Compiler Design and Implementation

Grade 2020

Why Compiler?

- 锻炼能力：
  - 代码能力：这是ACM班标准大作业中首个预计代码量超过1万行代码的作业。
  - 工程能力：你需要从头完全实现一个编译器。
  - 学习能力：我们不会开编译原理课详细讲怎么编译，需要自己学习。
  - 同伴帮助能力：互相交流很重要。
- 能力证明：
  - 如果你能实现一个编译器本身就很了不起。
- 传承：这作业快有 10 年了。

Compiler

- 实现一个从前端到汇编代码的编译器。
- Language：任意，有特殊环境请联系助教。
- 仅允许使用ANTLR等词法分析库。
- 分为 4 个 Part：Semantic、Codegen、Optimization、Bonus
- Bonus：GC、（暂定）Lambda演算、（暂定）多面体

时间安排

Semester 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	Phase 1: Semantic																放假
						Phase 2A: Codegen(IR)											

Semester 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Phase 2B: RISC																	
Phase 3: Optimize																	

编译器评分

- 总分 100分
- Semantic 50分

- Codegen 75分

- Optimize 94分

- Bonus 100分

迟交规则

- 原则上没有迟交的说法。
- 恢复 2017 级及以前的迟交扣分准则：累进扣分制。
- 超过x天提交的扣分不超过 $\sum_{i=1}^{x} i$。
- 特殊情况请联系 TAs。

阶段考核规则

- 增加阶段考核准则：每隔约 2 周会进行阶段检查。
- 只检查完成程度，未达到 baseline 的会有惩罚性扣分。
- 特殊情况请联系 TAs。
- 预计：10/15，10/30，11/15，11/30，12/15，12/30（可能期末取消）
- 下学期预计：1/30，2/28，3/15，3/30，4/15，4/30

课程计划

- 主要为实践课程，预计安排 3-5 节的公共部分课程与 2-3 节提高部分课程。
- 公共部分：
  - 补充ICS基本知识（Locality/Linking/Relocation & Loading/Symbol）
  - 编译原理的前端知识（Parser）

Parser & Lexer

@peterzheng98

Grade 2020, Lecture 1

目录

- Lexer
- Parser

语言定义

- 如何精准定义一个语言?

字符：从一个字符集合中的字符开始

```
int main(){
   int a = 1;
   return a;
}
```

语言定义

- 如何精准定义一个语言?

```
int main(){
   int a = 1;
   return a;
}
```

字符：从一个字符集合中的字符开始

词法结构（Lexical Structure）：表征了词的概念

语言定义

- 如何精准定义一个语言?

```
int main(){
   int a = 1;
   return a;
}
```

字符：从一个字符集合中的字符开始

词法结构（Lexical Structure）：表征了词的概念

句法结构（Syntactic Structure）：表征了句的概念，由一组词组成

语言定义

- 如何精准定义一个语言?

```
int main(){
  int a = 1;
  return a;
}
```

字符：从一个字符集合中的字符开始

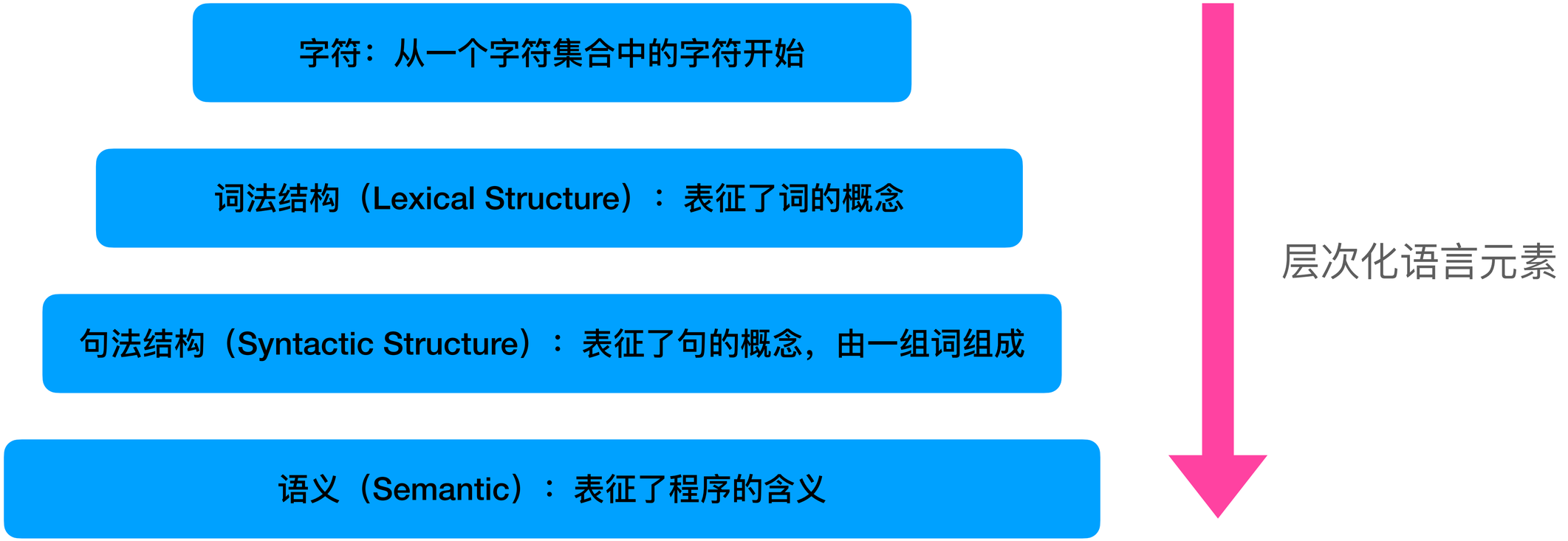
词法结构（Lexical Structure）：表征了词的概念

句法结构（Syntactic Structure）：表征了句的概念，由一组词组成

语义（Semantic）：表征了程序的含义

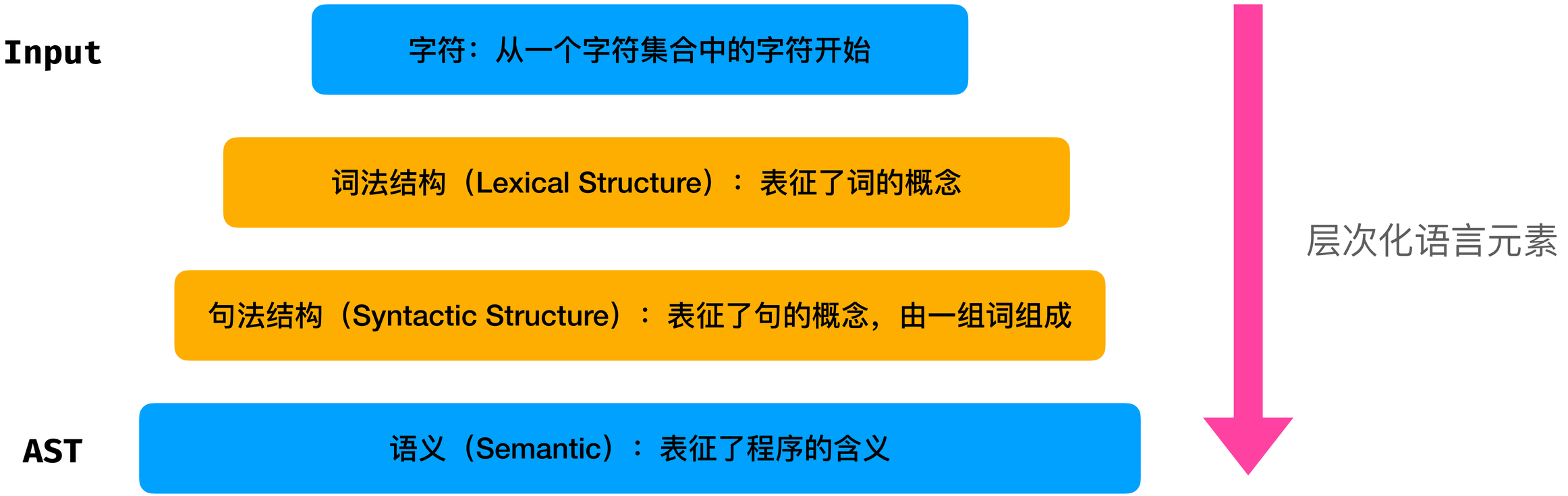
语言定义

- 如何精准定义一个语言?



语言定义

- 如何精准定义一个语言?



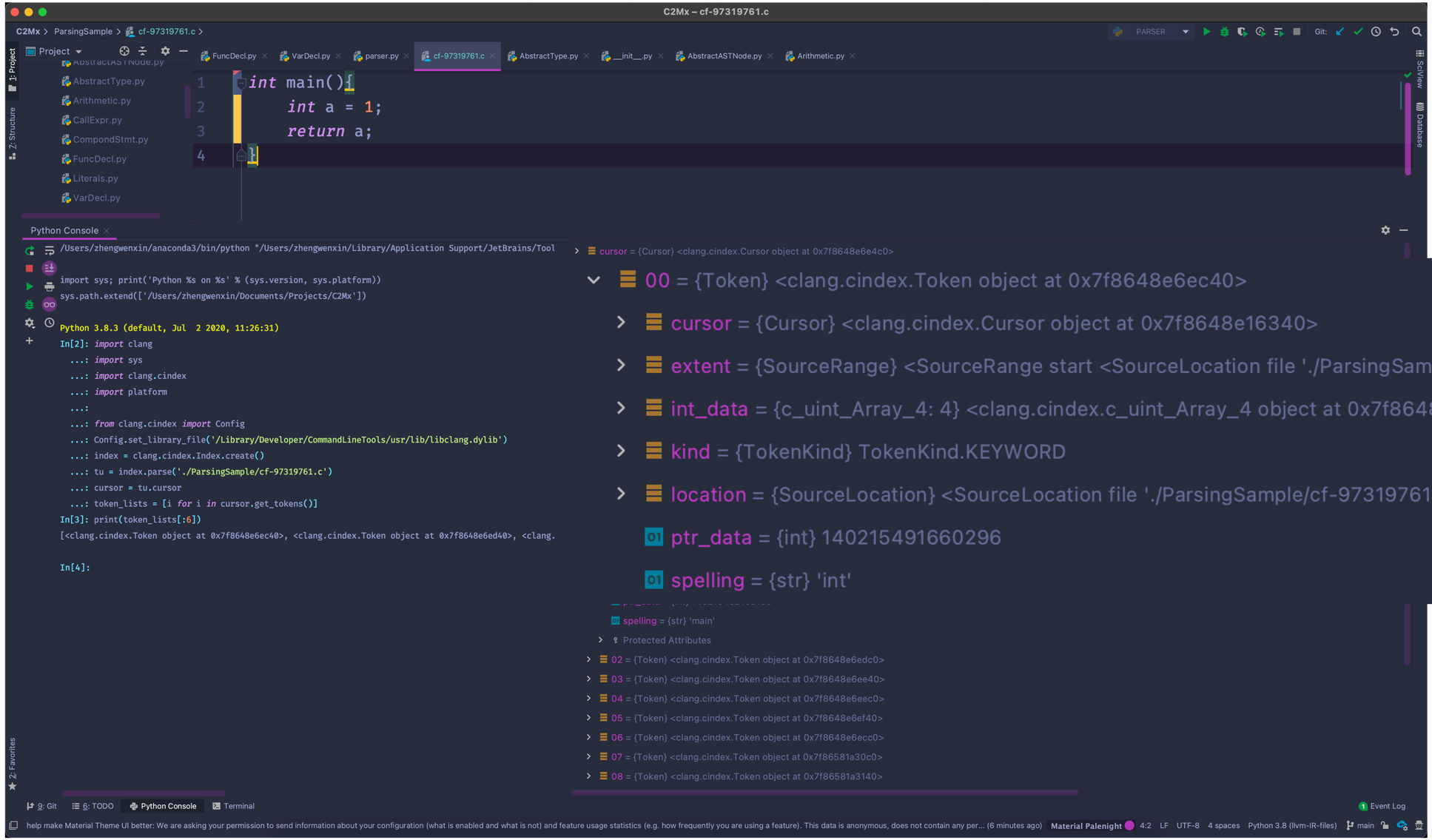
词法分析器 Lexer

- 词法分析
- 正则表达式
- 有限状态自动机
- 从正则表达式到自动机
- 例子

词法分析器 Lexer

- 流程：
  - 去除不必要的字符：注释，空格
  - 按照词素（token）类型分类：
    - 关键字（Keywords）、数字（Numbers）
    - 标点符号（Punctuation）、标识符（Identifiers）
  - 位置跟踪
  - 关联句法信息
- 目标：更容易解析

Clang Python Binding



词法分析器 Lexer

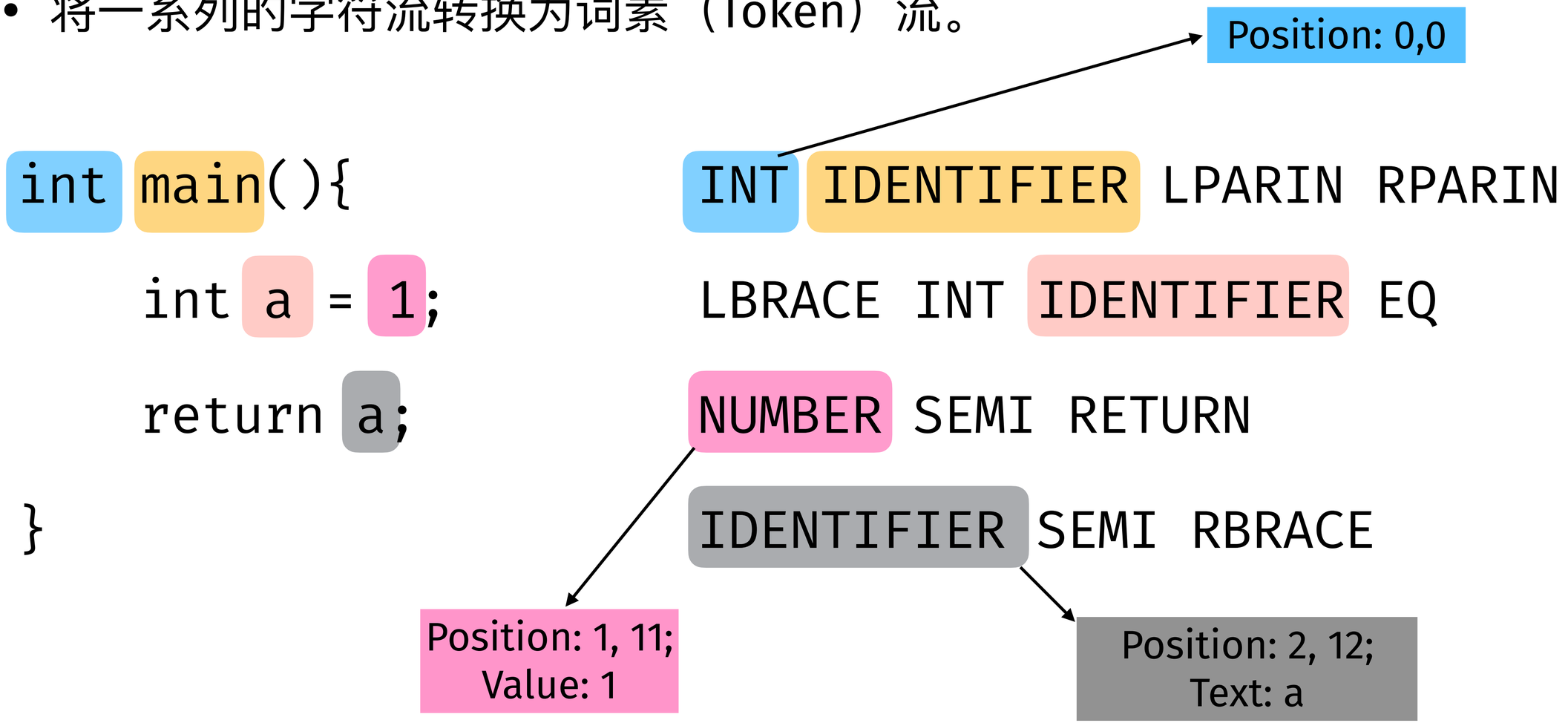
- 将一系列的字符流转换为词素（Token）流。

```
int main(){
    int a = 1;
    return a;
}
```

```
INT IDENTIFIER LPARIN RPARIN
LBRACE INT IDENTIFIER EQ
NUMBER SEMI RETURN
IDENTIFIER SEMI RBRACE
```

词法分析器 Lexer

- 将一系列的字符流转换为词素（Token）流。



词法分析器 Lexer

- 将一系列的字符流转换为词素（Token）流。

生成式方法：正则表达式、语法

识别式方法：自动机

正则语言 Regular Language

- 定义：给出一个有限字符串表 Σ，如下递归定义：
  - 空语言$\varnothing$，空字符串语言$\{\epsilon\}$是正则语言。
  - 对于每一个字符串$a \in \Sigma$，$\{a\}$是正则语言。
  - 如果 A 和 B 为正则语言，那么 $A \cup B, A \cdot B, A*$ 均为正则语言。
  - 其余在 Σ 的语言都不是正则语言。
- 换个定义方式：可以用正则表达式表达的语言。

正则表达式 Regular Expression

- 正式定义：
  - 基本情况：单个表中字符串，空字符串。
  - 四个运算：
    - 连接 RS：两两连接字符串。
    - R = {“ab”, “c”}, S = {“d”, “ef”}
    - RS = {“abd”, “abef”, “cd”, “cef”}

正则表达式 Regular Expression

- 正式定义：
  - 基本情况：单个表中字符串，空字符串。
  - 四个运算：
    - 集合并 R|S：连接集合。
    - R = {“ab”, “c”}, S = {“ab”, “d”, “ef”}
    - R|S = {“ab”, “c”, “d”, “ef”}

正则表达式 Regular Expression

- 正式定义：
  - 基本情况：单个表中字符串，空字符串。
  - 四个运算：
    - Kleene Star R*：表示包含ϵ且在字符串串接运算下闭合的的最小超集。这是可以通过R中零或有限个字符串的串接得到所有字符串的集合。
    - R={“0”, “1”}
    - R* = {空串, “0”, “1”, “00”, “01” ...} 表达所有有限二进制串。

正则表达式 Regular Expression

- 正式定义：
  - 基本情况：单个表中字符串，空字符串。
  - 四个运算：
    - 组 （R）：以组为单位。
  - 优先级：组、Kleene Star、连接、集合并
    - a|b*: {空串, a, b, bb, bbb, ...}
    - (a|b)*: {空串, a, b, aa, ab, ba, bb, aaa ...}

正则表达式 Regular Expression

- 例子：“a”, “b”
  - 偶数个a：
  - 奇数个b：
  - 偶数个a或奇数个b：

正则表达式 Regular Expression

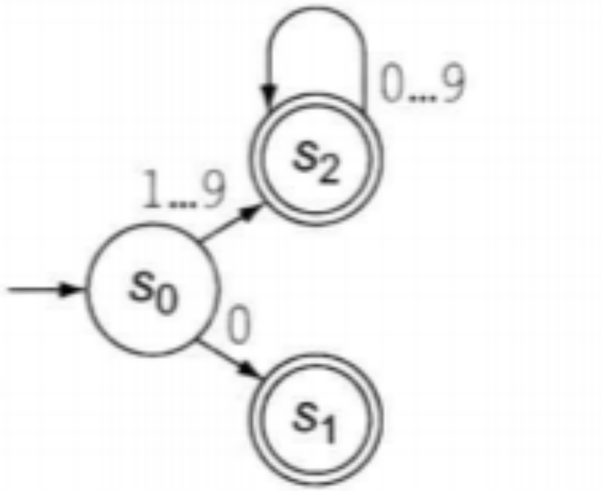
- 例子：“a”, “b”
  - 偶数个a：RA = b*(ab*ab*)*
  - 奇数个b：RB = a*ba*(ba*ba*)*
  - 偶数个a或奇数个b：RA | RB

正则表达式 Regular Expression

- 为什么使用正则表达式?
- 重要性质:
  - 非确定性有限自动机（NFA）、确定性有限自动机（DFA）接受的语言
  - 由正则语法或前缀语法生成
  - 交替有限自动机、双向有限自动机接受的语言
  - 可以由只读图灵机接受

有限状态自动机 Finite State Automaton

- 有限状态集合
- 通过五元组描述状态转移过程：
  - Σ是输入字母表（符号的非空有限集合）
  - S是状态的非空有限集合
  - δ转移函数 $\delta : S \times \Sigma \rightarrow S$
  - s_0初始状态，$s_0 \in S$
  - F最终状态集合，$F \subseteq S$

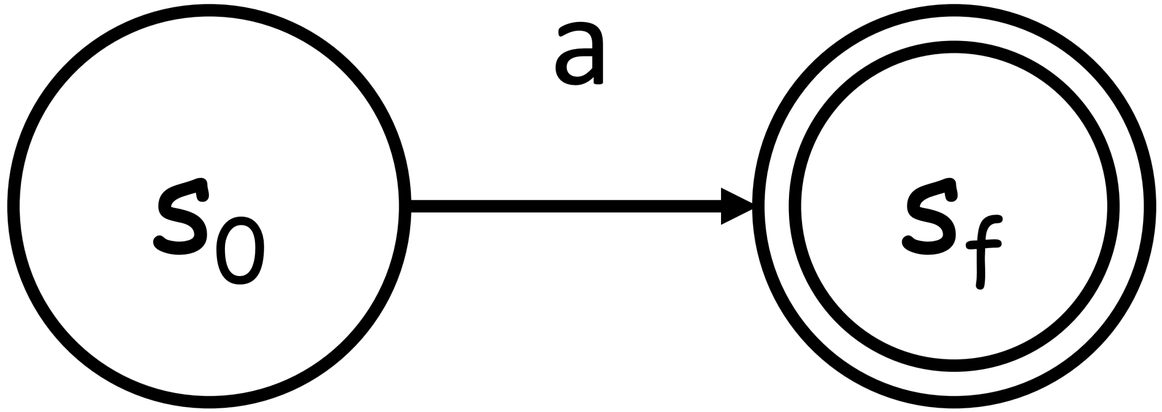


确定 v.s 非确定： DFA v.s. NFA

- NFA 非确定有限状态自动机：每个状态和输入符号对可以有多个可能的下一个状态的有限状态自动机。
- DFA 确定有限状态自动机：下一个状态是唯一确定的。
- 等价性：
  - 如果NFA可以识别一种语言，那么DFA也可以识别该语言，反之亦然。

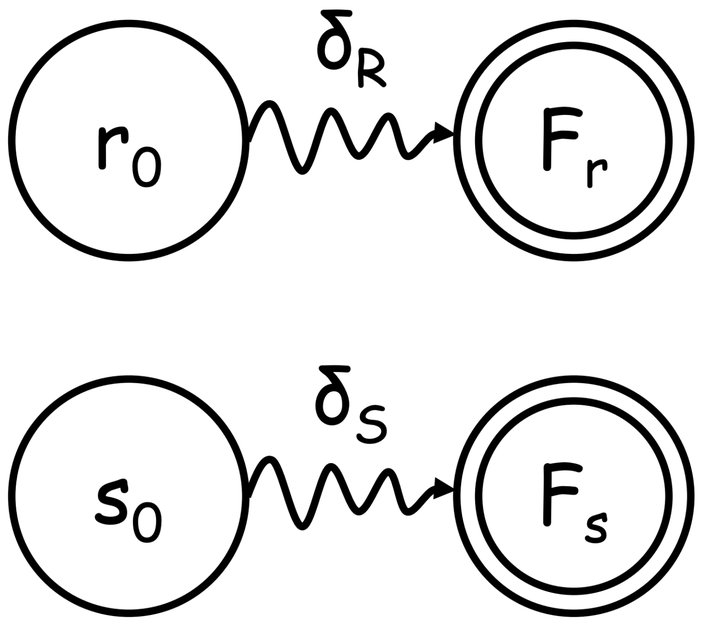
正则表达式和状态机的转换：基本情况

- 集合任意字符串：$a \in \Sigma, M_a = \{\Sigma, \{s_0, s_f\}, \delta, s_0, \{s_f\}\}$，$a$ 可以是空串。



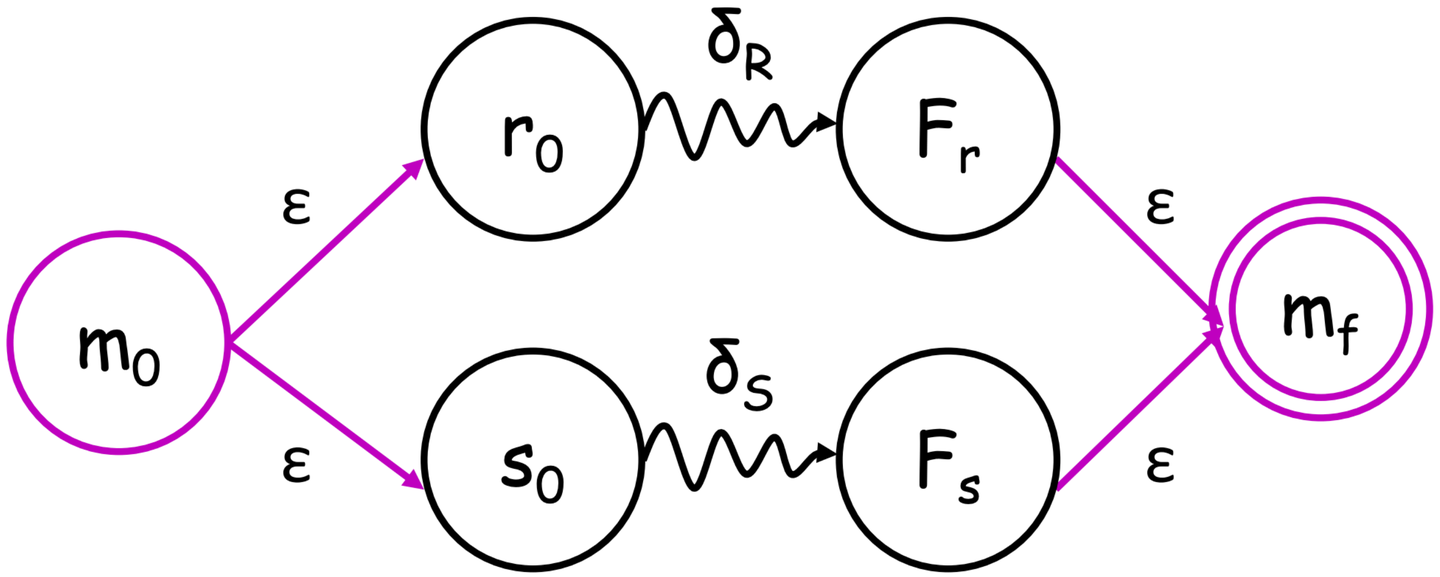
正则表达式和状态机的转换：基本运算

- 正则表达式对应状态机 $M_S = \{\Sigma, s_S, \delta_S, s_0, F_S\}, M_R = \{\Sigma, s_R, \delta_R, r_0, F_R\}$



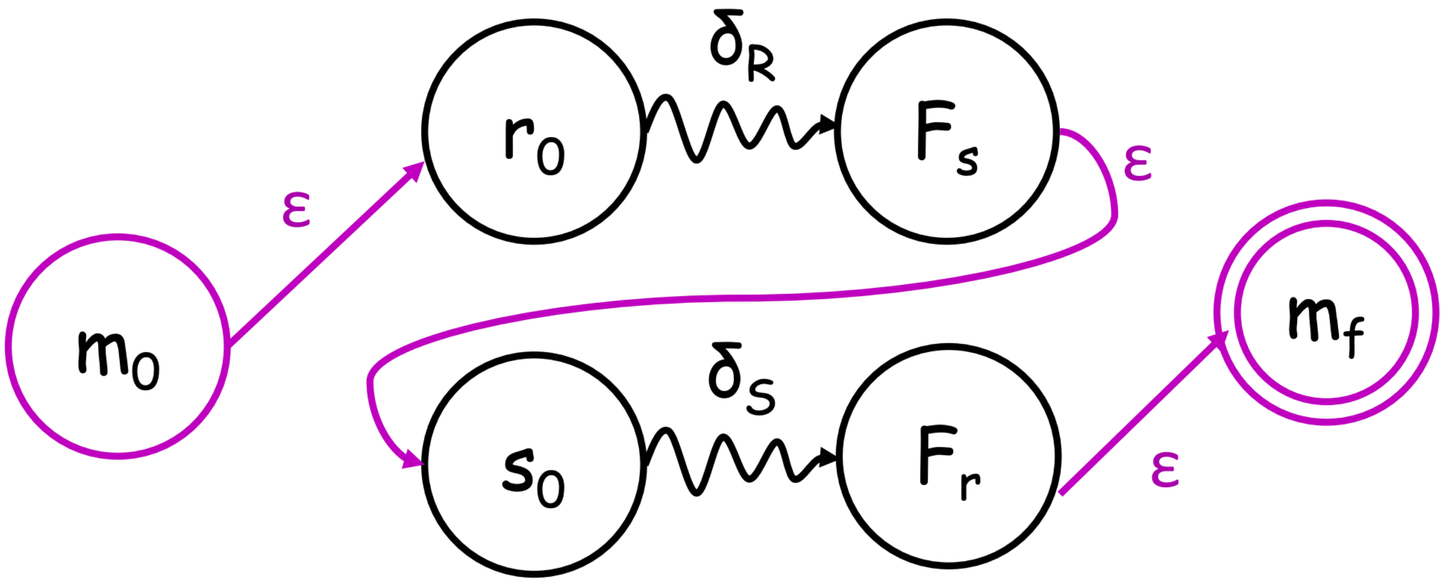
正则表达式和状态机的转换：基本运算

- 正则表达式对应状态机 $M_S = \{\Sigma, s_S, \delta_S, s_0, F_S\}, M_R = \{\Sigma, s_R, \delta_R, r_0, F_R\}$
- $M_{R|S} = \{\Sigma, s_S \cup s_R \cup \{m_0, m_f\}, \delta_{R|S}, m_0, m_f\}$



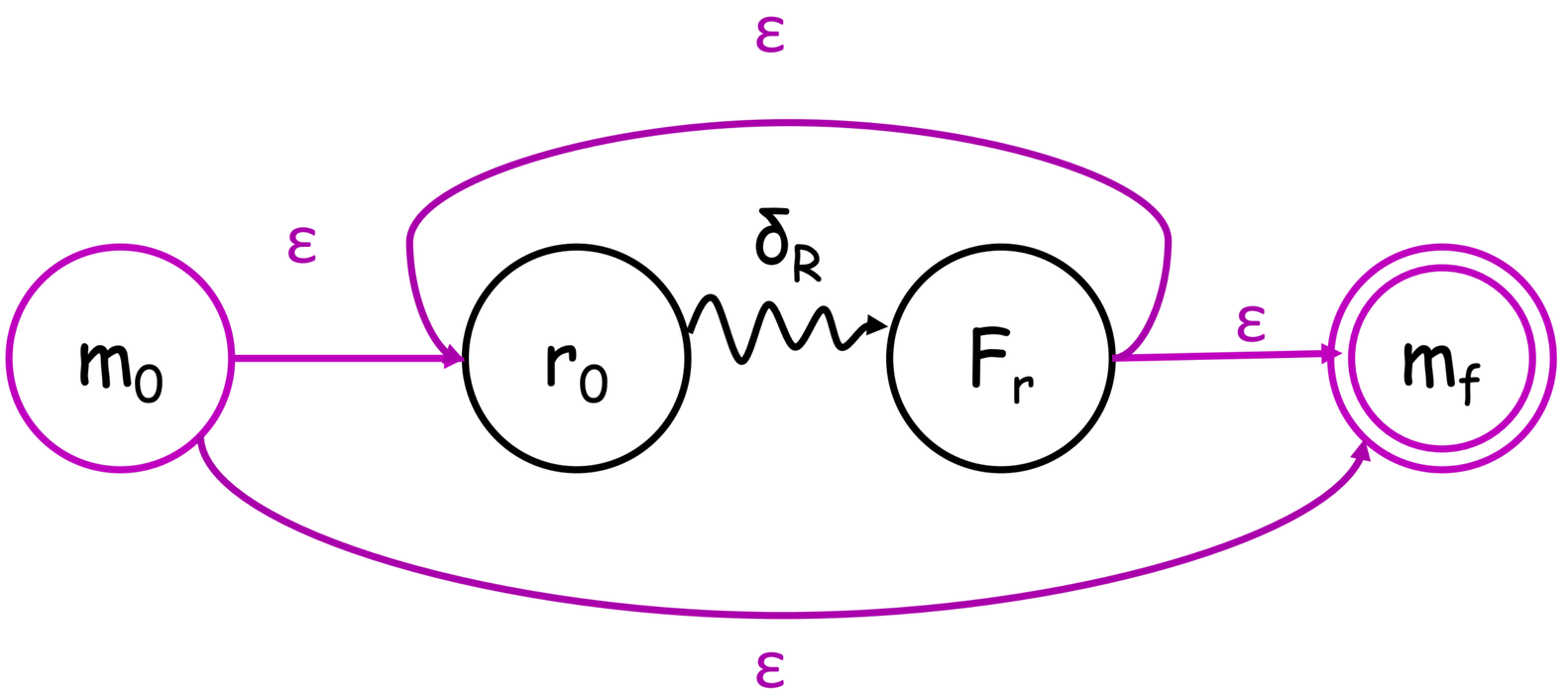
正则表达式和状态机的转换：基本运算

- 正则表达式对应状态机 $M_S = \{\Sigma, s_S, \delta_S, s_0, F_S\}, M_R = \{\Sigma, s_R, \delta_R, r_0, F_R\}$
- $M_{RS} = \{\Sigma, s_S \cup s_R \cup \{m_0, m_f\}, \delta_{RS}, m_0, m_f\}$



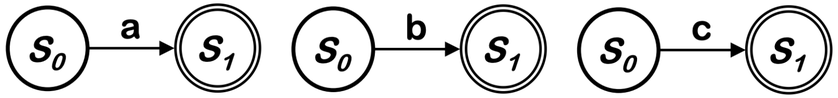
正则表达式和状态机的转换：基本运算

- 正则表达式对应状态机 $M_R = \{\Sigma, s_R, \delta_R, r_0, F_R\}$
- $M_{R*} = \{\Sigma, s_R \cup \{m_0, m_f\}, \delta_{R*}, m_0, m_f\}$

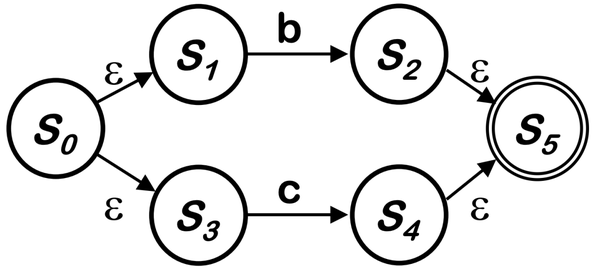


正则表达式和状态机的转换：基本运算

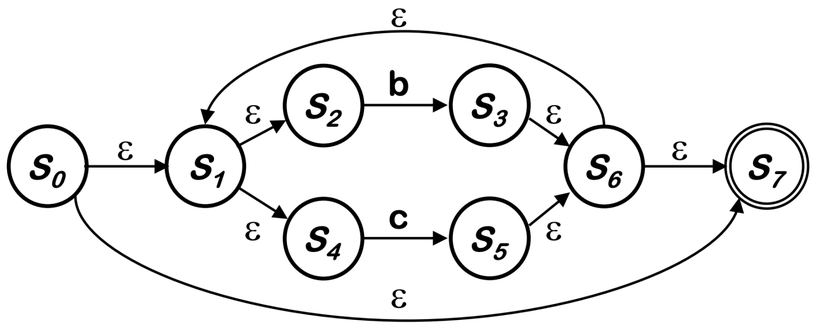
- 试一试 a(b|c)*
- a, b, c:



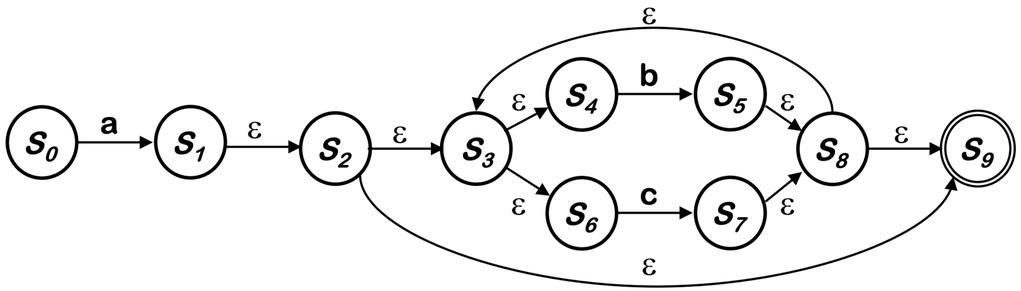
- b|c:



(b|c)*:

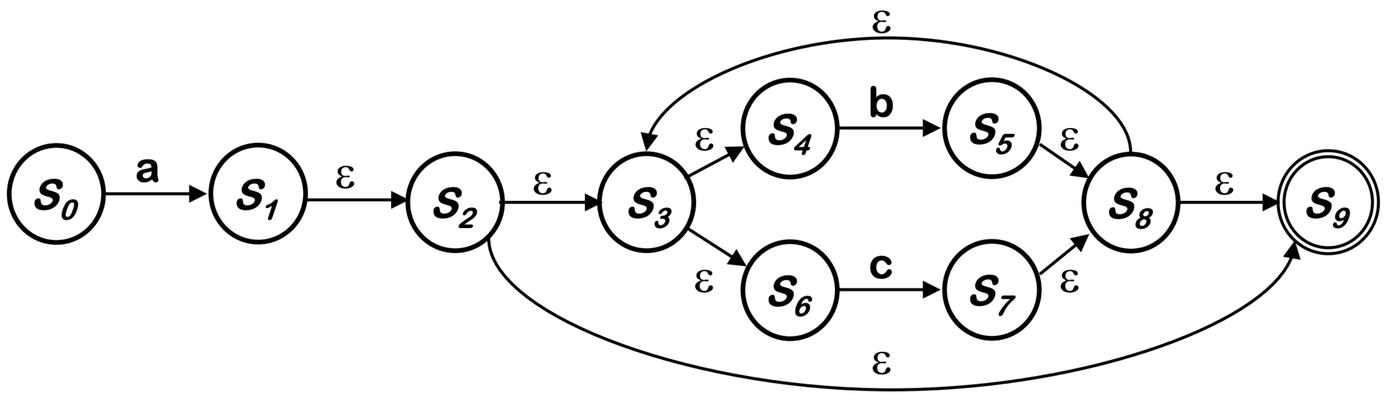


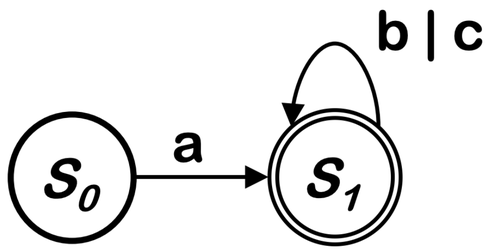
- a(b|c)*:



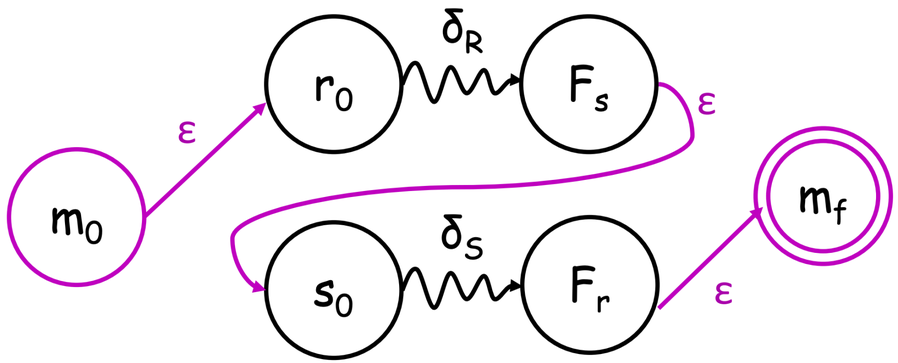
正则表达式和状态机的转换：基本运算

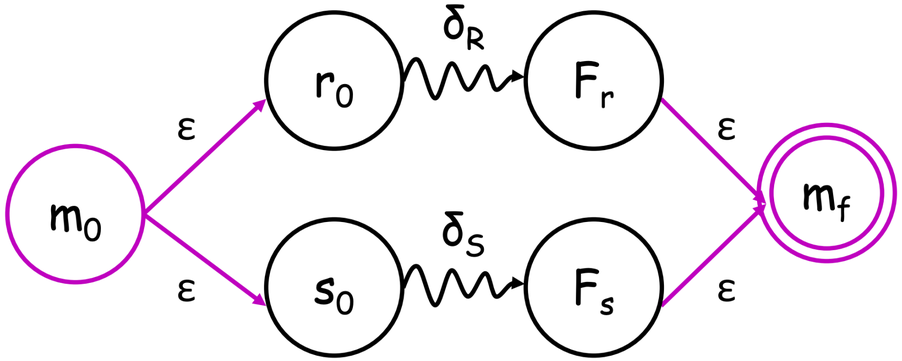
- 试一试 a(b|c)*
- a(b|c)*:

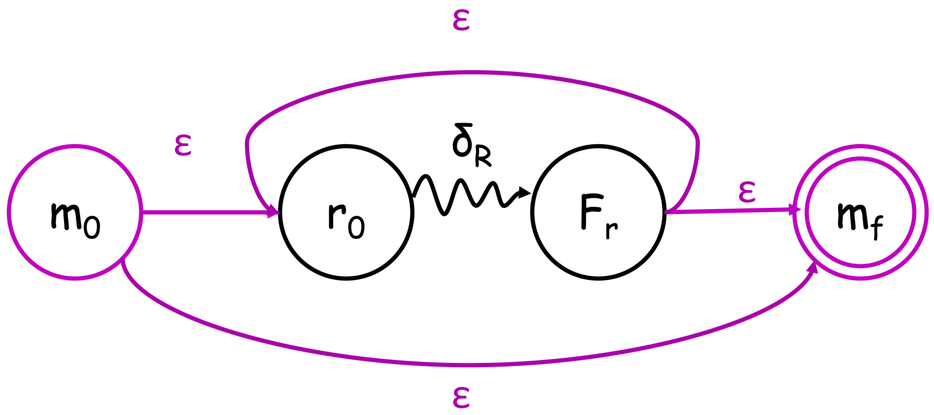




正则表达式和状态机的转换：基本运算







- 通过这个方法生成的状态机不一定是最优的。
- 可以执行简化

有用的东西：无符号整数

- 整数：非空数字字符串

```
digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
number = digit digit*
```

- 简写：$A^+ = A\ A^*$

```
number = digit+
```

有用的东西：标识符

- 标识符：字母或某个符号开头，由26个大小写英语字母，数字，下划线（_）

```
digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
letter = ‘A’ | … | ‘Z’ | ‘a’ | … | ‘z’ | ‘_’
identifier = letter (letter | digit) *
```

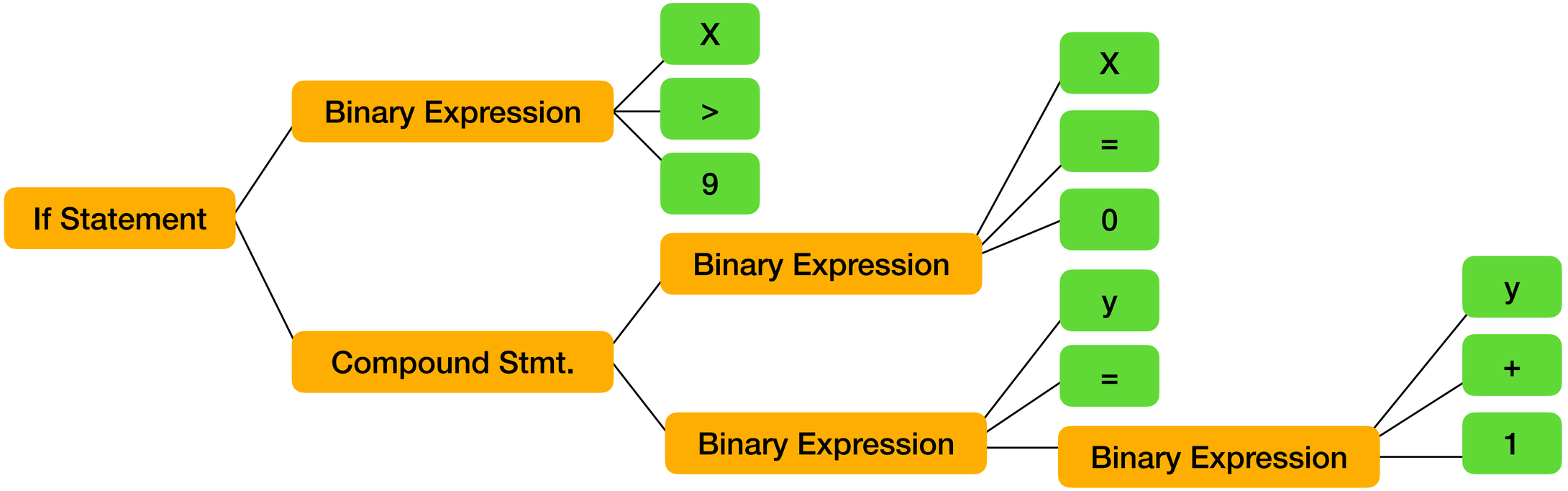
语法分析器 Parser

- 文法和语言
- 正则文法
- 上下文无关文法
- 自顶向下分析：LL(1)
- 自底向上分析：LR

语法分析器 Parser

- 对比：
  - 词法分析：字符 变为 词素
  - 语法分析：词素 变为 语法树/解析树

if(x>9){ x = 0; y = y + 1; }

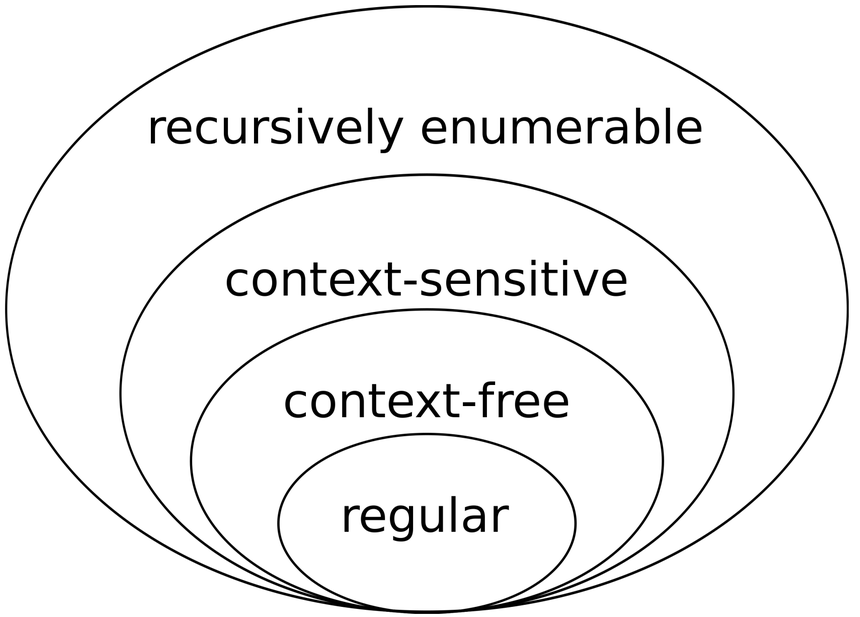


文法和语言 Grammars and Languages

- 一个文法识别一个语言，文法定义：
  - A 是“非终结”符号或变量的有限集合。它们表示在句子中不同类型的短语或子句。
  - Σ 是“终结符”的有限集合，构成句子实际内容。
  - S 是开始变量，$S \in A$。
  - P 是生成关系。
- 语言推导：按照生成关系从开始变量生成到给定字串w。
- 文法定义的语言指**所有可以通过S开始，关系P推导得到的字串**。

乔姆斯基分层 Chomsky hierarchy

文法	语言	自动机	产生式规则
0-型	递归可枚举语言	图灵机	α -> β（无限制）
1-型	上下文相关语言	线性有界非确定图灵机	αAβ -> αγβ
2-型	上下文无关语言	非确定下推自动机	A -> γ
3-型	正则语言	有限状态自动机	A -> aB A -> a



正则文法 Regular Grammar

- 正则表达式和非确定有限状态机可以由正则文法导出。
- 例子：a*bc*
  - S → aS，S → bA，A → cA，A → ϵ
  - 推导过程：S → aS → aaS → aabA → aabcA → aabc
- 这个正则文法是右正则文法，原因是非终结符号在右边。
- 左正则文法： A→a，A→Ba，A→ϵ，B→ϵ
- 右正则文法：A→a，A→aB，A→ϵ，B→ϵ

上下文无关文法 Context Free Grammar

- 一个文法识别一个语言，文法定义：
  - A 是“非终结”符号或变量的有限集合。
  - Σ 是“终结符”的有限集合。
  - S 是开始变量，$S \in A$。
  - P 是生成关系，要求生成关系都有$v \rightarrow a, v \in A, a \in (V \cup \Sigma)*$。
- 为什么是上下文无关的?
  - 生成关系左侧只有1个非终结符号。

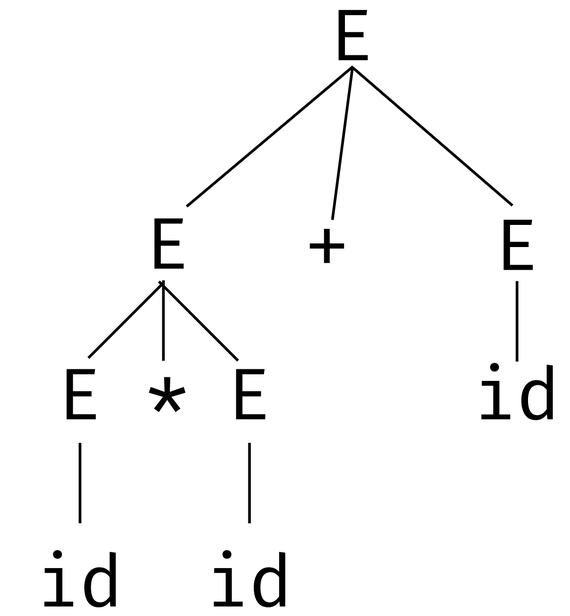
上下文无关文法 Context Free Grammar

- 基于上下文无关文法的推导例子：
  - 文法：$E \rightarrow E+E|E*E|(E)|id$
  - 字符串: id*id+id

上下文无关文法 Context Free Grammar

- 基于上下文无关文法的推导例子：
  - 文法：$E \rightarrow E+E|E*E|(E)|id$
  - 字符串: id*id+id

E
→ E+E
→ E*E+E
→ id*E+E
→ id*id+E
→ id*id+id

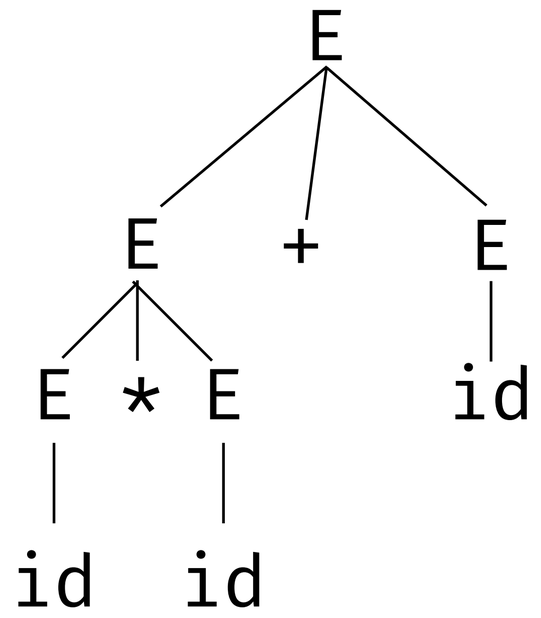


推导过程可以描述成一棵解析树。

上下文无关文法 Context Free Grammar

- 基于上下文无关文法的推导例子：
  - 文法：$E \rightarrow E+E \mid E*E \mid (E) \mid id$
  - 字符串: id*id+id

E	E
→ E+E	→ **E*E**
→ E*E+E	→ E*E+E
→ id*E+E	→ E*E+id
→ id*id+E	→ E*id+id
→ id*id+id	→ id*id+id

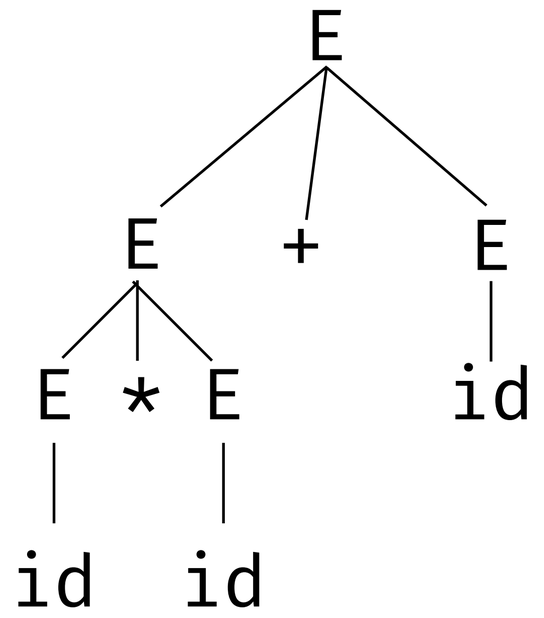


这个推导可以吗?

上下文无关文法 Context Free Grammar

- 基于上下文无关文法的推导例子：
  - 文法：$E \rightarrow E+E|E*E|(E)|id$
  - 字符串: id*id+id

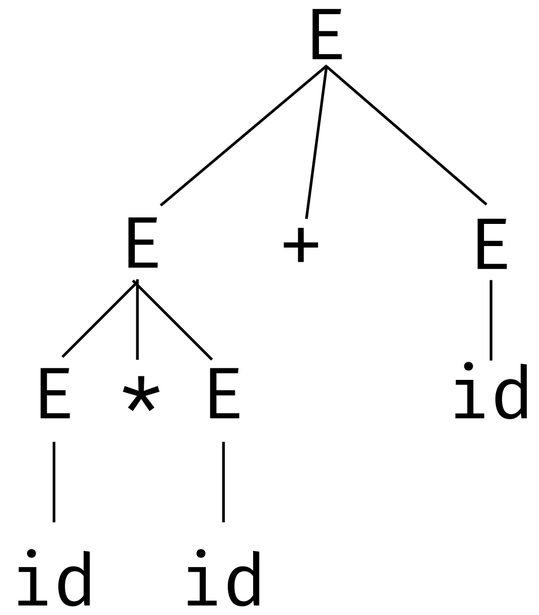
```
    E                E
→   E+E          →   E+E
→   E*E+E        →   E+id
→   id*E+E       →   E*E+id
→   id*id+E      →   E*id+id
→   id*id+id     →   id*id+id
```



还可以有右侧最先推导

推导树/语法树/分析树 Parse Tree

- 语法树/推导树/分析树：
  - 终结节点为叶节点。
  - 非终结节点为根节点或分支节点。
- 语法树叶子节点的”中序遍历”就是输入。
- 语法树表达了词素之间的结合关系，字符串不表达。



对CFG进行文法分析 Parsing a CFG

- 自上而下的分析：
  - 从根节点开始分析
  - 选择一个规则并且按照输入展开
  - 可能需要回溯，若不需要回溯，可以预测结果
- 自下而上的分析：
  - 从叶子节点
  - 根据规则识别前缀
  - 需要通过状态修改和输入识别进行匹配
  - 使用堆栈跟踪状态

可预测文法分析器 Predictive Parser

- 文法分析器可以预测使用哪一条规则
  - 提前查看接下来若干个词素
  - 不需要回溯
- 可预测文法分析器接受LL(k)语言，一般使用LL(1)。
  - 第一个L：从左往右扫描。
  - 第二个L：左侧最先推导（Left-most Derivation）
  - K：查看接下来k个词素。

LL(1) 文法

- LL(1): 对每个非终结节点和词素，只有一个生成规则可以成功。
- 写LL(1)是简单的。
- 可以用一张表格描述LL(1)文法：
  - 一维是待展开的当前非终结节点
  - 还有一维是下一个词素
  - 表格元素是生成规则

LL(1) 文法

```
S → if E then S else S
S → begin S L
S → print E
L → ; S L
L → end
E → num = num
```

```
enum token{IF,THEN,ELSE,BEGIN,END,PRINT,SEMI,NUM,EQ}
extern enum token getToken(void);
enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) { if (tok==t) advance(); else error();}
```

LL(1) 文法

```
S → if E then S else S  | begin S L | print E
void S(void) {
  switch(tok) {
  case IF:
  eat(IF); E(); eat(THEN); S(); eat(ELSE); S(); break;
  case BEGIN:
  eat(BEGIN); S(); L(); break;
  case PRINT:
  eat(PRINT); E(); break;
  default:error();
  }
}
```

LL(1) 文法

- 考虑以下这样的一个文法：
  - S → AddExp
  - AddExp → AddExp opt1 MulExp | MulExp
  - opt1 → + | -
  - MulExp → MulExp opt2 Exp | Exp
  - opt2 → * | /
  - Exp → (AddExp) | alphabet
- 是LL(1)文法吗?

LL(1) 文法

- 考虑以下这样的一个文法：
  - S → AddExp
  - AddExp → **AddExp** opt1 MulExp | MulExp
  - opt1 → + | -
  - MulExp → **MulExp** opt2 Exp | Exp
  - opt2 → * | /
  - Exp → (AddExp) | alphabet

无法确定 AddExp 和 MulExp 以什么终结符开头

可以无限展开 AddExp 的非终结节点

LL(1) 文法

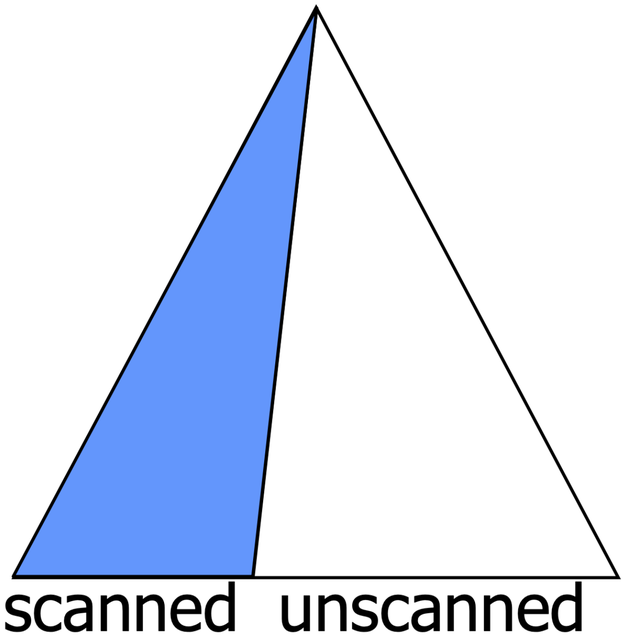
- 左递归的消除：
  - A → A a | b
  - 改写为如下
  - A → b A’, A’ → a A’ | e

自底向上分析 Botton-Up Parsing

- 自底向上比自顶向下更加一般化
  - 效率差不多
  - 想法和自顶向下有类似
  - 实际上现实中更多使用
- 称为LR分析
  - L表示从左像右
  - R表示右侧最先推导（Right-most Derivation）

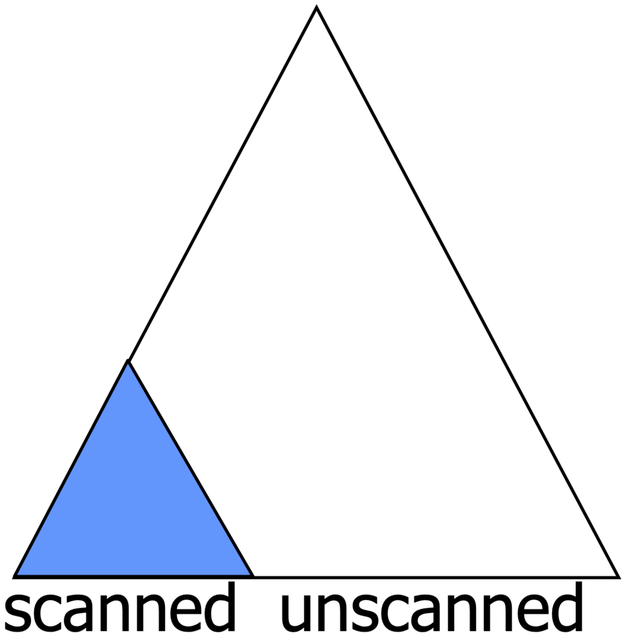
上到下还是下到上?

LL(k), recursive descent



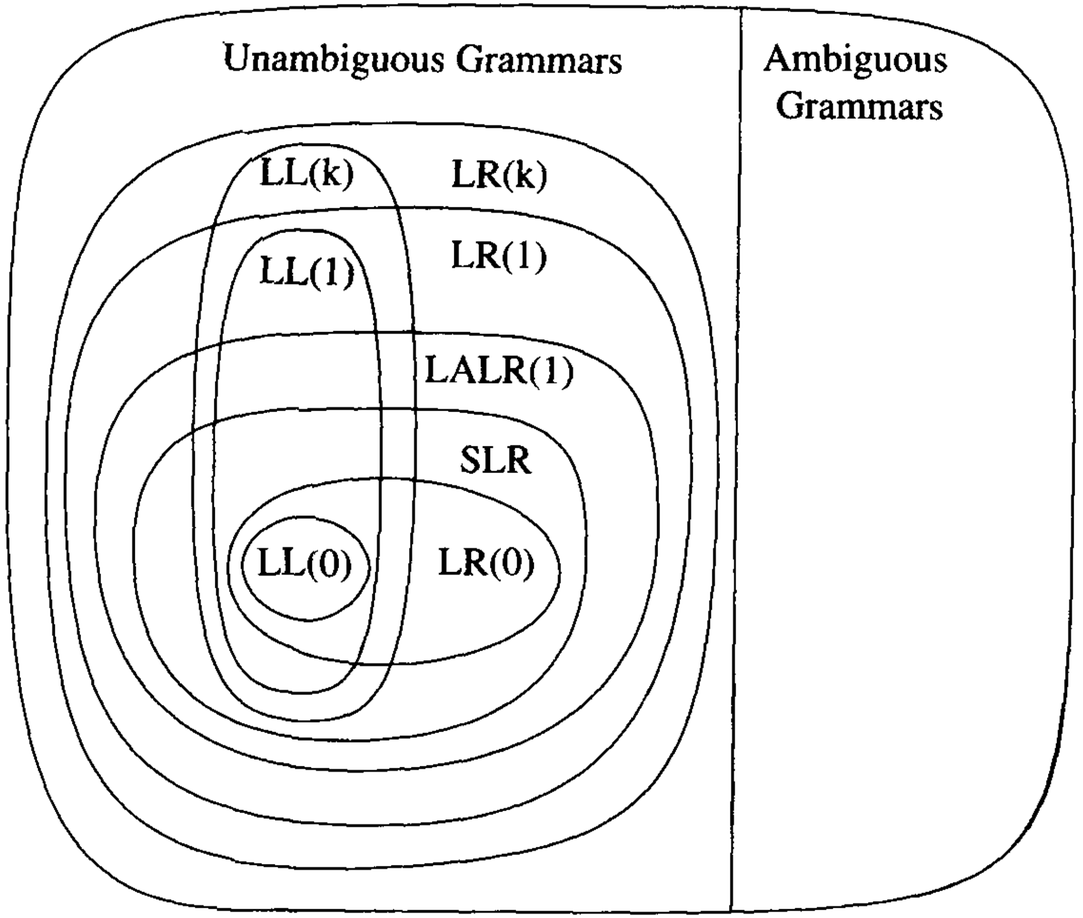
Top-down

LR(k), shift-reduce



Bottom-up

语言层次结构分类

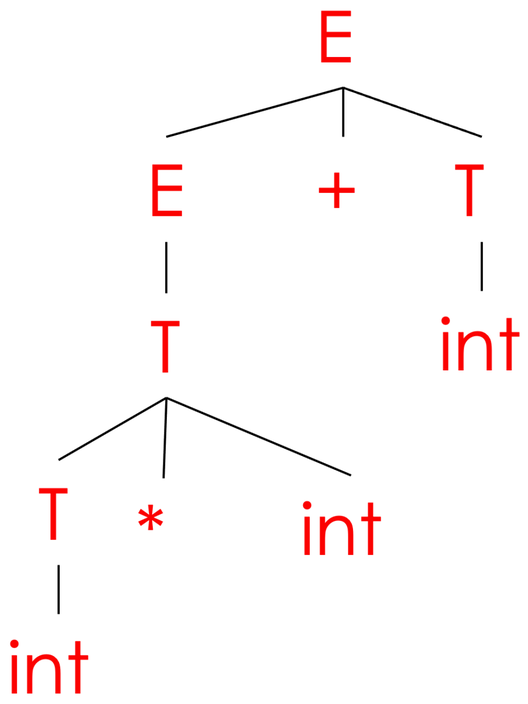

Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

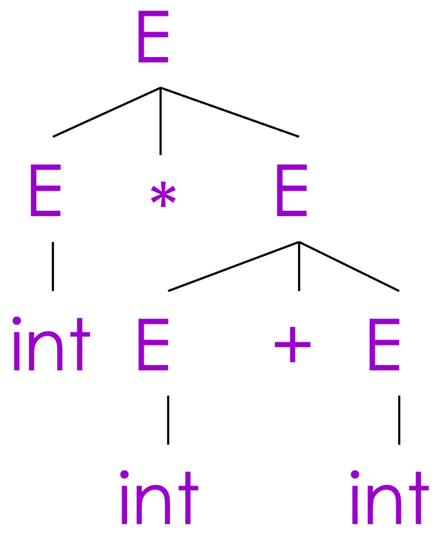
语言二义性 Ambiguity

- 一个语法是有歧义的如果对某个字符串有1种以上的语法分析树。
  - 也就是用左侧最先推导和右侧最先推导得到1种以上的语法分析树。

语言二义性 Ambiguity

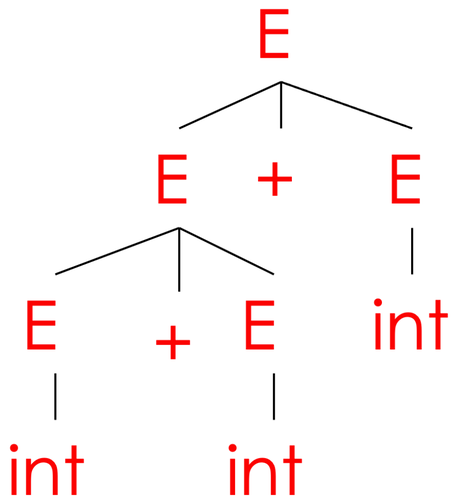
- `E → E + E | E * E | (E) | int, int * int + int`

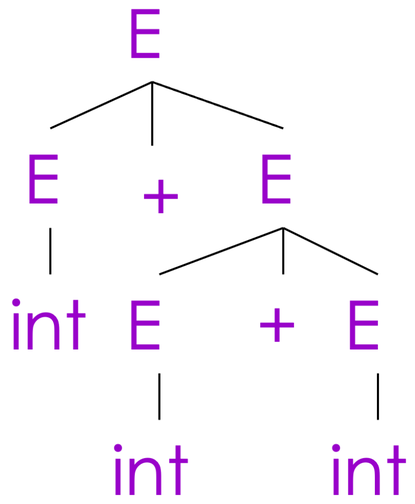




语言二义性 Ambiguity

- E → E + E | E * E | (E) | int, int + int + int





语言二义性 Ambiguity

- 二义性是不好的。
  - 导致部分程序的定义是未知的。

- 在程序中是经常出现的：
  - 算数语句
  - if then else等等

语言二义性 Ambiguity

- 解决二义性：
  - 规定优先级别
  - 规定结合律
- 例如：
  - `E → E + T | T, T → T * int | int | (E)`
  - 规定乘法优先级高于加法，规定必须向左结合。

参考资料 Reference

[1] CMU 15-411 Fall 20 Lecture 10
[2] MIT 6.035 Fall 18 Lecture 02, 03
[3] SE302 Compilers, Lecture 02, 03, 04, 05, 06
[4] https://www.tutorialspoint.com/automata_theory/context_free_grammar_introduction.htm
[5] https://en.wikipedia.org/wiki/Regular_expression
[6] https://en.wikipedia.org/wiki/Regular_language
[7] https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton
[8] M. O. Rabin and D. Scott, "Finite Automata and their Decision Problems", IBM Journal of Research and Development, 3:2 (1959) pp.115-125.
[9] Martin, John (2010). Introduction to Languages and the Theory of Computation. McGraw Hill. ISBN 978-0071289429.