])
}
procedure SCANNER((A → α•aβ, j), k, words)
    if a ⊂ PARTS-OF-SPEECH(words[k])
        ADD-TO-SET((A → αa•β, j), S[k+1])

procedure COMPLETER((B → γ•, x), k)
    for each (A → α•Bβ, j) in S[x]
        ADD-TO-SET((A → αB•β, j), S[k])
```


https://en.wikipedia.org/wiki/Earley_parser

Earley算法

如何根据下列语法规则解析(3+4)×5?

(num+num)×num

[1] $Expr \rightarrow Expr + Term$
[2] $\qquad | \; Expr \; - Term$
[3] $\qquad | \; Term$
[4] $Term \rightarrow Term \; \times \; Factor$
[5] $\qquad | \; Term \div Factor$
[6] $\qquad | \; Factor$
[7] $Facor \rightarrow (Expr)$
[8] $\qquad | \; num$
[9] $\qquad | \; id$

no	production	origin	comment
s(0) = ◦(num+num)×num			
1	$Expr \rightarrow \circ \; Expr + Term$	s(0)	start rule
2	$Expr \rightarrow \circ \; Expr - Term$	s(0)	start rule
3	$Expr \rightarrow \circ \; Term$	s(0)	start rule
4	$Term \rightarrow \circ \; Term \; \times \; Factor$	s(0)	Predict from [0][3]
5	$Term \rightarrow \circ \; Term \div Factor$	s(0)	Predict from [0][3]
6	$Term \rightarrow \circ \; Factor$	s(0)	Predict from [0][3]
7	$Facor \rightarrow \circ \; (Expr)$	s(0)	Predict from [0][6]
8	$Facor \rightarrow \circ \; num$	s(0)	Predict from [0][6]
9	$Facor \rightarrow \circ \; id$	s(0)	Predict from [0][6]
s(1) = (◦num+num)×num			
1	$Facor \rightarrow (\circ \; Expr)$	s(0)	Scan from [0][7]
2	$Expr \rightarrow \circ \; Expr + Term$	s(1)	Predict from [1][1]
3	$Expr \rightarrow \circ \; Expr - Term$	s(1)	Predict from [1][1]
4	$Expr \rightarrow \circ \; Term$	s(1)	Predict from [1][1]
5	$Term \rightarrow \circ \; Term \; \times \; Factor$	s(1)	Predict from [1][4]
6	$Term \rightarrow \circ \; Term \div Factor$	s(1)	Predict from [1][4]
7	$Term \rightarrow \circ \; Factor$	s(1)	Predict from [1][4]
8	$Facor \rightarrow \circ \; (Expr)$	s(1)	Predict from [1][7]
9	$Facor \rightarrow \circ \; num$	s(1)	Predict from [1][7]
10	$Facor \rightarrow \circ \; id$	s(1)	Predict from [1][7]

Earley算法

$$
\begin{array}{ll}
[1] & Expr \to Expr + Term \\
[2] & \quad\quad | \; Expr \; - Term \\
[3] & \quad\quad | \; Term \\
[4] & Term \to Term \; \times \; Factor \\
[5] & \quad\quad | \; Term \div Factor \\
[6] & \quad\quad | \; Factor \\
[7] & Facor \to (Expr) \\
[8] & \quad\quad\quad | \; num \\
[9] & \quad\quad\quad | \; id
\end{array}
$$

no	production	origin	comment
s(2) = (num◦+num)×num			
1	$Facor \to num \circ$	s(1)	Scan from [1][9]
2	$Term \to Factor \circ$	s(1)	Complete [1][7]
3	$Term \to Term \circ \times \; Factor$	s(1)	Complete [1][5]
4	$Term \to Term \circ \div Factor$	s(1)	Complete [1][6]
5	$Expr \to Term \circ$	s(1)	Complete [1][4]
6	$Expr \to Expr \circ +Term$	s(1)	Complete [1][2]
7	$Expr \to Expr \circ -Term$	s(1)	Complete [1][3]
8	$Facor \to (Expr \circ)$	s(0)	Complete [1][1]
s(3) = (num+◦num)×num			
1	$Expr \to Expr + \circ \; Term$	s(1)	Scan from [2][6]
2	$Term \to \circ \; Term \; \times \; Factor$	s(3)	Predict from [3][1]
3	$Term \to \circ \; Term \div Factor$	s(3)	Predict from [3][1]
4	$Term \to \circ \; Factor$	s(3)	Predict from [3][1]
5	$Facor \to \circ \; (Expr)$	s(3)	Predict from [3][4]
6	$Facor \to \circ \; num$	s(3)	Predict from [3][4]
7	$Facor \to \circ \; id$	s(3)	Predict from [3][4]

Earley算法

[1] $Expr \rightarrow Expr + Term$
[2] $\quad\quad | \, Expr - Term$
[3] $\quad\quad | \, Term$
[4] $Term \rightarrow Term \times Factor$
[5] $\quad\quad | \, Term \div Factor$
[6] $\quad\quad | \, Factor$
[7] $Facor \rightarrow (Expr)$
[8] $\quad\quad | \, num$
[9] $\quad\quad | \, id$

no	production	origin	comment
s(4) = (num+num◦)×num			
1	$Facor \rightarrow num \circ$	s(3)	Scan from [3][6]
2	$Term \rightarrow Factor \circ$	s(3)	Complete [3][7]
3	$Term \rightarrow Term \circ \times Factor$	s(3)	Complete [3][5]
4	$Term \rightarrow Term \circ \div Factor$	s(3)	Complete [3][6]
5	$Expr \rightarrow Term \circ$	s(3)	Complete [3][4]
6	$Expr \rightarrow Expr \circ +Term$	s(3)	Complete [3][2]
7	$Expr \rightarrow Expr \circ -Term$	s(3)	Complete [3][3]
8	$Expr \rightarrow Expr + Term \circ$	s(1)	Complete [3][1]
9	$Facor \rightarrow (Expr \circ)$	s(0)	Complete [1][1]
s(5) = (num+num)◦×num			
1	$Facor \rightarrow (Expr) \circ$	s(0)	Scan from [4][9]
2	$Term \rightarrow Factor \circ$	s(0)	Complete [0][6]
3	$Term \rightarrow Term \circ \times Factor$	s(0)	Complete [0][4]
4	$Term \rightarrow Term \circ \div Factor$	s(0)	Complete [0][5]
5			
6			
7			

Earley算法

[1] $Expr \rightarrow Expr + Term$
[2] $\quad | Expr - Term$
[3] $\quad | Term$
[4] $Term \rightarrow Term \times Factor$
[5] $\quad | Term \div Factor$
[6] $\quad | Factor$
[7] $Facor \rightarrow (Expr)$
[8] $\quad | num$
[9] $\quad | id$

no	production	origin	comment
s(6) = (num+num)×∘num			
1	$Term \rightarrow Term \times \circ\ Factor$	s(0)	Scan from [6][3]
2	$Facor \rightarrow \circ\ (Expr)$	s(0)	Predict from [6][1]
3	$Facor \rightarrow \circ\ num$	s(6)	Predict from [6][1]
4	$Facor \rightarrow \circ\ id$	s(6)	Predict from [6][1]
s(7) = (num+num)×num∘			
1	$Facor \rightarrow num\ \circ$	s(6)	Scan from [6][4]
2	$Term \rightarrow Term \times\ Factor\ \circ$	s(0)	Complete [6][1]
3	$Expr \rightarrow Term\ \circ$	s(0)	Complete [0][3]
4			
5			
6			
7			

Earley算法能否解析非CFG语法?

练习：

- 应用Earley算法解析正则num+num-num
- 应用Earley算法解析正则 (a|b)*

$$
\begin{array}{ll}
[1] & Expr \rightarrow Expr + Term \\
[2] & \quad\quad |\ Expr\ - Term \\
[3] & \quad\quad |\ Term \\
[4] & Term \rightarrow Term\ \times\ Factor \\
[5] & \quad\quad |\ Term \div Factor \\
[6] & \quad\quad |\ Factor \\
[7] & Facor \rightarrow (Expr) \\
[8] & \quad\quad\ |\ num \\
[9] & \quad\quad\ |\ id
\end{array}
$$

$$
\begin{array}{ll}
[1] & <\mathrm{regex}>::=<\mathrm{union}> \\
[2] & \quad\quad\quad\quad\quad |<\mathrm{concat}> \\
[3] & <\mathrm{union}>::=<\mathrm{regex}>\ "|"<\mathrm{concat}> \\
[4] & <\mathrm{concat}>::=<\mathrm{concat}><\mathrm{term}> \\
[5] & \quad\quad\quad\quad |<\mathrm{term}> \\
[6] & <\mathrm{term}>::=<\mathrm{element}>* \\
[7] & \quad\quad\quad\quad |<\mathrm{element}> \\
[8] & <\mathrm{element}>::=(<\mathrm{regex}>) \\
[9] & \quad\quad\quad\quad |<alphanum>
\end{array}
$$

自顶向下解析算法小结

- Earley算法
  - 支持所有CFG
  - 复杂度$O(n^3)$
    - 无歧义语法：$O(n^2)$
- 能否有更快的算法？
  - 要求CFG为LL(1)
    - 不存在左递归
    - 不存在回溯

四、自底向上分析

SLR/LALR/LR(1)

基本思路：基于规约的方法

- 如果在句法分析栈的上边缘找到β且$A \rightarrow \beta$，则将其规约为A；
- 否则移进

$$
\begin{array}{ll}
[1] & S \rightarrow \epsilon \\
[2] & \quad\ \mid [S]S
\end{array}
$$

	当前栈	待输入
		[ ]
第1步:	[	]
第2步:	[S	]
第3步:	[S]	
第4步:	[S]S	
第5步:	S	

移进和规约（Shift-Reduce）

Shift
移进表示：

$$\frac{w:\beta}{wa:\beta a}\text{shift}$$

Reduce
规约表示：

$$\frac{[r]\ X\to\alpha \quad w:\beta\alpha}{w:\beta X}\text{reduce(r)}$$

示例：

$$\frac{\epsilon:\beta_0}{[:\beta_0[}\text{shift}$$

$$\frac{[:\beta_1}{[]:\beta_1]}\text{shift}$$

…

$$\frac{[][[]][]:\beta_7}{[][[]][]]:\beta_7]}\text{shift}$$

$$\frac{[1]\ S\to\epsilon \quad w:\beta}{w:\beta S}\text{reduce([1])}$$

$$\frac{[2]\ S\to[S]S \quad w:\beta[S]S}{w:\beta S}\text{reduce([2])}$$

移进-规约应用

最右推导

[1]	$S \to \epsilon$
[2]	$\mid [S]S$

Stack		Input	Derivation	Action
	||	[][[][]]	$\epsilon : \epsilon$	shift [
[	||	[][[][]]	$[: [$	reduce([1])
[S	||	][[][]]	$[: [S$	shift
[S]	||	[[][]]	$[] : [S]$	shift
[S][	||	[][]]	$[][: [S][$	shift
[S][[	||	][]]	$[][[: [S][[$	reduce([1])
[S][[S	||	][]]	$[][[: [S][[S$	shift
[S][[S]	||	[]]	$[][[] : [S][[S]$	shift
[S][[S][	||	]]	$[][[][: [S][[S][$	reduce([1])
[S][[S][S	||	]]	$[][[][: [S][[S][S$	shift
[S][[S][S]	||	]	$[][[][] : [S][[S][S]$	reduce([1])
[S][[S][S]S	||	]	$[][[][] : [S][[S][S]S$	reduce([2])
[S][[S]S	||	]	$[][[][] : [S][[S]S$	reduce([2])
[S][S	||	]	$[][[][] : [S][S$	shift
[S][S]	||		$[][[][]] : [S][[S]S]$	reduce([1])
[S][S]S	||		$[][[][]] : [S][S]S$	reduce([2])
[S]S	||		$[][[][]] : [S]S$	reduce([2])
S	||		$[][[][]] : S$	

LR(0)句柄分析

[1] $E \to E + T$
[2] $\quad | \ T$
[3] $T \to T \times F$
[4] $\quad | \ F$
[5] $F \to (E)$
[6] $\quad | \ id$

增强语法 ⟹

$Goal \to E$
$E \to E + T$
$\quad | \ T$
$T \to T \times F$
$\quad | \ F$
$F \to (E)$
$\quad | \ id$

增强语法有唯一的目标符号Goal，不会出现在产生式的右侧。

$Goal \to E$ $\xRightarrow{\text{句柄状态}}$ $Goal \to \circ E$, $Goal \to E \circ$

可应用$E \to E + T$规约的句柄状态 ⟹
$E \to \circ E + T$
$E \to E \circ + T$
$E \to E + \circ T$
$E \to E + T \circ$

可应用$E \to T$规约的句柄状态 ⟹
$E \to \circ T$
$E \to T \circ$

可应用$T \to T \times F$规约的句柄状态 ⟹
$T \to \circ T \times F$
$T \to T \circ \times F$
$T \to T \times \circ F$
$T \to T \times F \circ$

可应用$T \to F$规约的句柄状态 ⟹
$T \to \circ F$
$T \to F \circ$

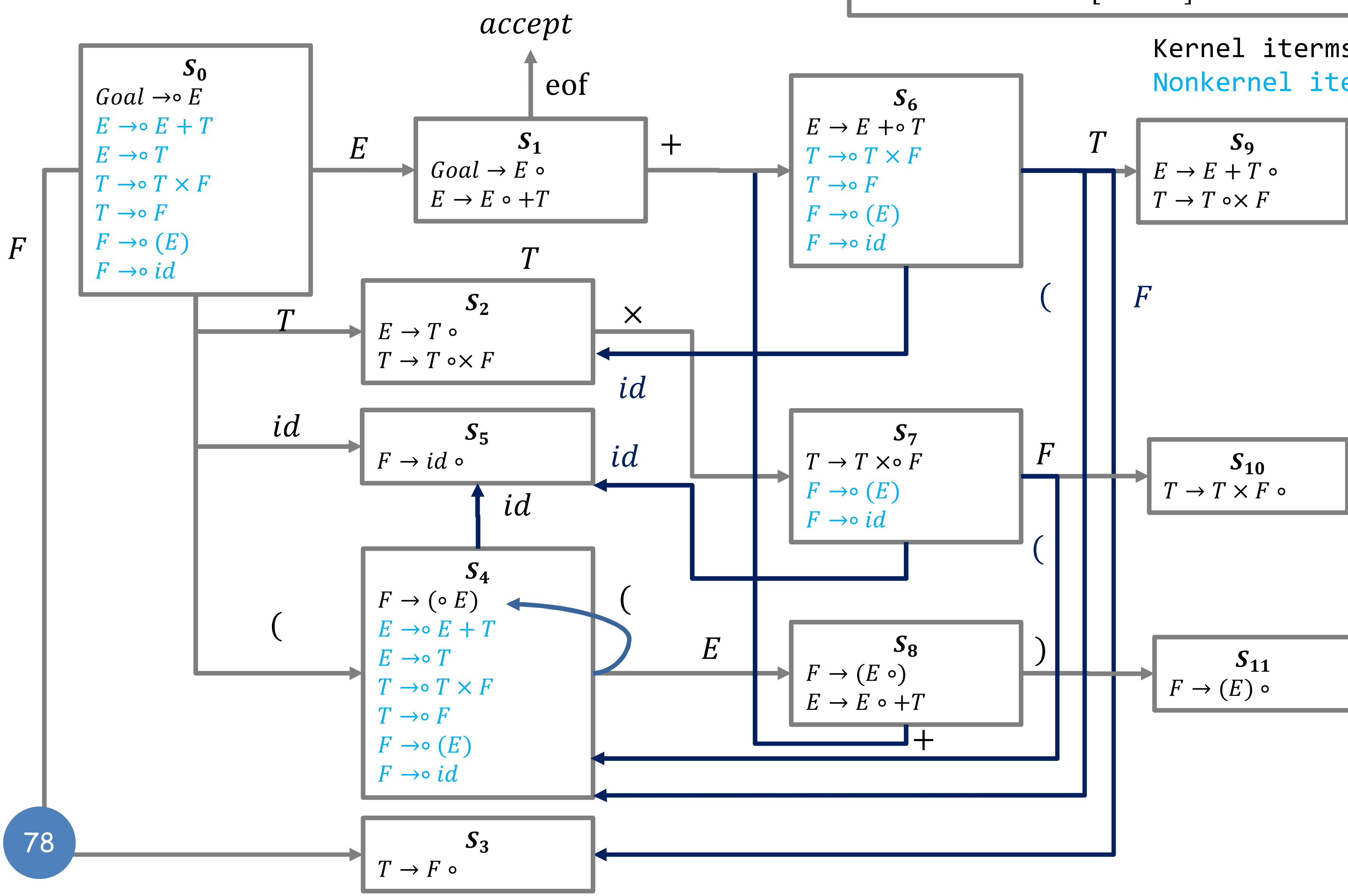
可应用$F \to (E)$规约的句柄状态 ⟹
$F \to \circ (E)$
$F \to (\circ E)$
$F \to (E \circ)$
$F \to (E) \circ$

可应用$F \to id$规约的句柄状态 ⟹
$F \to \circ id$
$F \to id \circ$

LR(0)自动机构建

```
While (S has changed)
  for each item [A → β ∘ Cδ, a] ∈ S
    for each production [C → λ] ∈ G
      if [C →∘ λ] ∉ S
        S ← S ∪ [C →∘ λ]
```

Kernel iterms
Nonkernel items



LR(0)自动机的状态转移关系表

迭代	规范项	id	+	×	(	)	eof	E	T	F
0	S_0	S_5	∅	∅	S_4	∅	∅	S_1	S_2	S_3
1	S_1	∅	S_6	∅	∅	∅	$accept$	∅	∅	∅
	S_2	∅	∅	S_7	∅	∅	∅	∅	∅	∅
	S_3	∅	∅	∅	∅	∅	∅	∅	∅	∅
	S_4	S_5	∅	∅	S_4	∅	∅	S_8	S_2	S_3
	S_5	∅	∅	∅	∅	∅	∅	∅	∅	∅
2	S_6	S_5	∅	∅	S_4	∅	∅	∅	S_9	S_3
	S_7	S_5	∅	∅	S_4	∅	∅	∅	∅	S_{10}
	S_8	∅	S_6	∅	∅	S_{11}	∅	∅	∅	∅
3	S_9	∅	∅	S_7	∅	∅	∅	∅	∅	∅
	S_{10}	∅	∅	∅	∅	∅	∅	∅	∅	∅
	S_{11}	∅	∅	∅	∅	∅	∅	∅	∅	∅

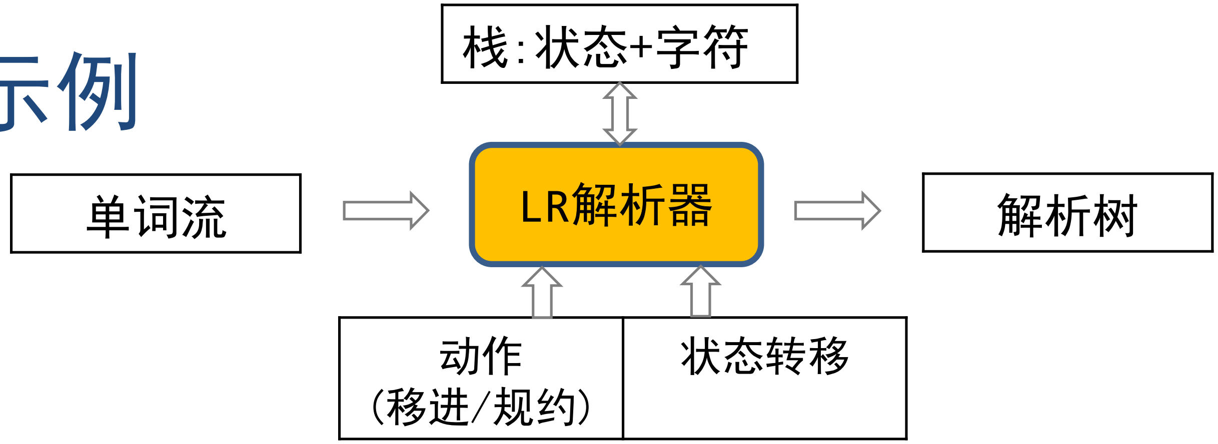
构建SLR解析表

移进条件：如果$A \rightarrow \alpha \circ a\beta \in S_i$，并且$Goto(S_i, a) = S_j$，设置$Action(S_i, a) =$ “$shift\ j$”

规约条件：如果$A \rightarrow \alpha \circ \in S_i$，$\forall a \in Follow(A)$，设置$Action(S_i, a) =$ “$reduce\ A \rightarrow \alpha$”

规范项	Action						Goto		
	id	+	×	(	)	eof	E	T	F
S_0	shift S_5			shift S_4			S_1	S_2	S_3
S_1		shift S_6				acept			
S_2		reduce [2]	shift S_7		reduce [2]	reduce [2]			
S_3									
S_4	shift S_5			shift S_4			S_8	S_2	S_3
S_5		reduce [6]	reduce [6]		reduce [6]	reduce [6]			
S_6	shift S_5			shift S_4				S_9	S_3
S_7	shift S_5			shift S_4					S_{10}
S_8		shift S_6			shift S_{11}				
S_9		reduce [1]	shift S_7		reduce [1]	reduce [1]			
S_{10}		reduce [3]	reduce [3]		reduce [3]	reduce [3]			
S_{11}		reduce [5]	reduce [5]		reduce [5]	reduce [5]			

SLR应用示例



Stack	**Symbols**	**Input**	**Action**
S_0		id×id \$	shift id, goto S_5
S_0S_5	id	×id \$	reduce by $F \rightarrow id$, back to S_0, goto S_3
S_0S_3	F	×id \$	reduce by $T \rightarrow F$, back to S_0, goto S_2
S_0S_2	T	×id \$	shift ×, goto S_7
$S_0S_2S_7$	T ×	id \$	shift id, goto S_5
$S_0S_2S_7S_5$	T × id	\$	reduce by $F \rightarrow id$, back to S_7, goto S_{10}
$S_0S_2S_7S_{10}$	T × F	\$	reduce by $T \rightarrow T \times F$, back to $S_7S_2S_0$, goto S_2
S_0S_2	T	\$	reduce by $E \rightarrow T$, back to S_0, goto S_1
S_0S_1	E	\$	

练习

- 下面的语法是否是LL(1)？是否是SLR(1)

[1]	$S \rightarrow AaAb$
[2]	$\quad\;\; \| BbBa$
[3]	$A \rightarrow \epsilon$
[4]	$B \rightarrow \epsilon$

是LL(1)

$First^{+}(S \rightarrow AaAb) = \{a\}$

$First^{+}(S \rightarrow BbBa) = \{b\}$

$\boldsymbol{S_0}$

$S' \rightarrow \circ\, S$

$S \rightarrow \circ\, AaAb$

$S \rightarrow \circ\, BbAa$

$A \rightarrow \circ$

$B \rightarrow \circ$

不是SLR(1)

$Follow(A) = Follow(B) = \{a, b\}$

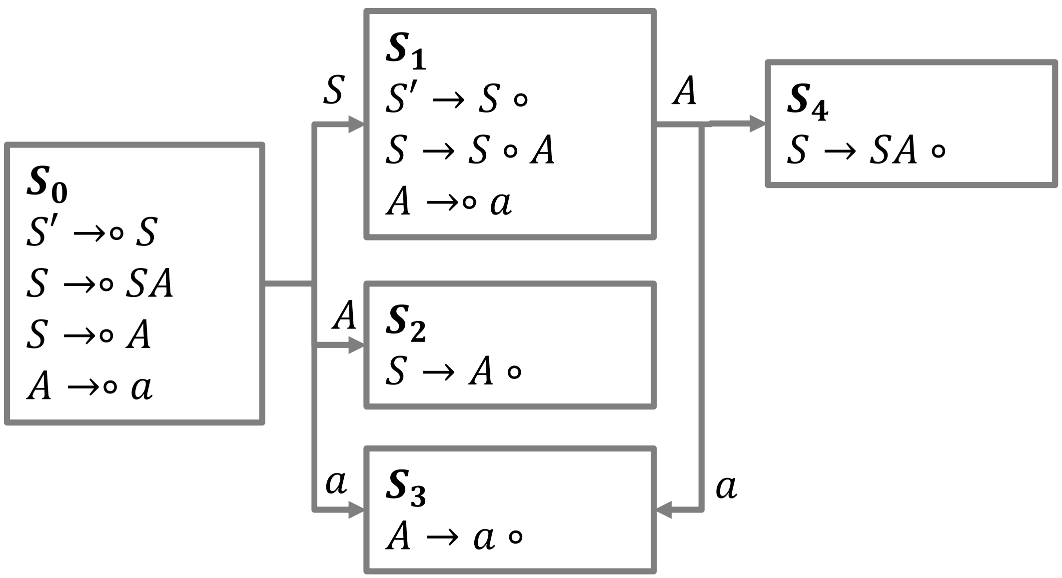
$Action(S_0, a) = reduce[3]$ 或 $reduce[4]$

练习

- 下面的语法是否是LL(1)？是否是SLR(1)

[1] $S \rightarrow SA$
[2] $\quad\;\; |A$
[3] $A \rightarrow a$

不是LL(1)
$First^{+}(S \rightarrow SA) = \{a\}$
$First^{+}(S \rightarrow A) = \{a\}$

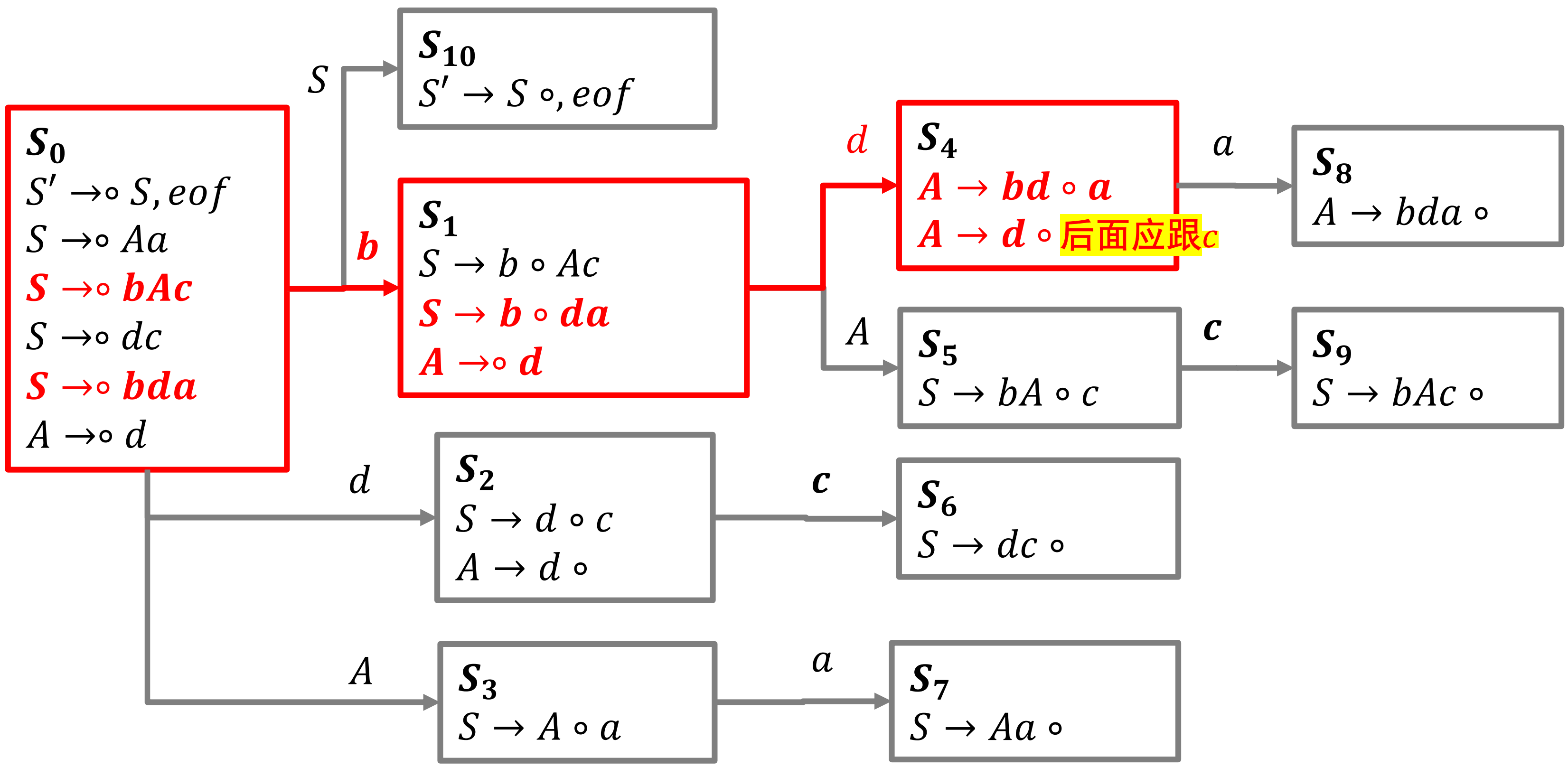


是SLR(1)

二义性语法：移进-规约冲突

- 构造SLR解析表则解析bda时存在移进-规约冲突
- S_4下一个字符为a，可移进
- $a \in Follow(A)$，可规约

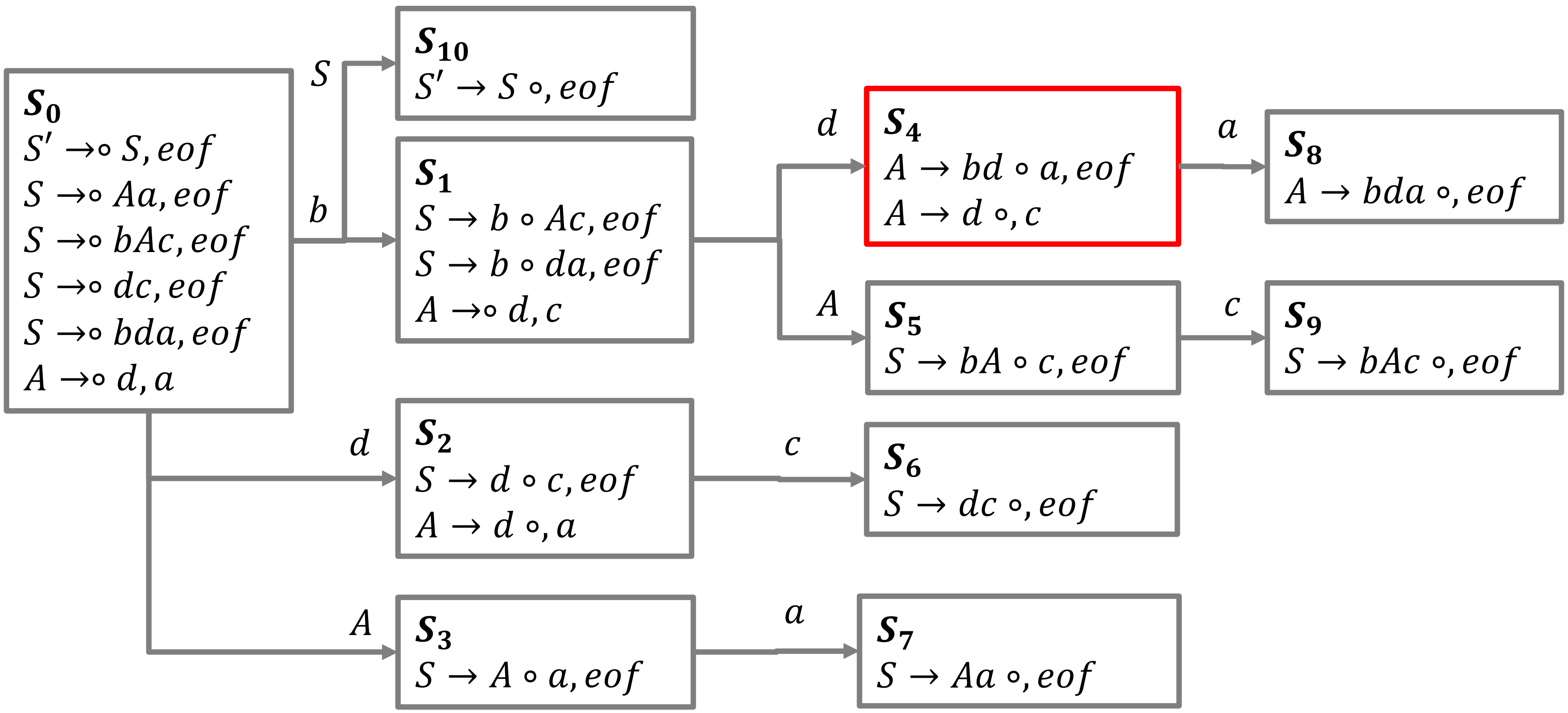
[1]	$S \rightarrow Aa$
[2]	$\quad\mid bAc$
[3]	$\quad\mid dc$
[4]	$\quad\mid bda$
[5]	$A \rightarrow d$



LR(1)自动机构造

[0] $S' \rightarrow S$
[1] $S \rightarrow Aa$
[2] $\quad|bAc$
[3] $\quad|dc$
[4] $\quad|bda$
[5] $A \rightarrow d$

1) 构造其LR(1)项的全集
2) 迭代过程
   - 通过闭包找到规范族
   - 分析规范族之间的状态转移关系



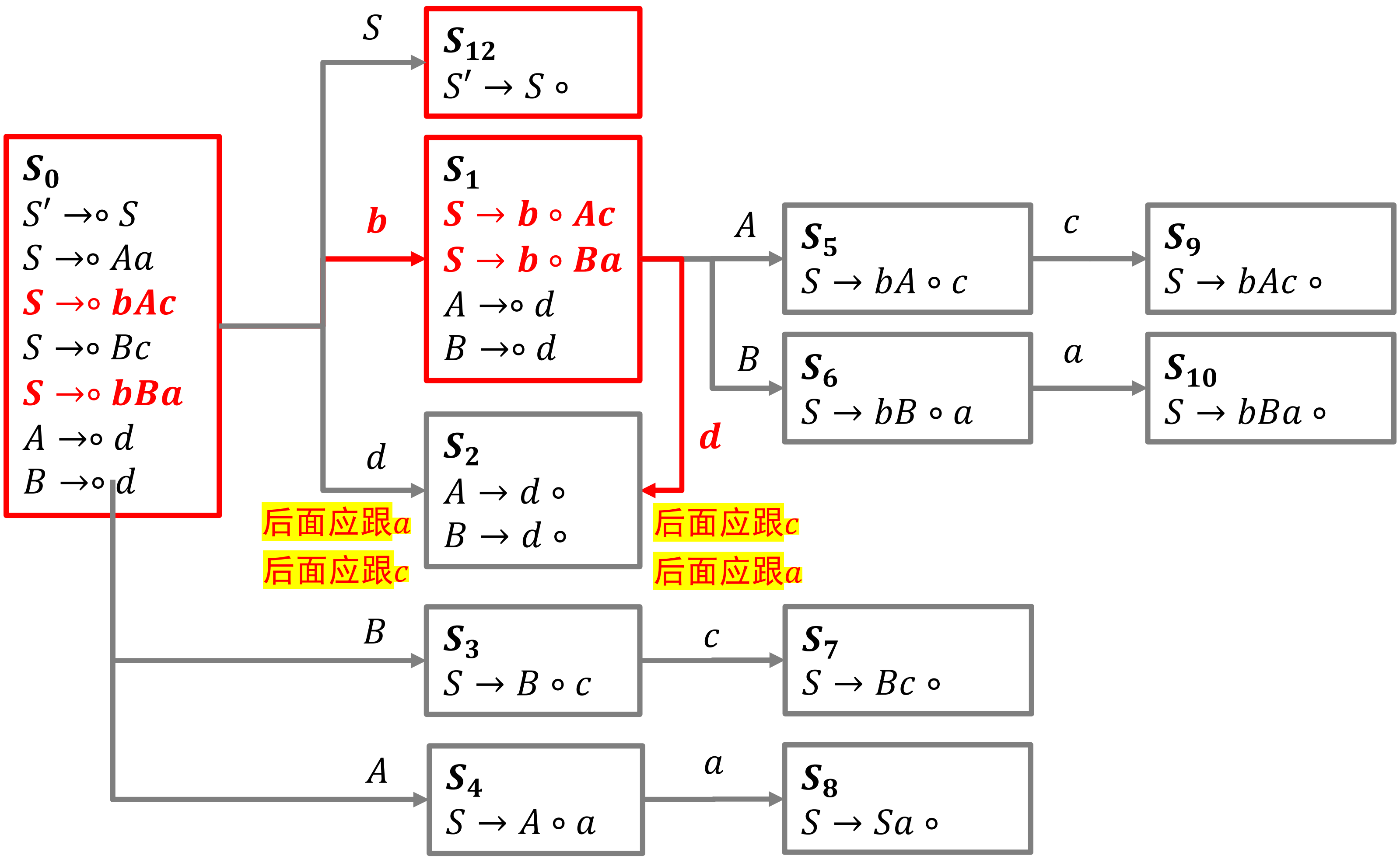
得到LR(1)解析表

规范项	Action					Goto	
	a	b	c	d	eof	S	A
S_0		shift S_1		shift S_2		S_{10}	S_3
S_1				shift S_4			S_5
S_2	reduce [5]		shift S_6				
S_3	shift S_7						
S_4	shift S_8		reduce [5]				
S_5			shift S_9				
S_6					reduce [3]		
S_7					reduce [1]		
S_8					reduce [4]		
S_9					reduce [2]		
S_{10}					*accept*		

二义性语法：规约-规约冲突

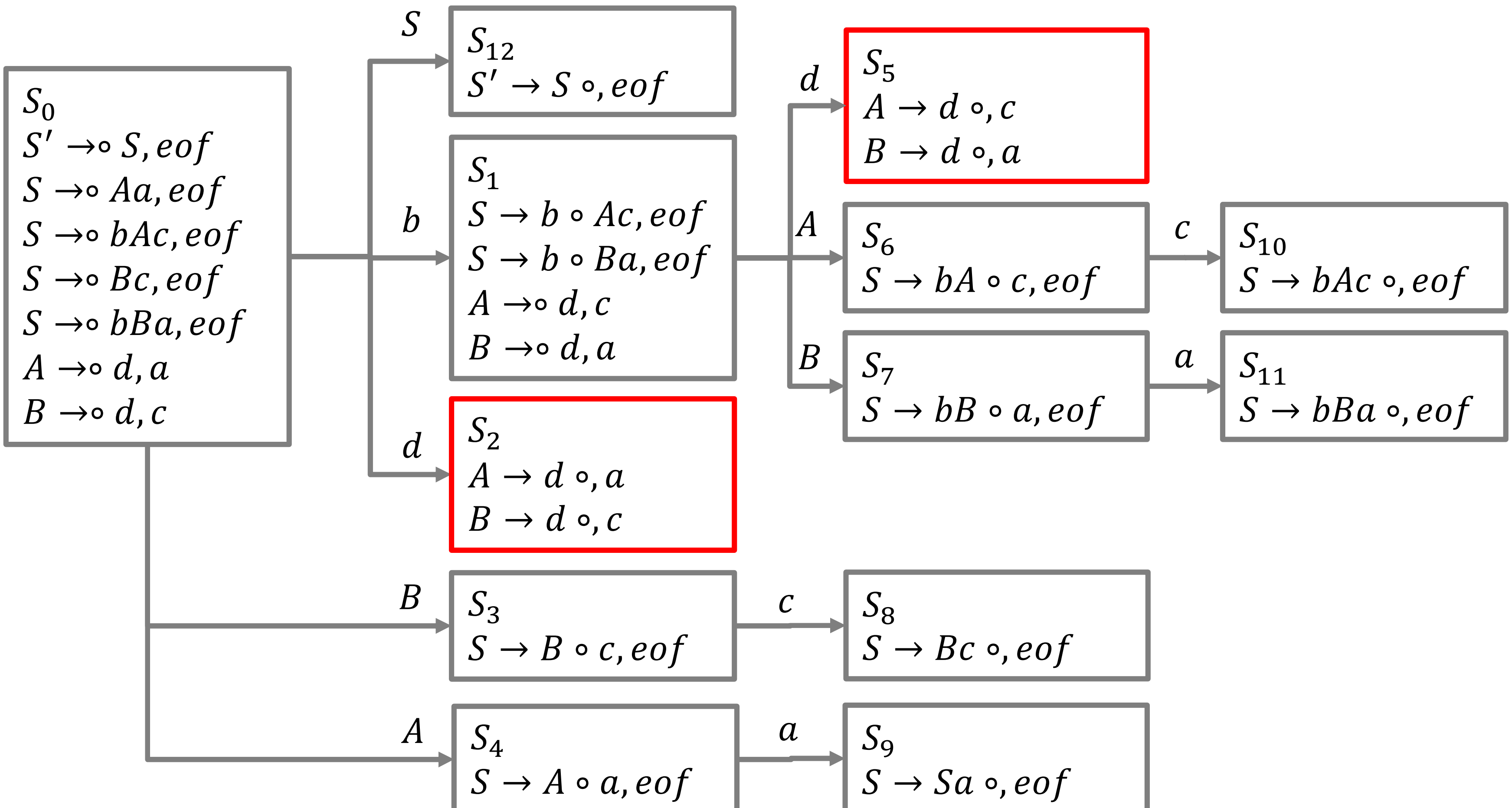
- 解析bdc时存在规约$(A \to d)$-规约$(B \to d)$冲突
- $a \in Follow(A)$ 且 $a \in Follow(B)$

[1]	$S \to Aa$
[2]	$\mid bAc$
[3]	$\mid Bc$
[4]	$\mid bBa$
[5]	$A \to d$
[6]	$B \to d$



LR(1)自动机构造

[1] $S \to Aa$
[2] $\quad|bAc$
[3] $\quad|Bc$
[4] $\quad|bBa$
[5] $A \to d$
[6] $B \to d$



如何选取移进-规约操作？

- 根据当前的栈顶句柄信息：SLR（Simple LR）
  - 通过构造LR(0)自动机和下个字符判断是否可以移进
  - 需要规约时根据Follow判断是否可行
- 自动机构造时考虑Follow信息：经典LR(1)：
  - 问题一：LR（1）支持语法有限，用GLR（Generalized LR）
    - 如果遇到冲突则分叉：复制栈状态，尝试不同规则或移进操作
  - 问题二：LR(1) 规范集太多，用LALR（Lookahead LR）
    - 自动机构造时考虑Follow信息
    - 同时精简规范集

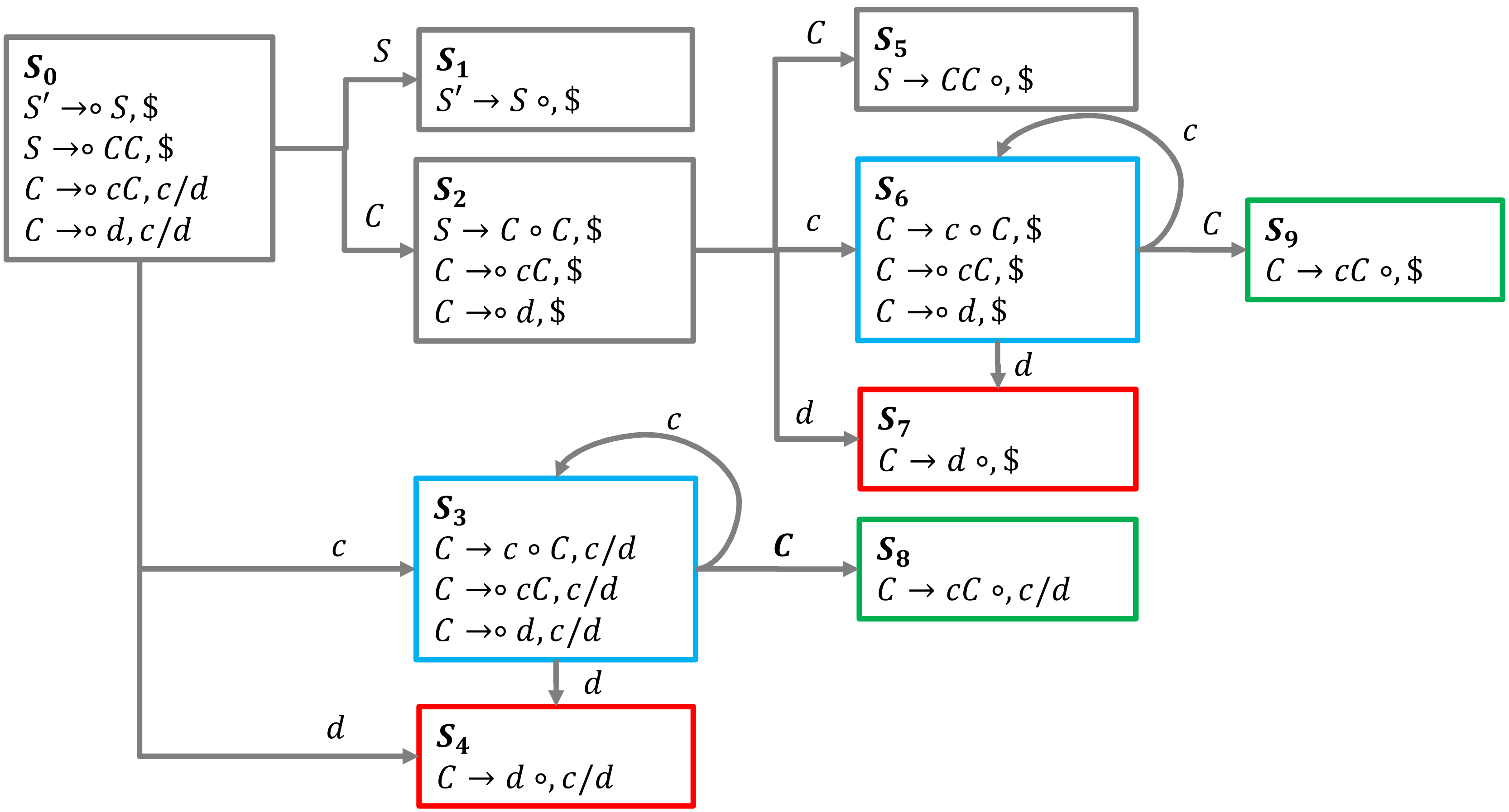
LALR

- SLR(1)存在移进-规约、规约-规约冲突问题，支持的语法范围小；
- LR(1)在规范集构造时融合了Follow信息，可以避免很多冲突问题，但解析表可能会比较大；
- LALR是一种折中方法，解析表大小和SLR相同；
- LALR构造思路：合并句柄状态完全相同的状态集

LALR语法举例

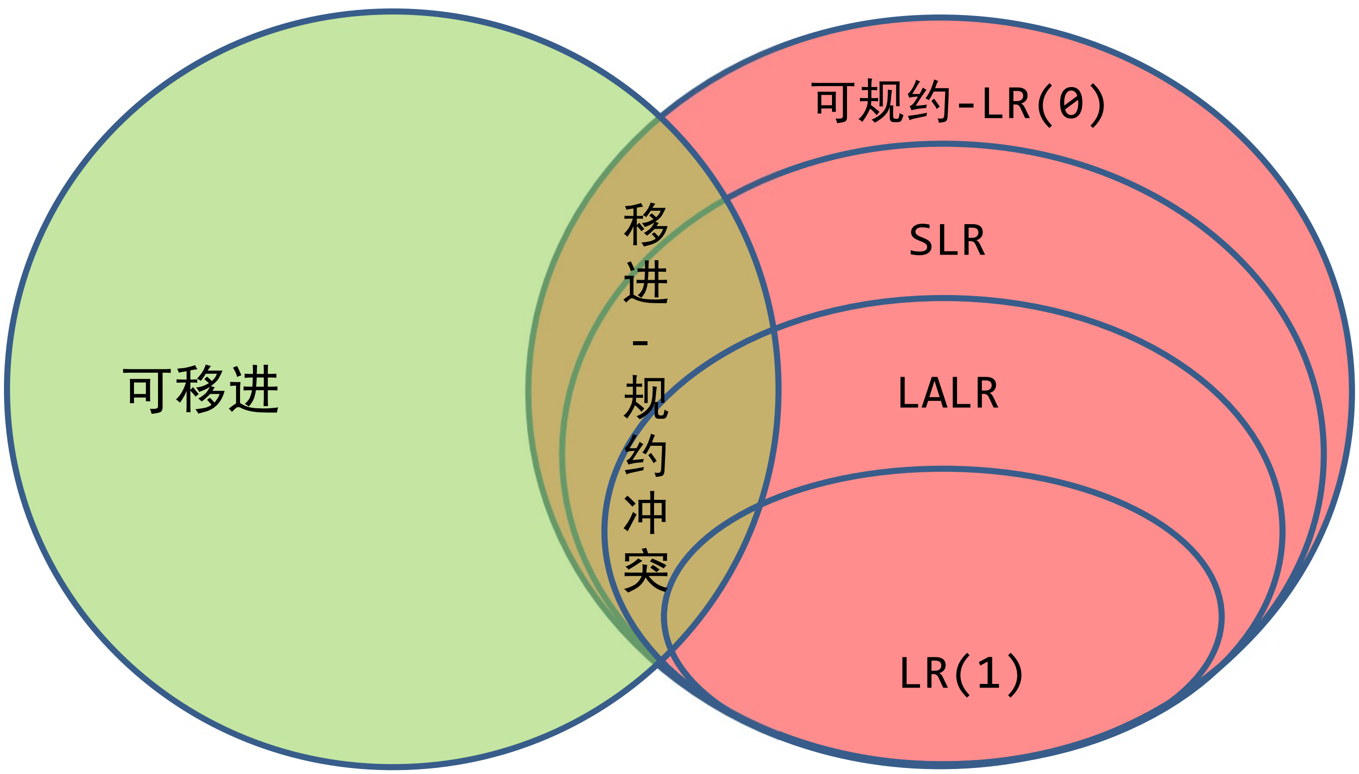
[1] $S' \to S$
[2] $S \to CC$
[3] $C \to cC$
[4] $\quad\quad | \ d$

- 可以合并的规范集
  - S3和S6、S4和S7、S8和S9。



几种语法的关系

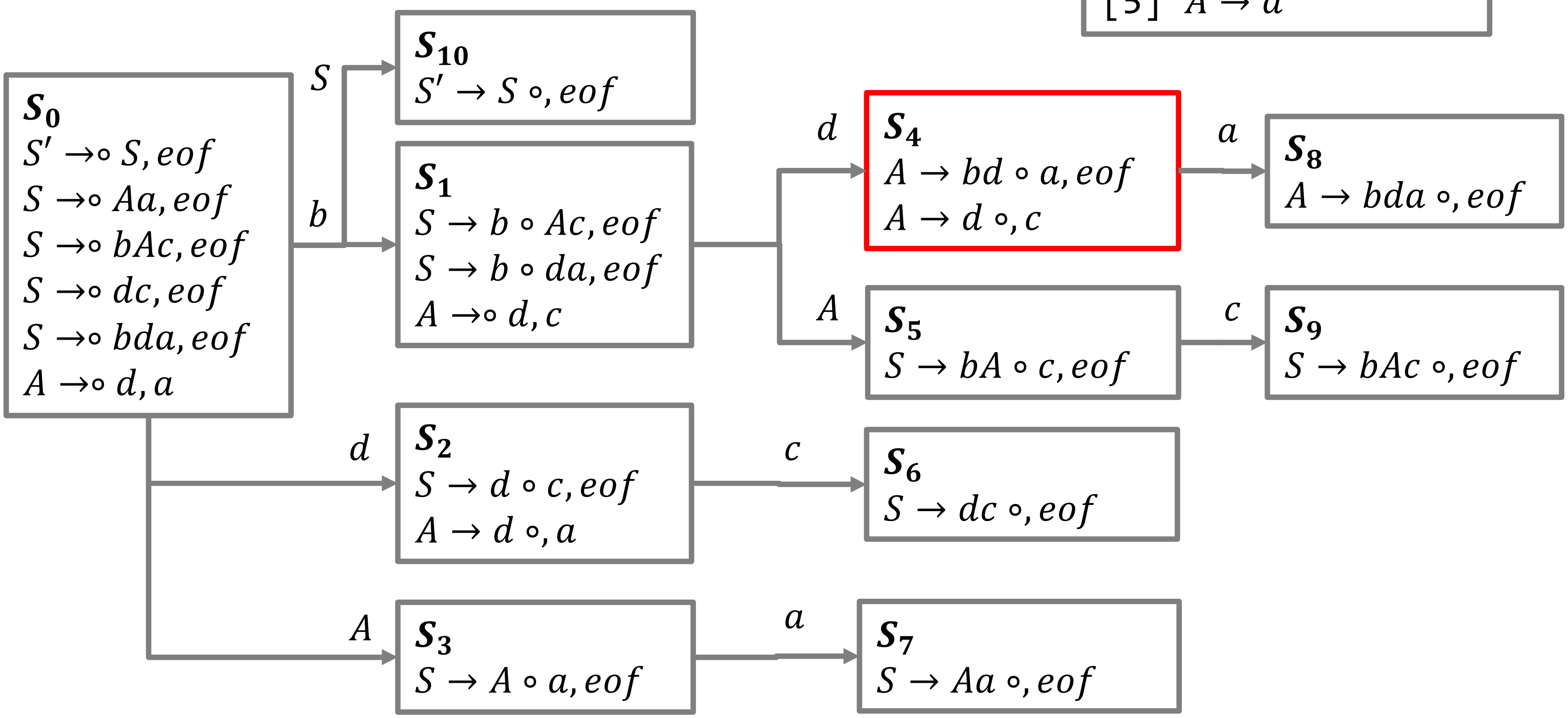
- 语法表达能力：LR(1)>LALR(1)>SLR
  - 规约条件严苛：LR(1)>LALR(1)>SLR
  - 移进条件同LR(0)?



举例：LALR，非SLR(1)语法

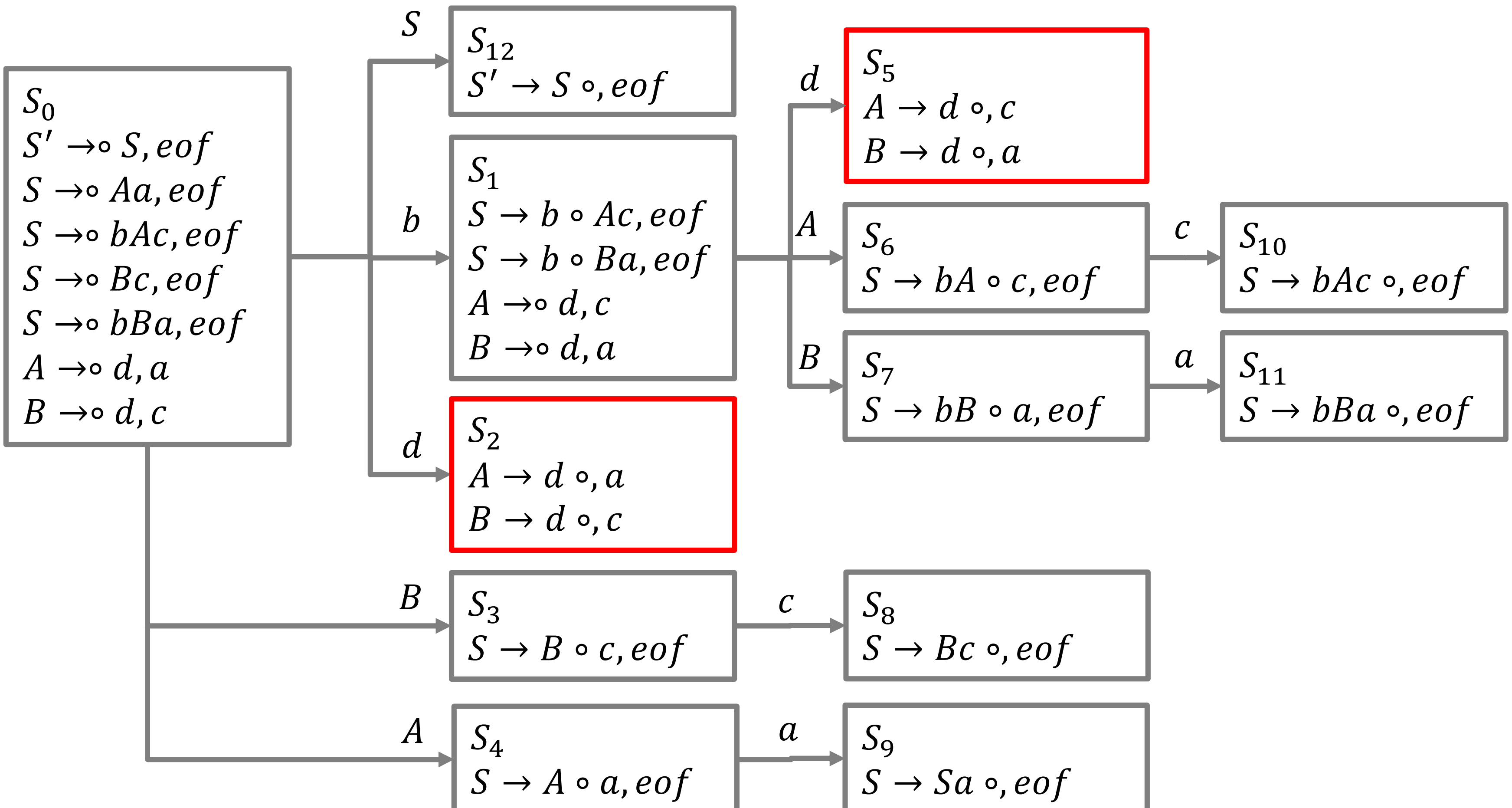
- 构造SLR解析表则解析bda时存在移进-规约冲突
- LALR解析方法可以避免冲突

[1] $S \rightarrow Aa$
[2] $\quad |bAc$
[3] $\quad |dc$
[4] $\quad |bda$
[5] $A \rightarrow d$



举例：LR(1)，非LALR语法

[1] $S \rightarrow Aa$
[2] $\quad |bAc$
[3] $\quad |Bc$
[4] $\quad |bBa$
[5] $A \rightarrow d$
[6] $B \rightarrow d$



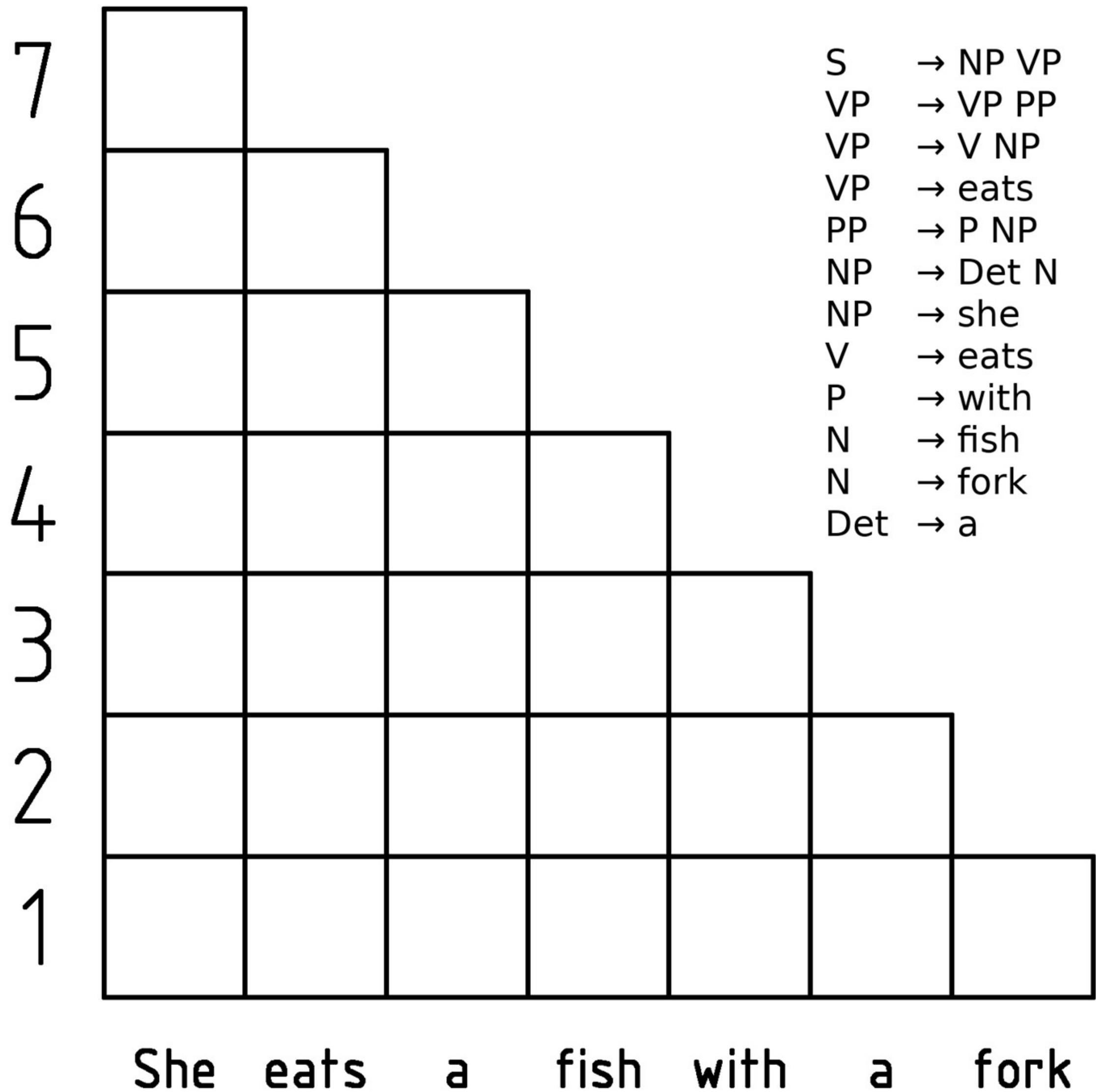
思考

- 下列语法是否是SLR或LR(1)？

$$
\begin{aligned}
&< regex >::=< union > \mid < concat > \\
&< union >::=< regex > "|" < concat > \\
&< concat >::=< concat >< term > \mid < term > \\
&< term >::=< element >* \mid < element > \\
&< element >::= (< regex >) \mid < alphanum >
\end{aligned}
$$

$$
\begin{array}{ll}
[1] & < regex >::=< concat >< regex' > \\
[2] & < regex' >::= "|" < concat >< regex' > \\
[3] & \qquad\qquad \mid \epsilon \\
[4] & < concat >::=< term >< concat' > \\
[5] & < concat' >::=< term >< concat' > \\
[6] & \qquad\qquad \mid \epsilon \\
[7] & < term >::=< element >< follow > \\
[8] & < follow >::=* \\
[9] & \qquad\qquad \mid \epsilon \\
[10] & < element >::= (< regex >) \\
[11] & \qquad\qquad \mid < alphanum >
\end{array}
$$

通用自底向上CFG分析： CYK算法



S → NP VP
VP → VP PP
VP → V NP
VP → eats
PP → P NP
NP → Det N
NP → she
V → eats
P → with
N → fish
N → fork
Det → a

CYK解析算法伪代码参考

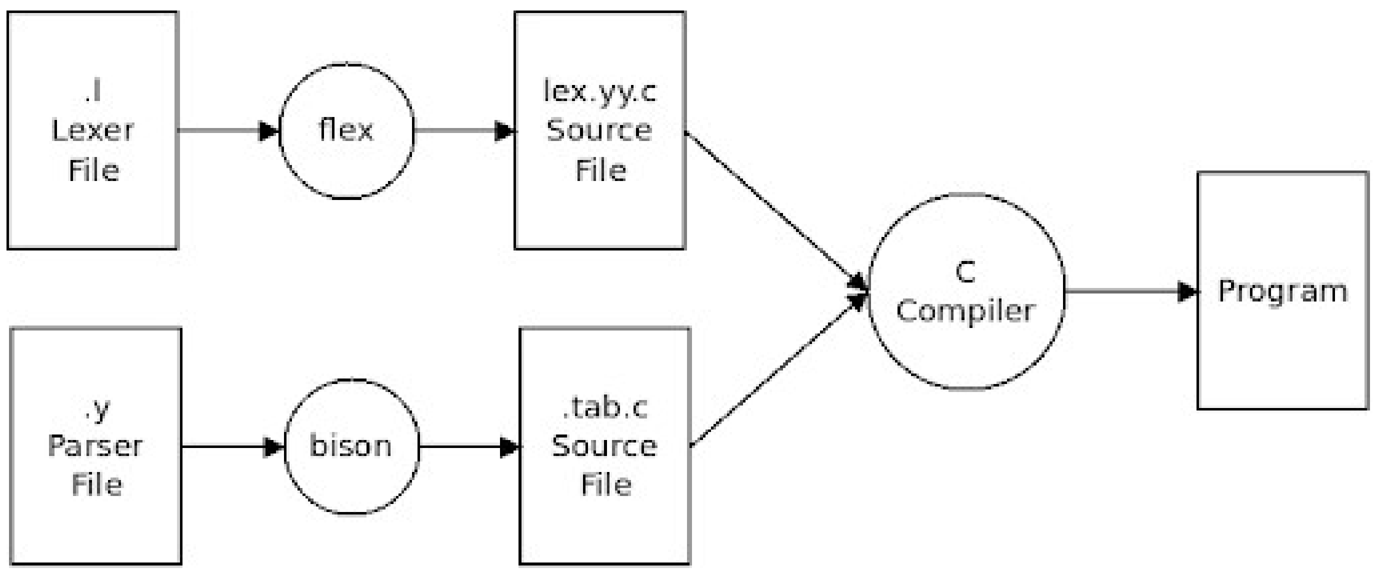
```
INIT:
    Gramma: R_1,R_2,…R_r
    String to parse: w = w_1,w_2,…,w_1
    P[n,n,r] inited with false

Foreach i = 1 to n:
    Foreach R_r->a_i
        P[1,i,r] = True
Foreach l = 2 to n:
    Foreach i = 1 to n–l+1:
        Foreach j = 1 to l-1:
            Foreach R_r->R_a R_b
                If P[j,i,a] and P[l-j,i+j,b]:
                    P[l,i,r]= True
If P[n,1,1]:
    w is a string of the language
Else:
    w is not a string of the language
```

五、语法分析工具

Bison

- 语法分析工具YACC(POSIX)/Bison (GNU)
  - 默认采用LALR(1)解析
  - 支持LR(1)等方法



https://www.gnu.org/software/bison/manual/html_node/

计算器程序示例

```
Main.c 计算器文件
Lexer.l 词法定义
Parser.y 语法定义
Expression.h 文件
Expression.c 功能函数
```

Lex文件

```
%{ /* Lexer.l file */
#include "Expression.h"
#include "Parser.h"
#include <stdio.h> %}

%option outfile="Lexer.c"
header-file="Lexer.h"
%option warn nodefault
%option reentrant noyywrap never-interactive nounistd
%option bison-bridge

%%
[ \r\n\t]* { continue; /* Skip blanks. */ }
[0-9]+ { sscanf(yytext, "%d", &yylval->value); return TOKEN_NUMBER; }
"*" { return TOKEN_STAR; }
"+" { return TOKEN_PLUS; }
"(" { return TOKEN_LPAREN; }
")" { return TOKEN_RPAREN; }
. { continue; /* Ignore unexpected characters. */}

%%
int yyerror(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
    return 0;
}
```

```
%{ /* * Parser.y file * */
#include "Expression.h"
#include "Parser.h"
#include "Lexer.h"
int yyerror(SExpression **expression, yyscan_t scanner, const char *msg) { /* Add error
handling routine as needed */ }
%}

%code requires { typedef void* yyscan_t; }
%output "Parser.c"
%defines "Parser.h"
%define api.pure
%lex-param { yyscan_t scanner }
%parse-param { SExpression **expression }
%parse-param { yyscan_t scanner }
%union { int value; SExpression *expression; }
%token TOKEN_LPAREN "("
%token TOKEN_RPAREN ")"
%token TOKEN_PLUS "+"
%token TOKEN_STAR "*"
%token <value> TOKEN_NUMBER "number"
%type <expression> expr


/* Precedence (increasing) and associativity */
%left "+"
%left "*" %

%%
input : expr { *expression = $1; } ;
expr : expr[L] "+" expr[R] { $$ = createOperation( eADD, $L, $R ); }
    | expr[L] "*" expr[R] { $$ = createOperation( eMULTIPLY, $L, $R ); }
    | "(" expr[E] ")" { $$ = $E; }
    | "number" { $$ = createNumber($1);
} ; %%
```

main.c

```
int yyparse(SExpression **expression, yyscan_t scanner);
SExpression *getAST(const char *expr) {
    SExpression *expression;
    yyscan_t scanner;
    YY_BUFFER_STATE state;
    if (yylex_init(&scanner)) return NULL;
    state = yy_scan_string(expr, scanner);
    if (yyparse(&expression, scanner)) return NULL;
    yy_delete_buffer(state, scanner);
    yylex_destroy(scanner);
    return expression;
}

int evaluate(SExpression *e) {
    switch (e->type) {
        case eVALUE: return e->value;
        case eMULTIPLY: return evaluate(e->left) * evaluate(e->right);
        case eADD: return evaluate(e->left) + evaluate(e->right);
        default: /* should not be here */ return 0;
    }
}

int main(void) {
    char expr[256];
    scanf("%s",expr);
    SExpression *e = getAST(test);
    int result = evaluate(e);
    printf("Result of '%s' is %d\n", test, result);
    deleteExpression(e);
    return 0;
```

Expression.h

```
#ifndef __EXPRESSION_H__
#define __EXPRESSION_H__

typedef enum tagEOperationType {
    eVALUE,
    eMULTIPLY,
    eADD
} EOperationType;

typedef struct tagSExpression {
    EOperationType type; /* /< type of operation */
    int value; /* /< valid only when type is eVALUE */
    struct tagSExpression *left; /* /< left side of the tree */
    struct tagSExpression *right; /* /< right side of the tree */
} SExpression;

SExpression *createNumber(int value);
SExpression *createOperation(EOperationType type, SExpression *left, SExpression *right);
void deleteExpression(SExpression *b);

#endif
```

Expression.c

```
static SExpression *allocateExpression() {
    SExpression *b = (SExpression *)malloc(sizeof(SExpression));
    if (b == NULL) return NULL;
    b->type = eVALUE;
    b->value = 0;
    b->left = NULL;
    b->right = NULL;
    return b;
}
SExpression *createNumber(int value) {
    SExpression *b = allocateExpression();
    if (b == NULL) return NULL;
    b->type = eVALUE;
    b->value = value;
    return b;
}
SExpression *createOperation(EOperationType type, SExpression *left, SExpression *right) {
    SExpression *b = allocateExpression();
    if (b == NULL) return NULL;
    b->type = type;
    b->left = left;
    b->right = right;
    return b;
}
void deleteExpression(SExpression *b) {
    if (b == NULL) return;
    deleteExpression(b->left);
    deleteExpression(b->right);
    free(b);
}
```

总结

一、句式分析的基本概念

二、LLVM案例分析

三、自顶向下分析

- LL(1)语言
- 通用算法：Earley算法

四、自底向上分析

- SLR、LALR、LR(1)语言
- 通用算法：GLR算法、CYK算法

五、语法分析工具