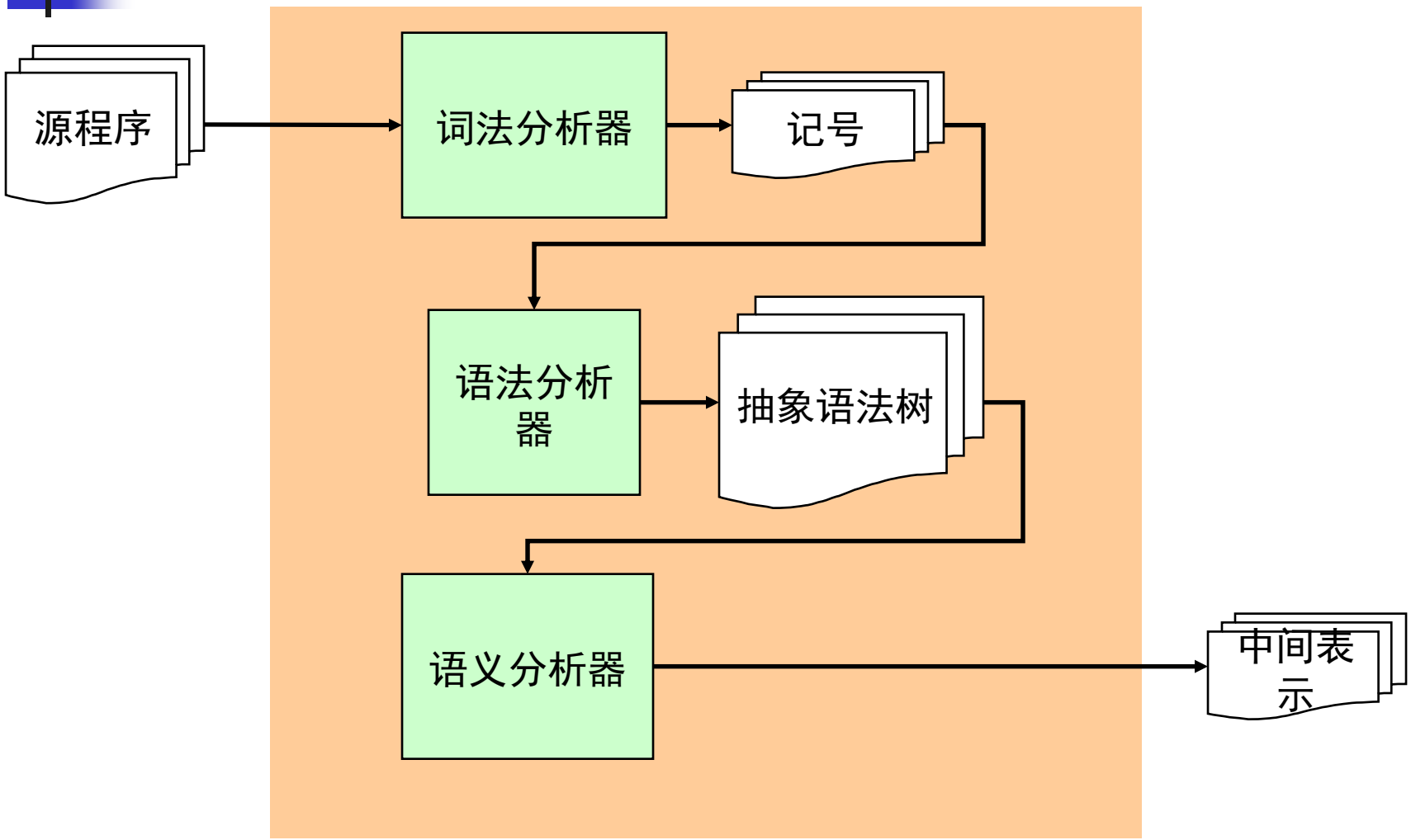
语法分析：LR(0)分析算法

编译原理

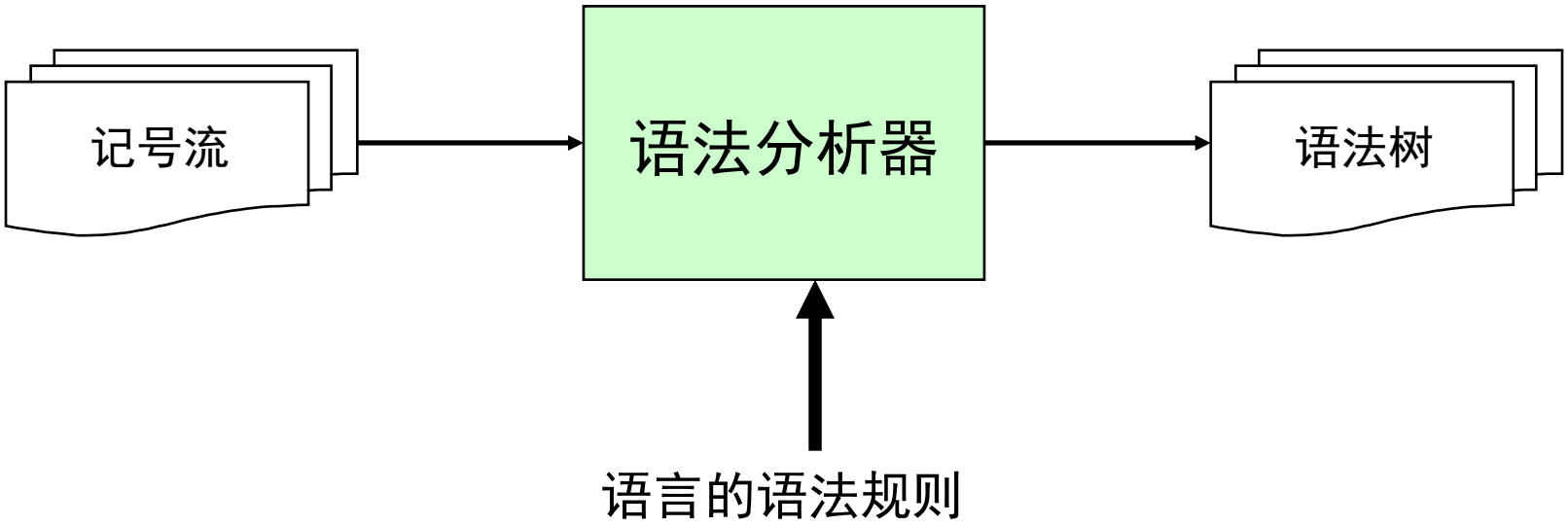
华保健

bjhua@ustc.edu.cn

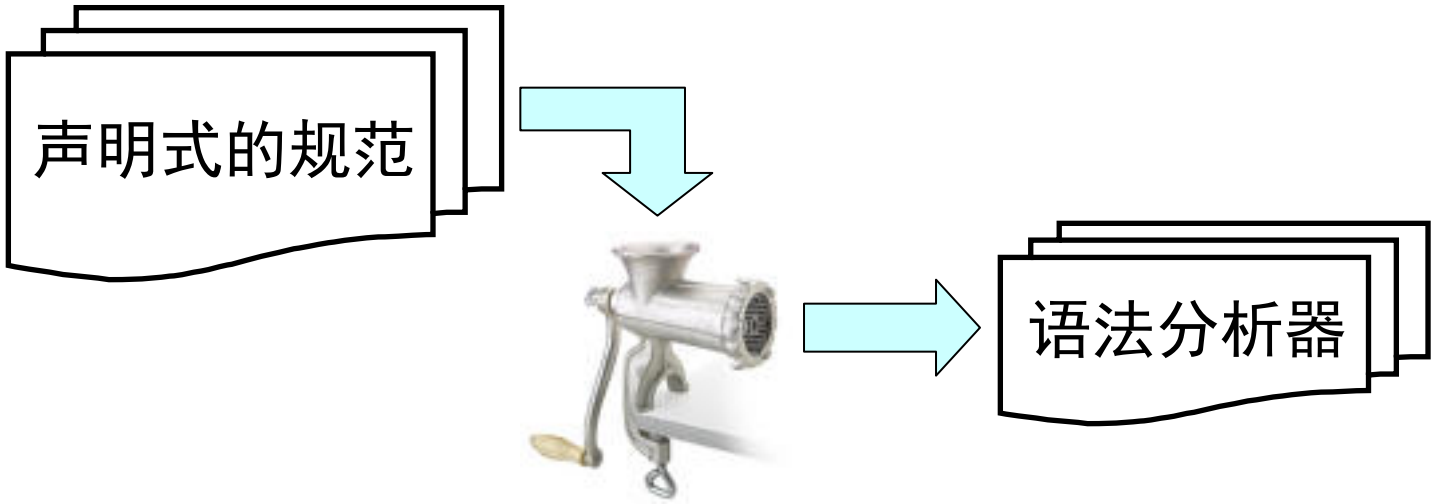
前端

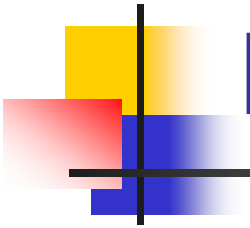


语法分析器的任务



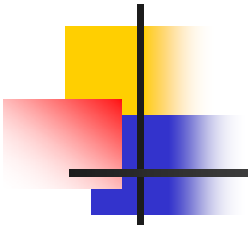
自动生成





LL(1)分析算法

- 从左（L）向右读入程序，最左（L）推导，采用一个（1）前看符号
  - 优点：
    - 算法运行高效
    - 有现成的工具可用
  - 缺点：
    - 能分析的文法类型受限
    - 往往需要文法的改写



自底向上分析算法

- 研究其中最重要也是最广泛应用的一类
  - LR分析算法（移进-归约算法）
    - 算法运行高效
    - 有现成的工具可用
  - 这也是目前应该广泛的一类语法分析器的自动生成器中采用的算法
    - YACC, bison, CUP, C#yacc, 等

自底向上分析的基本思想

```
0: S -> E
1: E -> E + T
2:    | T
3: T -> T * F
4:    | F
5: F -> n
```

2 + 3 * 4

F + 3 * 4

T + 3 * 4

E + 3 * 4

E + F * 4

E + T * 4

E + T * F

E + T

E

S

最右推导的逆过程!

点记号

- 为了方便标记语法分析器已经读入了多少输入，我们可以引入一个点记号 •

已经读入的　　剩余的输入

自底向上分析

2 + 3 * 4

F + 3 * 4

T + 3 * 4

E + 3 * 4

E + F * 4

E + T * 4

E + T * F

E + T

E

S

2 ● + 3 * 4

F ● + 3 * 4

T ● + 3 * 4

E + 3 ● * 4

E + F ● * 4

E + T * 4 ●

E + T * F ●

E + T ●

E ●

S ●

另外的写法

```
0: S -> E
1: E -> E + T
2:    | T
3: T -> T * F
4:    | F
5: F -> n
```

2	2 + 3 * 4
F	● + 3 * 4
T	● + 3 * 4
E	● + 3 * 4
E +	● + 3 * 4
E + 3	● 3 * 4
E + F	● * 4
E + T	● * 4
E + T *	● * 4
E + T * 4	● 4
E + T * F	●
E + T	●
E	●
S	●

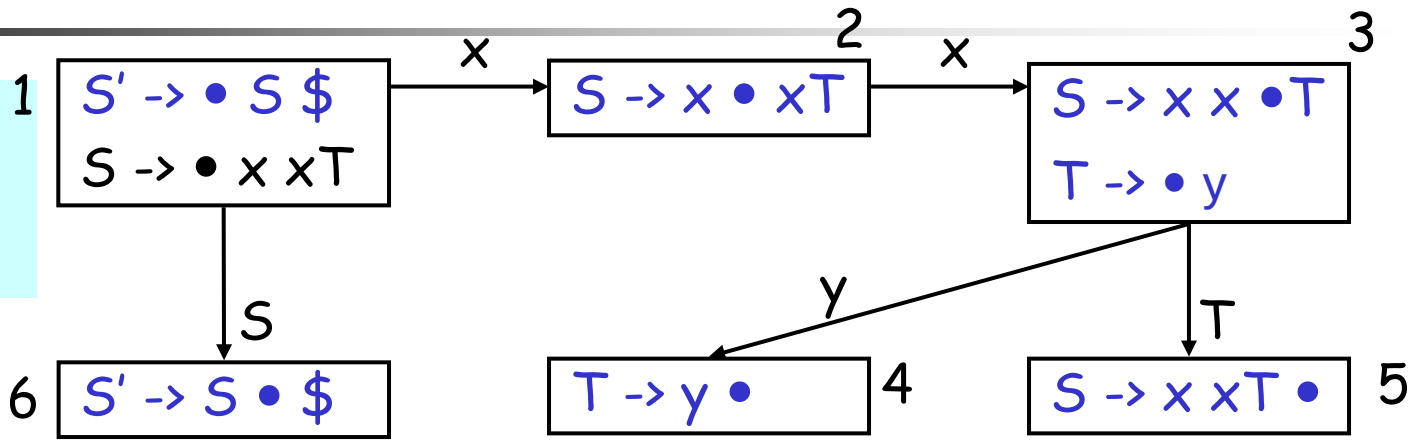
左方是什么数据结构？

生成一个逆序的最右推导

- 需要两个步骤：
  - **移进** 一个记号到栈顶上，或者
  - **归约** 栈顶上的n个符号（某产生式的右部）到左部的非终结符
    - 对产生式 A -> β1 … βn
      - 如果βn … β1在栈顶上，则弹出 βn… β1
      - 压入 A
- 核心的问题：如何确定移进和归约的时机？

算法思想

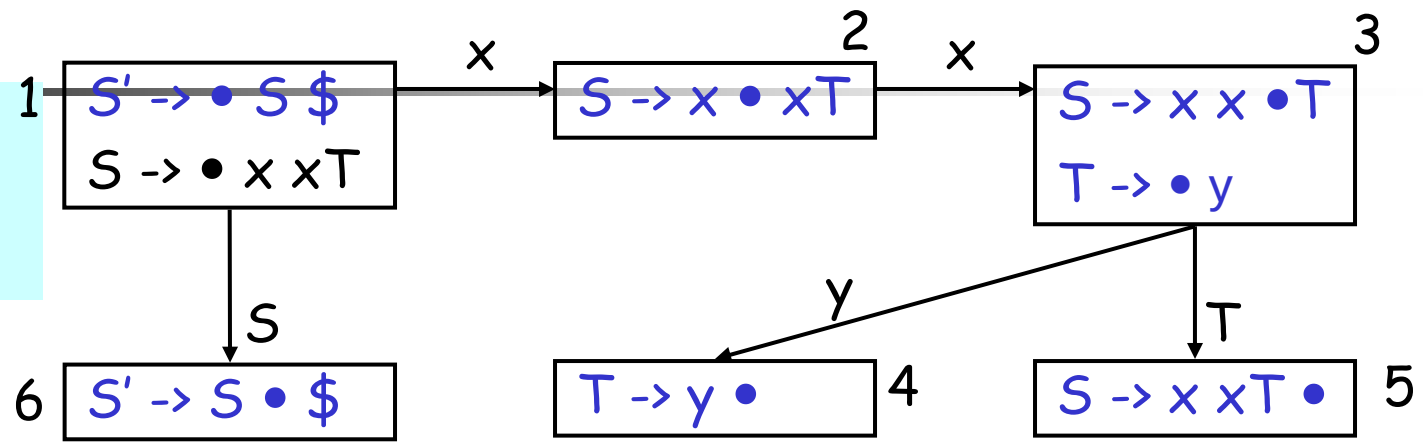
0: S'-> S$
1: S -> x x T
2: T -> y



分析这个输入串：x x y $

LR(0)分析表

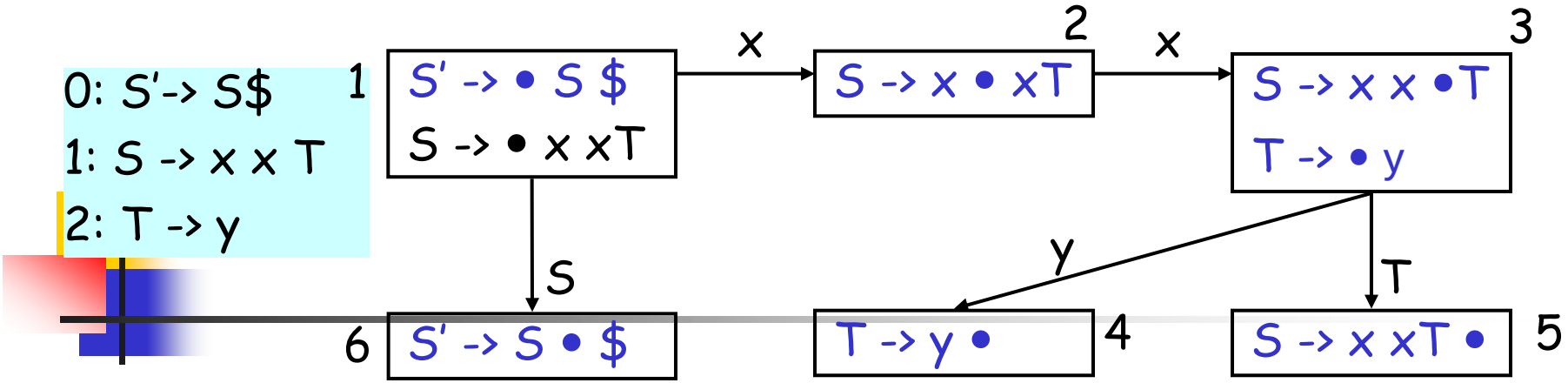
0: S'-> S$
1: S -> x x T
2: T -> y



	动作（ACTION）			转移（GOTO）	
状态\符号	x	y	$	S	T
1	s2			g6	
2	s3				
3		s4			g5
4	r2	r2	r2		
5	r1	r1	r1		
6			accept		

LR(0)分析算法

```
stack = []
push ($)        // $: end of file
push (1)        // 1: initial state
while (true)
  token t = nextToken()
  state s = stack[top]
  if (ACTION[s, t] == “si”)
    push (t); push (i)
  else if (ACTION[s, t] == “rj”)
    pop (the right hand of production “j: X -> β”)
    state s = stack[top]
    push (X); push (GOTO[s, X])
  else error (…)
```



分析这个输入串：x x y $

```
stack = []
push ($)        // $: end of file
push (1)        // 1: initial state
while (true)
  token t = nextToken()
  state s = stack[top]
  if (ACTION[s, t] == “si”)
    push (t); push (i)
  else if (ACTION[s, t] == “rj”)
    pop (the right hand of production “j: X -> β”)
    state s = stack[top]
    push (X); push (GOTO[s, X])
  else error (…)
```

LR(0)分析表构造算法

```
C0 = closure (S' -> • S $) // the init closure
SET = {C0}                 // all states
Q = enQueue(C0)            // a queue
while (Q is not empty)
  C = deQueue (Q)
  foreach (x∈ (N∪T))
    D = goto (C, x)
    if (x ∈ T)
      ACTION[C, x] = D
    else GOTO[C, x] = D
    if (D not ∈ SET)
      SET ∪= {D}
      enQueue (D)
```

goto和closure

```
goto (C, x)
  temp = {}            // a set
  foreach (C's item i: A -> β• x γ)
    temp ∪= {A -> βx • γ}
  return closure (temp)

closure (C)
  while(C is still changing)
    foreach (C's item i: A -> β• B γ)
      C ∪= {B -> …}
```